Self-Management of Adaptable Component-Based Applications

Liliana Rosa, *Member*, *IEEE*, Luís Rodrigues, *Senior Member*, *IEEE*, Antónia Lopes, *Member*, *IEEE*, Matti Hiltunen, *Member*, *IEEE*, and 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.

Index Terms—Adaptive systems, self-management, autonomic computing, goal policies

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 [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 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 managers while allowing the low-level management details and decisions to be left to the system. We assume that the behavior of the system can be controlled by applying one or more component adaptations. Our approach leverages 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 produces images of regular quality and another, low quality images. The designer, knowing the implementation details, is fully aware of the tradeoffs 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 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

- *L. Rosa and L. Rodrigues are with INESC-ID and the Instituto Superior Técnico (IST), Universidade Técnica de Lisboa (UTL), Rua Alves Redol 9,1000-029 Lisboa, Portugal. E-mail: lrosa@gsd.inesc-id.pt, ler@ist.utl.pt.*
- *A. Lopes is with the Departamento de Informática, Faculdade de Ciencias, Universidade de Lisboa, Campo Grande, 1749-016 Lisboa, Portugal. E-mail: mal@di.fc.ul.pt.*
- *M. Hiltunen and R. Schlichting are with AT&T Labs Research, 180 Park Avenue, Building 103, Florham Park, NJ 07932. E-mail: {hiltunen, rick}@research.att.com.*

Manuscript received 26 Sept. 2010; revised 28 Feb. 2012; accepted 1 Apr. 2012; published online 4 May 2012.
Recommended for acceptance by E. Di Nitto.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-09-0288.
Digital Object Identifier no. 10.1109/TSE.2012.29.

0098-5589/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

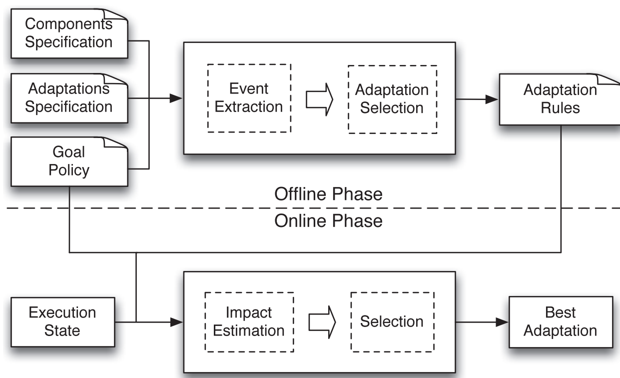

Fig. 1. Approach overview: offline and online phases.

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, 4, and 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 several aspects of the approach and Section 9 provides an overview of the related literature.

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 Fig. 1. Specifically, our approach takes a Components Specification (describing the components used in the system), an Adaptations Specification (describing the available components adaptations, explained in Section 3.3), and a Goal Policy (describing the desired system behavior, explained in Sections 3.1 and 3.2) 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 runtime when the system no longer behaves as desired, 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 the ranking of goals, that is, when it is not possible to fulfill all the goals, the goal with the lowest rank is violated first. Both offline and online phases are automatic with a few exceptions, which will be addressed throughout the description of the approach.

Even though the entire process could be done exclusively at runtime, the system reaction time (RT) can be reduced significantly by separating the evaluation in the two phases described above. Specifically, the first phase performs the analysis of the specifications and goals, producing a smaller set of possible adaptations that needs to be considered in the second phase. Therefore, the first phase is typically performed offline and the second phase online. However, the first phase may also be reexecuted online as a result of a change in the goal policy. In the next sections, we describe in detail these two phases, as well as the components specification, adaptations specification, goal policy, and performance metrics.

3 Adaptable Components and Goals

3.1 KPIs

We assume the desired system behavior can be described using a number of metrics that capture particular aspects of system performance. These metrics can address aspects such as the consumption of CPU or memory resources, user satisfaction (e.g., the response time), or service levels (e.g., SLAs), among others [2], [3], [4]. We call such metrics KPIs; 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.

The KPI definition includes the name, the combination function that states how the global value is estimated, the type of the expected value, and a relevance margin (*RMargin*) in any evaluation of the KPI. Any two values that differ from each other by less than the relevance margin are considered equivalent. Three examples of KPI definitions are presented below:

KPI *cpu_use*: double **CombFunc** Sum **RMargin** 0.01
KPI *mem_use*: double **CombFunc** Sum **RMargin** 0.01
KPI *resolution*: int **CombFunc** Sum **RMargin** 0

The first example states that system-level CPU usage KPI can be estimated as the sum of all components' cpu_use KPIs; also any two values of the KPI within 0.01 of each other are considered indistinguishable from the point of view of rule evaluation. The relevance margins could be

automatically calculated from the KPI's range, but we believe it is more useful to allow the operator to express what he or she views as a useful error for each defined KPI. The remaining examples describe the overall system memory consumption and overall system image quality.

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 monotonically nondecreasing *combination function* CF of $\{c.KPI : c \textit{ is in the current system configuration}\}$. The requirement for monotonicity for the combination function means that an increase in a component's KPI ($c.KPI$) either has no effect or increases the system-level KPI. Similarly, a decrease in a component's KPI causes a decrease in the system-level KPI or its value remains the same. 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, assume an adaptation of a component c that increases $c.cpu_use$. The system-level value of cpu_use will also increase and can be estimated as the sum of the components' CPU usages. The sum, average, and maximum are three common nondecreasing monotonic functions in the definition of the system-level KPIs.

3.2 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: 1) the KPIs relevant to the policy, and 2) 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 a *relevance_margin*:

CKPI ckpi_name $= JF(kpi1, kpi2, \ldots)$ **RMargin** relevance_margin
CKPI gdev $= 0.5 * |cpu_use - 0.6| + 0.5 * |mem_use - 0.4|$ **RMargin** 0.1

As an example, consider the specification of the CKPI *gdev* (above) that measures the weighted deviation from target CPU and memory utilization values. The 0.6 value is the target utilization for *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 *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 goal_name: kpi_name **Above** thr_lower
Goal goal_name: kpi_name **Below** thr_upper
Goal goal_name: kpi_name **Between** thr_lower thr_upper

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.

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 goal_name: kpi_name **Close** target **MinGain** mgvalue **Every** period
Goal goal_name: **Minimize** kpi_name **MinGain** mgvalue **Every** period
Goal 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 toward 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 relevance margin specified for the target KPI are considered indistinguishable, the *mgvalue* should always be larger than the relevance margin.

3.3 Specification of Component Adaptations

The adaptive behavior of a system is described using adaptations that change component parameters or the system configuration. The former tunes component parameters, while the latter replaces component implementations. The adaptations are specified in an *adaptations specification*, made in the context of a *components specification*. Both are described next.

The *components specification* describes the components available for use in the application. They are defined in terms of a type hierarchy reflecting the *is a* relationship, taking into account the functionality provided by the components [5]. This type of hierarchy supports multiple inheritance. Component types can be concrete, designating a specific component with an available implementation, or abstract, representing the characteristics of a group of components. A component's parents are defined using a *subtype* relationship. Below is the specification of a component that provides product webpages from a catalog, with *ImageQuality* controlling the image resolution:

Component Catalog
subtype StaticContentComponent
Parameters
ImageQuality:{low,regular}

The *adaptations specification* describes all the component adaptations that can be used to change the system behavior.[1] However, an adaptation is only applicable if

1. Note that at this stage, we do not support a parametric description of adaptations.

the target component actually exists in the current system configuration. The specification of an adaptation includes:

1. the concerned component,
2. the adaptation action(s) to be performed,
3. constraints such as the required component state or other adaptations that have to be performed simultaneously,
4. the impact of the adaptation on each KPI, and
5. the stabilization period for the adaptation.

