From Local Impact Functions to Global Adaptation of Service Compositions*

Liliana Rosa, Luís Rodrigues[1], Antónia Lopes[2], Matti Hiltunen[3], and Richard Schlichting[3]

[1] INESC-ID/IST
[2] Faculty of Sciences, University of Lisbon
[3] AT&T Labs Research

Abstract. The problem of self-optimization and adaptation in the context of customizable systems is becoming increasingly important with the emergence of complex software systems and unpredictable execution environments. Here, a general framework for automatically deciding on when and how to adapt a system whenever it deviates from the desired behavior is presented. In this framework, the adaptation targets of the system are described in terms of a high-level policy that establishes goals for a set of performance indicators. The decision process is based on information provided independently for each service that describes the available adaptations, their impact on performance indicators, and any limitations or requirements. The technique consists of both offline and online phases. Offline, rules are generated specifying service adaptations that may help to achieve the specified goals when a given change in the execution context occurs. Online, the corresponding rule is evaluated when a change occurs to choose which adaptations to perform. Experimental results using a prototype framework in the context of a web-based application demonstrate the effectiveness of this approach.

1 Introduction

Today's complex software systems and services (e.g., Apache, Tomcat, MySQL, virtual machines) offer different facilities for customizing their behavior, including loadable modules and numerous configuration options. Such facilities can be used to adapt the behavior of these services even during execution in response to changes in the operational envelope. These changes might be the result of, for instance, changes in system workload or in the available resources. While dynamic resource allocation (e.g., [10]) can be used to respond to such changes, adaptations that affect the service behavior itself can also be a powerful tool.

This paper addresses the problem of how to select appropriate service adaptations when the system behavior deviates from that which is considered optimal, for example, to provide a certain quality of service. This problem is extremely challenging since the best adaptation may depend not only on the particular configuration of the system—that is, the set of services and how these services are configured—but also on information that can be extremely dynamic and unpredictable, such as the pattern of service invocations. In this paper, we consider software systems built from one or more adaptable services. We assume that the behavior of such service compositions can be described using a set of *key performance indicators* (KPIs) that need to be maintained or optimized, and that the system behavior can be controlled by applying one or more adaptations.

* Parts of this work were published in the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems, 2009

There are several approaches to deciding on how to adapt a service composition. One approach is to consider the composition as a black box and use control theory and/or learning techniques [7,3,18] to derive adaptation policies. Unfortunately, this approach is expensive and the resulting policy is only valid for the specific configuration and workloads used during the learning process. Thus, if the system configuration changes, the entire process has to be repeated. The same applies for changes in the workload, where a small change can have a large impact on the set of adaptations that need to be selected. Another approach relies on the system architect or system administrator specifying a low-level adaptation policy for the system's service composition manually based on her own knowledge on the system operation [4]. Typically, these policies consist of declarative Event-Condition-Action (ECA) rules specifying how the system must adapt in the presence of specific events and conditions. Unfortunately, as the complexity of the system composition increases, this task becomes harder and more error-prone. Indeed, it often becomes impractical or even impossible for the system architect to manage all the possible interactions and side effects among the adaptations available for all services. The Cholla system [5] also addresses a similar problem, proposing a solution based on fuzzy control rules. While rules can often be developed independently, additional coordination rules specific to the chosen set of rules are often required. Also, this work does not provide an explicit mapping from KPI-based goals to adaptation rules. Note that our work is orthogonal to research on coordinating distributed adaptations [17,6]. In fact, such techniques could be combined with our approach in case distributed coordination is required.

While a complex system of this type is hard to understand, the developer of each individual component or module usually has a clear understanding of the ways the component can be adapted and the impact of each adaptation on the performance of the component in isolation. For instance, the designer of a graphical component G may implement two operational modes: one that produces high quality images and one that produces low quality images. The designer, knowing the implementation details, is fully aware of the tradeoffs involved, specifically that the low quality mode produces an image with lower image resolution, but consumes less memory and less processor time than the high quality counterpart. The challenge, of course, is to mesh this information with that from other components to devise the best solution.

The goal of this work, then, is to make services adaptive by leveraging information from service developers about the characteristics of each individual component considered independently of where it will be used. To realize this goal, we propose a technique that uses this information to select the best adaptations for a service when its execution deviates from the desired behavior. The selection process is driven by a high-level policy that specifies the desired behavior—and, hence, the goals the adaptations should strive to achieve—and relies on information provided for each component describing possible adaptations, their impacts on KPIs, and any limitations or requirements. The proposed technique consists of both offline and online phases; in the offline phase, a set of service adaptations that can help achieve the specified goals is created, while in the online phase, adaptations are selected from this set in response to a change in the execution context using the current system status and workload as input. For example, in the above example, if the graphical service G is heavily utilized (high workload), the change from high quality to low quality mode may yield significant memory, processor, and/or bandwidth savings,

while if G is in fact lightly utilized, the same adaptations may have negligible impact. Thus, the adaptations selected by our technique take into account not only the impact of each adaptation, but also the contribution of each service to the performance of the entire composition.

