Self-Adaptation through Incremental Generative Model Transformations at Runtime

Bihuan Chen*†, Xin Peng*†, Yijun Yu‡, Bashar Nuseibeh‡§, and Wenyun Zhao*†

*School of Computer Science, Fudan University, China

†Shanghai Key Laboratory of Data Science, Fudan University, China

‡Department of Computing and Communications, The Open University, UK

§Lero-The Irish Software Engineering Research Centre, University of Limerick, Ireland

{bhchen, pengxin, wyzhao}@fudan.edu.cn, {y.yu, b.nuseibeh}@open.ac.uk

ABSTRACT

A self-adaptive system uses runtime models to adapt its architecture to the changing requirements and contexts. However, there is no one-to-one mapping between the requirements in the problem space and the architectural elements in the solution space. Instead, one refined requirement may crosscut multiple architectural elements, and its realization involves complex behavioral or structural interactions manifested as architectural design decisions. In this paper we propose to combine two kinds of self-adaptations: requirements-driven self-adaptation, which captures requirements as goal models to reason about the best plan within the problem space, and architecture-based self-adaptation, which captures architectural design decisions as decision trees to search for the best design for the desired requirements within the contextualized solution space. Following these adaptations, component-based architecture models are reconfigured using incremental and generative model transformations. Compared with requirements-driven or architecture-based approaches, the case study using an online shopping benchmark shows promise that our approach can further improve the effectiveness of adaptation (e.g. system throughput in this case study) and offer more adaptation flexibility.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Methodologies*; D.2 [**Software Engineering**]: Requirements/Specifications; D.2 [**Software Engineering**]: Software Architectures

General Terms

Design, Management

Keywords

Self-adaptive system, runtime model, requirements, architecture, design decisions, model transformation

1. INTRODUCTION

In software engineering, requirements analysis seeks to "solve the right problem" while design space exploration seeks to "solve the problem right" [46]. Both activities traditionally happen at development time. After a software product is deployed, changes to the problem or to the solution require a time-consuming, at best iterative, development activity [34]. Increasingly, however, software systems are required to respond quickly to changing environments and requirements by dynamically adapting their architectures to their perception of customers' satisfaction.

Model-based approaches have been proposed as one way to achieve runtime self-adaptation [20]. Instead of relying on low-level and error-prone scripts for every possible adaptation, with models at runtime, a system can query, analyze and manipulate them to realize dynamic adaptations [7, 31]. Specifically, requirements models (e.g., [18, 6]) and architecture models (e.g., [22, 19]) have been used for self-adaptation. However, some problems still remain.

Requirements-driven approaches assume that requirements elements (e.g., goals) can be simply and directly mapped to architectural elements (e.g., components), and thus neglect the complexity of architectural design. Architecture-based approaches, on the other hand, assume that requirements are well-understood at design time and unchanged at runtime, and thus are unable to support dynamic adaptations to changing requirements. In any case, both these two kinds of self-adaptation approaches have a role to play and should be combined. In the prior attempts to combine them (e.g., [43, 45]) following Kramer and Magee's reference architecture [28], architectural elements are regarded as a collection of interfaces to functional requirements, hence lacking consideration of more complex architectural concerns.

A software architecture is not just a collection of functionalities but also a collection of design decisions concerning how those functionalities are structured and interacting with each other [25]. In that sense, architectural adaptations must also reflect the adaptations of design decisions. Manifest as the adapted design decisions, architectural adaptations may crosscut simultaneously multiple parts of a system; for instance, adding a logger function before any access to confidential information for better security. Architectural adaptations may also involve restructuring of existing functionalities to change the runtime structures or behaviors [35]; for instance, changing the structure of collaborating components from sequential to parallel for better performance.

In this paper, we propose a new model-based self-adaptation approach that combines requirements and architectural adaptations and supports complex architectural adaptations using model transformation techniques. We assume that requirements are available at runtime as goal models, architectural design decisions are available at runtime as design decisions models expressed as decision trees, and these models are consistent with each other.

Our approach periodically searches for a better architectural design solution using the design decisions model. If no solution exists given the constraints of current requirements and contexts, goal-oriented requirements reasoning is conducted to find a better goal specification within the constraints of quality expectations, and then the related design decisions are reconsidered to find a design solution reflecting the adapted goal specification. In either case, architectural adaptations are achieved by executing the automatically generated model transformation scripts in QVT-R (Query/View/Transformation-Relations) [4] that incrementally transform the current runtime architecture model into an adapted one. Given the adapted architecture model, the actual system reconfigurations can be delegated to any existing architecture-based management middleware (e.g., [40]).

We conducted a case study using an online shopping benchmark to evaluate the proposed approach. The results show promise that our approach can further improve the effectiveness of adaptation (e.g. system throughput in this case study) and offer more adaptation flexibility than requirements-driven or architecture-based self-adaptation approaches.

The rest of this paper is structured as follows. Section 2 motivates our work through a running example. Section 3 introduces some preliminaries required before Section 4 presents our proposed approach. Section 5 evaluates our proposal. Section 6 introduces and compares some related work before Section 7 draws our conclusions.

2. A MOTIVATING EXAMPLE

An online shopping company has a business department in charge of marketing strategies and a technical department in charge of development and maintenance of the IT systems. Suppose a sales promotion is launched for celebrating the New Year. It is anticipated that the system may suffer performance degradation due to the payload of a large number of concurrent requests. Without adaptation to the increased load, the response time and failure rate of order processing could increase, resulting in customer dissatisfaction.

Now consider the following two adaptation scenarios.

Scenario A: Jack, the chief architect of the technical department, is notified of the problem and finds out that the bottleneck lies in the order verification process, which has been implemented by several components including order information checking, credit checking and fraud checking. After reconsidering the design decisions of the current architecture, Jack decides to adapt the interaction structure of the order checking components from sequential to parallel processing for a better performance. This architectural adaptation resolves the problem and the whole adaptation process is made transparent to the business department.

Scenario B: Jack is notified and reconsiders the design decisions but cannot find any architectural design alternative to alleviate the problem. He raises the problem as an issue to Bob, the manager of the business department. After reconsidering the business decisions that led to the current requirements specification, Bob decides to adapt the order verification process to involve only order information and credit checking, and skip fraud checking, even though he is aware of the potential loss due to higher risks. This business adaptation may help accelerate the order processing process at the cost of a higher risk of malicious orders. Following this business requirement change, Jack can adapt the technical architecture accordingly. To this end, Jack first identifies the components that are influenced by this requirement change, and then removes the components that were introduced for the eliminated functionalities. Since the implementation of a requirement may be scattered across different parts of the architecture, such architectural adaptations often crosscut multiple architectural elements.

The above analogy of the business and technical departments helps explain the following two observations. First, runtime adaptations often involve both requirements and architectural decisions where different concerns (e.g., business versus technical) require different knowledge (e.g., requirements versus architectural design). Second, mappings from requirements to architecture are non-trivial, which involves complex traceability from requirements to architectural elements and architects' knowledge about design decisions.

Model-based self-adaptation can thus be regarded as an automation of these adaptation processes at runtime, based on the runtime representations of requirements and architectural design knowledge.

3. PRELIMINARIES

This section briefly introduces the preliminaries, i.e. goal-oriented requirements and architectural design decisions.