We address them one by one next.

The target component of an adaptation must exist in the components specification. An adaptation may refer to an abstract component, targeting all the concrete components that are a subtype of it, or a single concrete component. The adaptation describes the adaptation actions targeting that component. To apply the adaptation actions, the specification may also impose some requirements. For instance, it may require that a specific component be present in the system, that the system be in a particular state, or that some other adaptation must be executed with the given adaptation. We assume that meta-information about the deployed and executing components, as well as the current value of their parameters, is available at runtime so that any requirements may be checked. If the component exists and the requirements are satisfied, the actions can be performed.

The specification of an adaptation also includes the expected impacts of performing the adaptation actions. The impacts are specified using *impact functions* over the affected KPIs. If a KPI is omitted from the impacts, it means that the KPI is not affected. We assume that it is possible to define an impact function that provides a reasonable approximation of the impacts of executing the set of actions over a component c on each *c.KPI*. 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. As a result, we do not expect impact functions to be 100 percent accurate; some error is often acceptable for the predicted impacts, but it is fundamental that they correctly capture the direction of the change (increase or decrease). At runtime, if the observed impact does not match the impact estimated by the function, this fact is simply recorded in a log; the use of techniques (such as reinforcement learning) to improve the estimators at runtime is complementary to our approach and this is being addressed by other members of our research group [6], [7].

When the impact of an adaptation 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 requirements and impact functions. If the impacts on different KPIs are not independent, each impact function 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 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 in page-faults). The problem of deriving the impact functions for each adaptation is outside the scope of this paper (several approaches in the literature can be applied [2]).

The last element of an adaptation is the *stabilization* period. It refers to the time that must be allowed for an adaptation to take full effect, before subjecting the system to reevaluation. This period depends on the components affected and the type of adaptation actions. When a combination of adaptations is selected, the stabilization period in effect will be the maximum of all the stabilization periods.

The example below shows an adaptation that changes the image quality of a Catalog component to low, using a *setParameter(ImageQuality, low)* action. The application of this adaptation requires that the *imageQuality* is *regular*, so it can be reduced. In terms of impacts, the adaptation reduces the CPU use (*Catalog.cpu_use*), memory use (*Catalog.mem_use*), and image resolution (*Catalog.resolution*).

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

Note that while this adaptation changes the image quality from regular to low, it is possible to define a reverse adaptation that changes the image quality from low to regular; such an adaptation would have the inverse impacts: $Catalog.cpu_use *= 1.92$, $Catalog.mem_use *= 1.21$, and $Catalog.resolution += 1$. Whenever a reverse adaptation exists and its specification can be automatically derived, the adaptation is simply marked as *reversible*. While this helps to reduce the developer's effort of describing adaptations, the developer may still need to provide the reverse impact functions when they are too complex to be obtained automatically. One can opt not to mark an adaptation as *reversible* and explicitly provide the specification of reverse adaptations. An adaptation may have more than one reverse adaptation, too. For instance, if a component's parameter can take any of N states, then $N - 1$ reversible adaptations may be required to cover all the possible parameter adaptations.

The adaptations specification may also include additional constraints, specified through a listing of the adaptations that cannot or must 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. In summary, the complete

adaptations specification consists of the adaptations, the conflicts, and the dependencies.

Conflict name
Adaptations (cmpA.adapt, cmpB.adapt)
Dependency name
Adaptations (cmpA.adapt, cmpB.adapt)

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 set of adaptations to be executed is chosen 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 **When** e **Select** $\{AS_1, \ldots, AS_n\}$. The *When* clause defines the *event* that triggers the evaluation of the rule. The event may be a sporadic event, triggered when some KPI exceeds a threshold, or a periodic event, triggered by the passage of time. In any case, each event is associated with a KPI or CKPI of a particular goal. The *Select* clause lists alternative sets of adaptations AS_i to address this event. These sets are the viable combinations of all relevant adaptations, i.e., 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 actual impact of an adaptation offline, in many situations we can assess if it will increase or decrease a KPI. For instance, if a goal determines that a KPI must be above a threshold, only the adaptations that increase the KPI have a positive impact.

Assessing the usefulness of an adaptation offline depends of the impact function. In some cases, static analysis of the specified function can determine if the impact is positive or not. If it is impossible to determine automatically the direction of change for an impact function, this information can be provided by the component developer. In all scenarios where the usefulness of the adaptation cannot be assessed, for instance, if the impact functions depend on context information or when the event refers to a CKPI, the impact is marked as *undetermined*. When the adaptation is marked as undetermined, the adaptation is always included in the adaptation set to avoid eliminating potentially useful adaptations. This guarantees that only adaptations known not to help achieve the goal are discarded in the offline phase.

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.

During the rule generation process, a *human-readable* representation is created. This representation informs the system designer about the adaptations that are available to achieve each goal. Since the adaptation descriptions are provided by component developers, the system designer may not be aware of the available adaptations. The human-readable format allows the designer to identify goals that are misdescribed or goals that cannot be met using the set of available adaptations. As depicted below, this representation includes the event, a list of all relevant adaptations for the situation signaled by the event, and the pairs of conflicting and dependent adaptations.

TABLE 1
Event Types Generated for Each Type of Goal

Goal	Event 1	Event 2
Above	kpiBelow(kpi,x)	-
Below	kpiAbove(kpi,y)	-
Between	kpiBelow(kpi,x)	kpiAbove(kpi,y)
Close	kpiIncrease(kpi,θ,cnd)	kpiDecrease(kpi,θ,cnd)
Maximize	kpiIncrease(kpi,θ,cnd)	-
Minimize	kpiDecrease(kpi,θ,cnd)	-

When event
Adaptations:C1.A,C1.B,C2.X,C2.Y,C3.Z
Conflicts:(C1.A,C2.X) **Dependencies**:(C2.Y,C3.Z)

4.1 Event Extraction

Event extraction is the first task of rule generation. We assume the existence of a monitoring system that generates both the 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 $kpiIncrease(kpi, \theta, condition)$ and $kpiDecrease(kpi, \theta, condition)$ 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 monitoring system. 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. Fig. 2 exemplifies how events are extracted from some goals.

For an *Above* goal, an event of type *kpiBelow* needs to be triggered when the value of the KPI falls below the specified threshold by a margin greater than the KPI relevance margin. Similarly, for a *Below* goal, an event of type *kpiAbove* needs to be triggered when the value of the KPI exceeds the specified threshold. Since *Between* goals are a combination of the *Above* and *Below* goals, both previous events are needed. For the *Minimize/Maximize* goals, a periodic event of type *kpiDecrease*/*kpiIncrease*, respectively, needs to be triggered with the period specified in the goal. Finally, for the *Close* goals, both of these two events are possible. The triggered event depends on which condition is true: The *kpiIncrease* event has a condition that the KPI value has to be

```
Goal cpu_reserve: cpu_use Below 0.35
Event kpiAbove(cpu_use, 0.36)     //0.35+0.01

Goal target_cpu: cpu_use Between 0.3 0.4
Event kpiBelow(cpu_use,0.29)      //0.3-0.01
Event kpiAbove(cpu_use,0.41)      //0.4+0.01

Goal min_deviation: Minimize gdev MinGain 0.2
    Every 10
Event kpiDecrease (gdev, 10, true)

Goal target_cpu: cpu_use Close 0.3 MinGain 0.2
    Every 20
Event kpiDecrease(cpu_use,20,">0.31") //0.3+0.01
Event kpiIncrease(cpu_use,20,"<0.29") //0.3-0.01
```

Fig. 2. Example events extracted from goals.

