2012 IEEE 36th International Conference on Computer Software and Applications

Quality-Driven Self-Adaptation: Bridging the Gap between Requirements and Runtime Architecture by Design Decision

Liwei Shen, Xin Peng, Wenyun Zhao
School of Computer Science
Fudan University
Shanghai, P.R.China
{shenliwei, pengxin, wyzhao}@fudan.edu.cn

Abstract— **Running with static requirements and design decisions, a software system cannot always perform optimally in a highly uncertain and rapidly changing environment. Quality-driven self-adaptation, which enables a software system to continually adapt its structure and behavior to improve the overall quality satisfaction, thus becomes a promising capability of software systems. Existing researches on self-adaptive systems, although having proposed effective methods and techniques on requirements-driven self-adaptation and reflective components, do not well address the gap between requirements and runtime architecture. In this paper, we propose a quality-driven self-adaptation approach, which incorporates both requirements- and architecture-level adaptations. At the requirements level, value-based quality tradeoff decisions are made with the aim of maximizing system-level value propositions. At the architecture level, component-based architecture adaptations are conducted. To bridge the gap between requirements and runtime architecture, design decisions capturing alternative design options and their rationales are introduced to help map requirements adaptations and context changes to adaptation operations on the runtime architecture. To validate the effectiveness, we implement the approach based on a reflective component model and conduct an experimental study on it. The results show that the approach leads to better performance compared with traditional software and the overall quality satisfaction is kept maintained. Furthermore, the development effort is affordable but the approach still has shortage in extensibility.**

Keywords-self-adaptation; requirements; reflective component reconfiguration; design decision

I. INTRODUCTION

Traditional software systems are developed based on static requirements and design decisions. Software architectures are established conforming to the requirements and as a composition of a set of explicit design decisions [1]. This kind of requirements and design decisions are made at design time based on predictions on runtime environments, e.g. concurrent load and available resources. However, more and more modern software systems like service-oriented and cloud-based systems are running in highly uncertain and rapidly changing environments. And it is usual that decisions run well in one environmental setting may perform poorly in another setting.

Therefore, running with static requirements and design decisions, these systems cannot always perform optimally and the customer vision cannot be satisfied optimally at all time. Quality-driven self-adaptation, which enables a software system to continually adapt its structure and behavior to improve the overall quality satisfaction, thus becomes a promising capability of software systems. This kind of self-optimization, which means continually seeking opportunities to improve their own performance and efficiency, is one of the four aspects of self-management capabilities for self-adaptive systems [17].

According to the multi-layer reference architecture [18], a self-adaptive system consists of a set of interconnected components at the bottom layer, with the capability to support component creation, deletion and interconnection. At higher layers, the system manages its goals and plans for potential adaptations. Existing researches have proposed various reflective architecture and component models, e.g. Fractal [16], to support component-level adaptations. There are also researches on requirements-driven self-adaptation approaches [3][4][6], which incorporate requirements goal model and goal reasoning at runtime to support requirements-driven adaptation planning.

Existing researches, although having proposed effective methods and techniques on requirements-driven self-adaptation and reflective components, do not well address the gap between requirements and runtime architecture. Most requirements-driven self-adaptation approaches assume that each element (e.g. goal or feature) in a requirements model directly corresponds to one or several components/services in the runtime architecture and thus requirements-level adaptations can be simply mapped to component/service binding/unbinding or replacement. However, the mappings between requirements and architectural elements are usually quite complex. On the other hand, besides requirements-level decisions, there are also architecture-level technical decisions, which cannot be captured and reasoned about by requirements. These difficulties hinder requirements-level adaptations being simply mapped to architecture-level adaptations.

In this paper, we propose a quality-driven self-adaptation approach, which incorporates design decisions as the bridge between requirements- and architecture-level adaptations. In detail, in the requirements layer, requirements goal models are used to represent and reason about runtime requirements. In this layer, value-based quality tradeoff decisions are made

0730-3157/12 $26.00 © 2012 IEEE
DOI 10.1109/COMPSAC.2012.29

with the aim of maximizing system-level value propositions, and a preference-driven goal reasoner is used to reconfigure the runtime goal models based on the results of dynamic quality tradeoff. In the layer of runtime architecture, component-based architecture adaptations are conducted to reflect high-level adaptation decisions. Design decisions which capture design knowledge like alternative design options and their rationales are introduced as an intermediate layer to help map requirements adaptations and context changes to adaptation operations on the runtime architecture. To evaluate the effectiveness of our approach, we have conducted an experimental study with the proposed approach based on a reflective component model. The results show that the approach leads to better performance compared with traditional software and the overall quality satisfaction is kept maintained. Furthermore, the development effort is affordable and the implementations for adaptation mechanism are reusable. However, the approach still contains the shortage in extensibility.

The remainder of this paper is organized as follows. Section II introduces some background knowledge about requirements-driven self-adaptation and reflective architecture. Section III presents the proposed approach, including the overall process and the adaptation mechanisms in each of the three layers of requirements, design decision and runtime architecture. Section IV describes an experimental study of the approach with an online train ticket booking system. Section V evaluates the approach based on the experimental study and discusses related issues. Section VI presents some related work on quality-driven self-adaption and reflective component architecture reconfiguration. Finally, Section VII concludes the paper and outlines our future work.

II. BACKGROUND

A. Requirements Goal Model

In our approach, goal models [7][8][9] are used to represent and reason about runtime requirements. The meta-model of the requirements goal model used in our approach is depicted in Figure 1. It adopts the notation of AND/OR tree [12] and combines some characteristics from *i** goal modeling [10].

In a goal model, requirements are modeled as hard goals and softgoals, depending on whether they have clear-cut satisfaction criteria (goals) or not (softgoals) [5]. Hard goals, or abbreviated as goals, are stakeholder expectations or intentions which are posed on a software system or other stakeholders. Hard goals usually represent functional requirements derived from the process of modeling early requirements [11]. Softgoals, on the other hand, are used to model non-functional or quality requirements of a system. Furthermore, goals have to be refined until they can be assigned as responsibilities of single agents [20]. Therefore, we adopt tasks from *i** as the specific goals in the leaf level of goal models. A task determines the ways of doing things [10] which indicates the implementation of a certain requirement.

