Using QoS Contracts to Drive Architecture-Centic Self-Adaptation

Franck Chauvel, Hui Song, Xiang Ping, Gang Huang, Hong Mei

Using QoS-Contracts to Drive Architecture-Centric Self-adaptation*

Franck Chauvel, Hui Song, Xiang Ping Chen, Gang Huang, and Hong Mei

Key Laboratory of High Confidence Software Technologies, Ministry of Education
School of Electronics Engineering and Computer Science
Peking University, Beijing, 100871, PRC
{franck.chauvel,songhui06,chenxp04,huanggang,meih}@sei.pku.edu.cn

Abstract. Self-adaptation is now a promising approach to maximize the satisfaction of requirements under changing environmental conditions. One of the key challenges for such self-adaptive systems is to automatically find a relevant architectural configuration. Existing approaches requires a set of adaptation strategies and the rough estimation of their side-effects. However, due to the lack of validation methods for such strategies and side-effects, existing approaches may lead to erroneous adaptations. Instead of side-effects, our solution leverages quality contracts whose accuracy can be separately established and which can be dynamically composed to get a quality prediction of any possible architectural configurations. To support self-adaptation, we propose a reactive planning algorithm which exploits quality contracts to dynamically discover unforeseen architectural configurations. We illustrate our approach using a running HTTP server adapting its architecture with respect to the number and the similarity of incoming requests.

1 Introduction

The growing complexity of software systems and the need of continuously-running systems have resulted in the emergence of self-adaptive systems (SAS). Such systems adjust their internal architecture with respect to their execution environment in order to meet their functional or non-functional requirements. SAS are mainly built as a control-loop [6] which includes monitoring the running system, analyzing the collected data, planning the needed reconfigurations, and executing those reconfigurations.

Correctly planning the needed changes of the running system according to the environmental conditions is critical to get an effective self-adaptation. Existing approaches search among a finite set of predefined architectural changes named

* This work is partially sponsored by the National Key Basic Research and Development Program of China under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 60821003, 60873060; the High-Tech Research and Development Program of China under Grant No. 2009AA01Z16; and the EU FP7 under Grant No. 231167.

G.T. Heinemann, J. Kofron, and F. Plasil (Eds.): QoSA 2010, LNCS 6093, pp. 102–118, 2010.

adaptation rules in Plastic [1] and [14], *strategies/tactics* in Rainbow [11,7], *architectural aspects* by Morin et al. [15] or *actions* in [20,13]. In the presence of multiple and conflicting requirements, the selection takes into account the expected quality side-effects of each change in order to ensure a good trade-off between the desired quality properties (side-effects are modeled as help/hurt values and utility functions in Rainbow [11,7], MADAM [10], DiVA [9] and [20,13]).

The two key challenges in planning design are thus the definition of a set of possible architectural changes and the estimation of their side-effects on the desired quality properties with respect to the environmental conditions. However, although both of these activities are critical for SAS, they remain hand-crafted and highly error-prone: Ensuring that no architectural changes have been overlooked and that their side-effects are realistic remains extremely difficult [6].

Our contribution is to avoid the enumeration of architectural modifications and consequently the estimation of their side-effects. Instead, our solution leverages quality contracts whose accuracy can be independently established and which can be dynamically composed to get quality predictions for any meaningful architectural configuration.

Our solution combines a reactive planning algorithm with the parametric contracts proposed by Firus et al. [8]. The planning algorithm dynamically searches for an architectural change that better fits the environment and the quality objectives, whereas the parametric contracts predict the quality of each resulting architectural configuration. For the sake of interoperability, our prototype leverages standard model-driven techniques and can thus be connected to various execution platforms. We illustrate our approach on an HTTP server deployed on the Fractal platform [4], which adapts its architecture to the number and the density of incoming requests.

The remainder of the paper is organized as follows. Section 2 illustrates the limitations of existing techniques using the HTTP server. Section 3 gives an overview of our approach. Section 4 presents how we model contracts and component-based architectures whereas our self-adaptation algorithm is formalized in Section 5. Section 6 presents our prototype implementation and the related experimental results. Finally Section 7 discusses additional related works before Section 8 concludes and outlines some future works.

2 Motivating Example

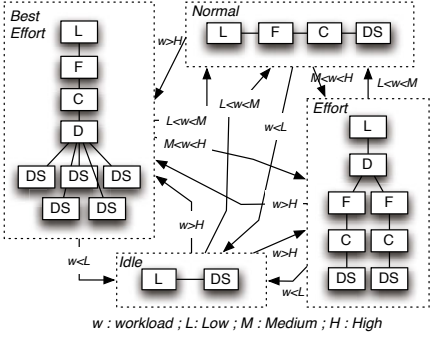
This section illustrates on an running scenario the limitations of existing approaches while designing SAS.

Let us consider an HTTP server made of two main components: a listener component (L) reads HTTP requests on a socket and transmits them to a data server component (DS) that returns the corresponding HTTP responses. In addition, three optional components may be inserted: A cache component (C) reduces the response time by caching solved requests ; a filter component (F) detects harmful requests (e.g. containing SQL code) and a dispatcher component (D) enables the combination of several data servers. Figure 1a illustrates several possible architectural configurations of the HTTP server.