The rest of the paper is organized as follows. Section 2 describes the way in which the impact of possible adaptations on system performance is specified, and also how high level goals can be captured in a policy. Section 3 then explains how ECA rules are derived offline from the policy, while Section 4 describes how these rules are evaluated online. The framework is illustrated and evaluated in Section 5 using a web based application built from the composition of several services that handle the process of replying to a HTTP request. Experimental results show that selected adaptations are effective for different compositions of the same services and different workloads. Section 6 concludes the paper.

2 Adaptable Services and Adaptation Goals

The proposed approach is based on adaptation goals defined in terms of a set of KPIs and requires information regarding adaptations, their impacts, and constraints for each service component. As mentioned above, KPIs are metrics that capture system performance, like CPU or memory use, among others [7].

The two key assumptions behind the approach are: (i) the value of each KPI for a service composition C is $\sum_{s \in C}\ s.KPI$, where $s.KPI$ is the "contribution" of service s to that performance indicator, and (ii) it is possible to express the (localized) impact of each adaptation of a service s in each of these KPI_s. For instance, the CPU used by a service composition is an example of a KPI that can be defined as the sum of the CPU used by each service in the composition. An adaptation of a service s that, if applied, changes the CPU used by s would have to give a function that estimates the new value of $s.cpu\_u$.

A KPI definition includes a name, the type of the expected value, and the acceptable *error_margin* in any evaluation of the KPI, as illustrated below.

```
KPI cpu_u: double Error 0.1
```

This means that two values of the KPI within *error_margin* of each other are considered indistinguishable from the point of view of goal evaluation.

2.1 Specification of Service Adaptations

Our approach relies on local information regarding each adaptation to assess how these adaptations can be used to change system behavior. These adaptations involve either changing service parameters or exchanging service implementations. The impacts of each adaptation on the system behavior is specified against a set of KPIs and a service model.

Service models describe the service components available for use in compositions and, for each component, the configurable parameters and available implementations. We consider service models as defined in our previous work [13], i.e., defined in terms of a type hierarchy reflecting the *is a* relationship, taking into account the functionality provided by the services. Service types can be concrete, designating a specific service for which an implementation is available, or abstract, representing simply the characteristics of a

group of other service types. Below is the model for a concrete service that provides static webpages with a configurable parameter *ImgQlt* that controls image quality (resolution):

```
Service StaticContent
  Parameters
    ImgQlt:{low,regular}
```

The service model is needed to support the specification of adaptations, which must include: a) the concerned service or service component, b) the adaptation action(s) to be performed, c) constraints such as the required service state or other adaptations that have to be performed simultaneously, and d) the impact of the adaptation on each KPI. If a KPI is omitted from the impacts, it means that the KPI is not affected. The following example shows the specification of an adaptation of the *StaticContent* service:

```
Adaptation ToLowStatic
 Service:
  StaticContent
 Actions:
  setParameter(ImgQlt,low)
 Requires:
  ImgQlt == regular
 Impacts:
  StaticContent.cpu_u /= 1.21   //decreases
  StaticContent.resolution = 1   //changes to low
```

This adaptation changes the image quality from regular to low, with the impact being to decrease the CPU used by the service and the image resolution. The effect of the adaptation on the KPIs is described by *impact functions* under the label *Impacts*, which provides an estimate for the new value of s.KPI if the adaptation is performed given its current value. Impacts can also be expressed in terms of current values of the configurable parameters, the current version of a service, or the presence or absence of a given service component. Even when not explicitly stated, any adaptation is only applicable if the target service or service component is present in the current service composition. We assume that meta-information about the deployed and executing service compositions, as well as the value of their parameters, is available at runtime. The problem of deriving the impact functions for each adaptation is outside the scope of this paper, but existing approaches can be applied [7].

Additional adaptation constraints can be specified by listing which adaptations of different services cannot be applied at the same time. By default, adaptations of the same service that have impact on the same KPI are assumed to conflict, but it is possible to specify a single adaptation that considers several actions provided the joint impact of these actions over the KPIs can be defined. These conflicts are simply described as pairs of adaptations:

```
Conflict conflict_name Adaptations (serviceA.adapt1,serviceB.adapt2)
```

The complete specification therefore consists of the service model, the adaptations, and the conflicts.

2.2 Policies

Adaptation goals are specified in terms of a policy that describes the desired values for a set of KPIs. A policy describes: a) the KPIs that are relevant to the policy, b) the goals to be met by the system, and c) the values of configuration parameters related to the

runtime operation of the adaptation engine. Besides identifying the relevant KPIs, the policy can further use them to specify *composite KPIs*, denoted by CKPIs. CKPIs are identified by a *ckpi_name* and their specification consists of a function of several KPIs, and an *error_margin*:

```
CKPI ckpi_name = f(kpi1, kpi2, ...) Error error_margin
```

This function also makes it possible to derive the impact of each adaptation in the CKPI from the impacts of the adaptations in *kpi1*, *kpi2*, etc. As an example, the definition of the CKPI *gdev* below measures the weighted deviation from target CPU and memory utilization values:

```
CKPI gdev = 0.5*|cpu_u-0.6| + 0.5*|mem_u-0.4| Error 0.1
```

Henceforth, we use KPI to refer to either a basic KPI or a CKPI.

