Towards Knowledge Management in Self-adaptable Clouds

Michael Maurer, Ivona Brandic, Vincent C. Emeakaroha and Schahram Dustdar
Vienna University of Technology, Distributed Systems Group
Argentinierstraße 8, 1040 Vienna, Austria
{maurer, ivona, vincent, dustdar}@infosys.tuwien.ac.at

***Abstract*—Cloud computing represents a promising computing paradigm where resources have to be allocated to software that needs to be executed. Self-manageable Cloud infrastructures are required to achieve that level of flexibility on the one hand, and to comply to users' requirements specified by means of Service Level Agreements (SLAs) on the other. Such infrastructures should automatically respond to changing component, workload, and environmental conditions minimizing user interactions with the system and preventing violations of agreed SLAs. However, identification of system states where reactive actions are necessary for the prevention of SLA violations is far from trivial. In this paper we investigate how current knowledge management systems can be used for the prevention of SLA violations in Clouds. First, we define a typical SLA use case and formulate the expected behavior of the knowledge management system in order to prevent possible SLA violations. Second, we investigate different methods for the knowledge management, e.g., situation calculus and case based reasoning (CBR). We discuss how these methods match the expected behavior for SLA violation prevention. We in particular examine the CBR method and devise several approaches for the knowledge management in Clouds based on CBR. Finally, we evaluate our approach based on the presented use case.**

I. Introduction

Cloud Computing is a novel computing paradigm, which offers computational power similar to utilities like water, electricity or gas. In order to achieve such a level of flexibility, computing resources have to be allocated to software waiting to be executed in a not only dynamic, but also autonomous way. These resources are allocated according to predefined *Service Level Agreements (SLAs)*, which consist of SLA parameters like response time, availability or storage. A *Service Level Objective (SLO)* defines for every SLA parameter the goal to be achieved, e.g., availability $> 98\%$. Usually, if SLOs are violated, the Cloud provider has to pay penalties to the costumer, which are also stated within the SLA. In many cases simple reactions, like moving virtual machines (VMs) from already allocated nodes to additional available nodes can prevent the violation of established SLAs. However, identifying such reactions is far from trivial.

The positive effects of sensing possible SLA violations before they occur are twofold. On the one hand it helps the Cloud provider to save money (not paying penalties), and to minimize user (e.g., system administrator) interaction in the Cloud itself on the other. Furthermore, effective SLA management also shuts down unnecessary resources for fulfilling the SLA, and therefore is an important component of Green IT and management of energy efficiency [19] as well.

On the one hand, large body of work has been done for the development of SLA management frameworks, which react to SLA violations [6], [16], but do not try to prevent the occurrence of these violations. On the other hand, considerable body of work has also been done for the prevention of SLA violations [20]. These approaches, however, are very tied to a single special parameter, e.g., CPU utilization. However, SLA management that proactively acts even before SLA violations occur – in a general way not specific to any SLA parameter – is still an open research issue. Thus, the development of knowledge systems for the management of reactive actions for the SLA violation prevention is the first step towards achieving this goal.

In this paper we facilitate the usage of knowledge bases for the self-management in Clouds. Therefore, we define a use case considering a typical SLA example and employ it to evaluate existing knowledge management approaches. We evaluate rule based systems, default logic, situation calculus, and case based reasoning. We particularly focus on one of these approaches, *case based reasoning (CBR)*, and devise a concept for the management of measurement inputs using active and passive system observation methods. We present and discuss a concept and methods for the derivation or even avoidance of threat thresholds used to sense future SLA violations. Our development of the knowledge management systems for Clouds is embedded into the *Foundations of Self-governing ICT (FoSII)* infrastructure [2]. The *FoSII* project aims at developing an infrastructure for autonomic SLA management and enforcement. Thus, the knowledge management system presented in this paper represents the first building block of the *FoSII* infrastructure. Furthermore, we discuss implementation choices of the presented concepts for the knowledge management in self-adaptable Clouds and present a preliminary evaluation thereof.

The major contributions of this paper are: (i) a specification of a complete use case for a knowledge management system in Cloud Computing; (ii) a theoretical evaluation of some promising knowledge management methods to be used in the field of Cloud Computing; and (iii) a design, an implementation, and a preliminary evaluation of a CBR-aware knowledge management system.

