Self-management of Adaptable Component-based Applications

Liliana Rosa
INESC-ID/IST
lrosa@gsd.inesc-id.pt

Luís Rodrigues
INESC-ID/IST
ler@ist.utl.pt

Antónia Lopes
University of Lisbon
mal@di.fc.ul.pt

July 2011
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 system’s target behavior is described as a high-level policy that establishes goals for a set of performance indicators. The decision process is based on information provided independently for each component 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 component adaptations that may help to achieve the established goals when a given change in the execution context occurs. Online, the corresponding rules are 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.
Self-management of Adaptable Component-based Applications

Liliana Rosa Member, IEEE, Luís Rodrigues Member, IEEE, Antónia Lopes Member, IEEE, Matti Hiltunen Member, IEEE, Richard Schlichting Fellow, IEEE

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 system’s target behavior is described as a high-level policy that establishes goals for a set of performance indicators. The decision process is based on information provided independently for each component 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 component adaptations that may help to achieve the established goals when a given change in the execution context occurs. Online, the corresponding rules are 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 (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 software components even at runtime in response to changes in the execution environment. For example, the system workload or the available resources may often change while the software system is executing. While dynamic resource allocation (e.g., [1]) can be used to respond to such changes, adaptations that affect the component behavior itself can also be a powerful tool.

This paper addresses the problem of how to select the appropriate individual component adaptations to address deviations from the system’s optimal behavior. This problem presents a number of challenges. One challenge is how to determine the impact of a component adaptation in the system behavior. The impact depends not only on the current configuration of the software system—the set of components and their configuration parameters—but also on other factors that can be extremely dynamic and unpredictable, such as the patterns of component invocations. Another challenge is how to combine different components’ adaptations

- L. Rosa and L. Rodrigues are with IST/INESC-ID
- A. Lopes is with Faculty of Sciences, University of Lisbon
- M. Hiltunen and R. Schlichting are with AT&T Labs Research
- This work was partially supported by FCT (INESC-ID multi-annual funding) through the PIDDAC Program funds and through project REDICO (PTDC/ELA/71752/2006).
to achieve a specific desired change in the system behavior. The impacts of a component adaptation are only known by the developers of the component. Furthermore, in a complex software system with numerous software components, this knowledge is not centralized in a single developer but distributed among many developers in different teams or even different companies.

To address this problem, we consider software systems built from one or more adaptable components. We describe the desired behavior of the system by using a set of key performance indicators (KPIs). KPIs allow a high-level description of the system behavior as desired by its users while allowing the low level management details and decision to be left to the system. We assume that the behavior of the software system can be controlled by applying one or more component adaptations. Our approach leverages on information from component developers concerning the characteristics of each individual component. For instance, the designer of a graphical component $G$ may implement two operational modes: one that produces regular quality images and one that produces low quality images. The designer, knowing the implementation details, is fully aware of the trade-offs involved, namely, that the low quality mode, while providing lower image resolution, consumes less memory and less processor time than the high quality counterpart.

We propose a technique that uses this component-level information to select the best adaptations for a system when its execution deviates from the desired behavior. The selection depends on the execution state, namely, the load for each service, and is driven by a high-level policy that specifies the desired behavior in terms of KPIs—and, hence, the goals the adaptations should strive to achieve. The selection relies on information provided for each component describing possible adaptations, their impacts on KPIs, and any limitations or requirements. For example, if the component $G$ is heavily utilized, changing the mode from regular to low quality may yield significant memory, processor, and/or bandwidth savings, while if $G$ is lightly utilized, the same adaptations may have negligible impact. Thus, our adaptation selection takes into account the impacts of each adaptation and the contribution of each component to the performance of the entire system.

The rest of the paper is organized as follows. Section 2 provides an overview of the proposed approach, while Sections 3 to 5 present its key elements in detail. Section 6 presents the case study used to evaluate the approach. Experiments and results of the evaluation are presented in Section 7. Section 8 discusses the limitations of the approach and provides an overview of the related literature. Section 9 concludes the paper.

## 2 Approach Overview

Our work targets software systems built by combining several components, where each component may have multiple different implementations and each implementation may have multiple configuration options (e.g., configuration parameters). We will use system configuration to refer to the current set of component implementations used to compose the system and the configuration parameters of these component implementations. The system is adapted either by changing the system composition (e.g., replacing the implementation of a component by another) or by altering one or more components (e.g., changing their configuration parameters).
Our approach for self-management of such adaptable component-based software systems is depicted in Figure 1. Specifically, our approach takes Component Specification (describing the components used in the system), Adaptation Specifications (describing the available components adaptations, explained in Section 3.2), and Goal Policy (describing the desired system behavior, explained in Sections 3.1 and 3.3) as inputs and produces a set of Adaptation Rules that are then used at runtime to adapt the system to maintain its desired behavior. The approach consists of offline and online phases. The offline phase generates a set of adaptation rules, where each rule defines a collection of component adaptations that may correct a particular deviation of the system from the desired behavior. The rule generation process itself is executed in two steps. The first step determines the types of deviations that are relevant for the desired system behavior. These are captured as events that are triggered when the deviation occurs. The second step determines the adaptations that may correct the deviation. The online phase is executed at system runtime when the system deviates from the desired behavior, signaled by the triggering of an event. In this phase, the set of adaptation rules is evaluated against the current system state and the goals defined in the policy. Given that the system’s current execution state can now be factored in the decision making, the system can choose those adaptations that will, if possible, adapt the system back to its desired behavior. If all goals cannot be met, the rule evaluation process presented in this paper supports graceful degradation based on ranking of the goals, that is, when it is not possible to fulfill all the goals, the goal with the lower rank is violated first.

In general, the offline phase takes place at system deployment time, but it could be performed at runtime as well (for instance, as a result of a change in the goal policy). Even though the entire process could be done exclusively at runtime, the system reaction time can be reduced significantly by separating the evaluation into two phases. Specifically, the offline analysis of the specifications and goals produces a smaller set of possible adaptations that needs to be considered at runtime. In the next sections, we describe in detail the two phases, as well as the service specification, adaptation specification, goal policy, and performance metrics.

Fig. 1. Approach overview: offline and online phases
3 ADAPTABLE COMPONENTS AND ADAPTATION GOALS

3.1 KPIs

We assume the desired system behavior can be described using a number of KPIs, which are metrics that capture particular aspects of system performance. These metrics can address aspects such as the CPU or memory resources, user satisfaction (e.g., the response time), or service levels (e.g., SLAs), among others [2]–[4]. KPIs can be measured for individual components or for the entire system. We use notation $c.KPI$ to denote the KPI value of an individual component $c$ (for instance, the CPU consumed by that component).

In this paper, we assume that the system-level KPIs can be obtained by combining the KPI measures of each individual component. Namely, the value of each system-level KPI can be estimated by a non-decreasing monotonic combination function $CF$ of $\{c.KPI : c \text{ is in the current system configuration}\}$. The requirement of monotonicity of the combination function means that an increase in a component’s KPI ($c.KPI$) either has no effect, or results in an increase, in the system-level KPI. This requirement, which expresses a rather natural property for KPIs, ensures that local reasoning about the type of impacts that adaptations may have in the KPIs of individual components have corresponding impacts for the whole system. For example, if an adaptation of a component $c$ decreases its $c.cpu.use$, it will also reduce system-level $cpu.use$ KPI, if the system-level CPU usage KPI can be estimated as the sum over the components’ CPU usages. The sum, average, and maximum are three common non-decreasing monotonic functions in the definition of the system-level KPIs.

The KPI definition includes the name, the combination function that states how the global value is estimated, the type of the expected value, and the acceptable error margin ($Error$) in any evaluation of the KPI. The error margin describes the error tolerance—any two values that differ from each other less than the error margin are considered equivalent. Three examples of KPI definitions are presented below.

\[
\begin{align*}
KPI \, cpu.use: & \quad double \quad CombFunc \quad Sum \quad Error \quad 0.01 \\
KPI \, mem.use: & \quad double \quad CombFunc \quad Sum \quad Error \quad 0.01 \\
KPI \, resolution: & \quad int \quad CombFunc \quad Sum \quad Error \quad 0
\end{align*}
\]

For example, the first example states that system-level CPU usage KPI can be estimated as the sum of all components’ $cpu.use$ KPIs and that any two values of the KPI within 0.01 of each other are considered indistinguishable from the point of view of rule evaluation. The error margins could be automatically calculated from the KPIs range, but we believe it is more useful to allow the operator to express what he or she understands as a useful error for each defined KPI.

3.2 Specification of Component Adaptations

The system’s adaptive behavior is specified in terms of adaptations of component’s parameters or system configuration. The former tunes component parameters, while the latter replaces component implementations.

The specification of adaptations is made in the context of a component specification, describing the components available for use in the application. We consider component specifications defined in terms of a type...
hierarchy reflecting the is a relationship, taking into account the functionality provided by the components [5]. Component types can be concrete, designating a specific component with an available implementation, or abstract, representing the characteristics of a group of component types. Below is the specification of a concrete component that provides product webpages from a catalog, with a parameter ImageQuality that controls image resolution. This is followed by an adaptation that changes the image quality from regular to low.