A policy can have one or more goals that are ranked to prioritize goals in situations where it is not possible to fulfill all goals. The rank is implicit in the order goals are listed in the policy, where the first goal has the highest rank. Additionally, there are two types of goals: *exact* and *approximation* goals. Exact goals separate the values of a performance indicator in two disjoint sets: *acceptable* and *not acceptable*. We consider the following types of exact goals:

```
Goal goal_name: kpi_name Above threshold_down MinimumGain gvalue
Goal goal_name: kpi_name Below threshold_up MinimumGain gvalue
Goal goal_name: kpi_name Between thr_down thr_up MinimumGain gvalue
```

An *Above* goal states that the value of the KPI should be kept above the stated threshold, a *Below* goal that the value should be kept below the threshold, and a *Between* goal that the value should be kept within lower and upper thresholds. In all three, the *MinimumGain* specifies the minimum change necessary to perform the adaptation; that is, if the estimated change in the KPI value is below *gvalue*, the adaptation is not worth performing. The *gvalue* should be greater than the *error_margin* specified for the target KPI.

In contrast, instead of simply classifying the values of a KPI as good or bad, approximation goals specify a total order between these values, that is, for any two values, it specifies which one is better. We consider the following types of approximation goals:

```
Goal goal_name: kpi_name Close target MinimumGain gvalue Every interval
Goal goal_name: Minimize kpi_name  MinimumGain gvalue Every interval
Goal goal_name: Maximize kpi_name MinimumGain gvalue Every interval
```

A *Close* goal states that the KPI value should be kept as close as possible to the *target* value, a *Minimize* goal states that the KPI value should be as small as possible, and a *Maximize* goal states that it should be as large as possible. As with exact goals, it is also possible to specify the expected minimum gain required in order to perform an adaptation. Furthermore, associated with each approximation goal, is a time *interval* that specifies how often the system should try to find an adaptation aiming for a better value for the KPI. Note that while adaptation towards an exact goal is only triggered when the current KPI value is unacceptable, an approximation goal opens the possibility of continuously attempting to improve the system behavior aiming for a better value.

Finally, a policy may also define the values of configuration parameters that control the runtime operation. For example, *mon_interval*, which controls how often the KPIs' current values are read, can be configured.

3 Rule Generation

Adaptation rules are generated offline from the policy using the specifications of the available adaptations. Each rule consists of an event and one or more alternative sets of adaptations A_i that may help achieve the specified goals when a change in the execution context occurs. These rules are evaluated at runtime to determine which set of adaptations should be executed given the current system state. The rules have the following format:

```
When event
  Select {A1, A2, ...}
```

The *When* clause defines the *event* that triggers the rule. This may be caused by a change signaled by a *sporadic event*—when some KPI exceeds a threshold, for example—or by the passage of time signaled by a *periodic event*. The *Select* clause lists all relevant sets of adaptations for dealing with that particular event. For instance, if a goal states that some KPI must be maintained above a given threshold, only those adaptations that affect this KPI and increase it are relevant. The sets A_i represent the viable combinations of the relevant adaptations, reflecting the fact that the combination of adaptations is subject to constraints imposed by conflicts or application conditions. Naturally, given that rules are generated offline, it is only possible to take into account the aspects that do not require runtime state information.

Extracting the rule sets offline in this way has two main advantages. First, it often simplifies the online phase and improves its performance as a result. Second, by capturing the online behavior in a human-readable form, the system operators can better understand the behavior of the system. This is especially valuable in cases where the observed behavior is counter-intuitive to the (expected) impact of the high-level policy.

3.1 Event Extraction

Event extraction is the first task of rule generation. This step relies on the assumption that the component that monitors the system performance, the *context monitor*, is able to generate different types of events divided into sporadic and periodic events. The *kpiAbove(kpi,x)* and *kpiBelow(kpi,x)* events signal when the value of *kpi* is detected to be above or below the value x, and needs to be decreased or increased, respectively. Similarly, *kpiIncrease(kpi,θ,condition)* and *kpiDecrease(kpi,θ,condition)* are periodic events generated every period θ, if the *condition* over the current value of *kpi* holds, and signals that the value of *kpi* needs to be increased or decreased, respectively.

As noted above, the high-level policy has two distinct types of goals. When an exact goal is violated, system adaptation should be triggered. For approximation goals, adaptations are triggered periodically, thus they require the use of periodic events. Table 1 summarizes the types of events generated for each type of goal and how these events are triggered.