The remainder of this paper is organized as follows: Section

II describes related work. In Section III we discuss the goals for the development of the knowledge management system, its relation to the *FoSII* project, and a use case. Section IV describes and theoretically evaluates knowledge management models to be applied in the field of Cloud Computing. Section V introduces a very promising method for knowledge representation and inference, CBR. Sections VI and VII show the implementation and a preliminary evaluation of CBR, respectively, whereas Section VIII concludes the work and gives an outlook and motivation for further research in this field.

II. Related Work

Currently most of the related work on the knowledge management in self-adaptable Clouds can be classified into the following groups: (i) work on SLA management which is however not preventive SLA management [12], [14], [16], [24]; (ii) knowledge management in general and not tied to Cloud computing [20], [23]; and (iii) autonomic management in various areas (e.g., SOA, workflow based systems, energy efficiency) [10], [17]–[19], [22], [25].

The Reservoir model [24] is a framework for Cloud Computing with the conceptual addition of SLA management. It states the need of dynamically adjusting resources (in addition to federating resources from peer providers) in order to meet SLAs, but does not specify a way to do that. Paschke et al. describe a rule based approach for dynamically dealing with SLAs in combination with ContractLog [23]. It specifies rules to trigger after a violation has occurred, but it does not deal with avoidance of SLA violations.

Some papers like [12] describe in detail the process of how to fulfill an SLA. In these papers, the SLA is very often limited to only one SLO and the analysis of this resource provisioning is closely tied to a special resource, e.g., CPU utilization. Others inspected the use of ontologies as knowledge bases only on a very conceptual level. [20] viewed the system in four layers (i.e., business, system, network and device) and broke down the SLA into relevant information for each layer, which had the responsibility of allocating required resources. Again, no details on how to achieve this have been given. The authors of [14] only describe the monitoring of SLAs.

Kephart et al. [18] get more specific when it comes to how to adjust resource allocations. They argue for avoiding a system based on action or goal policies, and opt for a utility-driven approach, where they give a detailed view on how to derive utility functions. However, it would be interesting to develop an automatic mapping of general SLAs to these utility functions, because as in [12], the authors only deal with one SLA parameter, response time, and relate it to the number of servers they use for satisfying a certain consumer load. Thus, the only actions to execute are *shut down server* and *start server*. Muthusamy at al.'s vision [22] is quite similar to our goals; yet, they do not investigate the use of a knowledge base.

Bahati et al. [10] also use policies, i.e., rules, to achieve autonomic management. They provide a system architecture including a knowledge base and a learning component, and

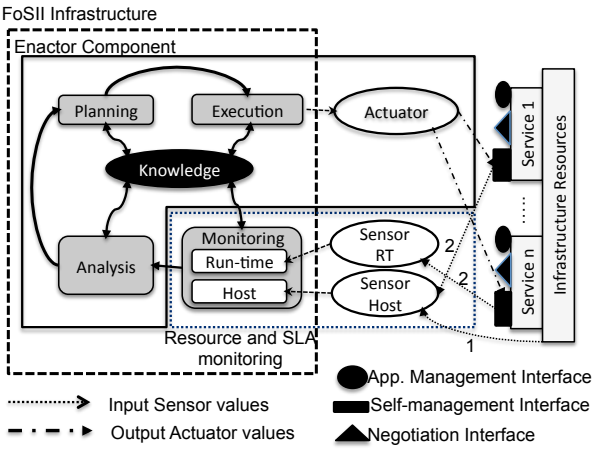

Figure 1. FoSII infrastructure

divide all possible states of the system into so called regions, which they assign a real value between 0 and 100 that signalizes the benefit of being in this region. The states themselves are derived from the SLA, i.e., response time > 500 (too slow), response time < 100 (too fast, consuming unnecessary resources) and $100 \leq$ response time ≤ 500, and the regions are therefore $0, 50$ and 100, respectively. With reward signals from the given metrics, the system learns whether different actions for one state were good or not. As in the other papers, this work deals with only one SLA parameter and a quite limited set of actions, and with violations and not with the avoidance thereof. However, combining the paper with the idea of threat thresholds (see section IV-A), could enhance this paper in the latter regard. Yet, the disadvantages of this approach as described in the same section remain upright.