Component Catalog
Parameters
ImageQuality:{low, regular}

Reversible Adaptation ToLowQuality
Component: Catalog
Actions:
setParameter (ImageQuality, low)
Requires:
ImageQuality = regular
Impacts:
Catalog.cpu_use ÷= 1.92 //decreases
Catalog.mem_use ÷= 1.21 //decreases
Catalog.resolution ÷=1 //decreases
Stabilization:
period = 60 secs

A component adaptation identifies one or more adaptation actions (to be executed together) concerning a single component c and specifies the impacts of those actions on all affected c.KPIs. We assume that it is possible to define an impact function that provides a reasonable approximation of the impacts of executing a set of actions over a component c on each c.KPI. In the above example, the adaptation is defined to impact three KPIs: Catalog.cpu_use, Catalog.mem_use, and Catalog.resolution. The impacts are reduced CPU use, memory use, and image resolution. While exact impact functions tend to be quite complex and may depend on several variables, rough approximations, like those used in the case study described in Section 6, are often sufficient for self-management. In the case the impacts on different KPIs are not independent, the impact function defined for each KPI must incorporate such dependencies. For instance, consider the case of an adaptation that replaces an algorithm by an optimized version that performs fewer operations at the expense of using more memory. The impact function of this adaptation on the execution time needs to take into account not only the gains obtained by the reduction in the number of operations, but also that the increase on memory usage may affect execution time negatively (e.g., due to the increase on thrashing pages in and out to disk).

The impact of an adaptation on a KPI may depend on many factors, among them, the type of adaptation actions, the affected component, the component configuration, the execution context, and the workload. When the impact is tied to a particular context, it should be stated in the adaptation requirements. If necessary, the component developer can describe different impacts for the same set of adaptation actions by defining multiple adaptations with the same set of actions but different impact functions, each adaptation applicable in a particular context. The problem of deriving the impact functions for each adaptation is outside the scope of this paper, but several existing approaches can be applied [2]. Knowledge about impacts is achieved mostly experimentally, doing benchmarks and comparing results. While component developers are usually not asked to provide such information, our approach requires the impacts to be quantified.
Each adaptation specification also define a *stabilization* period. It refers to the time that must be waited for an adaptation to take full effect, before subjecting the system to a reevaluation. The stabilization period depends on the components affected and the type of adaptation actions and, thus, it must be defined separately for each adaptation. If several adaptations are performed at the same time, the stabilization period in effect will be the larger of the considered adaptations.

In summary, the specification of an adaptation identifies: a) the concerned component, b) the adaptation action(s) to be performed, c) constraints such as the required component state or other adaptations that have to be performed simultaneously, d) the impact of the adaptation on each KPI, and e) the estimated stabilization period for the adaptation. If a KPI is omitted from the impacts, it means that the KPI is not affected. Even when not explicitly stated, any adaptation is only applicable if the target component actually exists in the current system configuration. We assume that meta-information about the deployed and executing components, as well as the current value of their parameters, is available at runtime.

All component adaptations whose reverse adaptation(s) are to be extracted automatically are marked as *reversible* adaptations. This helps to reduce the user’s effort in describing the adaptations. The *ToLowQuality* adaptation is an example of a reversible adaptation. Its reverse is an adaptation of the *Catalog* component that increases image quality to regular, with inverse impacts: \( \text{Catalog.cpu} \text{use} = 1.92, \text{Catalog.mem} \text{use} = 1.21 \) and \( \text{Catalog.resolution} += 1 \). When processing an adaptation specification, the implicit reverse adaptations are automatically generated (possibly, requiring help from the user for obtaining inverse functions, if complex impact functions are used). If the user wants to add the reverse adaptation(s) manually so that he can specify requirements or stabilization periods different from those automatically generated from the original adaptation, the adaptation should not be marked as reversible.

Additional adaptation constraints can be specified by listing the adaptations of different components that cannot be applied at the same time. By default, adaptations to the same component that have impacts on the same KPIs are assumed to be conflicting, but it is possible to specify a single adaptation that consists of several actions, provided that the joint impact of these actions over the KPIs can be defined. These conflicts are described as pairs of adaptations. Dependencies between adaptations are specified in the same manner.

The complete adaptation specification consists of the adaptations, the conflicts, and the dependencies.

| Conflict | conflict_name | Adoptions | (componentA.adapt1, componentB.adapt2) |
| Dependency | dependency_name | Adoptions | (componentA.adapt1, componentB.adapt2) |

### 3.3 Policies

Adaptation goals are specified in terms of a goal policy that describes the desired values for a set of KPIs. A goal policy describes: a) the KPIs relevant to the policy, and b) the goals to be met by the system. The policy can further use the relevant KPIs to specify composite KPIs, denoted by CKPIs. CKPIs are identified by a *ckpi_name* and their specification consists of a join function \( JF \) of several KPIs, and an *Error* margin:
As an example, consider the specification of the CKPI $g_{dev}$ presented above, that measures the weighted deviation from target CPU and memory utilization values. The 0.6 value is the target utilization for the $cpu_{use}$, while 0.4 is the target for $mem_{use}$. Henceforth, we use KPI to refer to either a basic KPI or a CKPI.

The KPIs are used to define specific behavior goals, constituting an adaptation policy. There are two types of goals: exact and optimization goals. Exact goals separate the values of a KPI in two disjoint sets: acceptable and not acceptable. We consider the following types of exact goals:

- **Goal name**: kpi_name **Above** threshold_down
- **Goal name**: kpi_name **Below** threshold_up
- **Goal name**: kpi_name **Between** thr_down thr_up

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 the stated lower and upper thresholds. Note that it would be possible to express the conjunction of $k$ **Above** $a$ and $k$ **Below** $b$ using the goal $k_{ab}$ **Above** 0, over the CKPI $k_{ab}$ defined by $k_{ab} = (k \div a) \times (b \div k)$, where $(x \div y) = (x - y)$ if $x > y$ and 0 if $x \leq y$. However, we opted to have explicit **Between** goals because it is easier to specify this type of goals and it is possible to handle the violation of $k$ **Between** $a \ b$ more efficiently than the violation of $k_{ab}$ **Above** 0. Specifically, in the latter case, it would not be possible to distinguish the situations in which the value is higher than $b$ or lower than $a$. In general, however, the logical combination of goals requires the use of CKPIs (e.g., the specification of a conjunction of goals over different KPIs requires to define a goal over a CKPI whose value is 0 only when at least one of the goals is violated).

Optimization goals, instead of simply classifying the values of a KPI as good or bad, specify a total order between these values. We consider the following types of optimization goals:

- **Goal name**: kpi_name **Close** target MinGain mgvalue Every period
- **Goal name**: Minimize kpi_name MinGain mgvalue Every period
- **Goal name**: Maximize kpi_name MinGain mgvalue Every period

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. The description of optimization goals may also include other parameters: the time period and the minimum gain. The **Every** time period specifies how often the system should try to find adaptations 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 optimization goal opens the possibility of continuously attempting to improve the system behavior. The minimum gain MinGain avoids the cost of reconfiguring the system for a profit that is below a defined threshold. If the estimated change in the KPI value is below mgvalue, the adaptation is not worth performing. Because any two values within the error margin specified for the target KPI are considered indistinguishable, the mgvalue should always be larger than the error margin.
The goal policy consists of a number of exact and optimization goals, presented in rank order. In situations where it is not possible to fulfill all goals, the selection process that is executed during the online phase can use this prioritization of the goals to achieve a graceful degradation of the system behavior. The selection process we present in Section 5 ensures that the system will never chose to correct a less important goal, if that results in the violation of a more important, higher ranked goal.

At first glance, the expressiveness of rank-based policies might seem not ample enough for expressing more complex scenarios such as $G_1 : k_1 \text{Below } v_1$ is more important than $G_2 : k_2 \text{Below } v_2$ and $G_2$ is more important than $G_3 : k_3 \text{Below } v_3$ but we prefer to have $G_2$ and $G_3$ both satisfied than to have just $G_1$. However, with the definition of appropriate CKPIs, we can easily address this type of scenarios. Specifically, we would just have to define a CKPI $k_{23} = (v_2 - k_2) * (v_3 - k_3)$, a goal $G_4 : k_{23} \text{Above 0}$, and a policy with the goals $G_4, G_1, G_2, G_3$, in this order. Note also that CKPIs can be used to define weight-based policies. For instance, one could also represent that we prefer the conjunction of $G_2$ and $G_3$ to $G_1$ by defining the following CKPI:

\[
k_w = w_1 \cdot \text{signal}(v_1 - k_1) + w_2 \cdot \text{signal}(v_2 - k_2) + w_3 \cdot \text{signal}(v_3 - k_3)
\]

where $\text{signal}(x) = 1$ if $x > 0$ and $0$ if $x \leq 0$, and then setting the weights as $w_1 = 0.4$, $w_2 = 0.35$, and $w_3 = 0.25$, respectively, together with the goal $G_w : \text{Maximize } k_w \text{ MinGain 0 Every 1200}$. We decided to use ranks because our goal is to make policy definition a high-level task. Using ranks, it is far simpler to specify the policy and be sure of what the policy does, than if one needs to rely exclusively on weights. A weight-based approach demands that the operator decide exactly the weight of each goal compared to others—a task that is far more complex than simply identifying which goals are more important than others. Furthermore, it is harder for the operator to verify if the given weights have the effect he or she expects in terms of system behavior. Finally, the extension of a weight-based policy with an additional goal also demands more effort than simply adding the new goal somewhere in the policy.