"< *target−relevance_margin*," while the *kpiDecrease* event has the opposite condition "> *target+relevance_margin*." For each extracted event, a rule is created with the *When* clause stating the event as the trigger for the rule evaluation.

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 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 are analyzed to verify if they change the KPI in the right direction. For instance, consider a rule with an event *kpiAbove(cpu_use,0.36)* where the goal is to decrease the value of the *cpu_use*. The adaptation *ToLowQuality* adapts the *Catalog* component with impact $Catalog.cpu_use = Catalog.cpu_use \div 1.92$. 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 $x \div 1.92 - x$ is always negative, the adaptation *ToLowQuality* decreases the CPU utilization. The corresponding reverse adaptation will increase the CPU consumption. The latter adaptation would be useful, for instance, to address the event *kpiBelow(cpu_use,0.29)* described before.

Once all the relevant adaptations for dealing with an event have been selected, the set of viable combinations of these adaptations is calculated. For a combination to be viable, it must respect all adaptation requirements and any specified interadaptation dependencies. Also, an adaptation is not considered viable if it violates any declared conflicts. Furthermore, to simplify the algorithm, we consider two adaptations that impact the same $c.KPI$ to be conflicting. We take this as an opportunity to significantly reduce the number of combinations that have to be compared at runtime. As a result, if the event concerns a KPI, because all selected adaptations have impact on that KPI, the viable combinations of adaptations have at most one adaptation per component. If the event concerns a CKPI, viable combinations of adaptations have 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 growth rate of the sets of adaptations as well as the number of the generated rules is 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 therefore it is necessary to evaluate the rule at runtime to choose the set of adaptations to perform. The evaluation of a rule **When** e **Select** $\{AS_1, \ldots, AS_m\}$ 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 policy goals.

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 nonapplicable adaptations from every AS_i. Then, the rule evaluation proceeds by searching for the combinations that *best match* the goals expressed in the adaptation policy, relying on the current system state.

While many different optimization criteria could be used in the selection process, our ranked-eager algorithm ensures that more important goals are met first, even if the total number of goals met is not maximized. The selection process starts by picking the sets of adaptations that are expected to satisfy the highest ranked goal k, then among those are selected the sets of adaptations that are expected to satisfy the second ranked goal $k+1$, and so on. There is one exception: If all sets selected in step k violate the $(k+1)$st goal, all sets are selected. In this manner, if it is not possible to satisfy the $(k+1)$st goal without violating the kth goal, the remaining goals are still used for tie-breaking. This selection mechanism is illustrated in Fig. 3.

An optimization goal specifies a total order between the possible values of a KPI. When several adaptation sets are evaluated against an optimization goal, those that put the KPI closer to the target specified in the goal are selected. This is illustrated by Fig. 4, where the best set of adaptations among those evaluated against the optimization goal B is represented by a pentagon shape—the AF set. Dashed lines are used to link adaptation sets that are considered equivalent with regard to goal B (the difference between their estimated impacts on the KPI of goal B is smaller than the relevance 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 closer to the target, and the other set is an equivalent one. The selection process is described in detail below.

The process starts with a search space $\mathcal{SS} = \{AS_1, AS_2, \ldots, AS_m\}$. 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. Let $[G_1, \ldots, G_n]$ be the list with the policy goals in the rank order (and, hence, G_1 is the goal with the highest rank and G_n is the goal with the lowest 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 that allows only a

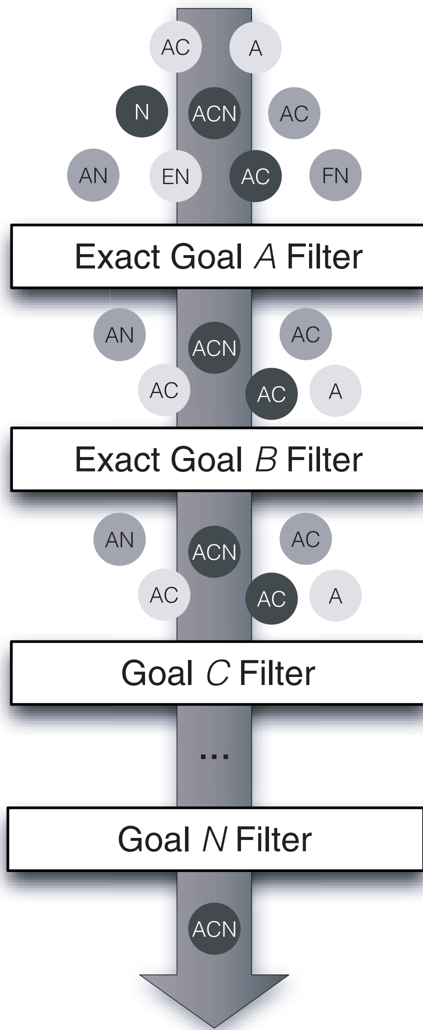

Fig. 3. Evaluation using the rank-eager criterion.

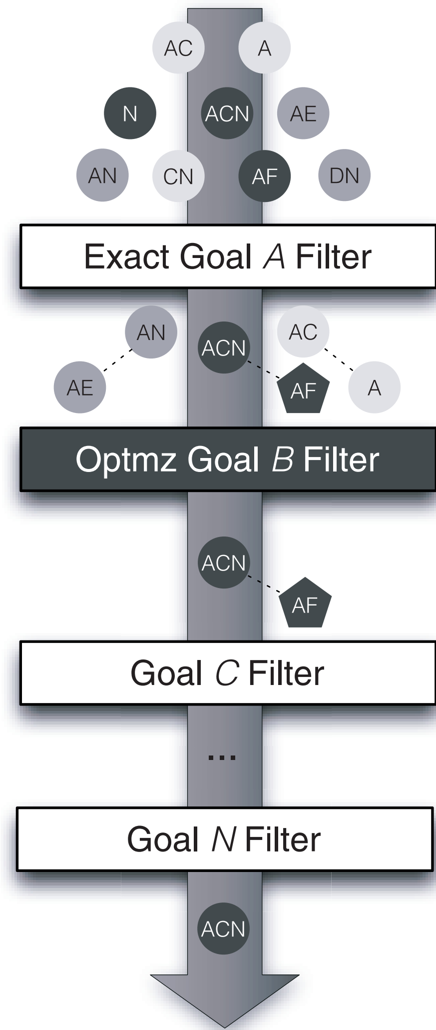

Fig. 4. Evaluation of optimization goals.

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_i associated with the goal. This value, which we denote by KPI_i^C, 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 noncomposite KPIs and, hence, the value of KPI_i^C is obtained after calculating the estimated impact of C in these KPIs.

The notion of C *matches a list of goals* $[G_1, \ldots, G_n]$ is defined inductively in the structure of the list as follows:

- C *matches* [ ].
- C *matches* $[G_1, ..., G_{k-1}, G_k]$, with $k >= 1$, if and only if the three following conditions hold:
  - C matches $[G_1, \ldots, G_{k-1}]$.
  - If the last goal of the list, G_k, is an exact goal, then KPI_k^C satisfies G_k or, for all other combinations C^+ in $\mathcal{SS}$ that match $[G_1, \ldots, G_{k-1}]$, $\mathrm{KPI}_k^{C^+}$ also violates G_k.
  - If the last goal of the list, G_k, is an optimization goal, then

$$|KPI_k^C - KPI_k^{C^*}| < \text{relevance_margin}^{KPI_k},$$

where C^* is, among the combinations in $\mathcal{SS}$ that match $[G_1, \ldots, G_{k-1}]$, the one that puts the KPI_k closer to the target specified in G_k.