The number of possible architectural configurations is potentially infinite since multiple instances of each component definition can be combined. From the functional point of view, a first constraint enforces that filter components (if deployed) protect cache components to avoid caching harmful requests. Another constraint establishes that both filtering and caching must happen only once for each data server. From the non-functional point of view, three main requirements must be met, namely the "minimization of the overall response time" (R.T.), the "maximization of the security level" (S.L.), and the "minimization of the memory consumption" (M.).

A first solution to the design of such a SAS is described in C2 [17], Plastic [1] or Genie [2]. It requires to statically identify a small set of frozen architectural configurations (also known as architectural modes) ; each one resulting from a trade-off between quality objectives and environmental conditions. In Figure 1a, four modes are selected: *Idle*, *Normal*, *Effort* and *Best Effort*. The *Idle* mode only includes one listener and one data server to handle the smallest workload ($w < low$). When the workload increases ($low \leq w < medium$), the system switches to the *Normal* mode to add a cache and a filter. If the workload keeps increasing, the system uses the *Effort* mode ($medium \leq w < high$) where two additional data servers are both cached and filtered. For heavier workloads ($w \geq high$), the system uses the *Best Effort* where a group of 5 servers is cached and filtered. The resulting behavior of the HTTP server is a meshed automaton where each state is a frozen architectural mode and each transition is triggered by a given state of the environment.

A better solution initially proposed in Rainbow [11,7] but also used in DiVA [9] and by Sykes et al. [20,13], is to identify adaptations strategies, their triggering conditions, and their respective costs or benefits with respect to the quality objectives. Table 1b outlines the adaptation strategies needed to deploy (rep. undeploy) each optional component (cache, filter, dispatcher and extra data servers) and their possible side effects on the quality objectives (memory in kB, reponse-time in ms, and security level).



(a) Using Predefined Configurations

Strategy	**Side-Effects** R.T.	M.	S.L.
AddCache	-50	+200	0
RemoveCache	+100	-200	0
AddFilter	+50	+200	+1
RemoveFilter	-50	-200	-1
AddDispatcher	+10	+25	0
RemoveDispatcher	-10	-25	0
AddServer	-200	+500	0
RemoveServer	+200	-500	0

(b) Using Adaptation Strategies

Fig. 1. Existing approaches for self-adaptation applied on the HTTP server

Predefined configurations (Figure 1a) enable a straightforward planning that consists in triggering the relevant transition in the automaton with respect to the current environment. In addition, it ensures that the functional constraints (e.g. the filter components protect the cache components) are respected. However, establishing that the predefined set of configurations is complete remains very difficult and configurations may easily be overlooked [6].

Adaptation strategies (Figure 1b) require the designer to roughly evaluate their quality side-effects (using absolute deviation or utility functions). However, estimating such side-effects at design time is difficult and error prone. Quality side-effects do not only depend on the selected strategy but also on the architectural configuration on which this strategy is applied, and on the environmental conditions. In the HTTP server for instance, the benefits of deploying a cache (row 1 of Tab. 1b) depends on the similarity of incoming requests and on its position with respect to the dispatcher (if deployed). In addition, using strategies that are only applicable on one environmental condition would roll back to the first solution.

Our solution is an alternative to these two solutions and avoids both the enumeration of predefined configurations and the rough estimations of quality side-effects.

3 Approach Overview

By contrast with existing approaches, which either predefine configurations or estimate quality side-effects, our approach leverages "composable" quality contracts whose accuracy can be separately established. Our solution gradually explores the possible architectural configurations by composing additional components, and concurrently builds the related quality model by composing their quality models (quality contracts).

As shown in Figure 2, we modify (on a model of the running system) the current configuration ($L-DS$ in Figure 2) by adding (resp. removing) one component instance and by rebuilding all the possible connection schemes between the remaining components. The use of quality contracts consequently enables the evaluation of the quality of each resulting configuration. Further details about the exploration of the configuration space are given in Section 5.

The key point of our approach is that the quality evaluation is based on complete architectural configurations and not on the expected side effects of the modification which led to new configurations. Such evaluation is made using quality contracts [8], that specify the quality of each individual component definition (L, F, C, D, and DS) as do classical performance models (c.f. Section 4.1 for further details on contracts). Contracts bring us the four following benefits:

1. They allow designers to specify compositional quality model for each component definition which accuracy can be established during unit testing [8].
2. They ensure (theoretically [3]) that all possible connections between components makes sense form the syntactic, behavioral, quality and semantic perspective. They thus ensure that the algorithm only explores meaningful configurations.