The specific events that are extracted from a high-level policy depend on the different values used in the goals and KPI declarations. Here, we explain how the values in the event attributes are defined for each type of goal. Figure 1 provides examples of events for some goal types. For an *Above* goal, an event of type *kpiBelow* needs to be triggered when the value of the KPI falls below the specified threshold by a margin greater than

Type	Goal	Event 1	Event 2	Trigger
Exact	Above	kpiBelow(kpi, x)	-	threshold exceeded
Exact	Below	kpiAbove(kpi, y)	-	threshold exceeded
Exact	Between	kpiBelow(kpi, x)	kpiAbove(kpi, y)	threshold exceeded
Approx	Close	kpiIncrease(kpi, θ, cond)	kpiDecrease(kpi, θ, cond)	periodic
Approx	Maximize	kpiIncrease(kpi, θ, cond)	-	periodic
Approx	Minimize	kpiDecrease(kpi, θ, cond)	-	periodic

Table 1. Events generated for each type of goal

the KPI *error_margin*. Similarly, for a *Below* goal, an event of type *kpiAbove* needs to be triggered when the value of the KPI exceeds the specified threshold. Since *Between* goals are a combination of the *Above* and *Below* goals, both previous events are needed. For the *Minimize/Maximize* goals, a periodic event of type *kpiDecrease*/*kpiIncrease*, respectively, needs to be triggered with the period specified in the goal. Finally, for the *Close* goals, two distinct events are extracted, one for when the KPI needs to be decreased and the other for when the KPI needs to be increased, as illustrated in Figure 1. For each extracted event or periodic event, a rule is created with the *When* clause stating the event as the trigger for the rule evaluation.

```
Goal cpu_reserve: cpu_u Below 0.6 MinimumGain 0.2
Event kpiAbove(cpu_u,0.7)     // 0.6+0.1

Goal target_cpu: cpu_u Between 0.4 0.6 MinimumGain 0.2
Event kpiBelow (cpu_u,0.3)     // 0.4-0.1
Event kpiAbove (cpu_u,0.7)     // 0.6+0.1

Goal minimize_deviation: Minimize gdev MinimumGain 0.2 Every 10
Event kpiDecrease (gdev,10, true)

Goal target_cpu: cpu_u Close 0.5 MinimumGain 0.2 Every 20
Event kpiDecrease (cpu_u,20, ">0.6")     // 0.5+0.1
Event kpiIncrease (cpu_u,20, "<0.4")     // 0.5-0.1
```

Fig. 1. Example events extracted from goals

3.2 Selecting Service Adaptations

The second task of offline rule generation is to identify the sets of adaptations that need to be included in each rule. The purpose of a given rule is either to increase or decrease the value of a given KPI. Thus, the impact functions declared in the adaptation descriptions need to be analyzed to check if the adaptation increases or decreases the value of the relevant KPI. Consider, for instance, a rule of the form **When** *kpiBelow(cpu_u,0.3)* **Select** ... where the aim is to increase the value of the *cpu_u*, and we have an adaptation X that applies to service S with the impact function *S.cpu_u *= 1.8*. To assess if adaptation X should be used in the rule, one simply checks whether the function $f(kpi) - kpi$ has a positive derivative. In this example, since the derivative of $1.8 \cdot x - x$ is 0.8, the adaptation X helps to increase the CPU utilization. Hence, this adaptation will be used in the construction of the sets of adaptations to be evaluated when the event *kpiBelow(cpu_u,0.3)* is triggered.

Once all adaptations that contribute to achieve the goal associated with the trigger event are known, rule generation proceeds with the calculation of the set of viable combinations, i.e., the sets of adaptations that can be executed at the same time. When there are adaptations that apply to the same service or conflicts between adaptations in the

main set, it is necessary to break the main set into several sets, where all adaptations in the same set are compatible, and have all their requirements satisfied. To help the system operator understand the behavior of the system, an intentional representation of the set of viable combinations is used. As illustrated in the example below (in human readable form), all adaptations that contribute to achieve the goal associated with the trigger event are listed, together with the pairs of conflicting adaptations and pairs of adaptations that need to be executed together.

```
When event
  Adaptations: S1.A, S1.B, S2.X, S2.Y, S3.Z
   Conflicts: (S1.A, S2.X)  Dependencies: (S2.Y, S3.Z)
```

4 Rule Evaluation

The rules that were generated offline are evaluated at runtime. The evaluation of a rule **When** e **Select** $\{A_1, ..., A_n\}$ occurs whenever event e is triggered, and consists of selecting a combination of adaptations from the subsets of A_i, for $i = 1, ..., n$. The selected set includes the adaptations to be applied to the system and, hence, the aim of the selection process is to find the combination that best satisfies the goals defined in the adaptation policy.

The process of rule generation ensures that each A_i includes only adaptations that can be executed at the same time. However, these sets may include adaptations that cannot be applied in the current configuration of the system. This happens if the target service is not part of the current composition or if the constraints expressed in the *Requires* section of the adaptation do not hold. Hence, the evaluation of the rule starts by removing non-applicable adaptations from every A_i. Then, rule evaluation proceeds by searching for combinations that best match the goals expressed in the adaptation policy, taking into account the current system state.