Hasselmeyer et al. [16] introduce a Conversion Factory, which on a design level combines the SLA, the system status, and the Business Level Objectives to create Operational Level Agreements (OLAs), which govern system configuration. Whereas the idea seems promising, there are no details on how to achieve these mappings to OLAs. In several papers Yousif et al. [19], [25] present autonomic resource management as far as power consumption is concerned by using fuzzy logic containing IF-THEN rules, for instance. Heinis et al. [17] experiment with self-configuring thresholds, but tied to a workflow execution engine.

III. Background

As our solution towards knowledge management is an integral component of the *FoSII* project [2], in this section we present an overview of the *FoSII* infrastructure and its relation to the knowledge management methods discussed in this paper. Thereafter, we present a case study used for the evaluation of different knowledge management approaches.

A. FoSII overview

As shown in Figure 1 the *FoSII* infrastructure is used to manage the whole lifecycle of self-adaptable Cloud services

[11]. The management is governed by a Monitor-Analysis-Plan-Execute (*MAPE*) cycle, whose components will be explained in this subsection. Furthermore, each FoSII service implements three interfaces: (i) negotiation interface necessary for the establishment of SLA agreements, (ii) application-management interface necessary to start the application, upload data, and similar application management actions, and (iii) the self-management interface necessary to devise actions in order to prevent SLA violations.

As part of the Monitor phase of the MAPE cycle, the host monitor sensors continuously monitor the infrastructure resource metrics (input sensor values arrow *1* in Figure 1) and provide the autonomic manager with the current resource status. The run-time monitor sensors sense future SLA violation threats (input sensor values arrow *2* in Figure 1) based on resource usage experiences and predefined threat thresholds.

Threat thresholds (TT) as explained in [4] are more restrictive thresholds than the SLO Values. The generation of TTs is far from trivial and should be retrieved and updated by the knowledge management system, and only at the beginning be configured manually, as described later on in the paper, where we even investigate a variation of CBR getting rid of TTs at all.

Next, we describe the mapping between the sensed host values and the values of the SLA parameters. Resource metrics – being monitored by the *host monitor* with the help of arbitrary monitoring tools (e.g., Ganglia) – include e.g., down-time, up-time, available incoming and outgoing bandwidth. Based on the predefined mapping rules stored in a database monitored metrics are mapped to the SLA parameters. An example SLA parameter is service availability Av (as shown in Table I), which is calculated using the resource metrics $downtime$ and $uptime$ defined by the mapping rule

$$Av = (1 - \frac{downtime}{uptime}) \cdot 100. \quad (1)$$

The mapping rules are defined by the provider using appropriate Domain Specific Languages (DSL). Calculated SLA values are compared with the predefined TT in order to react before SLA violations happen. As described in [13] we implemented a highly scalable framework for mapping of resource metrics to SLA parameter facilitating exchange of large numbers of messages.

Once an SLA violation threat is detected, reactive action has to be taken in order to prevent the possible violation. The actions are chosen in the Analysis phase in correspondence with the knowledge base, which this paper focuses on. The next steps are planing the order and timing of the actions (Plan phase) and finally executing them (Execution phase) with the help of actuators communicating with the self-management interface of the resp. services.

B. Use Case

In this section we define a use case for the examination of the knowledge management methods. An example SLA is depicted in Table I. We consider four Service Level Objectives (SLOs): incoming bandwidth, outgoing bandwidth, storage, and availability. The corresponding SLO values are shown on the right hand side in Table I. In order to evaluate the knowledge management approaches we describe the status of the system in terms of running *physical machines (PMs)* and a specific application running under this SLA at three different time points t_1, t_2, t_3. We assume that one application runs on one virtual machine (VM), but one VM can run on (1,*) PMs for scalability and/or reliability issues, and on one PM, there can run (1,*) VMs. Table II summarizes the measured system states.