Goals can be decomposed into a set of sub-goals or tasks with AND-decomposition. In addition, alternatives to achieve high-level goals are represented as OR-decomposed sub-goals or tasks in goal models. These alternatives usually have different influences on relevant quality requirements (i.e. softgoals). This kind of influences is represented by contribution links between goals/tasks and softgoals. Each contribution link is further assigned with a contribution degree which determines the extent of the influence. In our approach, the degree is categorized into five quantitative levels ($N = 5$) in both positive and negative directions, i.e. satisficing and denial directions according to [6]. Therefore, OR-decompositions and alternative sub-goals/tasks provide potential variability for requirements adaptations.

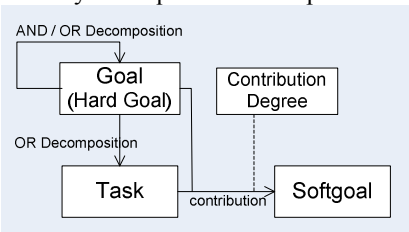

Figure 1. Meta-model of requirements goal model

B. Requirements-driven Self-Adaptation

Requirements-driven self-adaptation incorporates high-level requirements modeling and reasoning, e.g. goal models or other kinds of requirement models, to support adaptation planning on requirements level. Usually, requirements goal models are necessary to adapt under changing context and user preferences [4][6]. By conducting goal reasoning algorithms [14] based on condition-actions rules or quality requirements tradeoffs, a new configuration of goal model can be derived at runtime. As the selection decision of the alternative sub-goals/tasks of any OR-composed group has been changed in new configurations, the structure or behavior of a system is adapted. Furthermore, reasoning algorithms are desired to find optimal configuration that makes the adaptation in a smooth way [6].

The adaptation plan in requirements model subsequently guides the operations of runtime architecture. However, under the constraints of complex relations between goal models and architectures as well as the architecture-level technical decisions for adaptation, direct mapping the adaptation plan to architectural adaptation operations is no longer applicable.

C. Reflective Archtiecture and Component Model

Applications based on reflective architectures have the capability of changing their own structures during runtime. **Fractal** [16] is one of the most popular reflective component models for its ability to dynamic architecture adaptations through introspection and a set of reconfiguration APIs.

During build time, Fractal supports to establish the architecture with hierarchical levels. Furthermore, the embedded components are connected through interfaces. Some basic concepts for Fractal component model are explicitly expressed [16]: components are runtime entities; interfaces reside in components and work as the only

interaction points between components in terms of required/client and provided/server interfaces; bindings are communications between component interfaces with opposite interface types. The concepts compose the Fractal component model: Component = Membrane (controller part) + Content, Content = set of (sub-) components, Membrane = composition and reflection behavior.

Besides the component composition mechanism Fractal provides, membrane enriches the model with reflective and highly dynamic ability to achieve reconfiguration during runtime. In membrane, a series of standard controllers are defined. For instance, life cycle controller is used to start and stop the execution of the component for reconfiguration but not to terminate it. Content controller is used to add or remove a component from a composite component, i.e. the content of the composite component. Binding controller, on the other hand, is responsible for connecting/disconnecting a client interface to/from a server one. Attribute controller exposes getter and setter operations for modifying the attributes of a component. These controllers offer APIs that can be invoked at runtime. Therefore, the architecture of a composite component can be manipulated according to the specific operations, e.g. components can be added or removed, bindings can be added or removed, and attributes of components can be modified.

III. QUALITY-DRIVEN SELF-ADAPTATION APPROACH

In this section, we will first introduce the overall process of our approach, and then detail the adaptation mechanisms in each of the three layers of requirements, design decisions and runtime architecture.

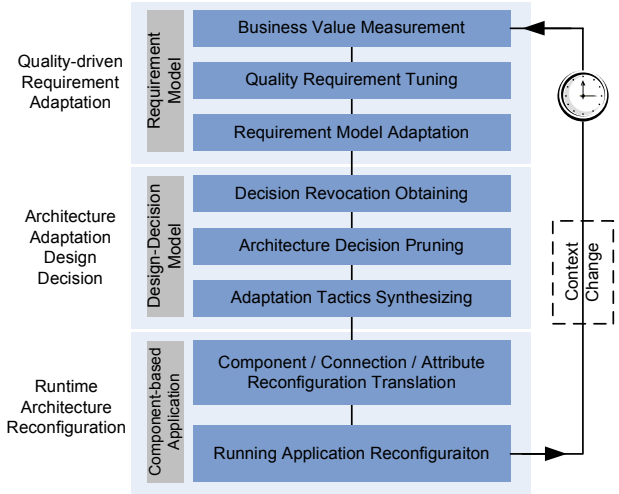

Figure 2. Overview of quality-driven self-adaptation approach

A. Approach Overview

Figure 2 depicts the overall process of our quality-driven self-adaptation approach. The process is iteratively conducted at runtime, involving decisions and adaptations in three layers, i.e. requirements, design decision and runtime architecture. The whole adaptation process follows the value-based self-optimization framework [6], which optimizes the runtime operation of a target system with the objective of maximizing system-level business value propositions.

In the requirements layer, a goal model is used to represent and reason about runtime requirements. The objective of adaptations in this layer is to dynamically make requirements-level adaptation decisions by reconfiguring the goal model. Due to the changing environment, quality tradeoff decisions should be adjusted dynamically at runtime to maximize the overall system satisfaction [6]. Therefore, in this layer, our approach tunes the priority ranks of relevant quality requirements and reconfigures the system goal models accordingly by goal reasoning. Concretely speaking, a feedback controller [13] dynamically tunes the priority ranks of relevant quality requirements (softgoals) based on the feedback of runtime earned value. Taking the tuned priority ranks of softgoals as input, a preference-driven goal reasoner is used to produce a new set of goal configurations that can best satisfy high-ranked softgoals. Each set of goal configurations is a set of customizations to the alternative sub-goals/tasks in the goal model and represents up-to-date adaptation decisions at the requirements level.

To bridge the gap between requirements and runtime architecture in runtime adaptations, an additional design decision layer is introduced to help map requirements adaptations and context changes to runtime architecture reconfigurations. As tasks in the goal model represent concrete functionalities to be implemented by the system, in this layer, a design decision model is created for each alternative task in the goal model. A design decision model captures design knowledge for the implementation of a specific requirement. It is organized in a decision tree, which consists of a series of decision nodes and corresponding design tactics. Each decision node in a decision tree is associated to a predication on quality tradeoffs and runtime environments, and each design tactic is described as general architectural operations like component/connection addition and deletion. There are three main activities conducted in this layer. First, revocation tactics of the previous design decision are obtained when the previous one is no longer valid in this adaptation process according to the requirements adaptation. Second, since the design decision tree may vary depending on binding or unbinding of tasks on requirements model, new architecture adaptation design decision is pruned as a valid path in the tree according to the current quality tradeoffs and context situations. Third, as several requirements may be affected, adaptation tactics derived from several corresponding design decisions, i.e. decision paths, should be synthesized to achieve the final tactics and to discover the underlying conflicts.