As mentioned above, the search space $\mathcal{S}$ is the set of all subsets of all A_i. Intuitively, the search involves analyzing the estimated effects of the different combinations on the KPIs addressed by the goals of the adaptation policy and deducing which ones best fit these goals. More precisely, recall that adaptation policies define a set of ranked goals $\{G_1, ..., G_n\}$, where G_1 is the goal with the highest rank. The comparison between different combinations of adaptations relies on their evaluation against these goals, starting from G_1. The evaluation of a combination against a goal G_i depends on the type of goal (exact or approximation), with the impact functions of the involved adaptations being used to estimate the effect on the KPI_i associated with the goal.

Let KPI_i^C be the estimated impact of a combination of adaptations C on KPI_i. (Note that if C is the empty set, then KPI_i^C is just the current value of KPI_i.) *C best matches* $\{G_1, ..., G_i\}$ only if the following conditions hold:

1. if $i > 1$, C best matches $\{G_1, ..., G_{i-1}\}$
2. if G_i is an exact goal:
   - if G_i is currently satisfied: KPI_i^C also satisfies G_i;
   - if G_i is currently violated: there is a gain w.r.t. the current value of KPI_i and it exceeds the specified minimum gain;
3. if G_i is an approximation goal:

– $|KPI_i^C - KPI_i^{C^*}| < error\_margin^{kpi}$ and, if C is not the empty set, the gain w.r.t. the current value of KPI_i exceeds the specified minimum gain;

where C^* is, among the combinations in $\mathcal{S}$ that best match $\{G_1, ..., G_{i-1}\}$, the one that puts the KPI_i closer to the target specified in G_i.

For instance, consider the exact goal *cpu_reserve* and assume that the current *cpu_u* value is 0.75 (the goal is currently violated). A combination with a single adaptation whose estimated effect brings *cpu_u* to 0.9 is excluded because it violates the goal. A combination with a single adaptation whose estimated effect brings *cpu_u* to 0.65 is also excluded because it does not meet the specified minimum gain. Two combinations with a single adaptation whose estimated effects bring *cpu_u* to 0.50 and 0.55, respectively, are both candidates for being selected. Thus, the next ranked goal would be used to tie-break among them.

5 Evaluation

To evaluate the proposed approach, we conducted a study to analyze how successfully the rules generated offline drive the runtime adaptation, given changes that carry the system outside the desirable or acceptable behavior defined in the goals. To do so, we implemented a prototype of the framework in Java^{TM}, and developed an experiment that illustrates the use of the proposed approach for the autonomic management of web-based applications.

5.1 Services, Adaptations and Policy

The case study consists of a web site that offers both secure and non-secure content; part of this content is static, and another part is dynamically generated. The content is produced by several services that are adaptable, which allows the quality of any provided content to be controlled.

Three services provide content: *StaticContent*, *DynContent*, and *SecureContent*. The first, *StaticContent*, provides the static content web pages that are not secure. The service can operate on *regular* or *low* mode; in *low* mode it offers lower image quality as well as de-animated GIFs. Thus, it is possible to have two adaptations of the *StaticContent* service: from regular to low quality and vice-versa. The first adaptation reduces resource consumption, while the second increases the quality of service. The second service, *DynContent*, generates user-tailored non-secure webpages. The service also features regular and low versions similar to *StaticContent*, which are implemented by adding, removing, or changing HTML tags using the approach described in [11]. Furthermore, two implementations of the *DynContent* service can be used: a *heavyweight* implementation that determines new recommendations and advertisements for a user on the fly, and a *lightweight* implementation that uses cached recommendations and advertisements [15]. Finally, the third service, *SecureContent*, handles webpages that deal with account login or sensitive data, such as order payment information; it also generates regular and low versions in terms of image quality and animated GIFs. The service specification is presented below. Space limitations prevent us from describing the entire set of services adaptations (which is presented in [14]), that includes the adaptation *ToLowStatic* introduced in Section 2.

```
Abstract Service DynContent
  Parameters
    ImgGIFFilter:{on, off}

Service LWDynContent
  subtype DynContent

Service HWDynContent
  subtype DynContent

Service StaticContent
  Parameters
    ImgQlt:{low, regular}

Service SecureContent
  Parameters
    Mode:{low, regular}
```

In our case study we used three KPIs. The monitored system resource is the consumed CPU (*cpu_u*); recent research has shown this to be the main bottleneck for this type of application [16]. The quality of service provided to the user is captured by two synthetic metrics, the *resolution* of the images returned to the user (*resolution*), and the *accuracy* of the recommendations included in the web pages (*harvest*). We have also considered a CKPI *qos*, defined as the composition of both the resolution and the harvest of the pages returned to the user as follows:

```
KPI cpu_u: double Error 0.1
KPI resolution: integer Error 0
KPI harvest: integer  Error 0
CKPI qos = (2*resolution + harvest) Error 0
```