Service Level Objective (SLO) Name	SLO Value
Incoming Bandwidth (IB)	$>$ 10 Mbit/s
Outgoing Bandwidth (OB)	$>$ 12 Mbit/s
Storage (St)	$>$ 1024 GB
Availability (Av)	$\geq$ 99%

Table I
SAMPLE SLA

	IB	OB	St	Av	PMs
t_1	12.0	20.0	1200	99.50	20
t_2	14.0	18.5	1020	99.47	17
t_3	20.0	25.0	1250	99.60	19

Table II
SAMPLE SYSTEM STATES

We define the goals of the knowledge management system as follows:

1) *Continuous generation and improvement of SLA threat thresholds.* Based on historical observations the knowledge management system should generate threat thresholds for the particular SLOs - if necessary distinguishing between different application domains of the specific SLA parameter.
2) *Identification of reactive actions.* In case threat thresholds do not exist (e.g., if an application from a new domain is deployed or if we deliberately decided to omit them as described later), the knowledge management system should suggest reactive actions based on the historical system state.

Based on the SLA defined in Table I and on the system states measured and depicted in Table II we discuss how system reactions could be defined. We assume that the knowledge management system acts in mode *Identification of reactive actions* without a particular predefined threat threshold for the SLA violations. Thus, we define a set of actions the *Execute component* of the MAPE cycle (cf. Figure 1) is able to trigger:

1) for individual applications (=VMs[1]):
   a) Increase incoming bandwidth share by $x\%$.
   b) Decrease incoming bandwidth share by $x\%$.
   c) Increase outgoing bandwidth share by $x\%$.

[1]Applications can be identified with virtual machines, since we assumed a one-to-one mapping of applications and virtual machines

d) Decrease outgoing bandwidth share by $x\%$.
e) Increase memory by $x\%$.
f) Decrease memory by $x\%$.
g) Add allocated storage by $x\%$.
h) Remove allocated storage by $x\%$.
i) Increase CPU share by $x\%$.
j) Decrease CPU share by $x\%$.
k) Outsource (move application) to other cloud.
l) Insource (accept application) from other cloud.
m) Add physical machine
n) Remove physical machine

2) for physical machines (computing nodes):
a) Add x computing nodes
b) Remove x computing nodes

3) Do nothing.

Thus, at the specified time points, the knowledge management system receives the measurements as inputs and should output an action that can be executed in order to prevent an SLA violation.

IV. Methods of Knowledge Management for SLA Management

In this section we present some well known knowledge management methods and evaluate how they fit to SLA management in a Cloud-like environments as discussed in Section III-B.

A. Rule-based system

A rule-based system as Jess [5] or Drools [1] contains rules in the "IF *Condition* THEN *Action*" format, e.g.,

(1) IF IB $<$ TT_IB THEN Add physical machine to VM.
(2) IF IB $<$ TT_IB THEN Increase IB share by 5% for VM.
(3) IF Av $<$ TT_Av THEN Add 2 comp. nodes to the cloud.
(4) IF Av $<$ TT_Av THEN Outsource app. to other cloud.

As already explained before, we here use *threat thresholds* to trigger some action *before* an SLA is violated. There are two drawbacks to this mechanism, though:

First, the question of how these TTs are obtained, has to be answered. They are very different from one SLA parameter to another, e.g., for SLO Storage $>$ 1024 GB, the *TT* could be already at 1300 GB (127% of the original threshold), whereas for the SLO IB $>$ 10 Mbit/s the TT could be at 11 Mbit/s (110% of the original threshold), as one might say that reallocating bandwidth shares is a lot quicker than reallocating storage. They can even differ a lot for the same parameter in a different domain, e.g., the TT for availability in some medical domain, where human lives can be at stake, must be much higher than for a 3D rendering service in the architectural domain. A way to get around this would be to have the TTs specified in DSLs or to include them in the SLA document. However, this would heavily depend on subjective estimations. Nevertheless, it would be possible to find some experience values that make sense for the most common parameters. Furthermore, it has to be specified whether these thresholds are derived from a constant function of the parameter's value, i.e., always add 5 units to the SLA parameter value, a linear one, i.e., always add 10% to the value, or even an exponential or any other function. So to solve this in a universally valid way, one would have to find an appropriate function for every SLA parameter in every domain.

The second question is how to solve two contradicting rules. Consider rules (3) and (4) depicted at the begining of this section. If availability for a certain service drops below the pre-specified TT, should the rule engine rather add computing nodes or outsource an application, or both? Using a salience concept to decide this, leads to a difficultly manageable load of rules. A good examination of this problem can also be found in [18].