C *best matches* $[G_1, \ldots, G_n]$ if and only if 1) C *matches* $[G_1, \ldots, G_n]$ and 2) 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*. 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 nonempty 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 with the two nonempty 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 makes it possible to describe scenarios where we want to give preference to an optimization goal; hence, the system adaptation is 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 product catalog, register for an account, and shop online. The website follows a multitier architecture and supports two types of users: private and business. Private users are consumers who buy products directly from the retailer. Business users are professional business users or other wholesalers who 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 back end. These components deal with different types of content: static, dynamic, secure, and nonsecure. Static content is retrieved directly, while dynamic content is generated upon request. The content is transferred through a nonsecure connection unless sensitive data are involved.

6.1 Adaptable Components

In this case study, we focus on the components running in the back end, namely, the adaptable components on which the web application relies: Catalog, User and Account.

The *Catalog* component provides the product description pages, which are static content webpages sent in a nonsecure manner to the client. This component is divided into *PrivateCatalog* and *BusinessCatalog*, for private and business users. 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 webpages customized for each user. These webpages are transmitted in a nonsecure manner. The component generates webpages with personalized product recommendations and searches when the user is logged in. The component is divided into *PrivateUser* and *BusinessUser* components, which rely on search and recommendation engines [8], [9]. Both engines can execute in two modes: *fresh* and *cache*. 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 [10]. 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 into *PrivateAccount* and *Business Account* components, the latter including invoice and budget management features. Both components have two configuration modes concerning the image quality (*regular/low*). The complete 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}
```

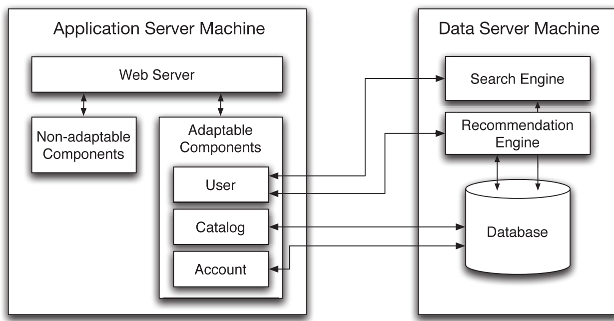

Fig. 5. The website's elements distribution in the back end.

Component BusinessAccount **subtype** Account
Component PrivateAccount **subtype** Account

The website relies on 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 separate machines to run the application and data servers, as illustrated in Fig. 5. The first machine runs a web server and the set of adaptable 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 websites face variable workloads, with predictable increases in the load, such as holiday season, but also with unexpected peaks, such as flash crowds [11]. The adaptive behavior in this case study aims at avoiding overload and maintaining an appropriate balance between resource consumption and service quality.

We consider that the system load is reflected by the CPU time consumed (other metrics could be used; see, for instance, [2], [12]). 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 service quality KPIs refer either to all private or to all business services, which is reflected in the choice of the combination function. For instance, the use of *B_Sum* in *resolution_bsn* means that only the resolution of business services are accounted, while *P_Sum* used in *resolution_prv* accounts only for private users. The processing time of requests is also divided in business and private. This is the time necessary to build a reply to a request, excluding the sending time.

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 16 reversible adaptations and, hence, a total of 32 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, some 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 $\div=$ 2.01
BusinessCatalog.resolution_bsn $-=$1
BusinessCatalog.ptr_bsn $\div=$ 1.99
Stabilization:
period $= 60$ secs

Reversible Adaptation ToLightCatalogP
Component:
PrivateCatalog

TABLE 2
KPIs Used in the Case Study

Name	Type	CF	Range	RMargin	Description
cpu_use	double	Sum	0-1	0.01	CPU consumption
resolution_bsn	int	B_Sum	2-4	0	Image quality
resolution_prv	int	P_Sum	2-4	0	Image quality
recommend_bsn	int	B_Sum	1-2	0	Freshness of recommendations
recommend_prv	int	P_Sum	1-2	0	Freshness of recommendations
search_bsn	int	B_Sum	1-2	0	Freshness of searches
search_prv	int	P_Sum	1-2	0	Freshness of searches
ptr_bsn	double	B_Sum	$\mathbb{R}^+$	0.2	Processing time of requests
ptr_prv	double	P_Sum	$\mathbb{R}^+$	0.2	Processing time of requests
query_load	int	Sum	$\mathbb{N}_0$	10	Searches and recommendations
bsn_components	int	B_Sum	$\mathbb{N}$	0	Number of business services
prv_components	int	P_Sum	$\mathbb{N}$	0	Number of private services

Actions:
setParameter(mode,low)
Requires:
mode = = regular
Impacts:
PrivateCatalog.cpu_use $\div =$ 1.92
PrivateCatalog.resolution_prv $- = 1$
PrivateCatalog.ptr_prv $\div = 1.43$
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 an additional 14 reversible adaptations. 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 it 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, it 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

In this case study, the goal is to avoid system overload while offering the best possible service quality. Overloading the system may be a result of overload in the application server or in the data server. To measure and detect it in the servers, two different metrics are relevant: the CPU usage in the application server, and the number of queries made to the search and recommendation engines in the data server. In the application server, besides the adaptable components, there are other components which are not adapted but still consume CPU. This must be considered when establishing a limit to the CPU consumption of adaptable components to avoid overload. In the data server, overload is controlled by imposing a limit to the number of queries.

The components contribute in different manners to the overall service quality. *Catalog* and *Account* components can provide images with different quality, the higher the quality the better, but the tradeoff 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 use the CKPIs in Table 3. Content quality is described by the CKPIs *qlt_prv* and *qlt_bsn*, while response time is described by CKPIs *mrt_prv* and *mrt_bsn*, which give feedback on the processing time for requests, independently of the request distribution (IRD).

TABLE 3
CKPIs Used in the Case Study

Name	JF	RM
mrt_bsn	(2*ptr_bsn)/bsn_services	0.2
mrt_prv	(2*ptr_prv)/prv_services	0.2
qlt_bsn	resolution_bsn+recommend_bsn+search_bsn	0
qlt_prv	resolution_prv+recommend_prv+search_prv	0

We have considered two different policies 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.

Policy A:
Goal limit_cpu : *cpu_use* **Below** 0.35