3.1 Goal-Oriented Requirements

In goal-oriented requirements analysis, typically functional requirements are modelled as *hard goals*, and quality requirements are modelled as *softgoals* [33]. A preference is specified for each softgoal to indicate its relative importance [30]. Goals can be refined into subgoals through AND/OR *decomposition links* until the leaves of the decomposition hierarchy as *tasks* that can be accomplished by either software or human agents. To satisfy an AND/OR-decomposed goal, all/at least one of its subgoals must be satisfied. Furthermore, goals can relate to each other through the weighted *contribution links* $w+$ and $w-$ where the normalized weight w is in the range of $[0, 1]$ [24]. A $+$ or $-$ sign indicates respectively that the satisfaction of the source goal contributes to w-level satisfaction or denial of the target goal.

The top part of Figure 1 presents the requirements goal model of a simplified online shopping system as a graph. The system is used as a running example throughout the paper. In this graph, hard goals, softgoals and tasks are syntactically shaped as rounded rectangles, clouds and hexagons, respectively. The semantics of the model are illustrated as follows. The goal `Order be strictly verified` is satisfied if all of its AND-decomposed subgoals `Check info`, `Check credit` and `Check fraud` are satisfied. The goal `Order be verified` can be satisfied by any one of the two OR-decomposed subgoals `Order be simply verified` and `Order be strictly verified`.

3.2 Architectural Design Decisions

Architectural design decisions manifest themselves in the system's architecture for assuring the satisfaction of the sys-

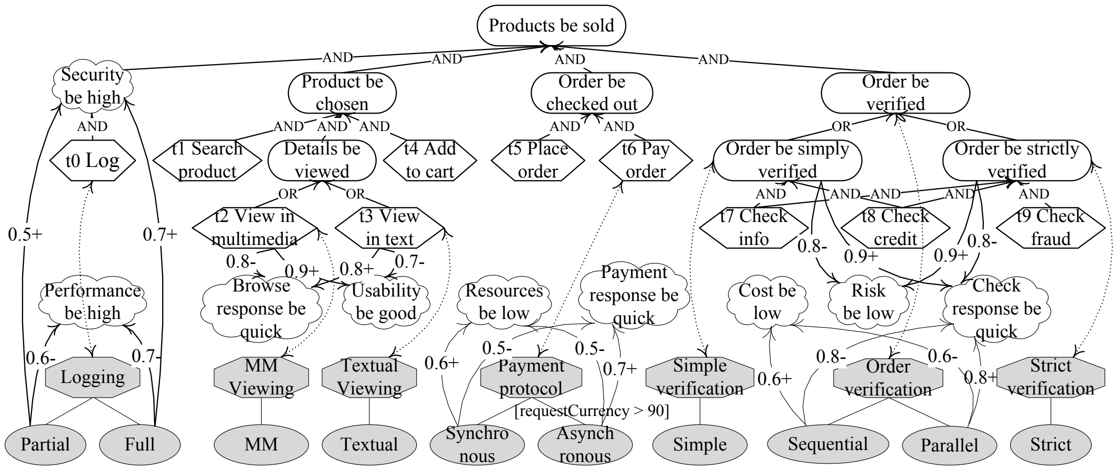


Figure 1: Requirements goal model and design decisions of a simplified online shopping system

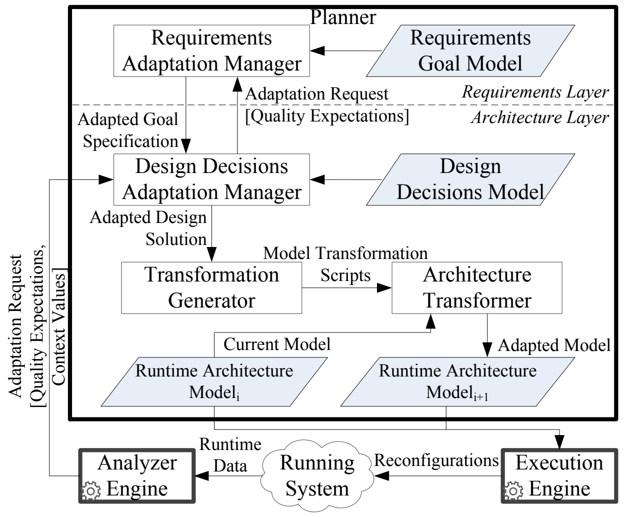

Planner
Requirements Adaptation Manager
Requirements Goal Model
Adaptation Request [Quality Expectations]
Requirements Layer
Architecture Layer
Adapted Goal Specification
Design Decisions Adaptation Manager
Design Decisions Model
Adapted Design Solution
Model Transformation Scripts
Transformation Generator
Architecture Transformer
Current Model
Adapted Model
Runtime Architecture Model$_i$
Runtime Architecture Model$_{i+1}$
Adaptation Request [Quality Expectations, Context Values]
Analyzer Engine
Runtime Data
Running System
Reconfigurations
Execution Engine

Figure 2: Overview of our approach.

tem's quality and business requirements [14]. Every decision has one issue describing the problem and some options describing the alternative solutions. Each option has some applicable contexts that have to be met for the option to be considered, some pros and cons for the quality requirements to record its impact on them, and some architectural modifications to realize the solution in the architecture [25]. For instance, to realize order payment in online shopping, different message protocols such as asynchronous protocol or synchronous protocol can be used. The former has a quicker response but needs more resources than the latter. However, due to its complexity, asynchronous protocol is considered only when the message concurrency is high.

4. OUR APPROACH

After an overview of our approach (Figure 2), this section introduces the adaptations of requirements and design decisions, and the generated architecture transformations.

4.1 The Overall Framework

Following the MAPE-K (Monitor, Analyze, Plan, Execute-Knowledge) control loop [26], our framework consists of an *Analyzer Engine*, a *Planner* and an *Execution Engine*. This control loop is periodically executed. Specifically, the *Analyzer Engine* aggregates quality values and context values based on the collected data during runtime monitoring. Using the aggregated and expected quality values, it tunes the expectations of quality requirements through a feedback controller proposed in our earlier work [36]. Expectations $\in$ [-1, 1] are used to indicate the expected satisfaction levels of quality requirements, which are different under different situations and thus need to be tuned. The expected quality values can be specified in advance or updated at runtime (e.g., by taking the average of the last n monitored samples).

The *Design Decisions Adaptation Manager* searches for the optimal design solution as a set of design options that best satisfy a given set of requirements and contexts using the design decisions model. If such a solution can be found, the *Transformation Generator* generates the model transformation scripts in QVT-R that incrementally transform the current runtime architecture model into an adapted one through the *Architecture Transformer*. On the other hand, if such a solution cannot be found, the *Requirements Adaptation Manager* is responsible for selecting the optimal goal specification as a set of tasks that best meet the quality expectations. No adaptation will be performed when no such goal specification can be found.

The *Execution Engine* reconfigures the running system according to the differences between the architecture models before and after adaptations. These system reconfigurations are currently supported by reflective component models such as Fractal [8] and OpenCOM [15], service-oriented adaptation techniques such as AO4BPEL [9] and VxBPEL [27], or architecture-based management middleware [40].

The steps of the self-adaptation are illustrated as follows.