4 Rule Generation

Goal policies and the description of available component adaptations are used to automatically generate a set of adaptation rules. Each rule consists of an event and one or more alternative sets of adaptations. Each set contains adaptations that can be performed simultaneously and may help achieve the specified goals when a change in the execution context occurs. The choice of the set of adaptations to be executed is performed at runtime, when the rule is evaluated, taking into account the current system state.

More precisely, during the offline phase, a set of adaptation rules is generated with the form $\text{When } e \text{ Select } \{AS_1, ..., AS_n\}$. The $\text{When}$ clause defines the event that triggers the evaluation of the rule. This may be caused by a change signaled by a sporadic event—for example, when some KPI exceeds a threshold—or by the passage of time signaled by a periodic event. In any case, each event is associated with a particular KPI or CKPI of a specific goal. The $\text{Select}$ clause lists alternative sets of adaptations $AS_i$ for dealing with this particular event.
These sets are the viable combinations of all relevant adaptations, i.e., of all adaptations that have a positive or unknown impact on the KPI or KPIs associated with the goal from which the event was extracted. While we cannot estimate the concrete impact of an adaptation offline, in many situations we can assess if it will increase or decrease a KPI. For instance, if a goal states that some KPI must be maintained above a given threshold, only the adaptations that increase the KPI have a positive impact. When it is not possible to assess if the impact is positive or not (for instance, because the impact function depends on context information or because the event concerns a CKPI), the adaptation is also considered relevant to avoid cutting down on potentially useful adaptations. The combinations of adaptations are determined considering dependencies and conflicts between adaptations, or any other requirements stated in the adaptations. Naturally, given that rules are generated offline, it is only possible to verify requirements that do not require runtime state information.

The generated rules can be made available to the operator who specified the goal policy for feedback purposes. However, these rules tend to be quite difficult to read as they usually include a large number of adaptations sets. If the operator wants a list of the relevant adaptations in each rule, it will be hard to identify them all in the mist of sets with common adaptations and subsets that unfold larger sets of adaptations. To address this difficulty, rules can be presented to the operator using a friendlier human-readable representation, which aims to facilitate the understanding of the system adaptive behavior. As depicted below, this representation includes the triggering event, the list of all relevant adaptations for the situation signaled by the event, and the pairs of conflicting and depending adaptations. In this manner, the system operators can check if an adaptation is missing from the list, or how the behavior goals translate into actual actions.

When event

Conflicts: (S1.A, S2.X) Dependencies: (S2.Y, S3.Z)

4.1 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 events of type kpiAbove(kpi,x) and kpiBelow(kpi,x) are sporadic and are generated when the value of kpi is detected to be above or below the threshold value x (and therefore needs to be decreased or increased, respectively). Similarly, events of type kpiIncrease(kpi,θ,condition) and kpiDecrease(kpi,θ,condition) are periodic, generated every period θ if the condition over the current value of kpi holds. They signal that the value of kpi needs to be increased or decreased, respectively.

As noted above, the policy has two distinct types of goals. When an exact goal is violated, the system adaptation is triggered as soon as the violation is detected by the context monitor. For optimization 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 when these events are triggered.

The specific events that are extracted from a goal 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 2 provides examples of events for some goals. For an Above goal, an event of type \text{kiBelow} needs to be triggered when the value of the KPI falls below the specified threshold by a margin greater than the KPI error margin. Similarly, for a Below goal, an event of type \text{kiAbove} 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 \text{kiDecrease} / \text{kiIncrease}, respectively, needs to be triggered with the period specified in the goal. Finally, for the Close goals, both of these two events are possible. One will be triggered depending on which condition is true: the \text{kiIncrease} event has a condition that the KPI value has to be \(< \text{target} + \text{error margin}\), while the \text{kiDecrease} event has the opposite condition \(> \text{target} + \text{error margin}\). For each extracted event, a rule is created with the When clause stating the event as the trigger for the rule evaluation.

<table>
<thead>
<tr>
<th>Type</th>
<th>Goal</th>
<th>Event 1</th>
<th>Event 2</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>Above</td>
<td>\text{kiBelow}(kpi, x)</td>
<td>-</td>
<td>threshold exceeded</td>
</tr>
<tr>
<td>Exact</td>
<td>Below</td>
<td>\text{kiAbove}(kpi, y)</td>
<td>-</td>
<td>threshold exceeded</td>
</tr>
<tr>
<td>Exact</td>
<td>Between</td>
<td>\text{kiBelow}(kpi, x)</td>
<td>\text{kiAbove}(kpi, y)</td>
<td>threshold exceeded</td>
</tr>
<tr>
<td>Approx</td>
<td>Close</td>
<td>\text{kiIncrease}(kpi, \theta, cond)</td>
<td>\text{kiDecrease}(kpi, \theta, cond)</td>
<td>periodic</td>
</tr>
<tr>
<td>Approx</td>
<td>Minimize</td>
<td>\text{kiDecrease}(kpi, \theta, cond)</td>
<td>-</td>
<td>periodic</td>
</tr>
</tbody>
</table>

**TABLE 1**
Event types generated for each type of goal

---

4.2 Selecting Component Adaptations

The second task of the offline rule generation is to identify the sets of adaptations that need to be included in each rule, according to the extracted event. The purpose of a given rule is either to increase or decrease the value of a given KPI. Thus, adaptations that do not declare impact on that KPI are discarded. The remaining adaptations’ impact functions is analyzed to verify if they change the KPI to the right direction. For instance, consider rule with event \text{kiAbove}(\text{cpu}_use, 0.36) where the goal is to decrease the value of the \text{cpu}_use. The adaptation \text{ToLowQuality} adapts the Catalog component with impact \text{Catalog.cpu}_use = \text{Catalog.cpu}_use ÷ 1.92. To assess if this adaptation should be used in the rule, one simply checks whether the function \(f(kpi) – kpi\) has a negative derivative. In this example, since the derivative of \(x ÷ 1.92 - x\) is always negative, the adaptation \text{ToLowQuality} decreases the CPU utilization. The corresponding inverse adaptation (since it is marked as
reversible) will increase the CPU consumption. This adaptation would be useful, for instance, for dealing with the event \( \text{kpiBelow}(\text{cpu}_\text{use}, 0.29) \) of Figure 2.

Once all the relevant adaptations for dealing with an event have been selected, the human-readable representation of the rule is built. Then, the set of viable combinations of these adaptations is calculated. These combinations respect all adaptation requirements and any inter-adaptation conflicts or dependencies that may exist. A combination that is always viable is the empty set of adaptations, which corresponds to leaving the system as it is. Also recall that any two adaptations to a component that have impact on the same KPI are considered conflicting by default. This is due to the fact that, in this case, it is typically not easy to estimate the joint impact on that KPI. In this way, adaptation rules that are generated for dealing with changes in KPIs have the following property: all sets of adaptations included in \( \text{Select} \) clause have at most one adaptation for each component. In the case of events that notify changes in CKPIs, there are at most as many adaptations per component as there are KPIs involved in the definition of the CKPI. These properties are important because they ensure that the rate of growth of the sets of adaptations as well as the number of the generated rules is, to some extent, controlled by the number of components and KPIs.

5 Rule Evaluation

An adaptation rule contains all the useful sets of adaptations for a particular event. However, the benefits of each of these sets depend on the system state and workload, and thus, the need to evaluate the rule at runtime to choose the set of adaptations to perform. The evaluation of a rule \( \text{When } e \text{ Select } \{ AS_1, ..., AS_m \} \) occurs whenever event \( e \) is triggered, and consists of selecting one from all the possible sets of adaptations \( AS_i \). The selected combination determines the adaptations to be applied to the system and, hence, the intent of the selection process is to find the combination that best satisfies the goals defined in the policy.

The process of rule generation ensures that each \( AS_i \) includes only adaptations that can be executed at the same time. However, these sets may include adaptations that cannot be applied in the current system configuration. This happens if the target component is not used in the current system configuration or if any constraints associated with the adaptation do not hold. Hence, the evaluation of the rule starts by removing non-applicable adaptations from every \( AS_i \). Then, rule evaluation proceeds by searching for the combinations that best match the goals expressed in the adaptation policy, relying in the current system state.

The selection process of a set of adaptations subsumes an optimization criterion and there are different reasonable choices. For instance, if we restrict our attention to policies with exact goals only, one possibility is to consider that being optimal means to satisfy as many goals as possible. In this case, the selection process must pick a set of adaptations \( AS_i \) that maximizes the number of goals that are expected to be satisfied if those adaptations in \( AS_i \) are performed (the ranking order of the goals is ignored in this case). A different optimization criteria, which we call ranked-eager, is to take into account the rank of goals in policies and satisfy more important goals first. In this case, the selection process must start by selecting the sets of adaptations
that are expected to satisfy the highest ranked goal, then among those are selected the sets of adaptations that are expected to satisfy the second highest ranked goal, and so on. If all sets selected in step $k$ violate the $k+1$-highest ranked goal, there is an exception to what was described above and all sets are selected. In this way, if there is no way to satisfy the $k+1$-highest ranked goal without violating a more important goal, then the remaining goals are still used for tie-breaking. This selection mechanism is illustrated in Figure 3.