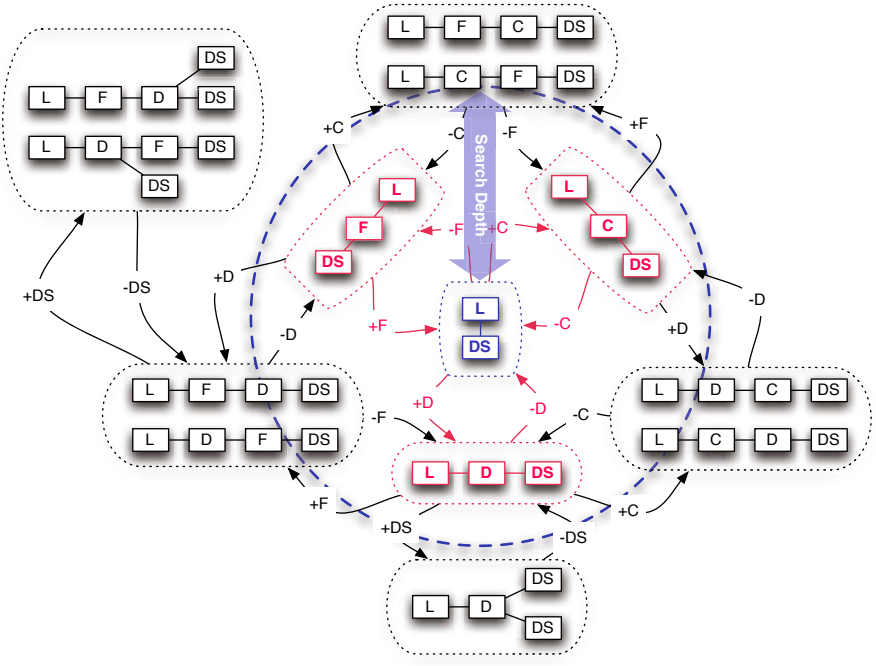


Fig. 2. Approach Overview: Gradual Exploration of the Configurations Space

3. They allow designers to express complex architectural constraints (such as Cache/Filter order) by propagating data on components interfaces.
4. They reduce the number of possible connection schemes, by providing a restrictive interface matching mechanism.

4 Modeling Self-adaptive Systems

This section presents the two main inputs required by our algorithm, namely: (i) the software architecture model specifying the available component definitions (such as Listener, Cache, etc.) and including the contracts specifying the possible connections and, (ii) the adaptation policy specifying the quality objectives that the system must try to satisfy and their relationship with the contracts.

4.1 Modeling Software Architecture

UML 2.x [16] is now the *de facto* standard commonly used by both academics and industrials to described software architectures. However, since UML goes beyond the scope of component-based software architecture, we extracted a minimal subset of concepts needed to perform architecture centric self-adaptation.

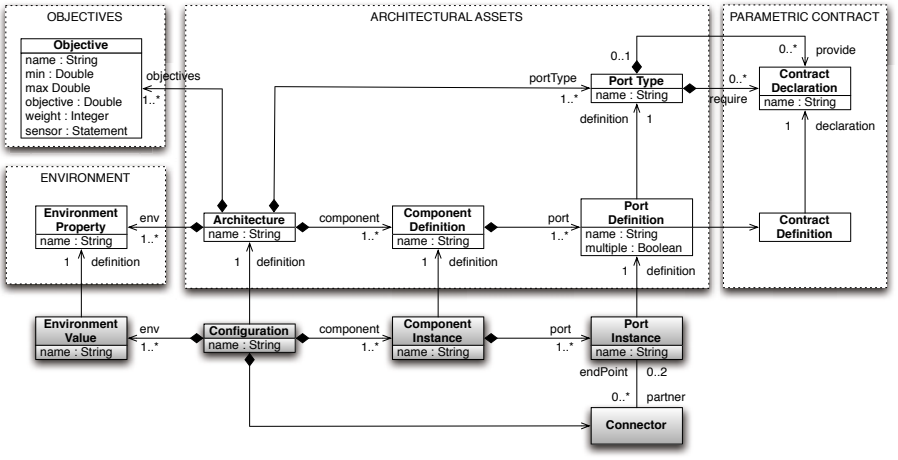


Fig. 3. Modeling Architectural Assets, Contracts, Objectives, and Environmental Properties

Figure 3 formalizes this subset of concepts. In the central frame, an *Architecture* contains a set of *Component Definitions* defining the potential connection points (so called "port") of component. For the sake of conciseness, Figure 3 does not include the definition of functional interfaces but they are needed and encapsulated into the contract definitions.

As explained by Beugnard et al. [3] the notion of contract is a suitable abstraction to describe and specify various properties on component interfaces, including syntax, behavior, synchronization, performances/quality and potentially semantics. Contracts formalize the relationship between provided and required interfaces (in an "assume/guarantee" manner).

Modeling Performances and Quality using Contracts. Non-functional properties such as performance or quality of service (QoS) are commonly modeled as crisp values added on components interfaces (CQML, QoSCL or SLA). By contrast, parametric contracts [8] advocate the use of numerical functions, which capture dependencies between provided and required interfaces. Then, once components interfaces are bound (to build a configuration), these functions can be evaluated to get a end-to-end quality prediction. Parametric contracts enable the encapsulation of different mathematical models for different QoS properties (ad-hoc, probabilistic, markovian, etc.) and their combination using function composition. Our contribution is not the definition of a new formalism to model QoS, but rather focuses on the use of parametric contract to enable self-adaptation.