1. *Analyzer Engine* periodically tunes quality expectations and aggregates context values, and then raises an adaptation request to *Design Decisions Adaptation Manager*;
2. *Design Decisions Adaptation Manager* tries to find an optimal design solution based on the design decisions model;
3. If a solution can be found, *Transformation Generator* generates the model transformation scripts, and *Architecture Transformer* incrementally executes them on the current architecture model;
4. If no solution can be found, an adaptation request is raised to *Requirements Adaptation Manager*, and then

it tries to find an optimal goal specification based on the goal model;

5. If a specification can be found, *Design Decisions Adaptation Manager* finds an optimal design solution that reflects the changed requirements, and triggers Step 3;
6. If no specification can be found, no adaptation is needed, which means that the current goal specification and design solution are already optimal;
7. *Execution Engine* reconfigures the running system based on the differences between the current and adapted architecture models.

Depending on the different possible outcomes, there can be three adaptation loops 1-2-3-7, 1-2-4-5-7, and 1-2-4-6, corresponding to successful architectural adaptation, successful requirements adaptation, and no adaptation respectively. Although it is also possible to directly raise an adaptation request to *Requirements Adaptation Manager*, i.e. 1-4-5-7, which could have bypassed the architectural adaptation in step 2, we do not choose that path because most of fine-grained adaptations could already be handled transparently to the fixed requirements through architectural adaptation.

To apply our approach, application-specific *Analyzer Engine* and *Execution Engine* are plugged in respectively to obtain quality values and context values, and to achieve actual system reconfigurations. As an input, three application-specific models; i.e., a goal model, a design decisions model and an initial architecture model, are required.

4.2 Requirements Adaptations

We use requirements goal models as the business abstraction of a running system. Goal models capture the space of alternative specifications (i.e. a set of leaf-level tasks) satisfying high-level goals in the form of OR-decompositions. For instance, there are 4 possible goal specifications for satisfying `Products be sold` in Figure 1. Further, each goal specification is often better in satisfying certain quality requirements but worse in satisfying some others. For instance, one possible goal specification GS is $[t0, t1, t2, t4, t5, t6, t7, t8, t9]$ in Figure 1, and it is better in `Usability be good` ($+0.9$[1]) and `Risk be low` ($+0.9$) but worse in `Browse response be quick` (-0.8) and `Check response be quick` (-0.8).

The *Requirements Adaptation Manager* focuses on business-driven decisions, and receives an adaptation request only if no design solution can be found. If requested, it will perform a goal reasoning process, which takes as input the goal model and the tuned quality expectations, and finds among all the possible goal specifications the optimal one that best satisfies the quality expectations.

Traditionally the optimal goal specification is the one with the highest weighted sum of the satisfaction levels of all softgoals [47]. This strategy tries to achieve the overall quality satisfaction as high as possible. As a result, some poorly-satisfied softgoals will be hidden under the well-satisfied ones, and it is unknown if these hidden softgoals meet their expectations. Therefore, here we adopt another strategy, which tries to meet as many softgoals' expectations as possible. In detail, we compute for each softgoal a *satisfaction delta*, which is the difference between its satisfaction level and its expectation. A positive/negative satisfaction delta means a softgoal does/does not meet its expectation. Then we compute for each goal specification a weighted sum of the negative satisfaction deltas (i.e. N_Score) and a weighted sum of the positive satisfaction deltas (i.e. P_Score). Finally, the goal specification with the highest N_Score is the optimal one that best satisfies the expectations; if multiple goal specifications have the same highest N_Score, one of them that has the highest P_Score is the optimal one.

[1]+/− indicates the negative/positive satisfaction level calculated by label propagation algorithms [24]

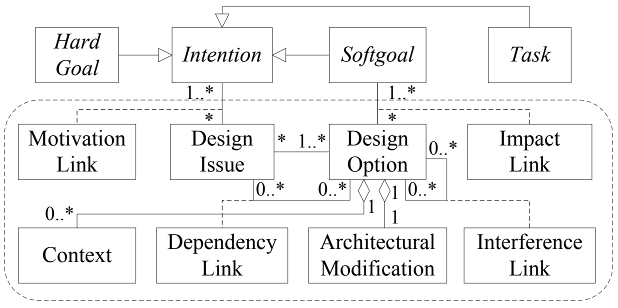


Figure 3: Metamodel of design decisions.

For instance, if the expectations for `Browse response be quick`, `Usability be good`, `Check response be quick` and `Risk be low` are respectively -0.2, $+0.6$, -0.2 and $+0.6$, their satisfaction deltas with goal specification GS are respectively $(-0.8) - (-0.2) = -0.6$, $(+0.9) - (+0.6) = 0.3$, $(-0.8) - (-0.2) = -0.6$ and $(+0.9) - (+0.6) = 0.3$. If the preferences for all softgoals are specified to 5, N_Score is $5 \times (-0.6) + 5 \times (-0.6) = -6.0$, and P_Score is $5 \times 0.3 + 5 \times 0.3 = 3.0$. Given these quality preferences and expectations, GS is actually the optimal goal specification, which configures the OR-decomposed goals (i.e. business-driven decisions) `Details be viewed` and `Order be verified` to `View in multimedia` and `Order be strictly verified`.

In general, of course, our approach is independent of the choice of such strategies. Specific strategies can be integrated according to specific adaptation objectives.

4.3 Design Decisions Adaptations

We introduce an architectural design decisions model to capture system architectural design and its candidate solutions. Figure 3 shows the metamodel of architectural design decisions we adopted. For a design decision, a *design issue* is related to certain requirements (*intentions*, which can be hard goals, softgoals or tasks [48]) through *motivation links* to indicate its motivation (i.e. to solve what problem or to meet what requirements), and has multiple *design options* for solving the problem or meeting the requirements. Multiple design issues can be motivated by multiple requirements. In addition, a design option has some applicable *contexts* for the option to be considered, contributes to certain quality requirements (*softgoals*) positively or negatively by *impact links*, and has some *architectural modifications* to realize at the architectural level.

The bottom part of Figure 1 shows the design decisions of the simplified online shopping system. Design issues and options are visually shaped as octagons and ellipses respectively. The issue `Logging` (i.e. what interactions should be logged) is to meet task `Log`. Its option `Full` (i.e. log all interactions) has a higher positive impact to softgoal `Security be high` but also a higher negative impact to softgoal `Performance be high` than option `Partial` (i.e. only log database interactions). The issue `Payment protocol` is to meet task `Pay order`. Its options `Synchronous` and `Asyn-`

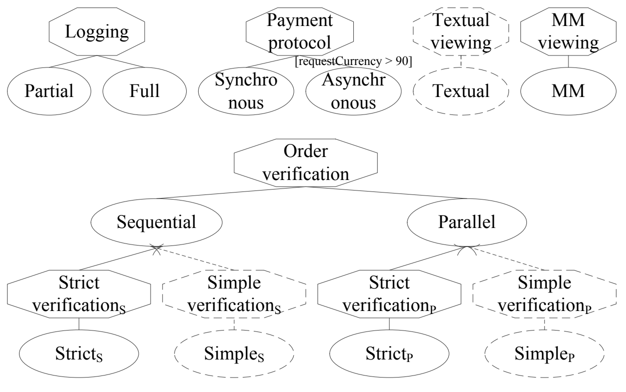


Figure 4: Design decisions model of the simplified online shopping system.

`chronous` have reverse impacts to softgoals `Resources be low` and `Payment response be quick`. The applicable context of option `Asynchronous` is when the request currency is larger than 90. The issue `Order verification` (i.e. what structure can be used to perform order information checking, customer credit checking and fraud checking) is to meet goal `Order be verified`. Its options `Sequential` and `Parallel` have reverse impacts to softgoals `Cost be low` and `Check response be quick`.