Goal limit_query_load : *query_load* **Below** 1200
Goal limit_mrt_bsn : *mrt_bsn* **Below** 1.6
Goal max_qlt_bsn : **Maximize** *qlt_bsn* **MinGain** 1
Every 900
Goal limit_mrt_prv : *mrt_prv* **Below** 1.9
Goal max_qlt_prv : **Maximize** *qlt_prv* **MinGain** 1
Every 1300

Policy B:
Goal limit_cpu : *cpu_use* **Below** 0.35
Goal limit_query_load : *query_load* **Below** 1200
Goal limit_mrt_prv : *mrt_prv* **Below** 1.9
Goal max_qlt_prv : **Maximize** *qlt_prv* **MinGain** 1
Every 1300
Goal limit_mrt_bsn : *mrt_bsn* **Below** 1.6
Goal max_qlt_bsn : **Maximize** *qlt_bsn* **MinGain** 1
Every 900

These policies reflect the insights provided by related research, including policies to achieve optimal resource use for web servers [2], [13], intermediary adaptation systems [12], [14], [15], and web server and user experience improvement [16]. 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 overload 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 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 than 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 more revenue at a given point in time.

TABLE 4
Events Extracted from the Goals

Goal	Event
limit_cpu	kpiAbove(*cpu_use*,0.36)
limit_query_load	kpiAbove(*query_load*,1210)
limit_mrt_bsn	kpiAbove(*mrt_bsn*,1.8)
max_qlt_bsn	kpiIncrease(*qlt_bsn*,900,true)
limit_mrt_prv	kpiAbove(*mrt_prv*,2.1)
max_qlt_prv	kpiIncrease(*qlt_prv*,1300,true)

6.5 Generated Rules

The given set of goals gives rise to several adaptation rules, as described in Section 4. 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, among other impacts. 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 when the *limit_cpu* goal is violated is the following:

When kpiAbove(cpu_use,0.36)
Adaptations:
ToLightCatalogB,ToLightCatalogP,ToLightAccountB, ToLightAccountP
Conflicts: none **Dependencies**: none

Overall, the offline phase generates six rules, one per goal. For each rule, the number of relevant adaptations range from four to 10. Each rule has at most 35 adaptation sets, each of which has up to four adaptations.

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 policies A and B are exactly the same. As discussed in Section 5, the goal ranking is taken into account at runtime when 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 push 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 server machine is a 8 $\times$ 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 data 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 data 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 us 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 back-end adaptable components are implemented as follows: The *Catalog* components are implemented using several HMTL 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 us to rewrite a URL on the fly, and thus 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

TABLE 5
Load Steps Used in the Experiments

Workload	Clients	Cat. Intv.	User Intv.	Acc. Intv.
Light	90	300	3500	3500
Medium	90	150	3500	3500
Heavy	90	150	3500	900

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 a large enough impact to satisfy a goal. We describe in more detail each of the scenarios, the experiments' goals, and results below.

7.2 CPU Use Overload

This scenario allows us to illustrate how the system behavior changes in the 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 load decreases. Components were initially deployed with a configuration that yields the best service quality: *Catalog* and *Account* webpages are served with regular quality, while *User* webpages have fresh recommendations and search results.

In this experiment, the system was subject to a workload that consists of four consecutive load steps: *light* (LW), *medium* (MW), *heavy* (HW), and finally back to a *light* step. 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 initial level, with the system offering the best service quality again.

The three load steps include requests to all components. Table 5 presents the number of clients and the interval (ms) 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, as the load increases, the system becomes overloaded. In the second case, two analyses were made, one for each of the policies described in Section 6.4. 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 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 Figs. 6, 7, and 8 compare the behavior of the nonadaptive 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. The horizontal lines mark the goal limit for a particular KPI.

Concerning CPU consumption, Fig. 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 over the desired level. This happens during the HW, where the adaptable components are already consuming all the allotted CPU for them. On the other hand, if the system adapts, the CPU use can be maintained at a reasonable threshold, able to sustain load peaks by degrading the service quality or offering the

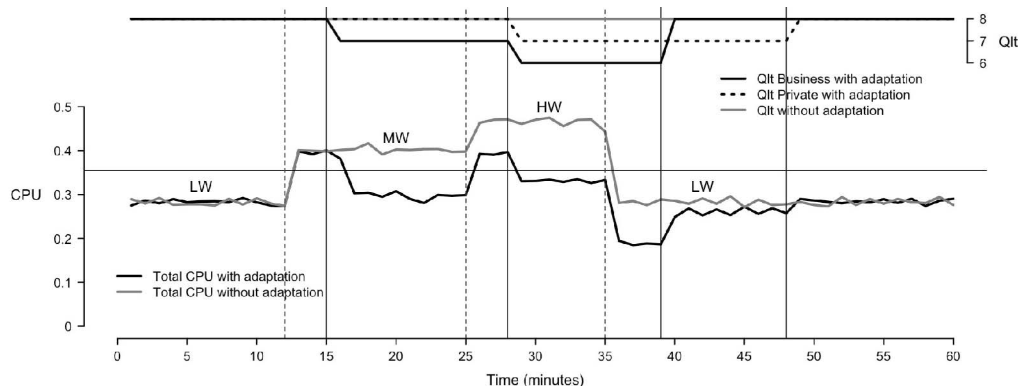

Fig. 6. Comparison of CPU consumption with and without adaptation.

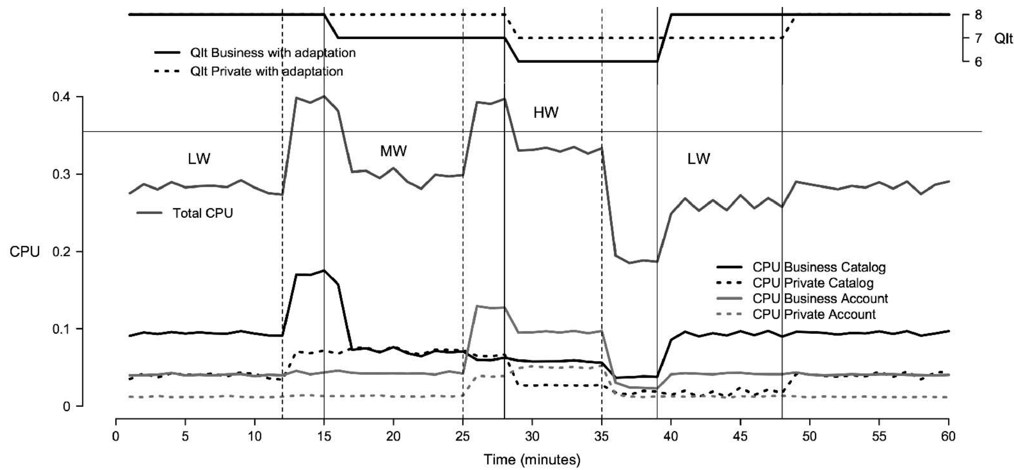

Fig. 7. CPU consumption by component and overall.

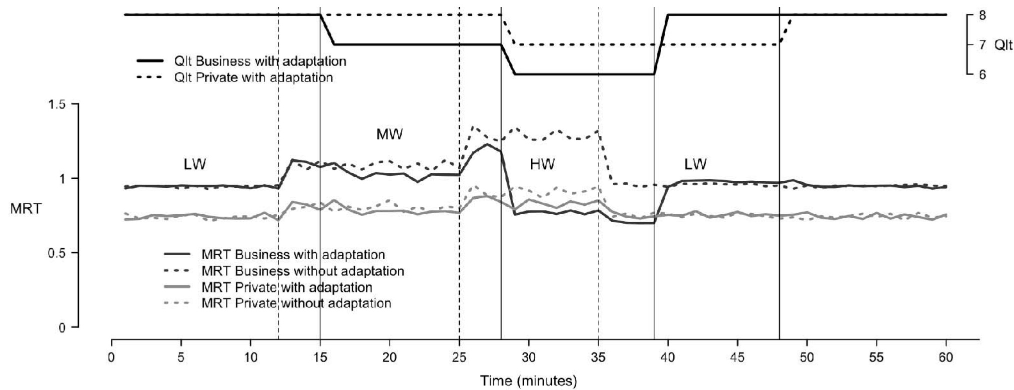

Fig. 8. *Business* and *Private* mean response time with and without adaptation.

best service quality if the CPU use 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 figure) 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 two policies for this concrete workload.

To analyze in more detail the impact of adaptation in terms of CPU savings, Fig. 7 shows the CPU use layered by *Catalog* and *Account* components. *User* components are not depicted because their CPU consumption 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 the *PrivateCatalog* component. Note that only *Catalog* and *Account* components can be adapted to cut down on CPU consumption as *User* components perform their computations at the data server machine.

Fig. 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 Fig. 6) it avoids resources exhaustion, maintaining the mean response time. This is particularly visible around minute 28 for *mrt_bsn*, which is reduced significantly. There are less significant decreases also in the *mrt_bsn* around minute 15 and in the *mrt_prv* around minute 28. Therefore, there is an improvement in terms of response time at the expense of service quality. Avoiding

TABLE 6
Inter Load

Workload	Clients	Cat. Intv.	User Intv.	Acc. Intv.
Inter	90	300	2600	3500

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 lower response 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 tradeoffs involved when setting the evaluation period for optimization goals.

In this experiment, the system was subject to a workload of three consecutive load steps: LW step, *inter* step (IW), and MW step. The new *inter* step increases the *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. After 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 further degrade service quality at the application server, but improves it at the data server since the number of queries is no longer close to the limit.