The top part of Figure 4 provides several illustrative examples of parametric contracts applicable on the HTTP server. The response time (RT) of the data server is defined as the average time to process a given workload (as a number of requests per sec). It is worth to note that the parametric contracts presented

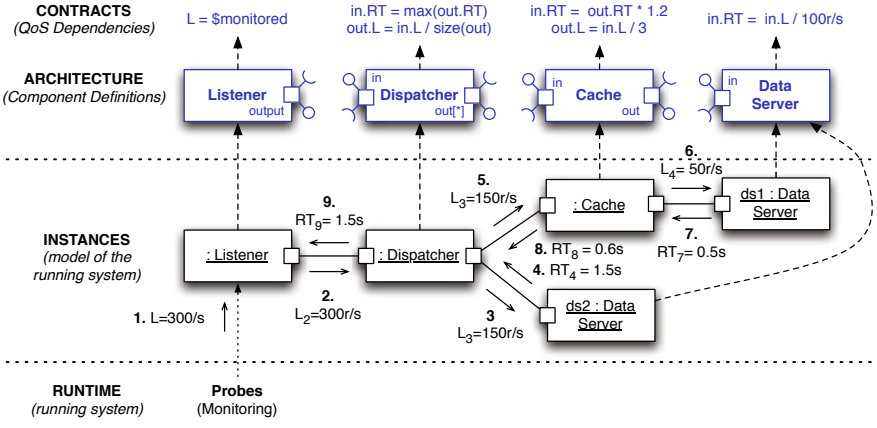


Fig. 4. End-to-End Quality Evaluation using Parametric Contracts

here are over-simplified, realistic and accurate models are proposed in [8]. In Figure 3, each *Contract Definition* encapsulates the imperative description of a function (the related abstract syntax is not detailed here).

Figure 4 illustrates such end-to-end QoS evaluation on a simplified example. At first, the running system is monitored to get initial data (e.g. the load delivered by the listener). Thanks to the parametric contracts, this initial load can be propagated to the other component instances (messages 2, 3, 5 and 6 at the instance level). For example, the dispatcher equally balances the load between its two outputs. Contracts then enable to compute the response time of the data servers and to back propagate it to the listener instance (messages 4, 7, 8 and 9).

Modeling Architectural Constraints using Contracts. Contracts can also be used to enforce architectural constraints. For the record, the HTTP server includes two functional constraints: *"filtering requests must happen before caching to avoid caching harmful requests"* and *"filtering and caching must happen only once for each data server"*. To enforce such constraints, component interfaces in the HTTP must specify if the HTTP requests have already been filtered and/or cached. For instance, the cache component assumes that its incoming HTTP requests have not yet been cached and guarantees that the requests which are transmitted to its backbone have been cached. Listing 1 illustrates the realization of such constraints using contracts.

Listing 1. Excerpt of the contracts needed to order cache and filter instances

```
1  cache.input
       provides processRequest(r1 : Request)
3          assume !cached(r1)

5  cache.output
       requires processRequest(r2 : Request)
7          guarantee cached(r2) and (isFiltered(r1) <=> isFiltered(r2))
```

4.2 Modeling Environment and Objectives

The quality objectives (class *Objective* in Figure 3) are modeled as numeric properties. They include a validity interval (defined by the *min* and *max* attributes) as well as an objective value. Although any value within the range may be used as an objective, only the bounds are used in practice since they reflect the "minimize" and "maximize" requirements respectively.

Listing 2. Minimization of the response time

```
issue response_time : Real
is
    range is [0, 20]
    objective is 0
    priority is 5
    sensor
    do
        value := component.select{ c | c.isKindOf("Listener") }.first()
            .output.rt
    end
end
```

Listing 2 shows an excerpt of the textual syntax used in our prototype to model objectives. It describes the *response time* of the HTTP server as a real value over $[0, 20]$ which optimal value is 0 and which priority is 5 (in $[0, 10]$). The last element defines the contract used to evaluate the response time on a given configuration. The response time of the HTTP Server is measured as the "rt" contract required by the port output of the listener component (See section 4.1).

Additional information such as deployment parameters (CPU speed, bandwidth, etc.) can be stored in our model as environmental properties and then used to further parameterize the quality contracts.

5 Self-adaptation Algorithm

The section formalizes the algorithm which exploits the quality contracts at runtime to enable self-adaptation.

5.1 Problem Formalization

In order to formalize the SAS problem and our self-adaptation algorithm, let us formalize the concepts introduced by Figure 3:

- $\mathcal{E}$ is the execution environment.
- $\mathcal{D}$ is the set of possible component definitions.
- $\mathcal{C}$ is the set of possible component instances.
- $new : \mathcal{D} \rightarrow \mathcal{C}$ is a function which creates a new instance of a given component definition.
- $\mathcal{P}$ is the set of port instances.
  - $match(p_1, p_2) : \mathcal{P} \times \mathcal{P} \rightarrow \mathbb{B}$ is a predicate which checks if two connection points are compatible i.e. if they can be bound together. This represent the enforcement of constraints using contracts.

- $pending(p) : \mathcal{P} \rightarrow \mathbb{B}$ is a predicate which checks if a connection point is pending, i.e. if it is not connected to any other connection points.
- $connectable(p) : \mathcal{P} \rightarrow \mathbb{B}$ is a predicate which checks if a (multiple) connection point is still connectable i.e. if it can still accept new bindings.

Then, let $\mathcal{K}$ be the set of architectural configurations where each element is a structure $k = \langle C_k, P_k, \beta_k \rangle$ such as:

- $C_k \subset \mathcal{C}$ is the set of component instances involved in the configuration k.
- $P_k \subset \mathcal{P}$ is the set of ports available in the configuration k.
- $\beta_k \subset P_k \times P_k$ is a reflexive relation which maps each connection point to his partners.