Using these KPIs, we defined the simple policy presented below that aims to provide users the best quality of service possible without exceeding a pre-defined threshold of CPU utilization. This policy is broadly similar to policies that have been used in related work, including policies to achieve optimal resource use for webservers [7,1], intermediary adaptation systems [11,9,8], and web server and user experience improvement [16]. The policy describes two goals. The first limits the value *cpu_u* to a pre-defined threshold of 0.6. This limitation is imposed to maintain an available CPU margin to deal with workload peaks. The focus of the second goal is to maximize the quality of the content provided to the user, ensuring that when resources are available, the best image quality, animated GIFs, and up-to-date recommendations are returned. The policy additionally specifies that the monitoring interval is 1 second.

```
Goal limit_cpu: cpu_u Below 0.6 MinimumGain 0.15
Goal max_qos: Maximize qos MinimumGain 1 Every 60
Configuration mon_interval 1
```

From this policy, event extraction and rule generation was performed offline. The extracted events are presented in Table 2. The rules, in their human readable form, are as follows:

```
When kpiAbove(cpu_u,0.7)
 Select {ToLowStatic, ActivateImgGIFFilter, ToLW+FilterOn, ToLW+FilterOff, ToLW+MaintainOn,
     ToLW+MaintainOff, ToLowModeSecure}

When kpiIncrease(qos,60,true)
 Select {ToRegularStatic, DeActivateImgGIFFilter, ToHW+FilterOn, ToHW+FilterOff, ToHW+
     MaintainOn, ToHW+MaintainOff, ToRegularModeSecure}
```

5.2 Experimental Setup

The prototype implementation consists of the overall framework, several static webpages (*StaticContent*), and the web site's dynamic generation components (*DynamicContent* and *SecureContent*). Each component is an adaptable *CGI* that offers two distinct behaviors that trade off the quality of service provided to the user with the resources used,

primarily CPU usage. Apache web server [2] running on Linux is used to execute requests. To monitor the execution context, i.e., CPU usage, a simple monitoring tool was implemented in Python and integrated with the framework prototype. The monitoring tool can be configured in terms of the interval between reads and the stabilization time after adapting.

To analyze how the policy drives changes in the quality of service when the resource consumption varies, we generated several workloads to force different adaptations. In periods when the load is high, then, the system will adapt one or more components to provide a lower quality of service, to keep CPU usage below the given threshold. After adapting, the KPIs readings are ignored until the end of a stabilization period.

The experimental testbed consists in three machines. One machine runs the Apache Web Server as well as the services, while the other two machines run a workload generator. The three machines are connected by a 100 Mbps Ethernet. The server machine is a 8 x 3.22 GHz processor with 8 GB RAM running Linux (kernel v2.6.24-21). We used Apache HTTP Server v2.2.8 configured with 150 *MaxClients* and a *KeepAliveTimeout* of 15 seconds, with CGI and SSL modules enabled. The client machines run Pylot [12], an open source tool for testing performance and scalability of web services based on an XML file that describes the workload. We modified the original Pylot tool to run several workloads in sequence, each for a period of time, thus, varying the workload.

The services in our case study are implemented as follows. First, the *StaticContent* service is implemented using several HMTL pages containing text and images with different sizes (from 5 to 500 KB), each one with a low and a regular version. Second, the *DynContent* service is implemented as a CGI that generates the HTML pages on the fly according to parameters passed in the HTTP request. The generated pages include images and text, again with two different implementations of the service. Finally, the *SecureContent* service consists of dynamically generated pages requested over HTTPS, with text and media.

In terms of adaptations, the change between different versions in the *StaticContent* service is achieved using file system links. The HTTP request will request a HTML file. If the low version is in use, the link will point to the low version. When the adaptation sets *ImgQlt* to regular, the link is redirected to the regular version. The same approach is used when the other remaining parameters are set, and, also, to exchange implementations of *DynContent* service.

The three different workloads used are determined by the type and frequency of requests, for each of the three services described above. The *light* workload allows all services to be offered with maximum *qos*. The *medium* workload requires the *qos* to be lowered in order to respect the *cpu_u* threshold. Finally, the *heavy* workload requires the system to operate with an even lower *qos*.

Type	**Goal**	**Event 1**	**Trigger**
Exact	*limit_cpu*	kpiAbove(*cpu_u*, 0.6 + 0.1)	*cpu_u*> 0.7
Approx	*max_qos*	kpiIncrease(*qos*, 60, true)	periodic

Table 2. Events generated for the case study

As defined in the policy, the consumed CPU is monitored every second. Due to the variability of the workload, a change is only signaled if it is observed for at least 10 out of 15 consecutive samples.

5.3 Results

Services were initially deployed with a configuration that yields the best quality of service: static web pages and secure content are served with regular quality, while dynamic content is deployed using the heavyweight version and with the content filter off. Then, we subject the system to a varying workload.