Figs. 9 and 10 present the individual mean response times for *private* and *business* services for both goal policies, showing the evolution of *query_load*, *mrt_prv*, and *mrt_bsn* KPIs, on a component basis, throughout the workload. Again, the vertical dashed lines mark a change in the workload, while the full vertical lines mark an adaptation. The horizontal line marks the *query_load* limit. 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. Under *policy A*, the selected adaptation degrades the *qlt_prv*, showing the preference for business clients. When the workload goes from IW to MW, the *limit_cpu* goal is violated and the system adapts as in the first experiment. Under *policy B*, when the *limit_query_load* goal is violated, the selected adaptation degrades the *qlt_bsn* instead, 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 tradeoff involved in the specification of such time periods. The period should be large enough to prevent the system from consuming an excessive amount of resources, but small enough to avoid delaying any adaptations necessary to achieve the optimal behavior for too long. Thus, its definition 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

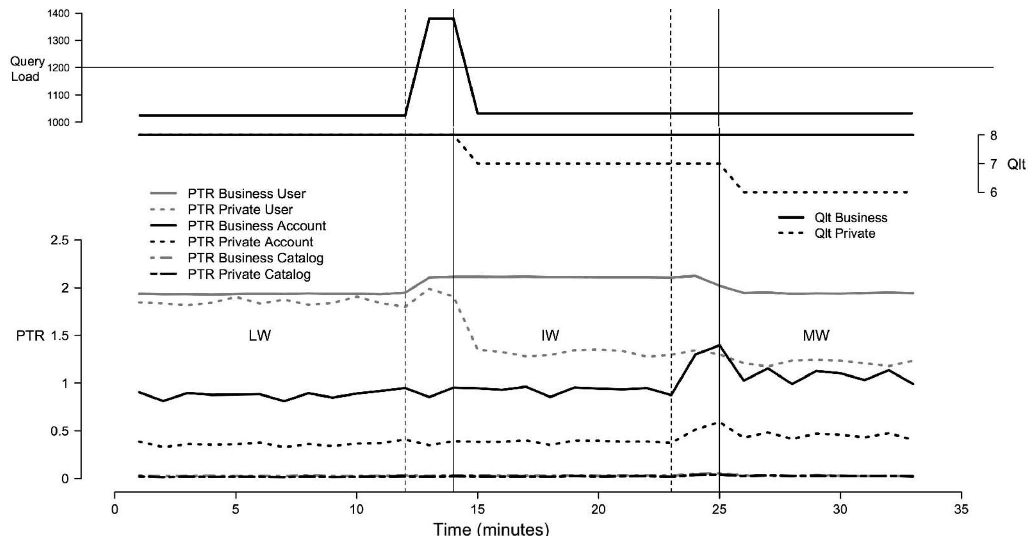

Fig. 9. Processing time for requests under the goal policy A.

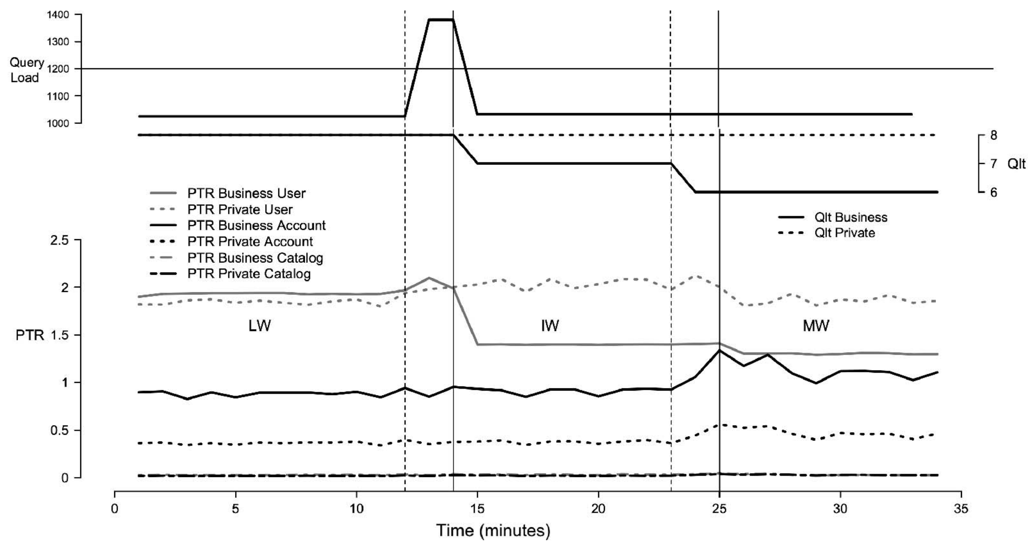

Fig. 10. Processing time for requests under goal policy B.

correct (but potentially suboptimal) 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

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*, also accepting *high*. The number of adaptations increased from 32 to 48. 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 significant, these results alone do not allow us to extract any conclusion about the scalability of the approach.

Therefore, to better analyze the scalability of the proposed approach in 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 adaptations specification size to virtually any size, while avoiding the effort and time necessary to devise the adaptations and their impacts. The policy for the artificial case studies has six goals, six components, and 20 KPIs. The size of the adaptations 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 number of adaptations ranges from 100 to 700. The number of impacts goes up to five impacts, on different KPIs. To evaluate the reaction time, we artificially trigger an event with a corresponding state (where the KPI value of the violated goal is out of the threshold). The results obtained are depicted in Table 8 and are organized by the size of the adaptations specification, showing the number of evaluated sets and the corresponding *Reaction Time* for the algorithm.

The results show that, as the number of adaptations increases, so does the reaction time to find the optimal solution. In fact, as depicted in Table 8, this time grows exponentially due to the increase in the number of sets of adaptations. However, in time constrained applications, one can often trade optimality for lower reaction times. In particular, when some exact goal is violated, what is important it is to quickly find a set of adaptations that corrects the problem. This can be achieved using a simple heuristic, as shown in Table 9. The *First* heuristic stops as soon as it finds a viable set that addresses the triggered event, instead of continuing the evaluation to find the optimal. Thus, substantially reducing the reaction times in the experiment (of course, in the worst-case scenario it will take as long as the regular algorithm). Although the research for more sophisticated heuristics is out of the scope of this paper, the results show that is possible to quickly find a set of adaptations that is able to address the state deviation identified, and search later for the optimal solution. For the latter purpose, our approach has the advantage of being automated and benefits from the offline preprocessing phase. Furthermore, the reaction times obtained for the different cases are substantially lower than any human reaction time.

7.5 Policy Evaluation and Reaction Time

We advocate the use of a two-phase approach. The offline phase identifies which adaptations may help to correct a policy goal violation. This selection relies on the static analysis of impact functions and can become quite complex

TABLE 7
Comparison between 32 and 48 Adaptations' Reaction Time (ms) in the *CPU Overload* Experiment

Event	Time (32 set)	Time (48 set)
kpiAbove(cpu_use,0.36)	13.3	27.1
kpiAbove(cpu_use,0.36)	9.6	20.3
kpiIncrease(qlt_bsn,900,true)	2.7	10.6
kpiIncrease(qlt_prv,1300,true)	2.9	11.4

TABLE 8
Times (ms) for Different Sizes of the Adaptations Specification

#Adaptations:	#Sets	Reaction Time	RT/Sets
100	95	0.141	0.67
200	539	0.327	1.65
300	2699	0.961	2.81
400	7055	1.483	4.76
500	15679	2.885	5.43
600	25199	4.545	5.34
700	125999	23.58	5.34

TABLE 9
Reaction Times (s) Using the *First* Heuristic

100	200	300	400	500	600	700
0.029	0.130	0.229	1.433	2.186	4.347	7.916

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 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 tied to the triggered event. The calculation of the viable combinations of adaptations that contribute to solving the violation would be performed afterward. 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 experiments of previous sections and measured the time necessary to decide to adapt the system in each case, using the two-phase and the single-phase approaches. The results are presented in Table 10 for the *cpu overload* and *query rate overload* experiments. As 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 show that the gains obtained with the two-phase approach increase as the number of adaptations that have impact on the same KPI increases. In the example, the fact that the number of adaptations that can be used to decrease *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 the *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 *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 *kpiAbove(cpu_use,0.36)* event are no longer applicable.

It is also of interest to see how both approaches compare in terms of scalability. Table 11 depicts the reaction time of the single-phase approach for the artificial case studies. The results obtained using both approaches (see Table 8 for the

TABLE 10
Reaction Times (ms) for the Experiments

CPU Overload Events	Two-phase	Single-phase
kpiAbove(cpu_use,0.36)	13.3	71.5
kpiAbove(cpu_use,0.36)	9.6	18.1
kpiIncrease(qlt_bsn,900,true)	2.7	10.7
kpiIncrease(qlt_prv,1300,true)	2.9	10.9
Query Rate Overload Events		
kpiAbove(query_load,1210)	12.5	29.2
kpiAbove(cpu_use,0.36)	13.1	32.7

TABLE 11
Single-Phase Reaction Times (s)

#Adaptations:	#Sets	Reaction Time
100	47	0.113
200	339	0.394
300	2591	1.911
400	3455	2.826
500	6911	21.981
600	20647	719.541
700	38879	1485.753

two-phase approach) 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 makes it possible to reduce the reaction time.

8 Discussion

8.1 On Rank-Based Policies