Furthermore, design decisions are often intertwined and crosscutting with each other [25] with respect to the decision-making process and architectural modification process. To model such relationships, we introduce *dependency* and *interference links* to relate design issues and options. Dependency links indicate that only if a set of design decisions are made together can the structure or behavior of certain part of a system be determined. For instance, issues `Order verification`, `Simple verification` and `Strict verification` have to work together to determine the behavior and structure of order verification. Interference links express the interferences among different design options when performing their architectural modifications, which will be introduced in Section 4.4.2.

To represent such intertwined design decisions and facilitate their decision-making process, we model them in decision trees. Figure 4 gives the design decisions model of the simplified online shopping system with each decision tree representing a set of intertwined design decisions. Subscripts are used to differentiate the design options with the same name but different architectural modifications under different decision-making process. For instance, following the different options of `Order verification`, options `Strict`$_S$ or `Strict`$_P$ will result in architectural modifications that respectively structure the checking components to sequential or parallel processing. For clarity, we only show the dependency links (lines with an arrow) in Figure 4, and omit the motivation and contribution links that are shown in Figure 1. For the pairs of issues `Textual viewing` and `MM viewing`, and `Simple verification` and `Strict verification`, only one of them will be considered in the decision-making process because they are exclusive to each other from the requirements perspective. Hence, Figure 4 shows the decision trees reflecting the goal specification GS by indicating the decisions that should not be considered by dashed lines.

Through motivation and impact links, the gap between requirements and architectural designs are narrowed. On the one hand, following motivation links, requirements changes (i.e. a new goal specification) will influence the set of design decisions that need consideration. For instance, if goal `Details be viewed` is reconfigured from `View in text` to `View in multimedia`, issue `Textual viewing` will not be considered while issue `MM viewing` should be considered.

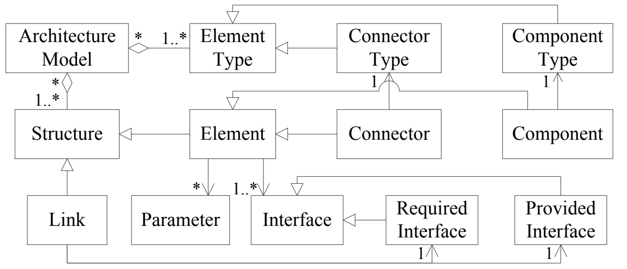


Figure 5: Metamodel of component-based architecture model.

On the other hand, using impact links sourced from design options to quality requirements, tuned quality expectations and monitored context values, the *Design Decisions Adaptation Manager* finds among all design solutions the optimal one, i.e. a set of design options that meet their applicable contexts and have the highest N_Score as introduced in Section 4.2. For instance, given the goal specification GS, if expectations for `Performance be high`, `Security be high`, `Resources be low`, `Payment response be quick`, `Cost be low` and `Check response be quick` are respectively $+0.6$, -0.2, $+0.6$, -0.2, $+0.6$ and -0.2, preferences for them are all specified to 5, and option `Asynchronous` does not satisfy its applicable context, then the optimal design solution DS is [`Partial`, `Synchronous`, `MM`, `Sequential`, `Strict`$_S$] with N_Score being -10.5 and P_Score being 3.5.

4.4 Architecture Transformations

To realize the changes of design solutions into the architecture, we propose a set of primitive adaptation operations to express the architectural modification of each design option, and then introduce the incremental and generative model transformations to execute the architectural modifications.

4.4.1 Expressing Architectural Adaptations

We use component-based architecture models as the design abstraction of the running system. Figure 5 gives the metamodel of architecture model we adopted. *Components* (i.e. computational elements and data stores) and *connectors* (i.e. interactions between components) are respectively instances of *component* and *connector types* that express the common behaviors. They have *provided interfaces* to specify the services they offer, and/or *required interfaces* to specify the services they need, and can have *parameters* to express their changeable characteristics. A *link* is a connection from a required interface to a provided interface. In addition, this metamodel could be extended to include application-specific properties for components and connectors.

Figure 6 shows the component-based architecture model of the simplified online shopping system manifesting the design solution DS. Components, connectors, provided and required interfaces are visually shaped as rectangles, rounded rectangles, solid and hollow circles respectively. Components `Logger` are only connected to database-related components `Product` and `Order`. Products details are viewed in multimedia mode through connector `getProdDetMM`. Order payment is realized using synchronous protocol. Order veri-

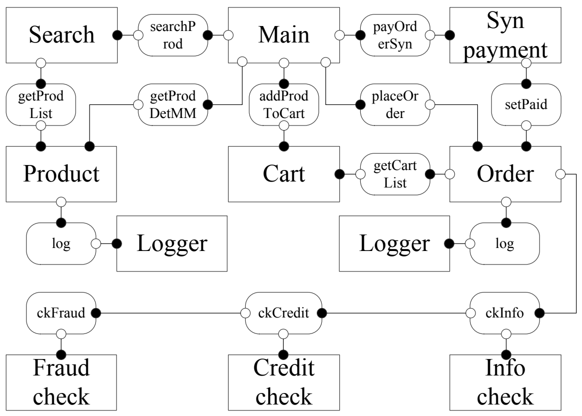


Figure 6: Architecture model of the simplified online shopping system.

fication is performed in sequential structure. And we extend components by including the property *dbRelated* to indicate whether or not a component is database-related.

Based on this metamodel, we propose a set of primitive adaptation operations to express the architectural modification of each design option. These primitive adaptation operations are parsed to generate model transformation scripts as will be shown in Section 4.4.2. These operations include

- `create (component | connector)` *id name type ...*
  `(interface` *id name type*`)+`
  `(parameter` *id name type value*`)*`
  `(when (component | connector)` *condition* `)?`
- `create link` *id name interface interface*
  `(when (component | connector)` *condition*`)?`
- `remove (component | connector | link)`
  `when` *condition*
- `tune parameter` *value* `when` *condition*

with `+`, `*` and `?` respectively indicate there is one or more, zero or more, and zero or one of the preceding element.

For component/connector creation, default properties (e.g., *id*), extended properties (e.g., *dbRelated*), its provided or required *interfaces*, and its *parameters* should be specified. For link creation, properties such as *id*, *name*, *required interface* and *provided interface* should be specified. These creation operations can also specify a crosscutting object (either *component or connector*) and a crosscutting *condition* to indicate that these operations are performed for every component or connector that satisfies the condition.

Removal operations should specify the satisfying *condition* of the to-be-removed component(s)/connector(s)/link(s). Parameter tuning operation should specified the new *value* and the satisfying *condition* of the to-be-tuned parameter(s).

The condition can be expressed by operators such as `=`, `<>`, `>`, `<`, `>=` and `<=` on the properties of components, connectors, links or parameters. Besides, composite conditions are also supported by using `and` and `or`.

With these operations, complex architectural adaptations can be expressed by combinations. We associate each design option with *adaptation* and *revocation* architectural modifications. The former indicates the modification (e.g., create a component) when the option is selected and the latter indicates the modification (e.g., remove a component) when the option is deselected.

For instance, the adaptation architectural modification for design option `Full` is shown as follows. For every component that is neither a logger nor a database-related component, it first creates a logger component, a logger connector and two links to connect them.