In the bottom layer, the runtime architecture is adapted based on the tactics produced by the design decision layer. Reflective component models are employed in this layer to support runtime architectural adaptations. Adaptation tactics are first translated into a set of reconfiguration operations of a reflective component-based architecture through an embedded reconfiguration manager. Common reconfiguration operations include addition or deletion of components, connections as well as modification of component attributes through executable APIs. Then the reconfiguration is conducted on the running application by

invoking the operations, leading to the dynamic change of the system structure and behavior.

B. Requirements-level Adaptation

To achieve value-based requirements adaptation, we incorporate the value-based feedback control loop proposed in [6]. The process is explained through a simple example illustrated in Figure 3. In the figure, round rectangles, rectangles and ellipses denote goals, tasks and softgoals separately. *Ticket Retrieval* is AND-decomposed into alternative tasks *Optimized Retrieval* and *Simple Retrieve*. The two tasks have opposite contributions to softgoals *Most Optimized Solution* and *Minimum Retrieval Response Time*.

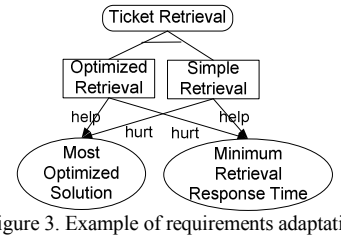

Figure 3. Example of requirements adaptation

First, the runtime earned value, i.e. value indicator, is regularly measured based on a value formula defined from the business perspective. The value formula is constructed based on monitored quantitative data related to softgoals, e.g. recorded average response time of retrieval and customer acceptance of retrieval. Secord, a feedback controller takes the value measurement as feedback and tunes the priority ranks of relevant quality requirements if necessary. For example, if the derivation between the feedback and the feedback in last recorded point goes beyond a prescribed threshold and the system response time goes up, the priority rank of *Minimum Retrieval Response Time* is upgraded in order to improve overall customer satisfaction. Third, a preference-driven goal reasoning algorithm is used to produce updated goal configurations based on the tuned priority ranks of quality requirements. In this activity, a SAT solver is used to find an optimal configuration which is a selection of the alternative tasks. Since the conflicting softgoals cannot be satisfied at the same time, the configuration is derived by removing the softgoal with the lowest rank iteratively. For example, only the task *Simple Retrieve* is selected in the adaptation when the priority rank of *Minimum Retrieval Response Time* is higher than that of *Most Optimized Solution*.

C. Design Decision Deduction

A design decision model corresponds to an individual task in the requirements model and is organized in a decision tree. As a bridge between the requirement-level adaptation and the runtime architecture reconfiguration, the result of the overall design decision deduction is a set of tactics generated from several decision trees. Then the tactics will be implemented upon the target architecture.

The meta-model of a design decision model is depicted in Figure 4. In the meta-model, each decision tree is composed of a set of decision nodes and decision edges. We give out the definitions of the decision tree elements as follows.

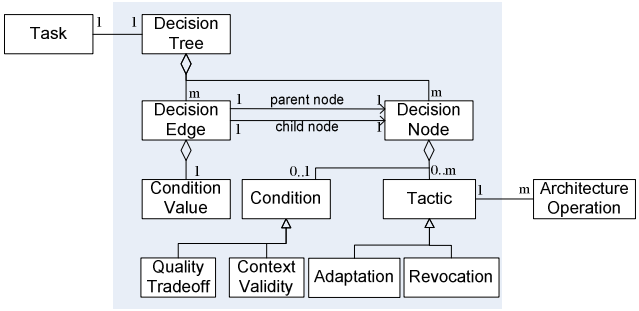

Figure 4. Meta-model of design decision model (decision tree)

Definition (decision node): A decision node $N = \{\text{Set}(A), \text{Set}(R), C\}$. A (Adaptation) and R (Revocation) are two specific types of tactics which denote the architecture-level modifications. In detail, A indicates an operation when the node is selected during design decision. R, on the other hand, always corresponds to adaptation and it means an operation implemented when the decision node is revoked. Furthermore, a tactic can be further mapped to a set of operations that is defined in the concrete reflective component-based architecture. Usually, adaptation always denotes the positive modification towards the architecture, i.e. adding new elements. Oppositely, revocation means the negative modification, i.e. removing elements. As a generalized decision model, the tactics are categorized into a set of generalized classes that can be adopted by different types of architectures. Since components and connections are the representative elements in almost all kinds of architectures, we give out the category in Table 1.

TABLE 1. GENERALIZED TACTIC CATEGORY

A (Adaptation)	Semantic of Adaptation	*R* (Revocation)
addComponent (Comp)	add a new component into the architecture	removeComponent (Comp)
addConnection (Comp1, Intf1, Comp2, Intf2)	create a connection between the interfaces of two components	removeConnection (Comp1, Intf1, Comp2, Intf2)
setAttribute (Comp, AttrName, NewVlaue)	modify an attribute of a component from an old value to a new value; reserve the old value for revocation	setAttribute (Comp, AttrName, OldValue)

The first column of Table 1 represents the adaptation patterns that are illustrated in design decision nodes. Furthermore, the third column is the reverse operations of adaptations which will be adopted if the adaptations are revoked. In particular, the adaptation *SetAttribute* demands to reserve the old value of the component attribute so that the value can be recovered in the revocation.

Besides, C (Condition) of a node denotes a condition which is used to determine whether the child node is selected in the decision process. The condition can be specified in two aspects. First, it may be condition related to quality tradeoffs, i.e. preference rank comparison between two softgoals of the requirements model. For example, the condition "rank(*minimum response time*) > rank(*maximum information detail*)" can be assigned to a decision node. Second, a condition may also be build towards the validity of

a system context. For instance, the condition can be described as "context(available memory)>500M".

Definition (decision edge): a decision edge $E = \{Np, V, Nc\}$. E connects two decision nodes Np and Nc where Np is the parent node and Nc is the child node. V is the condition value which gives out the answer, e.g. usually a Boolean value, to the condition defined in Np. As for the edges originated from a single parent node, the corresponding condition values should be exclusive so that only one child node can be selected along with the decision deduction.

Based on the definitions of the decision tree elements, the terms related to decision deduction is defined accordingly.