In our use case from Table II, the rules (1) - (4) above, and with $TT_{IB} = 12.5$ for incoming bandwidth and $TT_{Av} = 99.48$ for availability, the rule engine would fire rules 1 and/or 2 at time t_1; at t_2 it would fire rules 3 and/or 4, and at t_3 it would do nothing.

B. Default Logic

Default Logic [9] is a version of a rule-based system whose rules are no longer simple IF-THEN Rules, but can be described as *IF condition - and there are no reasons against it - THEN action*. We write such a rule as $\delta = \frac{\phi : \psi_1, \ldots, \psi_n}{\chi}$, where ϕ represents the condition, and χ is the action to execute, if the statements $\psi_1, \ldots, \psi_n$ are consistent with the current assumptions we hold of our system. A sample rule considering our case study can be written as

$$d_1 = \frac{IB < TT_{IB} : IncreaseIBshare}{IncreaseIBshare}. \qquad (2)$$

The rule means: *If incoming bandwidth is smaller than its threat threshold, and if there is no reason against increasing bandwidth share, then increase bandwidth share*. Reasons against could be that the bandwidth share is already at its maximum or that other (possibly more important) services issued a request for an increase at the same time. Contrary to ordinary rules in a rule-based system, it is easy for default rules to understand that resources cannot be increased indefinitely. However, default logic does not offer a remedy against the issues of retrieving TTs and dissolving contradicting rules.

Furthermore, default logic is especially used in fields with a lot of contradicting information. For Cloud Computing, however, we are rather interested in determining the reason of the current measurement information, e.g., why current incoming bandwidth has decreased. For example, we want to know whether the current bandwidth problem is caused by internal problems (e.g., too many service requests but too little resources provided), which the Cloud is capable of solving on its own, or by external factors (e.g., a DDoS attack), which cannot be influenced directly. Thus, we are rather confronted with more incomplete information than with contradicting one.

C. Situation Calculus

Situation Calculus [21] describes the world we observe in *states*, the so called *fluents*, and *situations*. *Fluents* are first-

order logic formulae that can be true or false based on the situation in which they are observed. *Situations* themselves are a finite series of actions. The situation before any action has been performed - the starting point of the system - is called the initial situation. The state of a situation s is the set of all fluents that are valid in s. Predefined *actions* can advance situations to new ones in order to work towards achieving a pre-defined goal by manipulating the fluents in this domain. For a world of three bricks that can be stacked upon each other lying on a table, fluents are quite easy to find: First, a brick can be on the table or not. Second, a brick can have another brick on it or not. Third, a brick x can lie on a brick y or not. Two possible actions are: Stack brick x on brick y and unstack brick y, i.e., put brick y onto the table. Now, a goal could be to have one pile of all three bricks in a specified order with an initial situation of them being piled in the reverse order. In each state of a situation, different fluents are true (e.g., brick x lies on brick y, brick y does not lie on brick x, brick z lies on the table), and stacking or unstacking generates a new situation.

To map this analogy to Cloud Computing is not as easy. As far as fluents are concerned, in a Cloud we have to consider the current value of each specific parameter, and whether the respective SLO is fulfilled or not. Furthermore, all the states of the Cloud itself like number of running virtual machines, number of physical machines available, etc., have to be modeled as fluents as well. Fluents for a specific application could be the predicate *has_value(SLAParameter p, Value v)* with $v \in \mathbb{R}^2$ meaning that the SLAParameter p holds the value v in the current situation, and *fulfills(SLO s)* meaning that the specified application fulfills a certain SLO s. The predicate *has_value(SLAParameter p1, x)* is valid for only one $x \in \mathbb{R}$ in a certain situation. The possible actions are provided by our use case.

Since we always observe the Cloud with all its states as a whole, it can be very difficult to derive exactly one action that could lead to an advancement of achieving a goal. The solution could be to view applications isolated from each other and to have one overall view that only takes into account some higher-level information like *fulfillsSLA(Application app)* meaning an application fulfills all its current SLOs at the moment. A doable way of defining goals could be to define utility functions that state the utility of a service fulfilling its SLA. Parameters of this utility function can be the importance of the consumer and the penalty that has to be paid when violating each SLO. The system then tries to find actions to maximize the utility.