`create component` *logComp logComp logCompT* `false`
  `interface` *logIntfL logIntf provided*
  `when component` *type* `<>` *'logCompT'* `and` *dbRelated* `= false`

`create connector` *logConn logConn logConnT*
  `interface` *logIntf_R logIntf* `required`
  `interface` *logIntf_P logIntf* `provided`
  `when component` *type* `<>` *'logCompT'* `and` *dbRelated* `= false`

`create link` *logLinkS logLinkS* `null` *logIntf_P*
  `when component` *type* `<>` *'logCompT'* `and` *dbRelated* `= false`

`create link` *logLinkT logLinkT logIntf_R logIntfL*
  `when component` *type* `<>` *'logCompT'* `and` *dbRelated = false*

The property *required interface* of the first link is set to `null`, which should be the crosscut components' required interface for logging. However, we cannot specify them here because there are multiple crosscut components. To solve this problem, we assume that one link's required and provided interfaces have the same interface name so that the *Architecture Transformer* will find this interface in the crosscut components according to the interface name when performing model transformations.

Besides, its corresponding revocation architectural modification is shown as follows. It removes the previously created components and connectors. The previously created links will be automatically removed because their referenced interfaces are removed with the components and connectors.

`remove component when` *name* `=` *'logComp'*

`remove connector when` *name* `=` *'logConn'*

4.4.2 Executing Architectural Adaptations

The architectural modifications of design options may affect the same part of the architecture and thus produce interferences. For instance, if option `Full` has been realized in the architecture, and there happens a switch from option `Synchronous` to option `Asynchronous`, the newly created component `Asyn payment` will not be linked to a logger connector because option `Asynchronous` is not aware of the architectural modification of option `Full` and vice versa. To compensate such interferences, the revocation and adaptation architectural modification of `Full` should be re-performed sequentially. Therefore, we introduce interference links to indicate such a relationship in order to facilitate the transformation process. In the online shopping system, there exist interference links sourced from `Synchronous`, `Asynchronous`, `Strict` and `Simple` to `Full`. These interference links can be manually constructed at design time by architects.

To ensure consistency, save efforts and reduce errors, we use model transformations to automate the modifications of architecture models. Besides, we use QVT-R as the language for model transformations since it is expressive enough, it is a standard defined by Object Management Group (OMG), and it is the most widely used declarative language to specify the relation between two models. Furthermore, we use mediniQVT [2] to execute model transformations because it implements OMG's QVT-R standard.

Specifically, the *Transformation Generator* is implemented by using the Java template engine FreeMarker [1]. It takes as input a new design solution and returns a set of ordered QVT-R scripts. The generator first makes a copy of the

```
top relation M2M {
  varId, varName : String;
  enforce domain source sM : am::ArchitectureModel {
    id = varId, name = varName
  };
  enforce domain target tM : am::ArchitectureModel {
    id = varId, name = varName
    ...
    -- template of creating links
    <#if linkList?? && linkList?size != 0>
    <#list linkList as link>
    ,structure = ${"tLink" + link_index} : am::Link {
      parent = tM, id = '${link.id}', name = '${link.name}',
      interface1 = ${"tIntf1" + link_index} : am::RequiredInterface {
        id='${link.interface1.id}' },
      interface2 = ${"tIntf2" + link_index} : am::ProvidedInterface {
        id='${link.interface2.id}' }
    }
    </#list>
    </#if>
    ...
  };
}
```

(a)

```
top relation M2M {
  varId, varName : String;
  enforce domain source sM : am::ArchitectureModel {
    id = varId, name = varName
  };
  enforce domain target tM : am::ArchitectureModel {
    id = varId, name = varName
    ...
    ,structure = tLink0 : am::Link {
      parent = tM, id = 'payAsynLinkT', name = 'payAsynLinkT',
      interface1 = tIntf10 : am::RequiredInterface {
        id = 'payAsynIntf_R' },
      interface2 = tIntf20 : am::ProvidedInterface {
        id = 'payAsynIntfA' }
    }
    ...
  };
}
```

(b)

Figure 7: A relation in QVT-R: (a) a template of creating links, (b) a script of create a link.

old/new design solution, and avoids redundant architectural modifications by removing their same design options satisfying condition C that the design option is not the target of an interference link, or is the target of an interference link but its source is not in the design solution. This condition is to avoid the negative effect of interference on model transformations. Then it has an iterative step to parse the revocation/adaptation architectural modification of design options, satisfying C, in the old/new design solution to generate a QVT-R script based on a template and remove corresponding design options until all design options are removed. Finally, it saves the new design solution to the old one.

For instance, $DS1$ and $DS2$ are respectively the old and new design solutions. First, the options with a strike-through line are the same options and thus are removed. Then, the options with a straight underline are the first set of options satisfying c. Finally, the options with a wave underline are the second set of options satisfying c. Therefore, two QVT-R scripts are generated. Figure 7 (a) shows a template of creating links in a relation that copies the *id* and *name* of an architecture model, and Figure 7 (b) gives part of the first generated script that adds a link between a connector and a component for asynchronous payment.

DS1 [`Full`, `Synchronous`, `Textual`, ~~`Sequential`~~, `Strict`$_S$]

DS2 [`Full`, `Asynchronous`, `MM`, ~~`Sequential`~~, `Simple`$_S$]

The *Architecture Transformer* is implemented by using the model transformation tool mediniQVT [2]. It takes as input the generated QVT-R scripts and the current architecture model, and returns an adapted architecture model. The scripts are executed incrementally in the order as they are generated: the first script is executed on the given architecture model, and then the following scripts are executed on the transformed architecture model of the previous script.

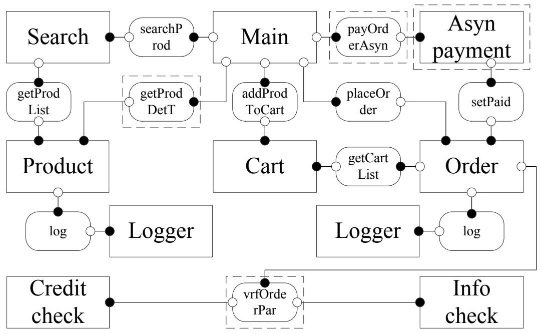


Figure 8: Architecture model after adaptation (the dashed rectangles indicate the modifications).

For instance, Figure 8 shows the architecture model after performing the 1-2-4-5-7 adaptation loop on Figure 6. The requirements adaptations include switching from multimedia mode and strict verification to textual mode and simple verification, and the architectural adaptations include changing synchronous payment and sequential verification to asynchronous payment and parallel verification.

5. CASE STUDY

To evaluate the proposed approach, we conducted a case study to answer the following two questions:

- **Q1**: Can improvements be achieved by combining requirements and architectural adaptations? (Section 5.2)
- **Q2**: Can the approach scale with the growth of requirements and architecture models? (Section 5.3)

5.1 Experimental Setup

Stress testing tool JMeter was used to simulate concurrent accesses and Badboy was used to record the test plan. The experiments were conducted on a ThinkPad E430c laptop with Intel Core i3 2.40 GHz processor and 4GB RAM.