Definition (adaptation design decision): an overall adaptation design decision AD = Set(RP) ∪ Set(AP). AP is an adaptation path from the root node down to a leaf node in a decision tree. We define the decision path $AP = \{N0, Nk, \ldots, Nl\}$ where $N0$ is the root node and Nk, …, Nl are the nodes in the path in sequential order. On the contrary, RP is the revocation path from the leaf node back to the root node along with an inverse direction of AP in a decision tree. Thus, RP contains decision nodes with an inverse sequence of those in AP. Typically, switching of the alternative tasks in a goal model leads to an adaptation design decision composed of RP and AP of two decision trees related with the two tasks.

Definition (adaptation design decision tactics): the derived tactics according to a design decision AD is defined as $ADT = \Phi\{\text{Set}(R) \cup \text{Set}(A) \mid \text{Set}(R)$ is revocations of nodes in Set(RP) and Set(A) is adaptations of nodes in Set(AP)$\}$. Furthermore, Φ denotes a set of functions including removing the counteracted tactics, reserving the single copy of tactics and uncovering the underlying conflicts.

In order to obtain the design decision tactics against a requirements-level adaptation, we introduce a *Design Decision Deduction* algorithm (Algorithm 1) and an *Adaptation Path Pruning* algorithm (Algorithm 2).

In Algorithm 1, the revocation paths Set(RP) of all the decision trees according to the previous requirements configuration are obtained (line 1-4). After that, the adaptation paths of all the decision trees according to the newly derived requirements configuration are obtained by invoking Algorithm 2 (line 6). The latter algorithm takes the current quality tradeoffs and context validity as input and deducts to prune a path from the root node of the decision tree down to a leaf node. In detail, the traversed decision nodes are added to AP in sequential order (line 23). Furthermore, the child node is selected based on its condition value towards the condition of the parent node according to the current quality tradeoffs and context validity (line 24-29).

Subsequently, Algorithm 1 suggests excluding the design decision tactics of the unchanged tasks from the final ADT (line 7-10). An unchanged task means the one exists in both requirements configurations and its AP is directly the inversion of its RP, i.e. current quality tradeoffs and context validity has no influence on the design options of the task. However, we should also note that the tasks exist in both configurations may also be adapted in design decision level since the changing context may influence the previous decision path in the same decision tree during runtime.

After obtaining the initial ADT (line 13), some additional operations have to be conducted on the tactic set. First, redundant tactics should be removed from ADT and only one copy of the tactics should be reserved (line 14). Second, counteracted tactics should be eliminated since they produce no effect during adaptation implementation (line 15). Third, underlying conflicts should be uncovered and be informed to developers or designers (line 16-17). For example, if two tactics of *SetAttribute* modify the attribute of the same component with different values, a conflict emerges and designers should be informed with it.

Algorithm 1. Design Decision Deduction
input: requirements model configuration in previous iteration *PRC*
requirements model configuration after adaptation *ARC*
output: adaptation design decision tactics *ADT*
procedure
begin
1. **for** each selected task *a* in *PRC*
2. obtain *RP* of the decision tree of the corresponding task
3. add *RP* to Set(*PR*)
4. **end for**
5. **for** each selected task *a* in *ARC*
6. obtain *AP* through performing **Algorithm 2**
7. **if** (*a* exists in PRC) and (*AP* of *a* is inversion of *RP* of *a*)
8. remove *RP* of *a* from Set(*RP*)
9. **continue** to next iteration
10. **end if**
11. add *AP* to Set(*AP*)
12. **end for**
13. obtain initial *ADT* from Set(*RP*) ∪ Set(*AP*)
14. remove redundant tactics from *ADT*
15. remove counteracted tactics from *ADT*
16. **if** conflicts exist in *ADT*
17. inform developers or designers of the conflicts
18. **else**
19. **return** *ADT*
20. **end if**
end

Algorithm 2. Adaptation Path Pruning
input: decision tree *dt*
current quality tradeoffs *qt*
current context validity *cv*
output: adaptation path *AP* of the decision tree
procedure
begin
21. decision node *cn* = root node of *dt*
22. **while** *cn* is not leaf node **do**
23. add *cn* to *AP*
24. **for** each decision edge *e* started from *cn*
25. evaluate answer *ans* to *cn.C* according to *qt* and *cv*
26. **if** *e.V* = *ans*
27. *cn* = *e.Nc* ; break
28. **end if**
29. **end for**
30. **end do**
31. **return** *AP*
end

D. Runtime Architecture Reconfiguration

The architecture reconfiguration in our approach is performed based on Fractal applications through two activities consecutively. First the design decision tactics resulted from the previous layer is translated into the reconfiguration operations of Fractal. In our approach, we provide an adaptation manager to receive the tactics and translate them into corresponding meta-operations to be

invoked (depicted in Figure 5). The adaptation manager is endowed with a reconfiguration template in which a set of *if-else* statements are prescribed. The conditions of the statements are used to match the description patterns of adaptations or revocations. Therefore, when the adaptation manager receives a tactic from the design decision, one of the *if-else* statements is identified. Based on the mapping between the meta-operations of Fractal and the tactics from design decisions (illustrated in Table 2), the constituents in the pattern are taken as the arguments of Fractal reconfiguration APIs. Finally, the APIs are invoked and implemented on the runtime architecture to make the structure modified.

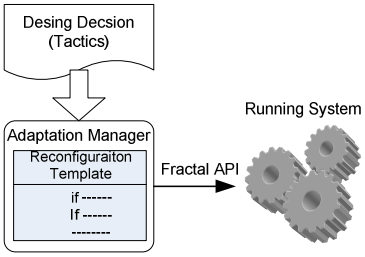

Figure 5. Adaptation manager and reconfiguration template

Among the mappings in Table 2, some of the arguments of Fractal meta-operations are constrained as instances of *Component* or *Interface* objects. However, tactics in design decisions only contain the textual name of components and interfaces. Therefore, before translating tactics, an additional Fractal API has to be invoked to get the component/interface instances according to their names in a reflective way. For example, a Fractal component instance can be obtained by invoking *ContentController.getFcSubComponents()* and locating the instance by matching the component name in the results set. In addition, since *AttributeController* of Fractal contains no intrinsic operations but indicating to expose the setter or getter operation of the desired attributes, we have the direct mapping from *setAttribute* tactics to the operation named as the combination of *set* and *AttrName*. Therefore, this tactic can also be translated naturally. Finally, two essential Fractal APIs must be placed around each reconfiguration, i.e. *LifeCycleController.stopFc()* must be invoked before reconfiguration, while *LifeCycleController. startFc()* has to be invoked when reconfiguration is finished.