In this work, we have opted for an optimization criterion that is an extension of the ranked-eager criterion, which is also applicable to policies with optimization goals. Given that an optimization goal specifies a total order between the possible values of a KPI, when several sets of adaptations are evaluated against an optimization goal, among these sets of adaptations are selected those that put the KPI close to the target specified in the goal. This is illustrated by Figure 4, where the best set of adaptations among those evaluated against the optimization goal $B$ is represented by a multi-point star shape—the $AF$ set. Dashed lines are used to link adaptation sets that are considered equivalent in what concerns goal $B$ (the difference between their estimated impacts on the KPI of goal $B$ is smaller than the error margin). In the example, when the optimization goal $B$ is evaluated, only two sets are selected. One is the set that puts the KPI of $B$ close to the target, and the other set is an equivalent one. The next paragraphs present a more detailed and rigorous description of the whole selection process.

The process starts with a search space $SS = \{AS_1, AS_2, ... AS_m\}$. The search involves analyzing the estimated effects of the different combinations on the KPIs addressed by the goals and deducing which ones best fit these goals. More precisely, recall that 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 with $G_1$. In other words, each goal works as a filter which only allows a number of $AS_i$ to pass. The filter of a goal $G_i$, which performs the evaluation of a combination of adaptations $C$ against a goal $G_i$, depends on the type of goal (exact or optimization) and the estimated impact of the adaptations on the KPI, associated with the goal. This value, which we denote by $\text{KPI}^C_i$, is calculated as follows:
- if KPI$_i$ is not composite and is declared to be calculated using a combination function $CF$, then $KPI_i^C$ is the result of applying $CF$ to the values $s.KPI_i^C$, for every component $s$ in the current system configuration. As noted before, the set $C$ includes at most one adaptation involving $s$. If it contains none, $s.KPI_i^C$ is just the current value of KPI$_i$. If it contains one adaptation, then $s.KPI_i^C$ is calculated by applying the impact function of that adaptation over the current value of KPI$_i$.

- if KPI$_i$ is composite (a CKPI), then KPI$_i$ is defined by a join function $JF$ involving non-composite KPIs and hence, the value of KPI$_i^C$ is obtained after calculating the estimated impact of $C$ in these KPIs.

We define that $C$ matches $\{G_1, \ldots, G_i\}$ only if the following conditions hold:

1) if $i > 1$, $C$ matches $\{G_1, \ldots, G_{i-1}\}$

2) if $G_i$ is an exact goal: $KPI_i^C$ satisfies $G_i$ or, for all other combinations $C^+$ in $SS$ that match $\{G_1, \ldots, G_{i-1}\}$, $KPI_i^{C^+}$ also violates $G_i$.

3) if $G_i$ is an optimization goal: $|KPI_i^C - KPI_i^{C^+}| < error\_margin^{KPI_i}$, where $C^+$ is, among the combinations in $SS$ that match $\{G_1, \ldots, G_{i-1}\}$, the one that puts the $KPI_i$ closer to the target specified in $G_i$.

$C$ best matches $\{G_1, \ldots, G_n\}$ only if i) $C$ matches $\{G_1, \ldots, G_n\}$ and ii) for some $1 \leq i \leq n$, either $G_i$ is an exact goal that is currently violated and $KPI_i^C$ satisfies it; or $G_i$ is an optimization goal and, compared to the current value of $KPI_i$, there is a gain that exceeds the specified minimum gain.

To illustrate the selection process, consider an adaptation policy whose highest ranked goal is $mem\_use\ Below\ 0.58$ and the second highest ranked goal is $cpu\_use\ Below\ 0.35$ (presented in Figure 2). Let’s assume that the current $cpu\_use$ value is 0.45 (the second goal is currently violated) and the rule available to deal with the violation of this goal is $When\ kpiAbove(\ cpu\_use,\ 0.36)\ Select\ \{AS_0, AS_1, AS_2, AS_3\}$, where $AS_0$ is the empty set and all adaptations included in the others sets $AS_i$ decrease $cpu\_use$ at the expense of increasing $mem\_use$. During the evaluation of the rule triggered by the event $kpiAbove(\ cpu\_use,\ 0.36)$, the selection process starts by selecting the sets whose estimated effects do not bring $mem\_use$ above 0.58. If the value of $mem\_use$ is already close to the limit and the estimated effect of all three non-empty sets is to bring its value above 0.58, then the result of the selection process is to leave the system as it is (the set of adaptations selected is the empty one). On the contrary, if the value of $mem\_use$ is far from the limit and the estimated effects of, say, all but $AS_3$ keep $mem\_use$ value below 0.58, then the process would continue by selecting among $AS_0, AS_1, AS_2$, the sets that are able to bring $cpu\_use$ below 0.36. Suppose this is the case of the two non-empty sets. Then, the next ranked goal in the policy would be used to tie-break among them.

The rule evaluation mechanism has two implicit advantages. First, even if a goal is violated and cannot be satisfied, the evaluation process will continue to see if it can improve the system by satisfying other goals, such as optimization goals. Second, when it is not possible to satisfy all goals, the proposed approach provides graceful degradation according to the rank. As a result, the goals with lower rank will be violated first to maintain the more important goals. Note that if an optimization goal ranks first in the policy, the rule evaluation mechanism will treat it in a greedy manner. This allows to describe scenarios where we want to
give preference to an optimization goal, hence, the system adapts mainly focused on that optimization.

6 Case Study

The case study used for evaluating the approach is an online retail website that allows users to browse the products catalog, register for an account, and shop online. The website follows a multi-tier architecture and supports two types of users: private and business. Private users are consumers that buy products directly from the retailer. Business users are professional business users or other wholesalers that acquire products for a company or resale purposes. Webpages are customized to the type of user, and can either be generated or retrieved from different components in the backend. These components deal with different types of content: static, dynamic, secure, and non-secure. Static content is retrieved directly, while dynamic content is generated upon request. The content is handled through a non-secure connection, unless sensitive data is involved.

6.1 Adaptable Components

In this case study, we focus on the components running in the backend, namely, the adaptable components in which the web application relies: Catalog, User and Account. In the context of this work, we refer to backend as the server side coding, which covers the website’s functionality and backend systems.

The Catalog component provides the product descriptions pages, which are static content webpages sent in a non-secure manner to the client. This component is divided into PrivateCatalog and BusinessCatalog, for private and business users, respectively. These components differ in the content provided to clients, namely, different prices and information displayed. Both components have two configuration modes: regular/low; in low mode the component offers lower image quality.

The User component generates user customized webpages, sent in a non-secure manner. The component generates webpages with personalized product recommendations and searches, when the user is logged in. The component is divided in PrivateUser and BusinessUser components, which rely in search and recommendation engines [6], [7]. Both engines can execute in two modes: fresh and cached. The former demands more resources and takes longer to generate a reply, while the latter consumes fewer resources and has faster response times. In the search engine, the fresh content is a list of products that fit the search keywords, sorted by current popularity indexes [8]. The cached content is a previously generated product list, whose popularity indexes may no longer be up to date. The recommendation engine provides a list of recommended products, a content that is used to customize the web experience to a particular user. In the recommendation engine, the fresh recommendations depend on information regarding the user session and previous orders while the cached recommendations are produced from time to time or are the result of previously generated recommendations.

Finally, the Account component handles webpages that deal with sensitive user information. This information refers to account login, credit card information, billing and shipping addresses, and account settings, among others. The component is divided in PrivateAccount and BusinessAccount components, the latter including
invoice and budget management features. Both components have two configuration modes concerning the image quality \((regular/low)\). The entire component specification is presented below:

- **Abstract Component Catalog**
  - Parameters
    - mode: \{regular, low\}

- **Component BusinessCatalog**
  - subtype Catalog

- **Component PrivateCatalog**
  - subtype Catalog

- **Abstract Component User**
  - Parameters
    - search: \{fresh, cache\}
    - recommendation: \{fresh, cache\}

- **Component BusinessUser**
  - subtype User

- **Component PrivateUser**
  - subtype User

- **Abstract Component Account**
  - Parameters
    - mode: \{regular, low\}

- **Component BusinessAccount**
  - subtype Account

- **Component PrivateAccount**
  - subtype Account

The website relies in a classical three-tier architecture: presentation, application, and data tiers. The application logic is executed in the middle-tier, typically in an *application server*; and the data tier consists of a database and its management services executed in a *data server*. In our prototype, we use different machines to run the application server and the data server, as illustrated in Figure 5. The first machine runs a web server and the set of components that implement the application logic. The data server runs the search and recommendation engines, as well as the database that stores the catalog and user data.

### 6.2 KPIs

Online retail web sites face dynamic workloads, with predictable periods of overload, such as holiday season, but also with unexpected peaks, such as flash crowds [9]. The adaptive behavior in this case study aims at avoiding overload and maintaining an appropriate balance between resource consumption and service quality, according to the workload.