The experiments were conducted on the online shopping benchmark, which is implemented in Java. Its requirements and architectural design decisions are illustrated in Figure 1 and Figure 4. The preferences for the softgoals in Figure 1 were all specified to 5, and the initial expectations for softgoals `Security be high`, `Performance be high`, `Usability be good`, `Browse response be quick`, `Cost be low`, `Risk be low`, `Check response be quick`, `Resources be low` and `Payment response be quick` were respectively set to +0.6, −0.2, +0.6, −0.2, +0.6, +0.6, −0.2, +0.6 and −0.2, and such tradeoffs have to be made because soft goals often cannot be all fully satisfied. The system was initially configured to the design solution [`Full`, `Synchronous`, `MM`, `Sequential`, `Strict`$_S$] that best satisfies these quality expectations.

The system workload varied in the experiments as follows to simulate dynamic environments. The workload first increased from zero users to 100 concurrent users in 10 minutes, i.e. adding a user every 6 seconds; then held for 4 minutes; finally decreased back to zero users in 10 minutes, i.e. removing a user every 6 seconds. The adaptation interval was set to one minute; i.e., the adaptation mechanism was periodically performed every one minute, which is enough for the adaptation process in this case study. For other applications, the interval should be accordingly specified.

In the experiments, the quality values such as performance (i.e. the time taken to process a request), browse response,

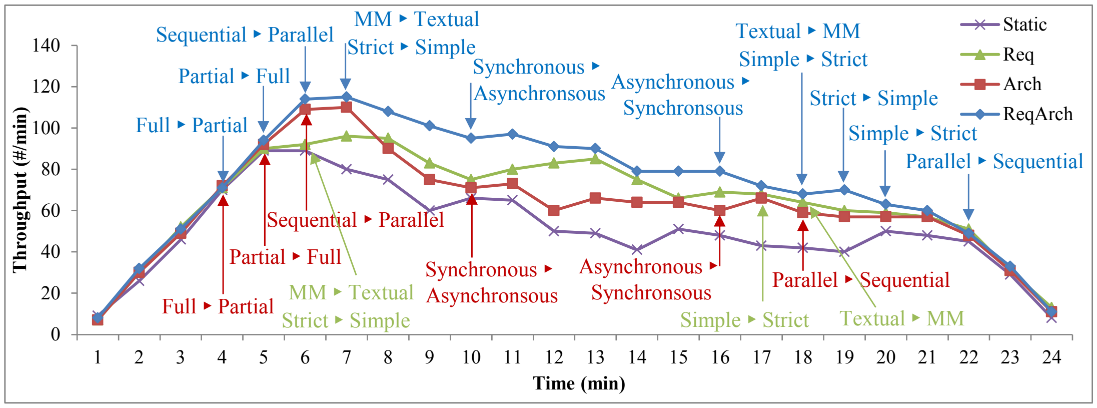


Figure 9: The adaptation process of the approaches with self-adaptation.

payment response, check response, and cost (i.e. the money paid to verify an order) were obtained by system log analysis, and resources (i.e. the memory consumed to complete an order payment) were measured by memory analysis.

However, to measure security, risk and usability, it often involves complex security analysis, risk analysis and customer feedback analysis. To simulate such real-life analysis, for simplicity, here we assumed that security was a random value between 88% and 92% with full logging and between 83% and 87% with partial logging since full logging can often achieve a higher security than partial logging. Similarly, risk was a random value between 9% and 13% with strict verification and between 14% and 18% with simple verification; and usability was a random value between 89% and 93% with multimedia mode and between 84% and 88% with textual mode. Such ranges of values were specified by the domain experts according to their experience.

For each of the following four approaches, we conducted the experiments using the same experimental settings.

- **Static**: the approach without self-adaptation
- **Req**: the requirements-driven self-adaptation approach
- **Arch**: the architecture-based self-adaptation approach
- **ReqArch**: the proposed self-adaptation approach

Specifically, **Req** only involves the proposed requirements adaptations and architectural adaptations are simply mapped to architectural elements. **Arch** assumes a static goal specification and only involves the proposed architectural adaptations.

5.2 Effectiveness Evaluation (Q1)

To compare the effectiveness of different approaches, we measured the system throughput (i.e. the successfully finished orders in one minute), which is the key performance indicator of this application. The higher the overall throughput, the more effective the adaptation is considered to be.

Figure 9 shows the adaptation process of the self-adaptation approaches (**Req**, **Arch** and **ReqArch**) to visually illustrate the differences of our approach from requirements-driven and architecture-based approaches. In Figure 9, the X axis denotes time intervals of one minute and the Y axis denotes system throughput in each time interval. The adaptations generated by each approach are respectively marked on the curves.

For **Static**, it can be observed that the system suffered a throughput loss when the workload increased to 60 concurrent users at time 6, and continuously had a low throughput until the workload decreased to 30 concurrent users at time 22. This is because the system suffered performance degradation due to the increasing workload.

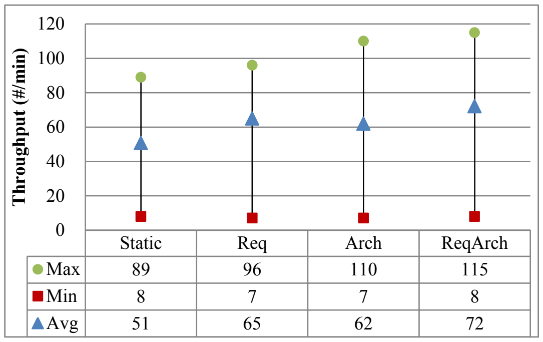
For **Req**, the system also suffered a throughput loss at time 6 when check response and browse response were very slow. Unlike **Static**, **Req** increased the expectations of check response and browse response, and thus reconfiguring the system from multimedia mode and strict order verification to textual mode and simple order verification for a better response. As a result, **Req** achieved a higher throughput than **Static** at the following time, but risk and usability were low achieving and their expectations were increased. When the workload decreased to 80 concurrent users, check response and browse response got better and their expectations were decreased. As a result, **Req** reconfigured the system to strict order verification and multimedia mode at time 17 and 18.

For **Arch**, the system suffered a performance degradation at time 4, which was not handled in the case of **Req** since the solution was out of the business-level adaptation space. But **Arch** reconfigured the system from full logging to partial logging for a better performance at the price of security. After a time interval, **Arch** reconfigured the system back to full logging because security was now more expected than performance. At time 6, check response was very slow, and **Arch** increased its expectation, and thus reconfiguring the system from sequential verification to parallel verification. At time 10, the applicable context of asynchronous payment was satisfied and the expectation of payment response was increased, and thus **Arch** reconfigured the system to asynchronous payment. When the workload decreased, **Arch** reconfigured the system back to synchronous payment and sequential verification at time 16 and 18. It can be seen that **Arch** achieved a higher throughput than **Static** but a lower throughput than **Req** from time 8, which means that in this case the solutions in the design-level adaptation space is less effective in terms of throughput than the ones in the business-level adaptation space.

For **ReqArch**, when design-level adaptation is not enough, business-level adaptation is involved; e.g., at time 7, 18, 19 and 20. As a result, as visually shown in Figure 9 and numerically shown in Figure 10, **ReqArch** achieved the highest maximum and average throughput. This shows that by combining requirements and architectural adaptations, our approach is promising to offer more adaptation flexibility and further improve the effectiveness of adaptation.

Table 1: Average results of the four approaches in terms of quality values.