Finally, let $\mathcal{O}$ be the set of quality objectives where each objective is a structure $o = \langle w_o, v_o \rangle$ such as w_o is the priority (weight) associated to the objective, and v_o is the optimal value. $\mathcal{O}$ forms an n-dimensions space where the overall distance to the objectives (Δ) of a configuration k under a given environment $\mathcal{E}$ is defined as:

$$\Delta(\mathcal{E}, \mathcal{O}, k) = \sqrt{\sum_{o \in \mathcal{O}} \Big(w_o \times |v_o - eval(o, k)| \Big)^2}$$

where $eval(o, k)$ stands for the evaluation of a dimension o on a given configuration k using the parametric contracts.

As shown by Equation 1 below, the decision problem of SAS is to find the configuration k which minimizes the distance to the objectives.

$$\Delta(\mathcal{E}, \mathcal{O}, k) = \min_{k' \in \mathcal{K}} (\Delta(\mathcal{E}, \mathcal{O}, k')) \tag{1}$$

5.2 Planning Algorithm

We use a *a reactive planning algorithm* to explore gradually the space of configurations. By contrast with traditional planning which searches for a sequence of actions satisfying predefined objectives, reactive planning searches for a single action which contributes to better satisfy the objectives.

We only consider two kinds of possible actions: adding and removing a component instance from a given configuration. To ensure the exploration of all possible connection schemes, our algorithm recomputes all of them after each addition or removal. More formally, we define two sets of actions:

- $\mathcal{A}^{\oplus}$ is the set of possible additions of a new instance. Each addition action is a structure $s = \langle \oplus, k, d \rangle$ where k is the target configuration and d the component definition to instantiate. The action execution is provided by the function $\chi : \mathcal{A}^{\oplus} \rightarrow \mathcal{K}$ such as $\chi(\langle \oplus, k, d \rangle) = \langle C_k + \{new(d)\}, P_k, \beta_k \rangle$.
- $\mathcal{A}^{\ominus}$ is the set of possible removals of a component instance. Each removal action is a structure $s = \langle \ominus, k, c \rangle$ where k is the target configuration and c the component instance to remove. The removal execution is provided by the function $\chi : \mathcal{A}^{\ominus} \rightarrow \mathcal{K}$ such as $\chi(\langle \ominus, k, d \rangle) = \langle C_k - \{any(d, C_k)\}, P_k, \beta_k \rangle$.

Here $any(d, C_k)$ is a random choice of a component instance in C_k such as its definition is d. Since we recompute all the possible connection schemes after each removal, it is not necessary to try to remove each instance of the same definition.

For a given configuration $k \in \mathcal{K}$ and a set of component definitions $\mathcal{D}$ we define the set possible actions $\mathcal{A}(k, \mathcal{D})$ as:

$$\mathcal{A}(k, \mathcal{D}) = \bigcup_{d \in \mathcal{D}} (\{\langle \oplus, k, d \rangle\} \cup \{\langle \ominus, k, d \rangle\})$$

According to the previous definitions, the planning problem addressed by our algorithm is to find the sequence of n actions $(a_1, \ldots, a_n)$ which execution best fits the quality objectives, as formalized by Equation 2 below:

$$\Delta(\mathcal{E}, \mathcal{O}, k) = \min_{a \in \mathcal{A}(k,D)^n} (\Delta(\mathcal{E}, \mathcal{O}, \chi(a))) \tag{2}$$

Algorithm 1 formalizes our solution for the problem above. Given the current configuration k, the set of actions that can be undergone on it ($\mathcal{A}$), and number n of actions to perform, it explores (recursively w.r.t. the length n) all the possible sequences of actions and their resulting configurations. For each resulting configuration, it evaluates the distance to the objectives (Δ) of each possible connection scheme (see line 6) and keeps the best one.

The enumeration of possible connection schemes is performed by algorithm 2. On line 1, it first filters the set of possible connections to remove the ones which cannot be instantiated on the current configuration (only multiple ports support multiple connection). Then, it recursively selects one of the possible connection and instantiates it until no more connections are possible (See line 9). At each step, we keep only the valid configurations (see line 5) i.e. the configurations which form a connected graph of components.

5.3 Worst Case Complexity

In terms of the number of analyzed configurations, the worst situation occurs when all the ports are multiple (i.e. potentially bounded to several other ports) and when all the ports share the same type (i.e. they provide and require the same interfaces with similar contracts). In such a situation, exploring all the possible bindings between a given set of components is boiled down to exploring all the possible connected graphs between their ports [12]. The overall worse case complexity, with respect to the number of objectives, the number of definitions, and the search depth is given by the following formula:

$$\begin{aligned} O\left[search\left(\mathcal{A}, n, k\right)\right] &= \sum_{i=1}^{n} \left(|\mathcal{O}| \times \mathbf{C}_i^{2 \cdot |\mathcal{D}| + i - 1} \times O\left[allConnectionSchemes\left(k_i, \beta_i\right)\right] \right) \\ &\in O\left(n \times |\mathcal{O}| \times (2 \cdot |\mathcal{D}|)! \times 2^{(n \cdot |P_k|)^2} \right) \end{aligned}$$

Algorithm 1. $search(\mathcal{A}, n, k) \rightarrow k_\perp$

Input: $\mathcal{A}$ a set of possible actions
Input: $n \in \mathbb{N}^+$ the maximal depth of the search
Input: $k \in \mathscr{K}$ the current configuration
Output: $k_\perp \in \mathscr{K}$ the best derived configuration