We consider that the system load is reflected by the consumed CPU time (other metrics could be used, see for instance [2], [10]). The closer this resource is to the limit, the closer the system is to overload. By maintaining the CPU consumption at a certain value, it is possible to avoid overloads. The service quality can be regarded as a combination of different factors: *Catalog* and *Account* components provide different quality of content to the client; *User* components also have impact on the service quality due to the freshness of recommendations and searches provided to the client. Table 2 presents the full list of KPIs considered in the case study, where *bsn* stands for business services and *prv* for private services. Note that all KPIs that address service quality refer either to all private or business services, which is reflected in the choice of the combination function. For
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>CF</th>
<th>Range</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_use</td>
<td>double</td>
<td>Sum</td>
<td>0-1</td>
<td>0.01</td>
<td>CPU consumption</td>
</tr>
<tr>
<td>resolution_bsn</td>
<td>int</td>
<td>B_Sum</td>
<td>3-6</td>
<td>0</td>
<td>Image quality</td>
</tr>
<tr>
<td>resolution_prv</td>
<td>int</td>
<td>P_Sum</td>
<td>3-6</td>
<td>0</td>
<td>Image quality</td>
</tr>
<tr>
<td>recommend_bsn</td>
<td>int</td>
<td>B_Sum</td>
<td>3-6</td>
<td>0</td>
<td>Freshness of recommendations</td>
</tr>
<tr>
<td>recommend_prv</td>
<td>int</td>
<td>P_Sum</td>
<td>3-6</td>
<td>0</td>
<td>Freshness of recommendations</td>
</tr>
<tr>
<td>search_bsn</td>
<td>int</td>
<td>B_Sum</td>
<td>3-6</td>
<td>0</td>
<td>Freshness of searches</td>
</tr>
<tr>
<td>search_prv</td>
<td>int</td>
<td>P_Sum</td>
<td>3-6</td>
<td>0</td>
<td>Freshness of searches</td>
</tr>
<tr>
<td>ptr_bsn</td>
<td>double</td>
<td>B_Sum</td>
<td>R⁺</td>
<td>0.2</td>
<td>Processing time of requests</td>
</tr>
<tr>
<td>ptr_prv</td>
<td>double</td>
<td>P_Sum</td>
<td>R⁺</td>
<td>0.2</td>
<td>Processing time of requests</td>
</tr>
<tr>
<td>query_load</td>
<td>int</td>
<td>Sum</td>
<td>N₀</td>
<td>10</td>
<td>Searches and recommendations</td>
</tr>
<tr>
<td>bsn_services</td>
<td>int</td>
<td>B_Sum</td>
<td>N</td>
<td>0</td>
<td>Number of business services</td>
</tr>
<tr>
<td>prv_services</td>
<td>int</td>
<td>P_Sum</td>
<td>N</td>
<td>0</td>
<td>Number of private services</td>
</tr>
</tbody>
</table>

TABLE 2
KPIs used in the case study.

instance, the use of B_Sum in resolution_bsn means that only the resolution of business services are accounted, in opposition to P_Sum used in resolution_prv, where only private users are accounted.

6.3 Component Adaptations

As discussed before, all Catalog, User and Account components have configuration parameters that can be changed at runtime. By taking advantage of their adaptability capabilities, we have defined sixteen reversible adaptations and, hence, a total of thirty two adaptations. The impact values used in the case study are either obtained from the component developers, such as impacts regarding changes to image quality, or from benchmarks, for instance to compare cache retrieved recommendations and recommendations generated on request. It is important to note that all adaptations have different impacts, ones greater than the others. Due to space limitations we only present two examples of adaptations.

Reversible Adaptation ToLightCatalogB
Component: BusinessCatalog
Actions:
setParameter (mode,low)
Requires:
mode = regular
Impacts:
BusinessCatalog.cpu_use ÷= 2.01 //decreases
BusinessCatalog.resolution_bsn ÷= 1 //decreases
BusinessCatalog.ptr_bsn ÷= 1.99 //decreases
Stabilization:
period = 60 secs

Reversible Adaptation ToLightCatalogP
Component: PrivateCatalog
Actions:
setParameter (mode,low)
Requires:
mode = regular
Impacts:
PrivateCatalog.cpu_use ÷= 1.92 //decreases
PrivateCatalog.resolution_prv ÷= 1 //decreases
PrivateCatalog.ptr_prv ÷= 1.43 //decreases
Stabilization:
period = 60 secs

These adaptations change the mode of Catalog component from regular to low. As stated in the Impacts statements, they reduce CPU consumption at the cost of degrading the service quality. In fact, the decrease in image quality reduces the CPU consumption and also the request’s processing time. Note also that the degradation of service quality in the business catalog allows a larger cut in CPU consumption than in the private catalog; this is due to the larger number and size of images that are involved in the business service.

In addition to the two adaptations presented above, the case study includes more 14 reversible adaptations (32 adaptations in total). We are aware that, for the sake of the strength of the evaluation of the proposed approach, the larger the number of adaptations the better. However, the larger the number of adaptations, the harder is to comprehend and analyze the obtained results. Therefore, the number of adaptations employed in the case study is a compromise that aims at highlighting the advantages of the approach while, at the same
time, should allow the reader an easy, or at least, understandable grasp of the achieved results and analyzed aspects. In Section 7.4 we address situations with larger numbers of adaptations.

### 6.4 Policies

The goal policies were designed to avoid system overload, while offering the best possible service quality. To avoid system overload, both the application server and the data server must be observed. We have considered relevant two different metrics: the CPU usage and the number of queries made to the search and recommendation engines. A limit to the CPU usage in the application server makes it possible to avoid its overload. Since several processes compete for the CPU, the limit only refers to the usage by adaptable components. For the same purpose, a limit to the number of queries is imposed on the data server.

The components contribute in different manners to the overall quality of service. Catalog and Account components can provide images with different quality, the higher the quality the better, but the trade-off is a longer response time. The User components can provide different levels of freshness for search and recommendation results. The fresher these results are the better, but the response time will be higher due to the additional computation time required. Therefore, we have the CKPIs presented in Table 3. Content quality is described by the CKPIs $qlt_{bsn}$ and $qlt_{prv}$, while response time is described by CKPIs $mrt_{bsn}$ and $mrt_{prv}$, which give feedback on the processing time for requests, independently of the request distribution (IRD).

We have considered two different policies in order to illustrate one of the main advantages of the proposed approach: different policies, reflecting different business strategies, can be obtained just by changing the order in which the goals are listed and, hence, changing policies requires little effort.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>$JF$</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mrt_{bsn}$</td>
<td>Mean processing time IRD $2^{ptr_{bsn}}/bsn_services$</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>$mrt_{prv}$</td>
<td>Mean processing time IRD $2^{ptr_{prv}}/prv_services$</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>$qlt_{bsn}$</td>
<td>Overall quality of services $resolution_{bsn}+recommend_{bsn}+search_{bsn}$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$qlt_{prv}$</td>
<td>Overall quality of services $resolution_{prv}+recommend_{prv}+search_{prv}$</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 3**

CKPIs used in the case study.

These policies reflect the insights provided by related research, including policies to achieve optimal resource use for web servers [2], [11], intermediary adaptation systems [10], [12], [13], and web server and user experience improvement [14]. Both have three common concerns: avoid overload, provide a satisfactory user experience to clients, and select which type of clients to favor, in case of overloads.

Policy A starts with the $limit\_cpu$ and $limit\_query\_load$ goals. The purpose of these goals is to avoid overloads by limiting the $cpu\_use$ of adaptable components in the application server and limiting the engines’ $query\_load$ in the data server. In this manner, it is possible to avoid resource exhaustion, without causing underuse of...
Events extracted from the goals used in the case study.

resources. The `limit_mrt-bsn` and `limit_mrt-prv` goals refer to response time, limiting the processing time for a request. Finally, the `max_qlt-bsn` and `max_qlt-prv` goals refer to content quality, urging the maximization of the content quality. The goals referring to the business clients come first in this policy to reflect their priority. In contrast, policy B gives priority to private clients. The same goals are employed, however, using a different order: the goals referring to private clients are given higher priority then the equivalent goals for business clients. By switching between policies A and B one can favor a given type of clients according to some business target, for instance, by favoring the type that provides a better revenue at a given point in time.