TABLE 2. FRACTAL META-OPERATIONS

Meta-operation of Fractal	Design decision tactic (adaptation)
ContentController.addFcSub Component(CompInstance)	addComponent(Comp)
BindingController.bindFc (Intf1, Intf2Instance)	addConnection(Comp1, Intf1, Comp2, Intf2)
setAttrName (NewValue)	setAttribute(Comp, AttrName, NewVlaue)

IV. EXPERIMENTAL STUDY

We have performed an experimental study on our quality-driven self-adaptive approach for an online train ticket system. Figure 6 illustrates the requirements goal model of the system. The legends of the elements of the requirements model including goals, tasks, softgoals, AND/OR decompositions and contribution links are depicted under the graph. The system is composed of four main parts: *Authentication* is responsible for registration and login for customers and has a *help* contribution to the softgoal *Security*; *Ticket Retrieval* filters and shows the tickets from databases based on the information including the departure city and destination city; *Ticket Booking* is the activity for customers that they select tickets, input travelers' information and then the system confirms the tickets; The goal *Payment* is defined to redirect customers to online payment sites for paying the booked tickets. Furthermore the tickets will be reserved for a period of retention time for customers to finish payment.

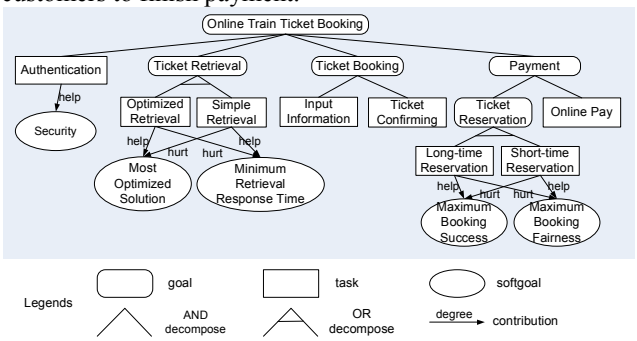

Figure 6. Requirements goal model of an online train ticket booking system

There are alternative tasks that are OR-decomposed from a same goal which indicate the different behaviors the system provides. As mentioned in III.B, *Optimized Retrieval* and *Simple Retrieval* are both decomposed from *Ticket Retrieval*. The former provides an advanced retrieval algorithm to search all the possible ticket solutions between two cities, e.g. it can provide the solution of transferring of two train lines in an intermediate city. The two tasks have opposite contributions towards the pair of softgoals of *Most Optimized Solution* and *Minimum Retrieval Response Time*. Thus the quality tradeoff between the softgoals can trigger the adaptation of the selection of the two tasks during runtime. Furthermore, *Long-time Reservation* and *Short-time Reservation* is another pair of alternative tasks decomposed from *Ticket Reservation*. The former suggests a long retention time in order to improve *Booking Success*, but with a side effect of hindering *Booking Fairness*. *Short-time Reservation* effects contrarily. Therefore, the selection of the tasks is also determined by the quality tradeoff at runtime.

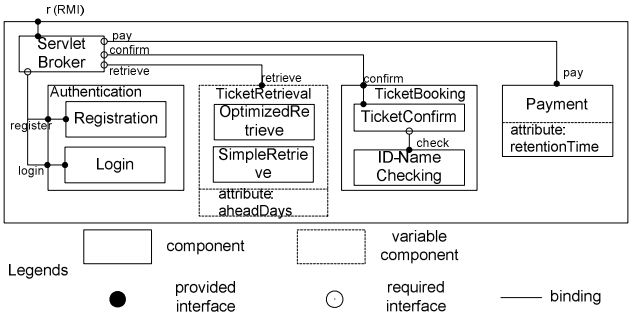

Figure 7. Fractal component model of an online train ticket booking system (part)

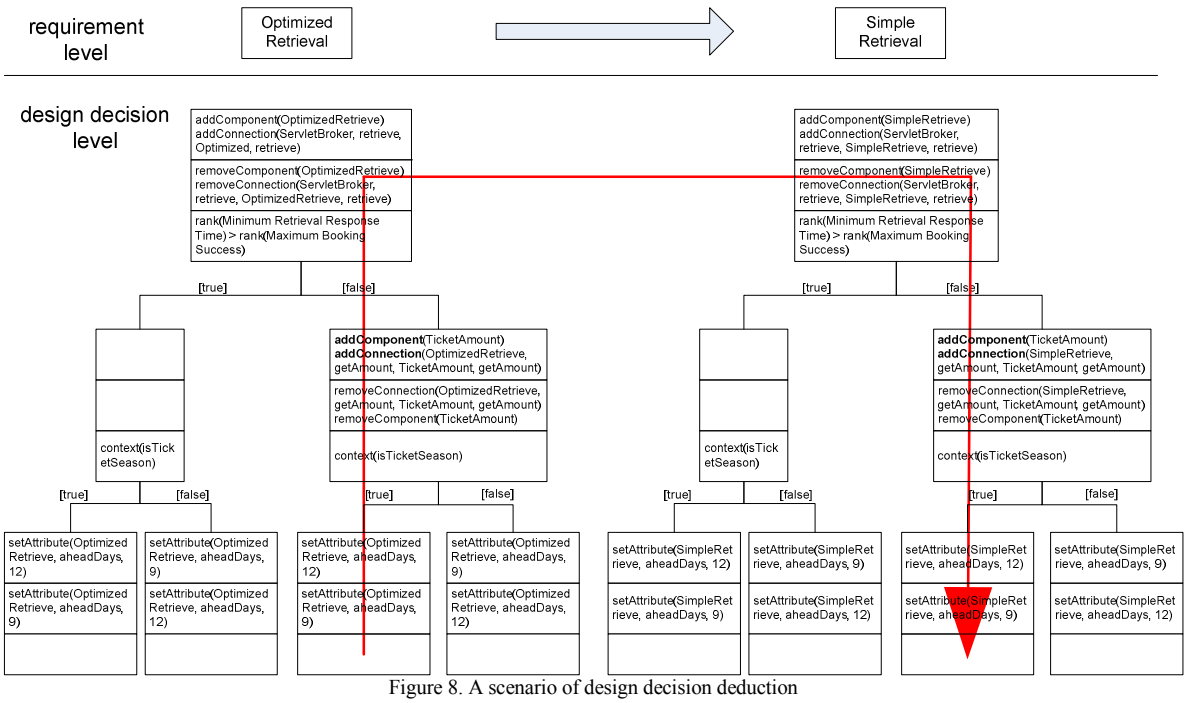

Figure 8. A scenario of design decision deduction

Figure 7 depicts the runtime architecture of the online train ticket booking system based on Fractal. Since the system is developed under a B/S structure, *Servlet Broker* is used to invoke the functions of components in the business logic layer through its RMI interfaces. Provided interfaces are connected with required interfaces with bindings. In particular, the rectangle with dashed frame *TicketRetrieval* denotes the variable component that will be replaced by one of the alternative concrete components *OptimizedRetrieve* and *SimpleRetrieve* after it is delivered. Furthermore, component *Payment* has an attribute *retentionTime* exposed whose value can be set at runtime. Similarly, the attribute *aheadDays* of *TicketRetrieve* indicates the retrieval time range of both the concrete components.