App.	Sec. (%)	Per. (ms)	Usa. (%)	bRe. (ms)	Risk (%)	Cost ($)	cRe. (ms)	Reso. (B)	pRe. (ms)
Static	89.99	1451.92	91.01	3721.98	11.02	0.34	4473.24	449.98	3797.45
Req	90.03	915.13	88.50	1877.36	13.29	0.32	2657.14	450.06	2036.87
Arch	89.78	888.28	90.98	2052.48	10.98	0.34	2919.15	462.44	2282.54
ReqArch	89.81	682.46	88.73	1477.01	13.49	0.32	2103.00	462.49	1466.79



	Static	Req	Arch	ReqArch
● Max	89	96	110	115
■ Min	8	7	7	8
▲ Avg	51	65	62	72

Figure 10: Throughput of the four approaches.

In addition, Table 1 reports the average results of quality values per interval of the four approaches. Columns 2–10 respectively list the results of security, performance, usability, browse response, risk, cost, check response, resources and payment response. It can be observed that, compared with the other three approaches, our approach was better in performance-related quality dimensions but worse in others. In other words, with the changing workload, our approach can achieve a better performance, which is always expected under high workload to improve throughput, with the acceptable sacrifice of other quality dimensions.

In summary, the observations from Table 1 and Figure 9 and 10 answer **Q1** positively that our approach is promising to further improve the effectiveness of adaptation in terms of system throughput in this case study and offer more adaptation flexibility than requirements-driven or architecture-based self-adaptation approaches with acceptable sacrifice of less expected quality dimensions.

5.3 Performance Evaluation (Q2)

The performance of our approach is determined by *Requirements Adaptation Manager*, *Design Decisions Adaptation Manager*, *Transformation Generator* and *Architecture Transformer*, whose time complexities are respectively exponential time with the size of OR-decomposed goals, exponential time with the size of design decisions, linear time with the size of primitive adaptation operations and linear time with the size of architecture model.

We conducted a set of experiments to evaluate the performance of our approach. Table 2 reports the experiment results. The first, third, fifth and seventh columns respectively list the size of OR-decomposed goals, design decisions, primitive adaptation operations and architecture model. And the other columns list the performance in milliseconds.

The *Requirements Adaptation Manager* takes around 1.1 seconds on the goal model with 18 OR-decomposed goals (with 46656 goal specifications), which is feasible in our approach, and returns an "out of memory" error when the number of OR-decomposed goals climbs to 24 (with 1679616 goal specifications). The *Design Decisions Adaptation Manager* takes around 3.7 seconds on the design decisions model with

Table 2: Performance of our approach.

RAM		DDAM		TG		AT	
OR(#)	P.(ms)	DD(#)	P.(ms)	Op.(#)	P.(ms)	AM(#)	P.(ms)
3	1	5	1	200	79	150	163
6	2	7	15	400	92	300	224
9	24	9	31	600	105	450	315
12	78	11	78	800	114	600	400
15	203	13	202	1000	120	750	600
18	1108	15	577	1200	125	900	702
21	22480	17	3620	1400	131	1050	977
24	out	19	out	1600	136	1200	1249

17 design decisions (with 131072 design solutions), which is feasible in our approach, and returns an "out of memory" error with 19 design decisions (with 524388 design solutions). The *Transformation Generator* takes less than 0.2 seconds with 1600 primitive adaptation operations, and the *Architecture Transformer* takes around 1.3 seconds with 1200 architectural elements. The above analysis answers **Q2** positively that our approach scales well with the growth of models and can be applied to real-life medium-sized software systems.

6. RELATED WORK

Instead of enumerating the related work in the area of self-adaptive systems, we refer readers to [12] and [29] for an introduction to the state-of-the-art. Here we only discuss the most related studies in three areas: requirements-driven self-adaptation, architecture-based self-adaptation, and earlier work that combines them.

6.1 Requirements-Driven Self-Adaptation

Approaches have been proposed to use requirements models as the knowledge for self-adaptation. Dalpiaz et al. [16] propose a conceptual architecture that provides systems with self-reconfiguration capabilities. Wang et al. [47] propose a requirements monitoring and diagnosing framework to provide systems with self-repairing capabilities. Elkhodary et al. [18] propose a feature-oriented self-adaptation framework FUSION that can learn the impact of adaptation decisions. Baresi et al. [6] present FLAGS to facilitate requirements-driven adaptations. Peng et al. [36] propose a requirements-driven self-tuning approach through dynamic quality tradeoff and value-based feedback loop. Chen et al. [10, 11] propose requirements-driven approaches for survivability assurance of Web systems and optimization of composite services. Fu et al. [21] propose a stateful requirements monitoring approach for self-repairing socio-technical systems. Salehie et al. [38] propose a requirements-driven approach to support adaptive security for protecting variable assets. Souza et al. [41] propose evolution requirements to define possible changes to the requirements, which can be integrated into our approach to provide richer requirements adaptations.

These approaches assume requirements (e.g., goals or features) can be mapped to architectural elements directly (e.g., components or services) and thus largely neglect the complexity and details in architectural design. In contrast, our approach introduces a design decisions model as intermediary and uses incremental and generative model transformations to implement more complex architectural adaptations.

6.2 Architecture-Based Self-Adaptation

Oreizy et al. [35] introduce the concept of architecture-based runtime software adaptation and evolution management. Garlan et al. [22] propose an architecture-based self-adaptation framework Rainbow, which provides a reusable infrastructure customizable for specific systems. Rainbow executes system-specific adaptation strategies written in Stitch language [13] after a violation of the invariant imposed by the architecture model. Floch et al. [19] propose a mobility- and adaptation-enabling middleware MADAM, which exploits architecture models for runtime adaptation of mobile computing applications. Georgas and Taylor [23] propose a policy-based approach to architectural adaptation management and establish the feasibility to apply the approach to robotic architectures. Morin et al. [32] propose to use model-level aspects to encapsulate variants and manage dynamic variability, and weave the corresponding aspects of a selection of variants into a base model to produce a new architecture model.

These approaches assume that requirements of self-adaptive systems are well-understood at design time and unchanged at runtime, thus are unable to support architectural adaptations resulting from requirements changes. Furthermore, most of these approaches support simple architectural adaptations such as adding, removing or replacing components but cannot support complex architectural adaptations such as crosscutting adaptations and restructuring architectural elements. The exceptions are Oreizy et al.'s restructuring [35], Garlan et al.'s strategy writing [22], and Morin et al.'s aspect weaving [32], however, these changes rely mostly on human experts. Compared with them, our approach further supports complex architectural adaptations such as crosscutting and restructuring ones using model transformation techniques.

6.3 Combining Requirements and Architectural Evolution or Adaptation

Nuseibeh [34] proposes to weave requirements and architectures for incremental software development and speedy delivery. Sawyer et al. [39] call for combining requirements and architectures at runtime for self-adaptive systems. Since then, several advances have been made in this direction.

Kramer and Magee propose to combine requirements and architectures for self-management, and propose a three-layer reference model [28]: a *goal management layer* for deliberative planning, a *change management layer* for reactive plan execution, and a *component control layer* for application-specific adaptation actions. As an instantiation, Sykes et al. use a planning-as-model-checking technique to generate plans [42], construct component configurations according to the planned actions and the interface dependencies among components [43], and choose the best one by utility functions on quality properties [44]. Their approach constructs a component configuration based on the functionality dependencies, thus regarding the architectures as a set of functionalities, neglecting the complexity of architectural design.

Tajalli et al. [45] propose a plan-based layered architecture for software model-driven adaptation PLASMA, which utilizes an architecture description language and a planning-as-model-checking technique to enable dynamic re-planning. It supports architectural adaptations resulting from requirements changes that are provided by architects at runtime.