### 6.5 Generated Rules

According to the generation process described in Section 4, a set of goals gives rise to a set of adaptation rules. These rules are built using the triggers extracted from the goals (presented in Table 4) and the set of available adaptations. For instance, the two adaptations presented in Section 6.3 decrease the CPU use, image quality, and response time for a service. Thus, both adaptations will be useful when the KPI `cpu_use` is above the limit. There are other two similar adaptations for the `Account` component. The adaptation rule that handles the violation of the `limit_cpu` goal is the following (which we present in the human readable representation):

<table>
<thead>
<tr>
<th>Type</th>
<th>Goal</th>
<th>Event</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td><code>limit_cpu</code></td>
<td><code>kpiAbove(cpu_use,0.35 + 0.01)</code></td>
<td><code>cpu_use &gt; 0.36</code></td>
</tr>
<tr>
<td>Exact</td>
<td><code>limit_query_load</code></td>
<td><code>kpiAbove(query_load,1200 + 10)</code></td>
<td><code>query_load &gt; 1210</code></td>
</tr>
<tr>
<td>Exact</td>
<td><code>limit_mrt-bsn</code></td>
<td><code>kpiAbove(mrt-bsn,1.0 + 0.2)</code></td>
<td><code>mrt-bsn &gt; 1.8</code></td>
</tr>
<tr>
<td>Approx</td>
<td><code>max_qlt-bsn</code></td>
<td><code>kpiIncrease(qlt-bsn,0.00,0.00,1.0)</code></td>
<td>Every 900 s</td>
</tr>
<tr>
<td>Exact</td>
<td><code>limit_mrt-prv</code></td>
<td><code>kpiAbove(mrt-prv,1.9 + 0.2)</code></td>
<td><code>mrt-prv &gt; 2.1</code></td>
</tr>
<tr>
<td>Approx</td>
<td><code>max_qlt-prv</code></td>
<td><code>kpiIncrease(qlt-prv,1.00,0.00,1.0)</code></td>
<td>Every 1300 s</td>
</tr>
</tbody>
</table>

**TABLE 4**

Overall, the offline phase generates 6 rules, one per goal. For each rule, the number of relevant adaptations range from 4 to 10. The rules have at most 35 adaptation sets which have up to 4 adaptations each.

Notice that the order of goals in the goal policy is not used for the generation of the set of adaptation rules. Hence, the rules that are generated for policy A and B are exactly the same. As discussed in Section 5, the goal ranking is taken into account when, at runtime, these rules are evaluated and it is necessary to find the set of adaptations that best satisfies the goals.

### 7 Experimental Evaluation

We conducted a study to evaluate the proposed approach, namely to analyze how successfully the rules generated offline drive the runtime adaptation, given changes that carry the system outside the desirable or acceptable behavior. We also analyze how the approach handles the two different adaptive behaviors described in Section 6.4. To do so, we implemented a prototype of the framework in Java™, and developed experiments for the autonomic management of web-based applications.
7.1 Experimental Setup

The prototype implementation consists of the overall framework and the website. The Apache web server (http://httpd.apache.org) running on Linux is used to execute requests. To monitor the execution context a monitoring tool was implemented in Python and integrated with the framework prototype. The monitoring tool can be configured in terms of the time interval between readings, among other options. These configuration options are defined in the framework configuration file. The tool monitors the CPU usage, the number of requests and queries, and the average processing time for each request. This information is collected per request and then interpreted to give information per service.

To analyze how the policy drives changes in the service quality when the resource consumption varies, we generated several workloads to force different adaptations. In periods when the load is high, the system will adapt one or more components to provide a lower quality. In periods when the load is light and the service quality is not at its best, the system will adapt to provide a higher service quality. After adapting, the KPIs readings are ignored until the end of the stabilization period.

The experimental testbed consists of four machines. The application server machine runs the Apache Web Server and all the components that implement the business logic. The data server machine runs the recommendation and search engines. The remaining two machines run workload generators, functioning as clients. All machines are connected by a 100 Mbps Ethernet. The application machine is a 8 x 3.22 GHz Xeon 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, SSL, and rewriting modules enabled. The application server machine is a 2.8 GHz Pentium IV processor with 2 GB RAM running Linux (Kernel v2.6.20-17). The client machines are similar to the application server and they run Pylot (http://www.pylot.org), an open source tool for testing performance and scalability of web services based on an XML file that describes the workload. The tool also allows to control the number of clients and the interval between requests. We modified the original Pylot tool to run several workloads in sequence, each for a period of time.

The backend adaptable components are implemented as follows. The Catalog components are implemented using several HTML pages containing text and images with different sizes (from 5 to 500 KB), each one with a lightweight and a regular version. The User components are implemented as several CGIs that generate the HTML pages on the fly, and perform queries to the application machine’s engines to retrieve the necessary information. The replies consist of a number of recommended products or search result products. The generated pages include images and text. Finally, the Account components consist of dynamically generated pages requested over HTTPS (with text and media), where a lightweight and a regular version are available.

To perform adaptations, the change of component mode is achieved using the rewrite module of Apache web server. This module allows the requested URL to be rewritten on the fly. It allows us to add a mode, search or recommendation argument to the url, to control the component execution. For example, if the BusinessCatalog component is using the lightweight version, the module will add “low” as an argument to the url.
To demonstrate the advantages and analyze particular aspects of the proposed approach, the system was subject to two distinct overload scenarios. The first scenario employs a workload that causes CPU overload, while the second scenario’s workload causes an overload in terms of queries performed to the engines. Both experiments aim to validate the proposed approach. Namely, they allow us to illustrate the rule evaluation process and the flexibility and ease of using different policies, by comparing and quantifying the gains of adaptation. In the analysis of these gains, it is important to note that the needed impact varies from workload to workload. In some workloads, an adaptation with a less dramatic effect is sufficient, while in others a larger impact is needed. Similarly, the impact of an adaptation may be far greater than necessary, as it may be the only available adaptation or the only one with large enough impact to satisfy a goal. We describe in more detail each of the scenarios, the experiments goals and the obtained results below.

### 7.2 CPU Use Overload

This scenario allows us to illustrate how the system behavior changes in face of a significant increase in the number of requests made by clients, causing overload in terms of CPU use. It is also of interest to see if the system is able to return to the best service quality when the overload ceases. Components were initially deployed with a configuration that yields the best service quality: Catalog and Account web pages are served with regular quality, while User web pages have fresh recommendations and search results.

In this experiment, the system was subject to a workload that consists in four consecutive load steps: light (LW), medium (MW), heavy (HW), and light (LW). The light step allows all services to be offered with maximum service quality. The medium step requires the service quality to be lowered in order to respect the cpu_use threshold. The heavy step requires the system to operate with an even lower service quality. Finally, the light step brings the load back to the start, allowing the system to offer the best service quality again.

The three load steps include requests to all components. Table 5 presents the number of clients and the interval between requests to each component. The difference between load steps is the decrease in the interval time, thus, increasing the request frequency. Each load step consists of a collection of URLs that are requested by each client. These requests are submitted in a random order. Each client waits for a reply before sending another request. Our experiment used 90 clients running concurrently. The client ramp up takes 5 seconds. The clients start sending requests as soon as they start.

We measured the system performance and resource consumption without and with adaptation. The first case corresponds to an empty goal policy. The system never adapts and the load continuously increases, resulting in overload. In the second case, two analysis were made, one for each of the policies described in Section 6.4.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Clients</th>
<th>Catalog Interval (ms)</th>
<th>User Interval (ms)</th>
<th>Account Interval (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light</td>
<td>90</td>
<td>300</td>
<td>3500</td>
<td>3500</td>
</tr>
<tr>
<td>Medium</td>
<td>90</td>
<td>150</td>
<td>3500</td>
<td>3500</td>
</tr>
<tr>
<td>Heavy</td>
<td>90</td>
<td>150</td>
<td>3500</td>
<td>900</td>
</tr>
</tbody>
</table>

**TABLE 5**
Load steps used in the first experiment’s workload
Under the workload described above, the behavior of the system under policies A or B is relatively similar and, hence, we opt for presenting only the results obtained with policy A. This happens because the two policies only differ on the type of clients they favor (business/private) and since the change in the load imposed on the system in this experiment is very significant, in order to keep the CPU use below the threshold, the system is required to decrease the quality of service for both types of clients.

The results that are depicted in Figures 6, 7, and 8 compare the behavior of the non-adaptive and of the adaptive system in terms of CPU use and response time in face of the same workload. The change of load step is marked by vertical dashed lines, thus, we can observe the load and the resource consumption increasing until stabilizing, or decreasing when returning to LW in the end. The adaptations (when they occur) are marked by vertical full lines. Straight horizontal lines mark the goal limit for a particular KPI.

Concerning CPU consumption, Figure 6 shows that the system can sustain significant load increases at the expense of degrading the service quality, when its behavior is adaptive. The figure illustrates that the increasing load results in higher CPU consumption. If the system is not adaptive, the load will push the CPU use to the limit. This happens during the HW, where the system is already consuming all the available CPU for running components. On the other hand, if the system adapts, the CPU consumption can be maintained at a reasonable threshold, able to sustain any load peaks by degrading the service quality or offering the best service quality if the CPU consumption allows. The top of the plot shows the service quality evolution. The baseline is the service quality without adaptation, at its maximum value. The remaining lines are the service quality layered by business and private. The adaptations are triggered by violations of the limit_cpu goal.

The decision on how to adapt is made according to the current system state and strongly depends on the ranking of goals in the policy. The first adaptation (around minute 15) degrades the quality of business services qlt_bsn, and we can observe the CPU use slowly decreasing until it stabilizes below the limit. The second adaptation (around minute 28) degrades both qlt_bsn and qlt_prv. These adaptations avoid system overload. The last two adaptations take place at the final workload, returning the system to its best quality, which is visible for qlt_bsn around minute 39, and for qlt_prv around minute 48. Note that policy B (not depicted in the

![Fig. 6. Comparison of CPU consumption with and without adaptation](image-url)
Fig. 7. CPU consumption by component and overall

Fig. 8. Catalog and Account mean response time with and without adaptation

The policy) returns the system to the best quality using a different sequence of adaptations: since policy B favors private users, the effects on \( qlt_{_prv} \) appear before the effects on \( qlt_{_bsn} \). For the reasons explained before, this is the only difference between the effects of the two policies for this concrete workload.