At first glance, the expressiveness of rank-based policies might seem not ample enough for expressing more complex scenarios such as $G_1 : k_1$ *Below* v_1 is more important than $G_2 : k_2$ *Below* v_2 and G_2 is more important than $G_3 : k_3$ *Below* v_3 but we prefer to have G_2 and G_3 both satisfied rather 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 $k_{23} = (v_2 \dot{-} k_2) * (v_3 \dot{-} k_3)$, a goal $G_4 : k_{23}$ *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 $signal(x) = 1$ 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: *Maximize* k_w *MinGain* 0 *Every 1200*. We chose 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. In a weight-based approach, the operator determines exactly the weight of each goal compared to others—a task that is more complex than simply identifying which goals are more important. Also, it is harder to verify if the given weights have the expected effect in terms of system behavior. Finally, extending a weight-based policy demands more effort than simply adding a new goal in a rank in the policy.

In terms of optimization criterion, we have opted to use a ranked-eager algorithm, which ensures that more important goals are met first even if the total number of goals met is not maximized. Other optimization algorithms could be devised. 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 satisfied (the ranking order of the goals would be ignored in this case).

8.2 On the "Between" Goal

The *Between* goal could be replaced by the conjunction of $k\ Above\ a$ and $k\ Below\ b$ using the goal $k_{ab}\ Above\ 0$, over the CKPI k_{ab} defined by $k_{ab} = (k \dot{-} a) * (b \dot{-} k)$, where $(x \dot{-} y) = (x - y)$ if $x > y$ and 0 if $x \leq y$. However, we opted to have explicit *Between* goals because they are easier to specify 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).

8.3 On Impact Functions

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 *n_listeners* is a context variable whose value is known at runtime: $Component.net_uti+ = unit_cost * n_listeners$.

8.4 Insights from Experience

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 policy. Therefore, the adaptation and self-management support does not require redevelopment. The approach is also easy to extend. From our experience developing the case study, new KPIs and corresponding goals can be added to the policy without much effort. It is also possible to make 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.

The approach is not applicable to systems with a component specification that might evolve during runtime; the dynamic creation and destruction of components of the types included in the component specification does not raise any difficulty in the selection algorithm. Rules are evaluated against the system state and the first step is to remove nonapplicable adaptations, namely, adaptations that target types of components that do not exist in the current system.

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 [6].

9 Related Work

Below, we discuss the most relevant works related to this work. Some can be seen as complementary to the proposal advocated in this paper.

9.1 Deriving Low-Level Policies from Goals

Work by Bandara et al. [17] 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 [18]), 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 the sequence of operations that 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 change of the high-level goal policies to be changed at runtime.

Work by Sykes et al. [19] 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 and thus the tradeoffs of performing an action. The work by Keeney and Wade [20] proposes an approach to combine low-level policies in a higher level policy for multiple goals. However, this solution only addresses policy refinement; thus only the same type of goals (again, described in temporal logic) are supported.

9.2 Decision-Making for System Adaptation

There are several approaches to deciding how to adapt a system. One approach is to consider the system as a black box and use control theory and/or learning techniques [2], [21], [22] to derive adaptation policies. In general, this is expensive and the resulting policy is only guaranteed to perform as expected for the 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 renders the approach useless, thus not allowing to experimentally determine the best adaptation [6].

Another approach relies on the system architect to manually specify a low-level policy for the system, using domain knowledge on the system operation [23]. Typically, they consist of Event-Condition-Action (ECA) [24] rules describing the adaptation of the system when specific events and conditions hold. 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 [25]. The ECA policies require detailed knowledge of adaptation impacts, similar to our approach. However, ECA policies are not an option when a very large number of adaptations is 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 between the adaptations available for all components. The Cholla system [26] 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 [27], [28]. In fact, such techniques could be combined with our approach if distributed coordination is required.

10 Conclusions and Future Work

We propose a novel approach to manage adaptive behavior in composed software systems. It relies on information provided by component developers regarding the impact of possible adaptations on the system KPIs. This information is key for the automatic offline generation of rules that describes the intended system behavior. The rules are then evaluated online to implement the adaptive behavior. Experimental results show that the approach is feasible and has several advantages. For one, each component configuration can be measured independently to quantify the impact of adaptation and still work for different configurations or workloads. For another, the approach is able to balance the tradeoffs of different adaptations to achieve particular goals. The results show that the approach considers not only how far the current state is from the optimal state—and, as a result, how large the impact has to be—but also uses each component's load to realistically estimate an adaptation impact.

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

Acknowledgments

The authors thank the anonymous reviewers for their valuable feedback. Also, they gratefully acknowledge the support of FCT (INESC-ID multi-annual funding) through the PIDDAC Program funds and project CMU-PT/ELE/0030/2009.

References

[1] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, "Generating Adaptation Policies for Multi-Tier Applications in Consolidated Server Environments," *Proc. Int'l Conf. Autonomic Computing*, pp. 23-32, 2008.

[2] Y. Diao, J. Hellerstein, S. Parekh, and J. Bigus, "Managing Web Server Performance with Autotune Agents," *IBM Systems J.*, vol. 42, no. 1, pp. 136-149, 2003.

[3] G. Tesauro and J. Kephart, "Utility Functions in Autonomic Systems," *Proc. First Int'l Conf. Autonomic Computing*, pp. 70-77, 2004.

[4] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-Based Self-Adaptation in the Presence of Multiple Objectives," *Proc. Int'l Workshop Self-Adaptation and Self-Managing Systems*, pp. 2-8, 2006.