Alferez et al. [5] propose a model-based framework that supports the dynamic evolution of context-aware systems to deal with unexpected context events. It uses goal and feature models to respectively represent the alternative space of requirements and architectures. However, it also focuses on the functionalities of the systems, and assumes each feature can be directly mapped to an architectural element, thus neglecting the complexity of architectural design.

Pimentel et al. [37] propose the STREAM-A approach, a systematic process to generate architectural design models from requirements models for adaptive systems using model transformations. Their focus is on the design-time development of adaptive-systems, whilst our approach focuses on the runtime self-adaptation.

In brief, the main differences of our approach from these approaches are that it further treats architectures as a set of design decisions concerning how the functionalities are structured and interact with each other, it supports both requirements and architectural planning, and it addresses crosscutting and restructuring adaptations using model transformation techniques.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a model-based self-adaptation approach that combines requirements and architectural adaptations. It uses architectural design decisions models to consider not only the functionalities of architectures but also their structures and behaviors. In addition, it treats requirements and architectural adaptations in a layered manner and supports crosscutting and restructuring architectural adaptations using incremental and generative model transformations. Our case study using an online shopping benchmark shows promise that our approach can further improve the effectiveness of model-based self-adaptation approaches (e.g. system throughput in this case study) and offer more adaptation flexibility.

Currently our approach does not support adaptations resulting from unanticipated changes; e.g., adding a new goal or a new design option. A possible remedy is to provide explicit management interfaces for the administrators to update relevant models at runtime. In addition, the interference links in the design decisions model are created manually, which could be difficult to understand and maintain for large systems. It is our future work to automate the creation of these links. We also plan to extend our approach by supporting widely-used architecture description languages (e.g., xADL 2.0 [17]) and supporting more model transformation tools (e.g., MMT [3]), integrate it with architecture-based management middleware (e.g., SM@RT [40]), and apply our approach to more software systems to further evaluate its effectiveness.

8. ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China under Grant No. 61361120097, National High Technology Development 863 Program of China under Grant No. 2013AA01A605, ERC Advanced Grant 291652 - ASAP, and Science Foundation Ireland Grant 10/CE/I1855.

9. REFERENCES

[1] FreeMarker. http://freemarker.org/.

[2] mediniQVT. http://projects.ikv.de/qvt/wiki.

[3] MMT. http://wiki.eclipse.org/Model_to_Model_-Transformation_-_MMT.

[4] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v1.0, 2008. http://www.omg.org/spec/QVT/1.0/PDF/.

[5] G. H. Alférez and V. Pelechano. Dynamic evolution of context-aware systems with models at runtime. In *MoDELS*, pages 70–86, 2012.

[6] L. Baresi, L. Pasquale, and P. Spoletini. Fuzzy goals for requirements-driven adaptation. In *RE*, pages 125–134, 2010.

[7] G. S. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[9] A. Charfi and M. Mezini. AO4BPEL: An aspect-oriented extension to BPEL. *World Wide Web*, 10(3):309–344, 2007.

[10] B. Chen, X. Peng, Y. Yu, and W. Zhao. Are your sites down? Requirements-driven self-tuning for the survivability of Web systems. In *RE*, pages 219–228, 2011.

[11] B. Chen, X. Peng, Y. Yu, and W. Zhao. Requirements-driven self-optimization of composite services using feedback control. *IEEE Trans. Services Computing*, 2014. Accepted.

[12] B. H. Cheng, R. Lemos, H. Giese, and et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, pages 1–26. Springer-Verlag, 2009.

[13] S.-W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.*, 85(12):2860–2875, 2012.

[14] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[15] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1:1–1:42, 2008.

[16] F. Dalpiaz, P. Giorgini, and J. Mylopoulos. An architecture for requirements-driven self-reconfiguration. In *CAiSE*, pages 246–260, 2009.

[17] E. Dashofy, A. van der Hoek, and R. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA*, pages 103–112, 2001.

[18] A. Elkhodary, N. Esfahani, and S. Malek. FUSION: A framework for engineering self-tuning self-adaptive software systems. In *FSE*, pages 7–16, 2010.

[19] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Softw.*, 23(2):62–70, 2006.

[20] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE*, pages 37–54, 2007.

[21] L. Fu, X. Peng, Y. Yu, J. Mylopoulos, and W. Zhao. Stateful requirements monitoring for self-repairing socio-technical systems. In *RE*, pages 121–130, 2012.

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

[23] J. C. Georgas and R. N. Taylor. Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In *SEAMS*, pages 105–112, 2008.

[24] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani. Reasoning with goal models. In *ER*, pages 167–181, 2002.

[25] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *WICSA*, pages 109–120, 2005.

[26] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[27] M. Koning, C.-a. Sun, M. Sinnema, and P. Avgeriou. VxBPEL: Supporting variability for web services in BPEL. *Inf. Softw. Technol.*, 51(2):258–269, 2009.

[28] J. Kramer and J. Magee. Self-managed systems: An architectural challenge. In *FOSE*, pages 259–268, 2007.

[29] R. Lemos, H. Giese, H. Müller, and et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems*. Schloss Dagstuhl, 2011.

[30] S. Liaskos, S. A. McIlraith, S. Sohrabi, and J. Mylopoulos. Integrating preferences into goal models for requirements engineering. In *RE*, pages 135–144, 2010.

[31] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.

[32] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *ICSE*, pages 122–132, 2009.

[33] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Softw. Eng.*, 18(6):483–497, 1992.

[34] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–117, 2001.

[35] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE*, pages 177–186, 1998.

[36] X. Peng, B. Chen, Y. Yu, and W. Zhao. Self-tuning of software systems through dynamic quality tradeoff and value-based feedback control loop. *J. Syst. Softw.*, 85(12):2707–2719, 2012.

[37] J. Pimentel, M. Lucena, J. Castro, C. Silva, E. Santos, and F. Alencar. Deriving software architectural models from requirements models for adaptive systems: the stream-a approach. *Requirements Eng.*, 17(4):259–281, 2012.

[38] M. Salehie, L. Pasquale, I. Omoronyia, R. Ali, and B. Nuseibeh. Requirements-driven adaptive security: Protecting variable assets at runtime. In *RE*, pages 111–120, 2012.

[39] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for RE for self-adaptive systems. In *RE*, pages 95–103, 2010.

[40] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. *J. Syst. Softw.*, 84(5):711–723, 2011.

[41] V. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos. Requirements-driven software evolution. *Comput. Sci. Res. Dev.*, 28(4):311–329, 2013.

[42] D. Sykes, W. Heaven, J. Magee, and J. Kramer. Plan-directed architectural change for autonomous systems. In *SAVCBS*, pages 15–21, 2007.

[43] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: A combined approach to self-management. In *SEAMS*, pages 1–8, 2008.

[44] D. Sykes, W. Heaven, J. Magee, and J. Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *SAC*, pages 431–438, 2010.

[45] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic. PLASMA: A plan-based layered architecture for software model-driven adaptation. In *ASE*, pages 467–476, 2010.

[46] H. van Vliet. *Software Engineering: Principles and Practice*. Wiley, third edition, 2008.

[47] Y. Wang and J. Mylopoulos. Self-repair through reconfiguration: A requirements engineering approach. In *ASE*, pages 257–268, 2009.

[48] E. S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *RE*, pages 226–235, 1997.