To analyze in more detail the impact of adaptation in terms of CPU savings, Figure 7 shows the CPU use layered by Catalog and Account components. The User components are not depicted because the CPU consumption of these components is the same during the entire experiment. The overall CPU use is also included, so that it is visible when the \( cpu_{_use} \) goal is violated. We can observe that the CPU consumption of each component evolves through the sequence of load steps. The first adaptation changes the BusinessCatalog component to low mode, during the MW step. During the HW step, both BusinessAccount and PrivateCatalog components are adapted to cut down on CPU consumption. When the load decreases, the Business components are first returned to their best quality, followed by PrivateCatalog component. Note that only Catalog and Account components can be adapted to cut down on CPU consumption: the contribution of User components is disregarded, since the majority of computations are performed at the application machine.

Figure 8 compares the response times in both scenarios. The results show that by avoiding the overload
with adaptation it is possible to maintain the baseline response times. The response times are layered by 
*business* and *private* services and the same applies to the service quality. If the system is not adaptive, there 
is a clear increase in the mean response time when the load step changes from LW to MW, and from MW to 
HW. However, if the system is adaptive, when the system reacts to the violations of `limit_cpu` goal (addressed 
in Figure 6), it avoids resources exhaustion, maintaining the mean response time. This is particularly visible 
around minute 28, where the `mrt_bsn` suffers an accentuated decrease. There are less perceptive decreases 
also in `mrt_bsn` around minute 15 and in `mrt_prv` around minute 28. Therefore, there is an improvement in 
terms of response time at the expense of service quality. Avoiding the overload is not the only factor that 
favors better response times: the decrease in the amount of content to be sent to clients also contributes to 
this goal. Sending less content not only allows faster sending times, but also frees more resources. When the 
load decreases, in the last LW step, the value of the mean response time per request is similar to the initial 
one, due to the adaptations that improve service quality again.

### 7.3 Query Rate Overload

We also provide results for a different workload scenario to illustrate how the system behavior changes in face 
of an increase in the number of requests handled by the *User* components, causing overload in terms of query 
load. Furthermore, the experiment in this scenario also illustrates how, under certain workloads, policies A 
and B cause the system to adapt in different manners. Finally, the experiment highlights the trade-offs involved 
when setting the evaluation period for optimization goals.

In this experiment, the system was subject to a workload that consists of three consecutive load steps: LW 
step, *inter* step (IW) and MW step. The newly introduced *inter* step increases the load in terms of *User* requests, 
without violating the CPU use limit but making the engines *query_load* go beyond the established limit. As 
a result, service quality must be degraded to avoid overload. The IW step is characterized in Table 6. When 
compared to IW, the MW step increases the number of requests made to *Catalog* components, but decreases 
the number of requests made to the *User* components. This load step makes the system again degrade service 
quality to avoid overload, but, at the same time, also improve the service quality since the number of queries 
to engines no longer is close to the limit.

Figures 9 and 10 present the individual processing times for all *private* and *business* services for both goal 
policies. They show the evolution of *query_load*, and `mrt_prv` and `mrt_bsn` KPIs, on a component basis, through 
out the workload. Again, the vertical dashed lines mark a change in the workload, while the full vertical lines 
mark when adaptation takes place. The horizontal line marks the *query_load* limit.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Clients</th>
<th>Catalog Interval (ms)</th>
<th>User Interval (ms)</th>
<th>Account Interval (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter</td>
<td>90</td>
<td>300</td>
<td>2600</td>
<td>3500</td>
</tr>
</tbody>
</table>

**TABLE 6**

*Inter load step used in the second experiment's workload*
Figure 9 shows how the system performs under policy A. When the IW step starts there is an accentuated increase in the query load that leads to the violation of the limit query load goal and to the system adaptation. The selected adaptation degrades the qlt prv, showing the preference for business users. When the workload goes from IW to MW, the limit cpu goal is violated and the system adapts as in the first experiment.

Figure 10 shows the performance under policy B. When the limit query load goal is violated, the selected adaptation degrades the qlt bsn instead of the qlt prv, reflecting the preference for private clients expressed in the policy. In the next workload, the limit cpu goal is violated and the system adapts as with policy A.

Recall that when an exact goal is violated, to correct the system behavior faster, only the adaptations that affect the violated KPI are considered. Any adaptations tied to optimization goals will wait for the next evaluation period. We can observe this phenomenon in this experiment: when the workload changes from...
IW to MW, the system does not converge immediately to the optimal configuration; only the adaptations that help in reacting to the limit_cpu violation are immediately applied, other adaptations reacting to the lower query rate are only applied when the optimization goals are next evaluated (not depicted in the figure).

The evaluation of optimization goals is performed according to the defined time period. The situation just described also illustrates the trade-off involved in the specification of such time periods. The time period should be large enough to prevent the system from consuming an excessive amount of resources, but small enough to avoid delaying too long the necessary adaptations to achieve the optimal behavior. The definition of the appropriate value for this parameter is an important part of the policy specification. The approach favors the correction of exact goals. Thus, when the system enters a state that violates an exact goal, the framework attempts to correct it as fast as possible. The framework is less eager with respect to optimization goals if the system is in a correct (but potentially sub-optimal) configuration. The larger the time period defined for the evaluation of optimization goals, the longer the system will take to reach the optimal behavior.

### 7.4 Scalability and Reaction Time

As mentioned previously, the size of the case study resulted from a difficult trade-off between the realism of the case study and the clarity of the exposition that can be provided in a paper. In this section, we report on the study conducted to evaluate the approach’s performance and scalability using larger case studies.

We extended the existing case study to allow another level in the mode of components Catalog and Account, now also accepting high. The number of adaptations increased to a total of 48, from the previous 32. The system was subject to the same workload, and despite some differences in the adaptations selected, the reaction times are depicted in Table 7 for the CPU overload scenario. As expected, the results show that there is an overall increase in the reaction times, a consequence of the larger number of adaptations and, consequently, adaptation sets to evaluate during the online phase. While, in the context of the case study this increase is not significative, these results alone do not allow to extract any conclusion about the scalability of the approach.

<table>
<thead>
<tr>
<th>Event</th>
<th>32 Adaptations Time (ms)</th>
<th>48 Adaptations Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>kpiAbove(cpu_use,0.36)</td>
<td>13.3</td>
<td>27.1</td>
</tr>
<tr>
<td>kpiAbove(cpu_use,0.36)</td>
<td>9.6</td>
<td>20.3</td>
</tr>
<tr>
<td>kpiIncrease(qll_bsn,300, true)</td>
<td>2.7</td>
<td>10.6</td>
</tr>
<tr>
<td>kpiIncrease(qll_prv,1300, true)</td>
<td>2.9</td>
<td>11.4</td>
</tr>
</tbody>
</table>

**TABLE 7**
Comparison between 32 and 48 adaptations’ reaction time during the cpu overload experiment.

Therefore, to better analyze the scalability of the proposed approach using larger case studies, we opted to use several artificial case studies (with adaptations and impacts randomly generated), instead of further extending the original case study. This offers two main advantages: to increase the adaptation specification size to virtually any size, while avoiding the effort and time necessary to devise the adaptations and their impacts. The devised artificial case studies rely on an adaptation policy with six goals, six components, and twenty KPIs. The size of the adaptation specification is configurable, as well as the maximum number of impacts per adaptation. The adaptations and their impacts are generated randomly, according to the configuration. The
The results show, as expected, that an increase in the number of adaptations also increases the reaction time. Up to 400 adaptations, the reaction time is negligible, but at 500 adaptations the reaction time becomes greater than the duration of an adaptation in the case study (at most one second). While, the prototype implementation could be improved to lower the reaction times, eventually, the reaction time will be too large. This limit can be pushed further by enriching the selection process with a number of techniques that cut back on reaction time at the expense of optimality. These techniques include choosing the first set that would improve the violated goal KPI; limit the number of sets to be evaluated; or use only major sets (only the larger sets, for instance, with adaptations A, B, and C available only evaluate the set ABC, eliminating all subset evaluations).

### 7.5 Policy Evaluation and Reaction Time

In this paper, we advocate the use of a two-phase approach to evaluate the goal policies. The offline phase identifies which adaptations correct each kind of violation of the policy goals. This selection relies on the static analysis of the impact functions and can become quite complex if we do not limit the classes of impact functions that can be used (e.g., to linear functions). The calculation of the viable combinations of adaptations, which can also be statically determined, is also performed in this phase. During the online phase, it is only necessary to get the actual system state and calculate the impact of each of the adaptations previously selected.

Alternatively, the policy evaluation could be carried out exclusively at runtime by calculating the impact of every adaptation that affects the KPI of the goal which is tied to the triggered event. The calculation of the viable combinations of adaptations that contribute to solve the violation, would be calculated afterwards. Given that this process is simpler, it is important to evaluate the performance gains achieved with the two-phase approach. In this section, we evaluate these gains in the context of the case study (the original with 32 adaptations) and of the artificial case studies introduced in the previous section.