The workload consists of a collection of urls that are requested by each client. The order of this list is randomized for each client to ensure that the sequences will differ. Each client waits for a response before sending another request; this interval is 10 milliseconds. Our experiment used 100 clients that run concurrently. The client rampup takes 25 seconds, therefore, 4 clients are launched every second. The clients start sending requests as soon as they start. The workload is changed between three different levels: light (LW), medium (MW), and heavy workload (HW) characterized as follows:

LW: 60% of requests for static content, 30% for dynamic content, and 10% for secure content. This workload is not enough to violate any of the KPI constraints. The experiment starts and finishes with this workload.

MW: 35% of requests for static content, 55% for dynamic content, and 10% for secure content. This workload violates the CPU threshold defined by the first goal, thus, triggering an adaptation to decrease CPU use.

HW: 20% of requests for static content, 30% for dynamic content, and 50% for secure content. With this workload it is impossible to satisfy the CPU threshold without a substantial decrease in CPU use, forcing an adaptation with greater impact.

Figures 2 and 3 depict the described scenario under varying workloads. Each dotted vertical line marks a change in the workload. We begin with LW, changing to MW around time 134, then to HW at around time 405, and finally switch back to LW around time 740. The impact of the workload change on the CPU usage may be delayed, depending

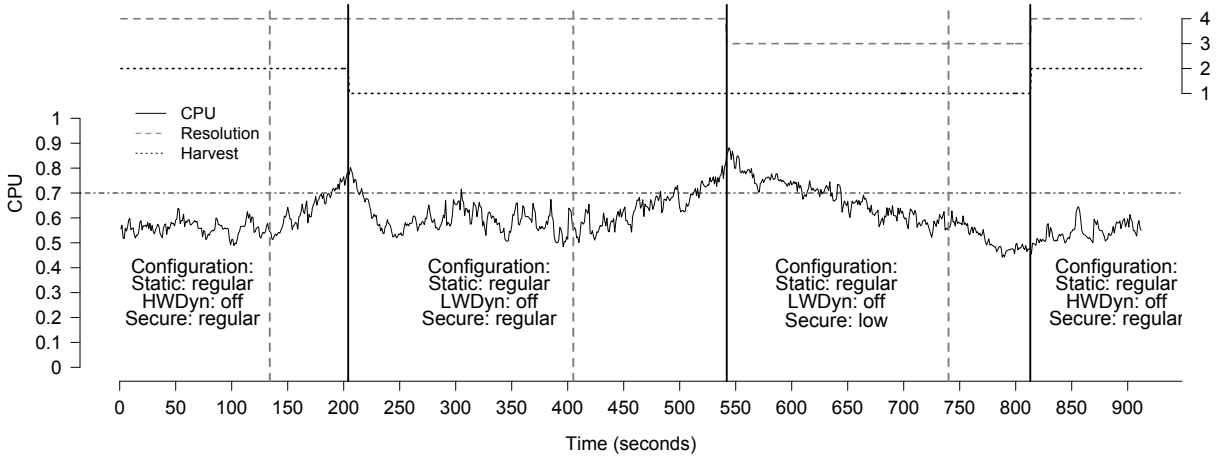

Fig. 2. Evolution of the KPIs of the system in the described scenario.

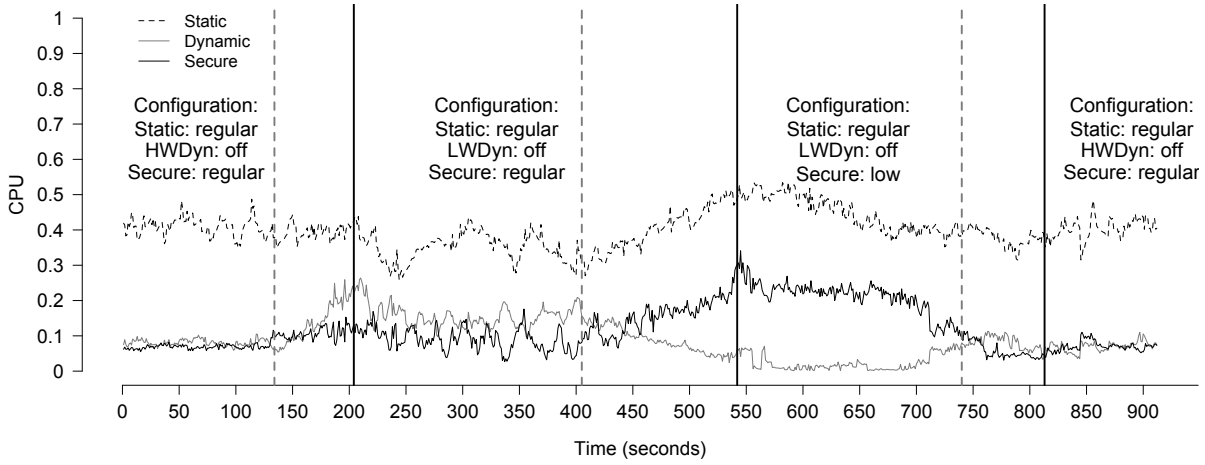

Fig. 3. CPU consumed by each service

of the request distribution. Between each workload change, there's the current service composition and configuration. The solid vertical lines mark when adaptations take place. After each adaptation, the monitoring device ignores the readings during a stabilization period to allow ongoing requests to be processed until they are completed by the original components.