Consider a Cloud servicing 100 applications with five SLA parameters each. This leads to $100 * (5 + 1) = 600$ different fluents, like *has_value(SLAParameter p1, x)*, *has_value(SLAParameter p2, y)*, etc. for every application. Thus, the largest obstacles to this approach are the large number of fluents and the immense search space for the possible actions as a result thereof.

[2]Instead of $\mathbb{R}$ one could consider using different sets with an only finite number of elements, as the set of floating point numbers.

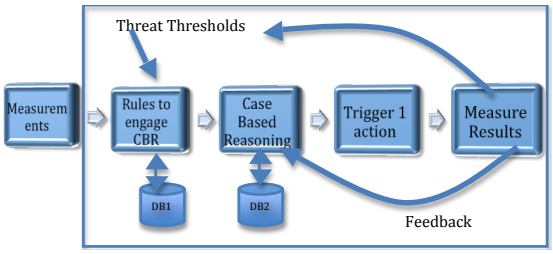

Figure 2. The process of Case Based Reasoning

V. Case Based Reasoning and Autonomic SLA Management

A. Case Based Reasoning (CBR)

Case Based Reasoning is the process of solving problems based on past experience [7]. In more detail, it tries to solve a *case* (a formatted instance of a problem) by looking for similar cases from the past and reusing the solutions of these cases to solve the current one. In general, a typical CBR cycle consists of the following phases assuming that a new case was just received:

1) Retrieve the most similar case or cases to the new one.
2) Reuse the information and knowledge in the similar case(s) to solve the problem.
3) Revise the proposed solution.
4) Retain the parts of this experience likely to be useful for future problem solving.

In step 4, the new case and the found solution is stored in the knowledge base. In the following section, we will show how we adopt CBR to the needs of SLA management in the field of Cloud computing.

B. CBR adapted to SLA Management

In this section we discuss the basic CBR model used for SLA management and some of its variations.

Following the diagram in Figure 2, the basic idea is to have rules stored in database 1 that engage the CBR system once a TT value has been reached for a specific SLA parameter. The measurements are fed into the CBR system, surrounded by the frame, as a new case by the monitoring component. Then, CBR prepared with some initial meaningful cases stored in database 2, chooses the set of cases which are most similar to the new case by various means as described in section VI. From these cases we select the one with the highest utility measured before. Now we trigger the action that was executed in the selected case. Finally, we measure the result of this action in comparison to the initial case some time intervals later and store it with the calculated utilities as a new case to CBR. Summing up, we have the following basic process (cf. Figure 2): New Measurements arrive (Measurements) → Check whether the TTs reached for some parameter (Rules to engage CBR). If yes, choose a set of most similar cases in CBR and from them choose the one with the highest utility (Case

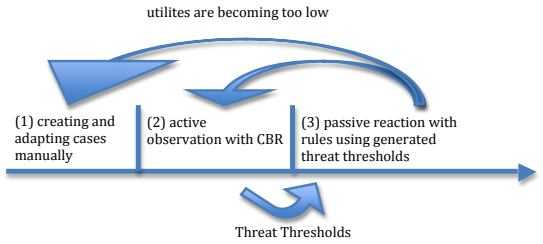

Figure 3. Active and Passive phases in the CBR management

Based Reasoning) → Execute action of this case (Trigger 1 action) → Calculate utility of this action by measuring results (Measure results) → Store case in CBR (Feedback). Doing this, we can constantly learn new cases and evaluate the usefulness of our triggered actions. By measuring the utility after more than one time interval, CBR is also able to learn whether an action was carried out too late (when utilities improved following the time intervals, but the improvement was too late in order to prevent an SLA violation) or even unnecessary. Thus, the TTs, which tell us when to engage the CBR mechanism, can be constantly ameliorated as well.