We considered the events triggered during the workload sequences described in previous sections and measured the time necessary to decide how the system should adapt in each case, using the two-phase and the single-phase approaches. The results are presented in Table 9 for the cpu overload and query rate overload experiments. As it can be seen, for all events, gains can be obtained when using the two-phase approach. A large fraction of the work is performed offline, improving the response time of the system. The results also

<table>
<thead>
<tr>
<th>Number of Adaptations:</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Sets</td>
<td>95</td>
<td>539</td>
<td>2699</td>
<td>7055</td>
<td>15679</td>
<td>25199</td>
<td>125999</td>
</tr>
<tr>
<td>Reaction Time (s)</td>
<td>0.141</td>
<td>0.327</td>
<td>0.961</td>
<td>1.483</td>
<td>2.885</td>
<td>4.545</td>
<td>23.58</td>
</tr>
</tbody>
</table>

**TABLE 8**

Reaction times for different sizes of the adaptation specification
show that the gains obtained with the two-phase approach increase as the number of adaptations that have impact on the same KPI also increases. In the example, the fact that the number of adaptations that can be used to decrease \texttt{cpu\_use} is larger than the number of adaptations available for the other events is reflected in the larger difference between reaction times observed for the first instance of \texttt{kpiAbove(cpu\_use,0.36)} event. This difference is much smaller in the second instance of this event. This happens because, when the second instance of \texttt{kpiAbove(cpu\_use,0.36)} event is triggered, the system has already been subject to adaptation and, as a result, some adaptations available for dealing with \texttt{kpiAbove(cpu\_use,0.36)} event are not applicable anymore.

It is also of interest to see how the both approaches compare in terms of scalability. Table 10 depicts the reaction time of the single-phase approach for the artificial case studies. When comparing the results obtained from both approaches (see Table 8 for the two-phase approach), they show that the two-phase approach takes less time to find the best set, despite analyzing a larger number of sets. This outcome is due to the offline generation of viable combinations of adaptations, which allows to reduce the reaction time.

8 Discussion and Related Work

Our approach relies on the assumption that is possible to establish a reasonable approximation for the impact of adaptations on KPIs in terms of their current value. While exact functions tend to be quite complex, rough approximations might be good enough for the purpose at hand, as our case study shows. Also, although our examples do not illustrate this, nothing prevents impact functions from having context variables as input. For instance, the impact on the network utilization of an adaptation that would require notification to be sent to a set of listeners could be expressed as a function of the number of listeners, where \texttt{n\_listeners} is a context variable whose value is known at runtime: \texttt{Component.net\_uti+ = unit\_cost * n\_listeners}.

The analysis of the case study shows that the proposed approach is suitable for the evaluated target systems. The approach provides enough flexibility to change the system’s adaptive behavior, which is achieved by changing the goal policy. Therefore, the adaptation and self-management support does not require re-development. The proposed approach is also easy to extend. From our experience developing the case study,

<table>
<thead>
<tr>
<th>CPU Overload Events</th>
<th>Two-phase Reaction Time (ms)</th>
<th>Single-phase Reaction Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{kpiAbove(cpu_use,0.36)}</td>
<td>13.3</td>
<td>71.5</td>
</tr>
<tr>
<td>\texttt{kpiAbove(cpu_use,0.36)}</td>
<td>9.6</td>
<td>18.1</td>
</tr>
<tr>
<td>\texttt{kpiIncrease(qll_fun,300,true)}</td>
<td>2.7</td>
<td>10.7</td>
</tr>
<tr>
<td>\texttt{kpiIncrease(qll_priv,1300,true)}</td>
<td>2.9</td>
<td>10.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query Rate Overload Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{kpiAbove(query_load,1210)}</td>
</tr>
<tr>
<td>\texttt{kpiAbove(cpu_use,0.36)}</td>
</tr>
</tbody>
</table>

**TABLE 9**

Reaction times for the events triggered during cpu overload and query rate overload experiments

<table>
<thead>
<tr>
<th>Number of Adaptations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>300</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>600</td>
</tr>
<tr>
<td>700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
</tr>
<tr>
<td>399</td>
</tr>
<tr>
<td>2591</td>
</tr>
<tr>
<td>3435</td>
</tr>
<tr>
<td>6911</td>
</tr>
<tr>
<td>20647</td>
</tr>
<tr>
<td>38879</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reaction Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.115</td>
</tr>
<tr>
<td>0.394</td>
</tr>
<tr>
<td>1.911</td>
</tr>
<tr>
<td>2.826</td>
</tr>
<tr>
<td>21.981</td>
</tr>
<tr>
<td>719.541</td>
</tr>
<tr>
<td>1485.753</td>
</tr>
</tbody>
</table>

**TABLE 10**

Single-phase reaction times for different sizes of the adaptation specification
making the system evolve through the addition of new components or more adaptations to existing components can be done without need for a new goal policy. New KPIs and corresponding goals can be added to the goal policy without much effort. The description of adaptations is a more delicate issue. In some cases, the adaptation impact may be dependent on the system configuration prior to the system evolution. As a result, instead of a single adaptation, several descriptions may be necessary to cover all the different impacts. The estimation of an adaptation impact may also pose some challenges. For example, the exchange of components requires some experimental testing to quantify the impact of changing from one component to the other.

Still, we recognize that our assumptions may constrain the domain of applicability of the approach. However, it is important to recognize that other approaches also have their drawbacks. For instance, we have experimented with the use of machine learners to predict the behavior of adaptive systems and we were faced with the complexity of feature selection [15]. In the next paragraphs, we discuss the most relevant works related to our own. Some can be seen as complementary to the proposal advocated in this paper.

8.1 Deriving Low-level Policies from Goals

Work by Bandara et al [16] derives low-level policies from high-level goal policies, through policy refinement. The user is expected to provide a representation of the system description, namely, the properties and the behavior of system components (using Event Calculus [17]), together with a specification of the goal policy in temporal logic. The approach maps the abstract entities in goals to concrete system objects and devices, and relies on abductive reasoning to find which sequence of operations allows the goal to be achieved. This solution does not support optimization goals, nor graceful degradation, if it is not possible to achieve all goals. Furthermore, the goals only address component properties, excluding any global system properties. Another limitation is that it does not provide a means of deriving the correct value of a parameter in a set parameter adaptation. Finally, this approach does not allow to change the high-level goal policies during runtime.

Work by Sykes et al [18] also proposes a three-layered architecture to derive adaptation rules from high-level goals, specified with temporal logic. The approach automatically assembles a configuration of components and the necessary actions to achieve the goals, which is called a plan. This plan is built by searching breadth-first for paths that go from the current state to the desired state, where each transition is an action. The sequence of states that leads to the desired state results in a collection of actions. This approach is affected by the same issues mentioned in the previous approach. Nonetheless, it allows changes to the goals during runtime.

Both approaches lack the ability to balance conflicting goals, therefore, the trade-offs of performing an action. The work by Keeney and Wade [19] proposes an approach to combine low-level policies into a higher-level policy for multiple goals. However, this solution only addresses policy refinement, therefore, only the same type of goals (again, described in temporal logic) are supported.
8.2 Decision-making for System Adaptation

There are several approaches to deciding on how to adapt the system. One approach is to consider the system as a black box and use control theory and/or learning techniques [2], [20], [21] to derive adaptation policies. In general this approach is expensive and the resulting policy is only guaranteed to perform as expected for the same system 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. Furthermore, machine learning approaches may be inadequate to some scenarios, where the complexity of feature selection render the approach useless, thus not allowing to experimentally determine the best adaptation [15].

Another approach relies on the system architect or administrator specifying a low-level adaptation policy for the system manually based on her own knowledge on the system operation [22]. Typically, these policies consist of declarative Event-Condition-Action (ECA) [23] rules specifying how the system must adapt in the presence of specific events and conditions. Less complex policies may consist of declarative if-then rules, together with causal networks (graphical representations of the system), to form a model-based reasoning mechanism [24]. While it is true that the proposed approach relies in the knowledge of impacts of adaptations, so the ECA policies demand detailed knowledge of the impacts of adaptations. Furthermore, ECA policies are not an option when very large number of adaptations are used. As the complexity of the system composition increases, the task of specifying a low-level policy becomes harder and more error-prone. Often, it 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 [25] 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 [26], [27]. In fact, such techniques could be combined with our approach in case distributed coordination is required.

9 Conclusions and Future Work

This paper proposes a novel approach to managing adaptive behavior in composed software systems. This approach relies on information provided by each service designer regarding the impact of possible adaptations on the system KPIs. This information is key for the automatic offline generation of a set of rules equivalent to the policy that describes the intended system behavior. 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 one, each service configuration can be measured independently to quantify the impact of adaptation, and still work for different configurations or workloads. For another, the approach is also able to balance the trade-offs of different adaptations to achieve particular goals. The experimental results show that the approach
have greatly improved its overall quality.

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.

Acknowledgments
We would like to thank the anonymous reviewers for their constructive and detailed comments on the manuscript, which have greatly improved its overall quality.

REFERENCES