$k_\perp \leftarrow k$
while $\mathcal{A} \neq \varnothing$ **do**
  $a \leftarrow any(\mathcal{A})$
  $k' \leftarrow \chi(a)$
  $\beta \leftarrow \{(p_1, p_2) \in P_{k'} \times P_{k'} \mid p_1 \neq p_2 \wedge match(p_1, p_2)\}$
6 **foreach** $k'' \in allConnectionSchemes(k', \beta)$ **do**
    if $\Delta(\mathcal{E}, \mathcal{O}, k'') < \Delta(\mathcal{E}, \mathcal{O}, k_\perp)$ **then**
      $k_\perp \leftarrow k''$
    end
  end
  if $n > 1$ **then**
    $k'' \leftarrow search(\mathcal{A} - \{\complement(a)\}, n-1, k')$
    if $\Delta(\mathcal{E}, \mathcal{O}, k'') < \Delta(\mathcal{E}, \mathcal{O}, k_\perp)$ **then**
      $k_\perp \leftarrow k''$
    end
  end
  $\mathcal{A} \leftarrow \mathcal{A} - \{a\}$
end
return $k_\perp$

Algorithm 2. $allConnectionSchemes(k, \beta) \rightarrow R$

Input: $k \in \mathscr{K}$ a configuration without any connector
Input: $\beta \in (P_k)^2$ the set of possible connectors to create
Output: $R \in \wp(K)$ the resulting set of complete configurations

1 $\beta' \leftarrow \{\{p_1, p_2\} \in \beta \mid connectable(p_1) \wedge connectable(p_2)\}$
while $\beta \neq \varnothing$ **do**
  $b \leftarrow choice(\beta')$
  $\beta_k \leftarrow \beta_k \cup \{b\}$
5 **if** *valid(k)* **then**
    $R \leftarrow R \cup \{k\}$
  end
  if $\exists \{p_1, p_2\} \in \beta'$, $connectable(p_1) \wedge connectable(p_2)$ **then**
9 $R \leftarrow R \cup allConnectionSchemes(k, \beta')$
  end
  $\beta_k \leftarrow \beta_k - \{b\}$
  $\beta' \leftarrow \beta' - \{b\}$
end

Although the complexity is exponential when the number of possible connectors explodes, our algorithm remains applicable since the depth search is not supposed to be larger than 2 (generally 1), as we illustrate using the HTTP server where all components definition share the same port types.

6 Experimental Evaluation

6.1 Prototype Implementation

We provide a first prototype of our algorithm as a Java library named "Brain" [1]. It is built upon the ECore framework: the architectural definitions, the contracts and the objectives (See Figure 3) are initially defined as a standard ECore model which can thus be reused for other purposes. Brain also provides a textual syntax for contracts and the related interpreter which enables the end-to-end evaluation of quality contracts.

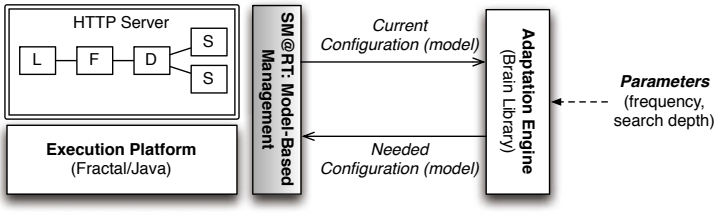


Fig. 5. Deployment and Configuration of the Adaptation Framework on the Fractal Platform

Figure 5 depicts the deployment and the configuration of the Brain library on the Fractal platform. The monitoring and modification of the real running system are done using the SM@RT tool[2] [19] which enables the generation of a synchronization engines between a running system and its model based view. During the runtime, this synchronization engine monitors the running system and instantiates the relevant instance model (the gray part of Figure 3). In addition, the synchronization engine also deploys automatically the new configuration calculated by the Brain library by computing the difference between the current and the new configuration. The HTTP server is implemented on the Fractal platform [4] as a composite component extended with a specific controller. This "Brain" controller combines the Brain library and the synchronization engines generated by SM@RT to monitor and adjust the running system at a given frequency.

[1] Available at `http://code.google.com/p/pku-brain/`
[2] Available at `http://code.google.com/p/smatrt`

6.2 Experimental Setups

We carried out two experiments aiming at ensuring the feasibility and the effectiveness of our approach respectively.

In the first experiment, we defined two adaptation policies (A and B) which differ only from the weight related to each quality dimension (our approach does not require the definition of architectural modifications). As shown by Figure 6, the first adaptation policy focuses on the response time of the server and thus balances the quality dimensions as follows: $R.T. = 9$, $M. = 1$ and $S.L. = 1$. In contrast, the policy B targets a consensus between those three dimensions and therefore defines equal weights ($R.T. = 5$, $M. = 5$ and $S.L. = 5$). Finally we monitor the architectural configurations which are produced by these two policies with respect to a load increasing and decreasing.

The second experiment compares our approach with a set of predefined architectural configurations. We implement the fixed set of configurations presented in Figure 1a (c.f. Section 2) and a selection engine selecting the best of these configurations. For the sake of the comparison, both our planning algorithm and the static selection engine evaluate the configurations using the same set of contracts. In Figure 7, our adaptive planning algorithm found different but better fitting configurations, by accepting to spend more memory in order to save response time.