Without design decisions, there exists a gap between the requirements model and the runtime architecture when adaptation is triggered. Developers may encounter difficulties in coding the self-adaptation behavior of the system since the reasoning process for adaptation is ignored. Although system developers may specify adaptation rules for the architecture reconfiguration according to requirement-level adaptation, the effort for establishing rules as well as matching the rules may grow rapidly when the scope of the system grows.

Design decision model works as the bridge between the requirements model adaptation and the architecture configuration. Figure 8 depicts a scenario of design decision deduction when quality tradeoff is adapted during runtime. In the figure, a decision node is represented by three separate constituents from top to bottom: adaptation (*A*), revocation (*R*) and condition (*C*). Each edge is assigned with a Boolean value towards the condition of the parent node.

It is notable that the earned business value during runtime is mainly influenced by response time and user acceptance of the ticket solution. Suppose in the previous adaptation iteration, *Optimized Retrieval* was selected under *Ticket Retrieval* in the requirements model. The design decision path is illustrated in the left part of Figure 8 along the red line from the root node down to the leaf node. As a sequence, component *OptimizedRetrieve* and *TicketAmount* were added into the architecture and the connections were also established. Furthermore, the attribute *aheadDays* of *OptimizedRetrieve* was set according to the condition of context validity.

Self-adaptation is triggered by the measurement of business value in a new adaptation iteration. In the requirements model level, quality tradeoff has been made and the preference rank between the softgoals has been tuned as "rank(*Minimum Retrieval Response Time*) > rank(*Most Optimized Solution*) > rank(*Maximum Booking Success*) > rank(*Maximum Booking Fairness*)". Therefore, a design decision deduction process is conducted following the algorithms introduced in III.C.

1) Revocation path (*RP*) of task *Optimized Retrieval* is obtained. In this activity, the revocation parts of the decision nodes are accumulated by going through the decision tree with a reverse direction of the previous decision path.

2) Adaptation path (*AP*) is pruned in the design decision tree of task *Simple Retrieval*. By evaluating the answers to the conditions posed in the decision nodes, the new decision path is built gradually from the root node to the leaf node. Since the answer to "rank(*Minimum Retrieval Response Time*) > rank(*Maximum Booking Success*)" is false and the answer to "context(*isTicketSeason*)" is true, the adaptation path is obtained as illustrated in the right part of Figure 8 along the red line.

3) Synthesize the design decisions including revocations and adaptations. In this scenario, only one task selection is

adapted in the requirements level, thus the overall architecture decision is derived by combing the revocations (of the nodes in the left part of Figure 8) and the adaptations (of the nodes in the right part of Figure 8) along with the red line in the graph. In particular, the effects by adaptation and revocation on the runtime architecture can be counteracted. For example, revocation removeComponent(*TicketAmount*) in the left decision node indicates to remove the component *TicketAmount* from the architecture, while adaptation addComponent(*TicketAmount*) in the right decision node works oppositely. Thus the two operations cancel out and both of them are not included in the final tactics. The result of the design decision leads to the design decision tactics (*ADT*) showed in Table 3.

TABLE 3. DERIVED DESIGN DECISION TACTICS

Revocation	**setAttribute**(OptimizedRetrieve, aheadDays, 9) **removeConnection**(OptimizedRetrieve, getAmount, TicketAmount, getAmount) **removeConnection**(ServletBroker, retrieve, OptimizedRetrieve, retrieve) **removeComponent**(OptimizedRetrieve)
Adaptation	**addComponent**(SimpleRetrieve) **addConnection**(ServletBroker, retrieve, SimpleRetrieve, retrieve) **addConnection**(SimpleRetrieve, getAmount, TicketAmount, getAmount) **setAttribute**(SimpleRetrieve, aheadDays, 12)

The tactics are then mapped to the operations of Fractal component model. By means of the adaptation manager embedded in self-adaptive systems, the operations illustrated in Table 4 are obtained and actually invoked to modify the runtime architecture.

TABLE 4. OPERATIONS ACTUALLY INVOKED FOR RECONFIGURATION

```
//stop the outer component for reconfiguraiton
Fractal.getLifeCycle(ServletComp).stopFc();
//set attribute of component OptimizedRetrive with a setter method
//predefined
Fractal.getAttributeController(OptimizedRetrieve).setAheadDays(9);
//unbind the interface getAmount of OptimizedRetrieve from the server
//interface
Fractal.getBindingController(OptimizedRetrieve).unbindFc("getAmount");
//unbind the interface retrieve of ServletBroker from the server interface
Fractal.getBindingController(ServletBroker).unbindFc("retrieve");
//remove component OptimizedRetrieve from the outer component
Fractal.getContentController(ServletComp).removeFcSubComponent(Opti
    mizedRetrieve);
//add component SimpleRetrieve to the outer component
Fractal.getContentController(ServletComp).addFcSubComponent(Simple
    Retrieve);
//bind the interface retrieve with the service interface of SimpleRetrieve
Fractal.getBindingController(ServletBroker).bindFc("retrieve",
    Fractal.getContentController(SimpleRetrieve).getFcInternalInterface
    ("retrieve"));
//bind the interface getAmount with the service interface of TicketAmount
Fractal.getBindingController(SimpleRetrieve).bindFc("getAmount",
    Fractal.getContentController(TicketAmount).getFcInternalInterface
    ("getAmount"));
// set attribute of component SimpleRetrive with a setter
Fractal.getAttributeController(SimpleRetrieve).setAheadDays(12);
//start the outer component after reconfiguraiton finished
Fractal.getLifeCycle(ServletComp).startFc();
```

V. EVALUATION AND DISCUSSION

A. Effectiveness

In order to evaluate the approach effectiveness, we conduct an experiment to compare performance of the train ticket booking system with self-adaptation capability as well as that of a modified version without the capability, i.e. *OptimizedRetrieval* at all time. The experiment environment is tomcat 5.0 running on a test server with Intel Core i3 2.13GHz CPU and 2G RAM, running Window 7. The testing script is recorded by Badboy 2.1 and executed in JMeter 2.5.1. In the testing, 25~500 concurrent user threads are simulated to access the system twice. In each access thread, a series of usual operations including *login*, *ticketQuery*, *ticketBooking, ticketPayment* and *logout* are executed in order and the average response time is recorded.