Figure 2 depicts the evolution of the KPI values during system execution. After changing the workload from LW to MW, the system detects that the CPU use is above the CPU limit plus the error margin (0.7), thus, it selects an adaptation that decreases the harvest KPI. Later, the workload is switched to HW, forcing an adaptation that lowers the resolution KPI to decrease the CPU use; note that this adaptation requires longer to take effect. Finally, the workload is changed back to LW and another adaptation takes place, increasing both the resolution and harvest KPIs. This increases the quality of service to a maximum, as in the beginning.

Figure 3 shows the contribution of each service to the global CPU utilization for the same scenario, allowing us to assess the impact of each adaptation. As a result of the first workload change, the system adapts around time 204 by changing the dynamic content implementation. This adaptation is selected because it lowers the CPU use to below the limit and also offers the highest *qos* CKPI value. This follows since it only decreases harvest, which has a lower weight in the *qos* CKPI. When the second workload change takes place, the system adapts around time 542 by changing the secure content from regular mode to low. This adaptation is selected because the CPU usage by secure content is clearly higher than the others, giving this adaptation a greater impact that the total of the others with a higher *qos* value. Finally, when the workload goes back to LW, the system adapts to its initial configuration with highest *qos*; this occurs at around 813 seconds and is triggered by a periodic event. These results demonstrate that the system adapts as expected given the characteristics of the workload and the performance of the deployed components, always offering the highest possible *qos*.

6 Conclusions and Future Work

This paper proposes a novel approach to managing adaptive behavior in customizable software systems. This approach uses information provided by each service designer about the impact of possible adaptations on the system KPIs to perform the automatic offline generation of a set of rules corresponding to a policy that describes the intended system behavior for those KPIs. These rules are then evaluated online to implement the adaptive behavior. Experimental results show that this approach is feasible and has a number of advantages. For example, each service configuration can be measured independently a single time to quantify the impact of adaptation, and still work for different configurations or workloads. The approach is also able to balance the trade-offs due to different goals when choosing an adaptation. Finally, as shown by experimental results, the approach considers not only how far the current state is from the optimal state—and, as a result, how large the impact has to be—but also uses the load of each service to realistically estimate the impact of an adaptation.

As future work, we plan to broaden application of the approach. Currently, for instance, we do not explicitly consider dependencies among services, so that when such dependencies exist, each adaptation must be applied separately. We plan to extend our model to consider such constraints.

Funding

This work was funded by REDICO project (PTDC/EIA/71752/2006).

References

1. Tarek Abdelzaher and Nina Bhatti. Web content adaptation to improve server overload behavior. In *WWW8 / Computer Networks*, pages 1563–1577, 1999.
2. Apache. See httpd.apache.org.
3. K.J. Astrom. Adaptive feedback control. *Proceedings of the IEEE*, 75(2):185–217, Feb. 1987.
4. Raphael M. Bahati, Michael A. Bauer, and Elvis M. Vieira. Policy-driven autonomic management of multi-component systems. In *CASCON '07*, pages 137–151, NY, USA, 2007. ACM.
5. P. Bridges, M. Hiltunen, and R. Schlichting. Cholla: A framework for composing and coordinating system software adaptations. *IEEE Transactions on Computers*, (to appear) 2009.
6. W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *ICDCS' 01*, pages 635–643, Apr 2001.
7. Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing web server performance with autotune agents. *IBM Syst. J.*, 42(1):136–149, 2003.
8. R. Grieco, D. Malandrino, F. Mazzoni, and D. Riboni. Context-aware provision of advanced internet services. In *PerCom Workshops 2006*, pages 4 pp.–603, March 2006.
9. Gennaro Iaccarino, Delfina Malandrino, and Vittorio Scarano. Personalizable edge services for web accessibility. In *W4A '06*, pages 23–32, NY, USA, 2006. ACM.
10. G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu. Generating adaptation policies for multi-tier applications in consolidated server environments. In *ICAC '08*, pages 23–32, June 2008.
11. Francesca Mazzoni. *Efficient provisioning and adaptation of Web-based services.* PhD in computer science, Università di Modena e Reggio Emilia, 2006.
12. Pylot. See www.pylot.org.
13. Liliana Rosa, Antónia Lopes, and Luís Rodrigues. Modelling adaptive services for distributed systems. In *SAC '08*, pages 2174–2180, NY, USA, 2008. ACM.
14. Liliana Rosa, Luís Rodrigues, Antónia Lopes, Matti Hiltunen, and Richard Schlichting. From local impact functions to global adaptation of service compositions. Technical report, 2009.

15. Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. *Internet Computing, IEEE*, 11(1):60–66, Jan.-Feb. 2007.
16. Steve Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, 2008.
17. Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. *Softw. Pract. Exper.*, 28(9):963–979, 1998.
18. Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *ICDCS '02*, page 301, Washington, DC, USA, 2002. IEEE Computer Society.