6.3 Discussion

Feasibility. Figure 6 illustrates the feasibility of our approach. On the top part, using Policy A, the adaptation algorithm first increases the number of

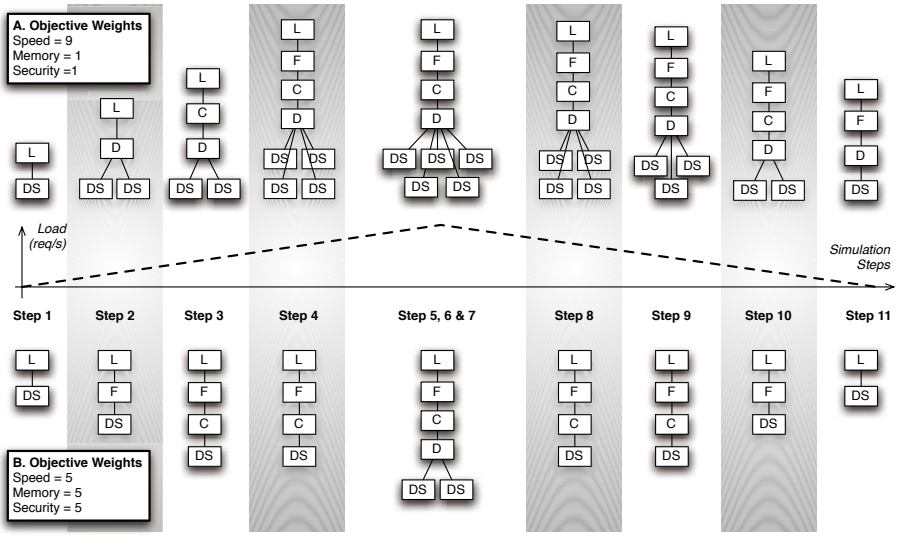


Fig. 6. Driving Self-Adaptation Using Weighted Quality Objectives

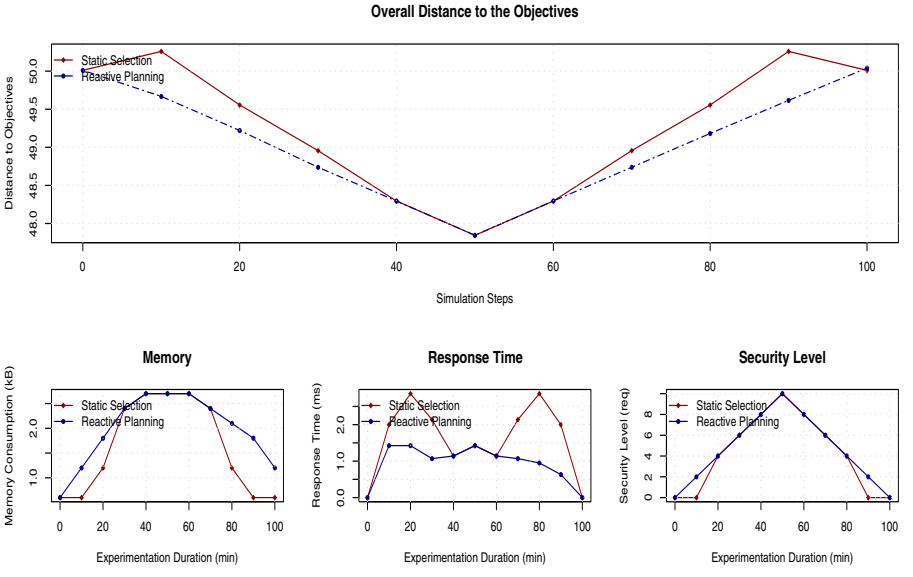


Fig. 7. Adaptation relevance: Predefined Configurations vs. Reactive Planning

data servers connected to the dispatcher component in order to balance the load increase (step 2, 4 and 5). In contrast, using Policy B, the adaptation algorithm first deploys a filter and a cache (step 2 and 3) and finally increases the number of data servers (5). The use of such objective-based policies boils down the management of adaptation to weight modifications.

Effectiveness. By avoiding the static specification of architectural modifications, our approach simplifies the specification of the desired self-adaptive behavior. In addition it also increases the effectiveness of self-adaptation. As shown in Figure 7, our planning algorithm discovers new architectural configurations which result in a better trade-off with respect to the quality objectives. By contrast with side-effects, the specification of parametric contracts must be the responsibility of a QoS expert and since their accuracy can be independently established, they therefore enforce both the separation of concerns and the accuracy of the adaptation.

Performance. The theoretical complexity of the planning algorithm limits its applicability due to the combinatorial explosion of the number of possible configurations in the most complex cases. Possible applications of this algorithm exclude large scale architectures where the number of similar component (similar port/interface type) is maximal. The HTTP server example illustrates such a situation, since each possible configuration is potentially valid. Above all, the benefits of building self-adaptation upon parametric contracts is not related to a specific exploration algorithm such as our reactive planning algorithm.

7 Related Works

As explained in Section 2, existing approaches can be divided in two categories. Solutions from the first category advocate the definition, at design time, of a set of architectural configurations which freezes the adaptation space. For instance, C2 [17], Genie [2] and Plastic [1] address architecture-centric self-adaptation in that way. Our approach avoids such static enumeration of predefined architecture configurations and can explore dynamically unforeseen configurations.

Solutions in the second category (Rainbow [11,7], MADAM [10], Sykes et al. [20,13], DiVA [9]) proposed to combine, at runtime, predefined architectural actions. Although this results in a potentially infinite set of architectural configurations, it requires the designers to roughly evaluate at design time the side-effects of such architectural actions. Our approach to self-adaptation leverage quality contracts as third-party quality-models. By contrast with side-effects evaluation, the accuracy of such quality contracts can be separately assessed as shown in [8].