The same experiment is conducted on the two versions of the system. Figure 9 depicts the curves of average response time of the entire HTTP requests of each user thread as well as that of the HTTP request for *ticketQuery*. We list the item *ticketQuery* individually since the request contains invocation to *TicketRetrieval* which provides variability at runtime. In the figure, we can see that the response time is almost the same when the concurrent thread number is lower than about #125. It indicates the overall satisfaction is not decreasing to an extent that self-adaptation has to be triggered. In addition, since the implementation for self-adaptation is lightweight and is almost programmed in the RMI service, there is no considerable running cost for HTTP requests.

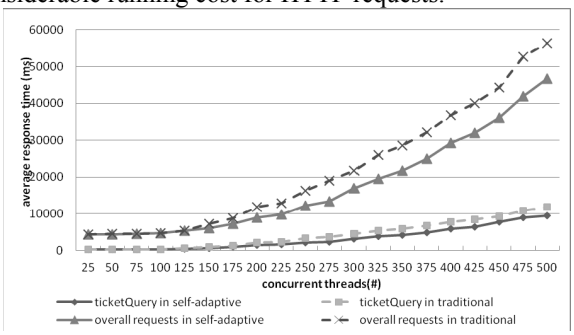

Figure 9. Comparison of average response time of *TicketRetrieval*

When the number of concurrent threads exceeds #125, the overall quality satisfaction indicated by the measured business value is no longer met. Therefore, adaptation is triggered that *OptimizedRetrieval* in the requirements model is to be replaced by *SimpleRetrieval.* As the concurrent threads grows, the performance of the self-adaptive system exceeds the traditional one all the time.

However, adaptations continuously take place since quality tradeoffs are changed depending on runtime environment. For example in Figure 10, the nodes denote the percentage of the reduced average response time of the HTTP request for *ticketQuery* when the mechanism for self-adaptation takes effect. The value rises rapidly and keeps at

a high level from #125 to #350 due to adaptation, but it goes down from #350 to #450 because more and more customers cannot get the optimal solutions thus *OptimizedRetrieval* is selected again in a new adaptation. However, the value goes up from #450 since the increasing response time violates the overall quality satisfaction and *SimpleRetrieval* is selected.

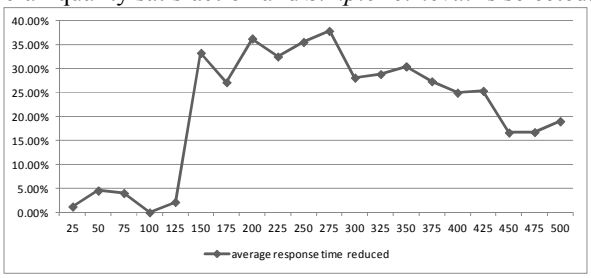

Figure 10. Percentage of average response time reduced

B. Qualitative discussion

By evaluating the overall customer satisfaction and the subsequent reconfiguration, the system can always be running in a status where the most preferred qualities are met. In addition, the adaptation is carried out smoothly without disturbing the running system intensively due to the timed evaluation and cost-effective goal reasoning.

In the design decision period, the deduction in the decision trees is regarded more efficient than traditional rule matching in large scale systems, i.e. get decisions by searching the rule with consistent conditions. Certainly, the synthesizing of decisions from different decision trees may bring conflicts that have to be handled by users.

However, the underlying mechanisms restrict the self-adaptive systems to adapt within the predefined scope. It means the design decisions are finite solutions and the operations of the runtime architecture are prescribed. Thus, this limits the extensibility of architecture adaptation.

C. Development Effort

Compared with a self-adaptive system without design decision support, developing a system by adopting our approach requires efforts in the following aspects.

Efforts for developing goal model reasoning mechanism based on Feedback Control Theory [13]. Different from traditional self-adaptation which is directly triggered by monitored context changes and plans the adaptation based on prescribed rules, the requirements model adaptation in our approach demands to establish a requirements goal model for reasoning as well as a value indicator [6] to describe the real time customer satisfaction. These two requisites rely heavily on designer's experience, thus requiring efforts. In particular, the value indicator is calculated based on a value formula in which several parameters should be adjusted properly.

Efforts for developing design decision model as well as the deduction logics. The creation of a design decision model depends on designers' expertise. They should consider the requirements-level elements in the decision model. In addition, the conditions involved and their positions to establish decision trees also require efforts. However, the decision trees are individually defined so that no combination explosion problem emerges, i.e. define exhaustive rules for all the possible adaptations.

Efforts for developing Fractal component model, composition logic of the model as well as the implementation of the adaptation manager. Fractal-based system requires to invoking embedded APIs to compose the system at the initial time according to the architecture which takes efforts. Furthermore, the template residing in the adaptation manager should be developed conforming to the mapping patterns between design decision tactics and Fractal operations (in Table 2).

All the efforts mentioned above are required before the system delivery. However, the mechanisms for goal reasoning, design decision deduction and the adaptation manager can be reused for different self-adaptive systems, thus the efforts are considered affordable.

VI. Related Work

Several frameworks or approaches have been proposed for requirements-driven self-adaptation. Dalpiaz et al. [3] proposed a conceptual architecture that provides systems with self-reconfiguration capabilities. The constituents of architecture are activated to take effect during a model-based adaptation process based on requirements models. The adaptation in the requirement level is thus achieved especially through monitoring and diagnosing failures. In addition, Ali et al. [4] proposed a goal-oriented requirement engineering modeling and reasoning framework for systems operating in and reflecting varying contexts. In the framework, available goal model variants are chosen to meet goals satisfaction influenced by contexts. Thus, a runtime reasoning technique is applied to derive goal model variants that reflect contexts and user priorities. This work is similar with ours in requirement-level adaptation. The reasoning particularly assumes to find optimal solutions towards user priorities predefined at design time. However, we imply a value-based approach for reasoning in goal model where user preference of the qualities is changeable. Furthermore, the scope of these approaches is on the requirement level which has been extended to guide architecture reconfiguration in our approach.