[5] L. Rosa, A. Lopes, and L. Rodrigues, "Modeling Adaptive Services for Distributed Systems," *Proc. 23rd ACM Symp. Applied Computing*, pp. 2174-2180, 2008.

[6] M. Couceiro, P. Romano, and L. Rodrigues, "A Machine Learning Approach to Performance Prediction of Total Order Broadcast Protocols," *Proc. Fourth IEEE Int'l Conf. Self-Adaptive and Self-Organizing Systems*, pp. 184-193, 2010.

[7] M. Couceiro, P. Romano, and L. Rodrigues, "Polycert: Polymorphic Self-Optimizing Replication for In-Memory Transactional Grids," *Proc. 12th ACM/IFIP/USENIX Middleware Conf.*, vol. 7049, pp. 309-328, 2011.

[8] G. Lohse and P. Spiller, "Electronic Shopping," *Comm. ACM*, vol. 41, no. 7, pp. 81-87, 1998.

[9] B. Mobasher, R. Cooley, and J. Srivastava, "Automatic Personalization Based on Web Usage Mining," *Comm. ACM*, vol. 43, no. 8, pp. 142-151, 2000.

[10] G. Linden, B. Smith, and J. York, "Amazon.com Recommendations: Item-to-Item Collaborative Filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76-80, Jan./Feb. 2003.

[11] I. Ari, B. Hong, E. Miller, S. Brandt, and D. Long, "Managing Flash Crowds on the Internet," *Proc. 11th IEEE/ACM Int'l Symp. Modeling, Analysis, and Simulation of Computer Telecomm. Systems*, pp. 246-249, 2003.

[12] R. Grieco, D. Malandrino, F. Mazzoni, and D. Riboni, "Context-Aware Provision of Advanced Internet Services," *Proc. Fourth Ann. IEEE Int'l Conf. Pervasive Computing and Comm. Workshops*, pp. 600-603, 2006.

[13] T. Abdelzaher and N. Bhatti, "Web Content Adaptation to Improve Server Overload Behavior," *Proc. Eighth Int'l Conf. World Wide Web*, pp. 1563-1577, 1999.

[14] F. Mazzoni, "Efficient Provisioning and Adaptation of Web-Based Services," PhD thesis, Universita di Modena e Reggio Emilia, 2006.

[15] G. Iaccarino, D. Malandrino, and V. Scarano, "Personalizable Edge Services for Web Accessibility," *Proc. Int'l Cross-Disciplinary Workshop Web Accessibility*, pp. 23-32, 2006.

[16] S. Souders, "High-Performance Web Sites," *Comm. ACM*, vol. 51, no. 12, pp. 36-41, 2008.

[17] A. Bandara, E. Lupu, J. Moffett, and A. Russo, "A Goal-Based Approach to Policy Refinement," *Proc. Fifth IEEE Int'l Workshop Policies for Distributed Systems and Networks*, pp. 229-239, 2004.

[18] K. Eshghi, "Abductive Planning with Event Calculus," *Proc. Fifth Int'l Conf. Logic Programming,* pp. 562-579, 1988.
[19] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "From Goals to Components: A Combined Approach to Self-Management," *Proc. Int'l Workshop Software Eng. for Adaptive and Self-Managing Systems,* pp. 1-8, 2008.
[20] J. Keeney and V. Wade, "Towards Policy Decomposition for Autonomic Systems Governance by Applying Biologically Inspired Techniques," *Proc. IEEE Network Operations and Management Symp. Workshops*, pp. 309-313, 2008.
[21] K. Astrom, "Adaptive Feedback Control," *Proc. IEEE,* vol. 75, no. 2, pp. 185-217, Feb. 1987.
[22] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic, "Controlware: A Middleware Architecture for Feedback Control of Software Performance," *Proc. 22nd Int'l Conf. Distributed Computing Systems,* pp. 301-310, 2002.
[23] R. Bahati, M. Bauer, and E. Vieira, "Policy-Driven Autonomic Management of Multi-Component Systems," *Proc. Conf. Center for Advanced Studies on Collaborative Research,* pp. 137-151, 2007.
[24] D. McCarthy and U. Dayal, "The Architecture of an Active Database Management System," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 215-224, 1989.
[25] G. Wang, C. Wang, A. Chen, H. Wang, C. Fung, S. Uczekaj, Y.-L. Chen, W. Guthmiller, and J. Lee, "Service Level Management Using QoS Monitoring, Diagnostics, and Adaptation for Networked Enterprise Systems," *Proc. Ninth IEEE Int'l Enterprise Distributed Object Computing Conf.*, pp. 239-250, 2005.
[26] P. Bridges, M. Hiltunen, and R. Schlichting, "Cholla: A Framework for Composing and Coordinating System Software Adaptations," *IEEE Trans. Computers,* 2009.
[27] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building Adaptive Systems Using Ensemble," *Software Practice and Experience,* vol. 28, no. 9, pp. 963-979, 1998.
[28] W.-K. Chen, M. Hiltunen, and R. Schlichting, "Constructing Adaptive Software in Distributed Systems," *Proc. 21st Int'l Conf. Distributed Computing Systems,* pp. 635-643, 2001.

Liliana Rosa received the MS degree in computer science from the University of Lisbon in 2006 and is working toward the PhD degree in computer science at the Instituto Superior Técnico, Portugal. She is a researcher in the Distributed Systems Group of INESC-ID Lisboa. She is currently working in adaptive systems and autonomic computing. She is a member of the IEEE.

Luís Rodrigues is a professor *Catedrático* in the Departamento de Engenharia Informática, Instituto Superior Técnico, Universidade Técnica de Lisboa, and a senior researcher in the Distributed Systems Group of INESC-ID Lisboa. He works on distributed dependable systems. He is a coauthor of two books on distributed computing. He is a senior member of the ACM and the IEEE.

Antónia Lopes received the PhD degree in informatics from the University of Lisbon in 1999. She has been an associate professor at the University of Lisbon, Portugal, since March 2006. Her research interests include design principles, theories, and techniques that support the modeling and analysis of various types of software intensive systems, namely, service-oriented systems and self-adaptive systems. She is a member of the IEEE.

Matti Hiltunen received the MS degree in computer science from the University of Helsinki and the PhD degree in computer science from the University of Arizona in 1996. He is a principal member of the technical staff in the Dependable Distributed Computing and Communication Department at AT&T Labs-Research in Florham Park, New Jersey. His research interests include dependable distributed systems and networks, cloud computing, and adaptive systems. He is a member of the IEEE, ACM, the IEEE Computer Society, and IFIP Working Group 10.4 on Dependable Computing and Fault-Tolerance.

Richard Schlichting received the BA degree in mathematics and history from the College of William and Mary, and the MS and PhD degrees in computer science from Cornell University. He is currently an executive director of Software Systems Research at AT&T Labs in Florham Park, New Jersey. He was on the faculty at the University of Arizona from 1982-2000, and spent sabbaticals in Japan in 1990 at the Tokyo Institute of Technology and in 1996-1997 at the Hitachi Central Research Lab. He has served on the editorial boards of a number of IEEE magazines and journals, and has been on the technical program committees for more than 70 conferences and workshops. He is also the current chair of IFIP Working Group 10.4 on Dependable Computing and Fault-Tolerance, and has been active in the IEEE Computer Society Technical Committee on Dependable Computing and Fault Tolerance, serving as chair of that organization from 1998-1999. His research interests include highly dependable computing, distributed systems, and networks. He is a fellow of the ACM and the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.