Caporuscio et al. proposes PMF [5] a framework to manage the performance of software systems at run time using model-based performance evaluation. PMF goes further than C2 and Rainbow and MADAM since it generates new configuration using performances models. However, the objectives are fixed at design time while our approach supports their dynamic evolution as shown in Figure 6.

Finally, Ramirez et al. propose Plato [18] a framework which generates new configurations fitting the environmental conditions using genetic algorithms. Plato shares some similarities with our approach but requires the designer to provide a global fitness function to evaluate architectural configurations. By contrast, our approach relies on local quality contracts which can be obtained from components provider.

8 Conclusion

One of the key challenges of the development of SAS is to correctly plan how to adjust the current architectural configuration to better fit the environmental conditions and maximize the satisfaction of the requirements. Current approaches either require the enumeration, at design time, of a fix set of predefined configurations or of a set architectural modifications and their expected side-effects. Both remains difficult and error-prone activities and may lead to erroneous adaptation.

This paper addresses these two limitations by exploring at runtime the possible changes which can be undergone on the current architectural configuration of the system. Our algorithm dynamically searches for modifications of the current configuration and concurrently modifies the related quality model needed to evaluate the resulting configurations. This is achieved by combining a reactive planning algorithm with quality contracts.

Although the complexity of our algorithm prohibits its use for large scale systems (such as Multi Agent systems), we illustrated its use to dynamically adapts an HTTP server. Compared to the traditional design-time selection of

configurations, it provides a better adaptation effectiveness with regard to the quality objectives and avoids the rough estimation of architectural modifications at design time.

As future works we plan to address the automated discovery of quality models to easy the integration of third-party components and improve adaptation capabilities of legacy systems.

References

1. Batista, T., Joolia, A., Coulson, G.: Managing Dynamic Reconfiguration in Component-Based Systems. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 1–17. Springer, Heidelberg (2005)
2. Bencomo, N., Grace, P., Flores, C., Hughes, D., Blair, G.: Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems. In: ICSE: Proceedings of the 30th international conference on Software engineering, pp. 811–814. ACM Press, New York (2008)
3. Beugnard, A., Jezequel, J., Plouzeau, N., Watkins, D.: Making components contract aware. Computer 32(7), 38–45 (1999)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.: The Fractal Component Model and its Support in Java. Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems 36(11-12), 1257–1284 (2006)
5. Caporuscio, M., Di Marco, A., Inverardi, P.: Model-based System Reconfiguration for Dynamic Performance Management. Journal of Systems and Software 80(4), 455–473 (2007); Software Performance, 5th International Workshop on Software and Performance
6. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H., de Lemos, R. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
7. Cheng, S., Garlan, D., Schmerl, B.: Architecture-based Self-Adaptation in the Presence of Multiple Objectives. In: Proceedings of the Intl. Workshop on Self-Adaptation and Self-Managing Systems, pp. 2–8. ACM Press, New York (2006)
8. Firus, V., Becker, S., Happe, J.: Parametric Performance Contracts for QML-specified Software Components. In: Proceedings of the 2nd Int. Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA 2005), vol. 141, pp. 73–90 (2005)
9. Fleurey, F., Solberg, A.: A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 606–621. Springer, Heidelberg (2009)
10. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using Architecture Models for Runtime Adaptability. IEEE Software 23(2), 62–70 (2006)
11. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure. Computer 37(10), 46–54 (2004)
12. Harary, F., Palmer, E.: Graphical Enumeration. Academic Press, London (1973)

13. Heaven, W., Sykes, D., Magee, J., Kramer, J.: A Case Study in Goal-Driven Architectural Adaptation. In: Goos, G., Hartmanis, J., Leeuwen, J.V. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, p. 127. Springer, Heidelberg (2009)
14. Mei, H., Huang, G., Lan, L., Li, J.G.: A Software Architecture Centric Self-Adaptation Approach for Internetware. Science in China, Series F: Information Sciences 51(6), 722–742 (2008)
15. Morin, B., Barais, O., Nain, G., Jézéquel, J.M.: Taming Dynamically Adaptive Systems using Models and Aspects. In: ICSE: 31st Intl. Conference on Software Engineering, pp. 122–132. IEEE, Los Alamitos (May 2009)
16. OMG: OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. OMG Available Specification (ptc/03-08-02), Object Management Group (November 2007)
17. Oreizy, P., Gorlick, M., Taylor, R., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems and Their Applications 14(3), 54–62 (1999)
18. Ramirez, A.J., Knoester, D.B., Cheng, B.H., McKinley, P.K.: Applying Genetic Algorithms to Decision Making in Autonomic Computing Systems. In: Proceedings of the 6th intl. conference on Autonomic Computing (ICAC 2009), pp. 97–106. ACM, New York (2009)
19. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating Synchronization Engines between Running Systems and Their Model-Based Views. In: Bencomo, N., Blair, G., France, R. (eds.) MRT 2009: Proceedings of the Workshop on Models at Runtime 2009. Springer, Heidelberg (2009) (to be published)
20. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From goals to components: a combined approach to self-management. In: SEAMS 2008: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, pp. 1–8. ACM, New York (2008)