On the other hand, reconfiguration on runtime architecture receives many attentions. Rainbow [19] is an architecture-based framework which enables self-adaptation based on a reusable infrastructure and an externalized approach. In Rainbow framework, architecture models are considered as the most suitable abstraction level to address adaptation mechanism so that the mechanisms can be directly reused at runtime. By comparison, we lift the adaptation decision up to the requirement level and the actual reconfiguration is guided by design decisions towards requirement adaptation.

Therefore, it is promising to combine requirement level adaptation and runtime architecture reconfiguration as a

whole. There are also contributions in this aspect. For example, Sykes et al. [2] proposed a combined approach for adaptable software architecture guided by high-level goals. In their approach, component configurations are derived through searching in a component selection search tree according to the condition-action rules defined in the domain model. In particular, the configuration is directly mapped and generated by design decisions of syntax structures such as interface types. Our approach works in the similar way by combining quality-driven requirement adaptation and reflective component model reconfiguration, i.e. Fractal. The design decision is performed associated with factors of quality tradeoffs as well as changing context. Therefore, our approach provides systematic way to combine the two aspects and the introduced design decision model is helpful for relieving the gap between requirement level and architecture level.

Architectural design decision is another popular field in architecture community [1][15]. Typically, design decisions are made at design time to construct software architectures according to static design issues and anticipated running context. In our approach, design decision is adopted at runtime aiming at the dynamic changing design issues. Furthermore, a design decision model is proposed in this paper to facilitate the deduction.

VII. CONCULSION AND FURTHER WORK

A self-adaptive system should be able to reflect the requirements-level adaptation decisions on the runtime architecture. However, the gap between requirements and runtime architecture makes it difficult to map high-level adaptations to runtime architecture. In this paper, we have proposed a quality-driven self-adaptation approach with an intermediate design decision layer to bridge the gap between requirements-level adaptations and runtime architectural reconfigurations. With design knowledge of alternative design options and their rationales explicitly captured in design decision models, our approach can dynamically adapt design decisions and map requirements adaptations and context changes to reconfigurations of the runtime architecture. We have implemented the proposed approach with a reflective component model and conducted an experimental study on an online ticket system. The results of the study have confirmed the effectiveness of our approach.

Our future work lies in two aspects. One is to develop a general and reusable framework to facilitate the application of our approach. The other is to conduct more sophisticated case studies with industrial applications to further evaluate the efficiency of the approach.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China under Grant No. 90818009, National High Technology Development 863 Program of China under Grant No. 2012AA011202 and No. 2011AA010101.

REFERENCES

[1] A.Jansen and J.Bosch, "Software Architecture as a Set of Architectural Design Decisions", in Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, 2005.

[2] D.Sykes, W.Heaven, J.Magee and J.Kramer, "From Goals To Components: A Combined Approach To Self-Management", in Proceedings of the 2008 international workshop on Software Engineering for Adaptive and Self-Managing System, 2008.

[3] F.Dalpiaz, P.Giorgini and J.Mylopoulos, "An Architecture for Requirements-Driven Self-reconfiguration", in Proceedings of the 21st International Conference on Advanced Information Systems Engineering, 2009.

[4] R.Ali, F.Dalpiaz and P.Giorgini, "A goal-based framework for contextual requirements modeling and analysis", in Journal of Requirements Engineering, Volume 15 Issue 4, 2010.

[5] S.Liaskos, "Acquiring and Reasoning about Variability in Goal Models", PhD thesis, University of Toronto, 2008.

[6] X.Peng, B.Chen, Y.Yu and W.Zhao, "Self-Tuning of Software Systems through Goal-based Feedback Loop Control", in Proceedings of the 18th Intenaltional Conference on Requirements Engineering, 2010.

[7] E.Yu, "Towards modelling and reasoning support for early-phase requirements engineering", in Proceedings of the 3rd IEEE Int. Symposium on Requirements Engineering, 1997.

[8] J.Mylopoulos, L.Chung, and B.Nixon, "Representing and using nonfunctional requirements: A process-oriented approach", IEEE Transactions on Software Engineering, 18(6):483–497, 1992.

[9] J.Castro, M.Kolp, and J.Mylopoulos, "Towards requirements-driven information systems engineering: the Tropos project", Information Systems, 27(6):365–389, 2002.

[10] E.Yu and J.Mylopoulos, "From E-R to "A-R" – Modelling Strategic Actor Relationships for Business Process Reengineering", in Proceedings of the13th International Conference on the Entity-Relationship Approach, pp.548-565, 1994.

[11] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition", Science of Computer Programming, Vol. 20, Issue 1-2, pp. 3-50, 1993.

[12] P. Klaus, "Requirements Engineering-Fundamentals, Principles, and Techniques", Springer, Berlin, 2010.

[13] G.F.Franklin, J.D.Powell and A.E.Naeini, "Feedback Control of Dynamic Systems", 5th ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2006.

[14] P.Giorgini, J.Mylopoulos, E.Nicchiarelli and R.Sebastiani, "Formal Reasoning Techniques for Goal Models", Journal on Data Sementaics I, Vol 1, pp. 1-20, 2003.

[15] X.Cui, Y.Sun and H.Mei, "Towards Automated Solution Synthesis and Rationale Capture in Decision-Centric Architecture Design", in Proceedings of Seventh Working IEEE/IFIP Conference on Software Architecture, 2008.

[16] E. Bruneton, T.Coupaye, M.Leclercq, V.Quema and J.B.Stefani, "The Fractal Component Model and Its Support in Java: Experiences with Auto-adaptive and Reconfigurable Systems", Software Practice and Experience, 36:1257-1284, 2006.

[17] J.O.Kephart and D.M.Chess, "The Vision of Autonomic Computing", Computer, Vol 36, Issue 1, pp.41-50, 2003.

[18] J.Kramer and J.Magee, "Self-Managed Systems: an Architectural Challenge", in Proceedings of Future of Software Engineering, 2007.

[19] D.Garlan, S.W.Cheng, A.C.Huang, B.Schmerl and P.Steenkiste, "Rainbow: architecturebased self-adaptation with reusable infrastructure", Computer 37(10) pp. 46–54, 2004.

[20] A.V.Lamsweerde and E. Letier, "From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering," in Radical Innovations of Software and Systems Engineering in the Future, 2002.