Further thoughts on the basis concept lead to the following variations:

a) Instead of using rules with TTs, CBR continuously receives new cases by the measurement device. Thus, CBR is not triggered due to TTs, but constantly active. CBR decides when to do something (actions 1-2 in use case from section III-B) and when to do nothing (action 3). This way we can get rid of TTs, which is especially useful in the early stage when the system doest not have historical measurements.
b) As in a) and depicted in Figure 3, we divide the system status into (1) a manual phase, where we create or adapt cases manually, (2) an active CBR phase as usual, and (3) a passive rule based phase, where we only do something, if the TT is attained, which we learned in the active phase. When in phase 3, we also calculate utilities of our actions as in phase 2. If the utilities get too low, depending on the severity, we either reactivate the active phase (phase 2) to learn new cases or even go into the manual phase (phase 1). When utilities ameliorate, we finally go back to the passive phase (phase 3).
c) For simple parameters (parameters whose causes are easy to understand and model), we have simple TTs and actions using rules instead of using CBR, which helps to relieve computing resources.

VI. Implementation of CBR

This section describes implementation details of CBR and methods we used for learning and reacting, as well as the utility measurements employed. The implementation follows variation (a) of the previous section.

We implemented the testbed in Java, based on FreeCBR [3], a generic implementation of step (1) of the list in subsection V-A. As can be seen in Figure 4 a complete case consists of:

```
1.  (
2.   (SLA,  1),
3.   (
4.    ((Incoming Bandwidth, 12.0),
5.    (Outgoing Bandwidth, 20.0),
6.    (Storage, 1200),
7.    (Availability, 99.5),
8.    (Running on PMs, 1)),
9.    (Physical Machines, 20)
10.  ),
11.   "Increase Incoming Bandwidth share by 5%",
12.  (
13.   ((Incoming Bandwidth, 12.6),
14.   (Outgoing Bandwidth, 20.1),
15.   (Storage, 1198),
16.   (Availability, 99.5),
17.   (Running on PMs, 1)),
18.   (Physical Machines, 20)
19.   ),
20.  0.002
21. )
```

Figure 4. CBR case example

(a) the id of the application being concerned (i.e., instance ID) (line 2), (b) the initial case (measurements by the monitoring component) consisting of the SLO values of the application and global Cloud information like number of running virtual machines (lines 3 – 10), (c) the executed action (line 11), (d) the resulting case (measured some time interval later (cf. Section V-B)) as in (b) (lines 12 – 19), and (e) the resulting utility (line 20).

To evaluate the actual utility a specific action helped in a specific case, we compare the utility of the initial case to the utility of the resulting case. Let α_{old} and α_{new} be the actual values of some parameter α measured at the initial and the resulting case, respectively, and α_T the specified SLA parameter threshold. We define the relative utility for a parameter $\alpha \geq \alpha_T$. In case that $\alpha \leq \alpha_T$, the definition has to be multiplied by -1. We define utility $u(\alpha)$ for $\alpha_T \neq 0$ as

$$u(\alpha) = \frac{\alpha - \alpha_T}{\alpha_T} \tag{3}$$

The gain in (or maybe loss of) utility from the initial to the resulting case for a parameter α can be described as

$$u(\alpha_{old}, \alpha_{new}) = \frac{\alpha_{new} - \alpha_T}{\alpha_T} - \frac{\alpha_{old} - \alpha_T}{\alpha_T} = \frac{\alpha_{new} - \alpha_{old}}{\alpha_T}. \tag{4}$$

As a next step we have to define the utility for parameters not stated in the SLA of the application, like “running on PMs” or the global parameter “Physical Machines”. Considering our use case from section III-B we define that the less PMs the application runs on, the better it is, since this frees up resources for other applications. The same is true for the impact of the number of running physical machines. Shutting down every physical machine that is not needed to guarantee the SLAs is seen as a positive effect on our utility. Thus, we also compare the number of running PMs from the resulting to the initial case with $u(PMs_{old}, PMs_{new}) = \frac{PMs_{old} - PMs_{new}}{PMs_{old}}$. The same principle is true for “running on PMs”.

We now derive the final utility by taking the average of the utilities $u(\alpha_{old}, \alpha_{new})$ for all SLA parameters α, of the utilities of running PMs, and of the global parameters. Of course, one

could also take into consideration building a weighted average. Generally speaking, there may be more sophisticated methods to define utilities than this linear approach, but for simplicity we decided to start with this one.

For our complete case depicted in Figure 4 and the SLA from our use case in Table I, the utility is thus calculated as follows:

$$u(case) = \frac{(\frac{12.6-12.0}{10.0} + \frac{20.1-20.0}{12.0} + \frac{1198-1200}{1024} + \frac{99.5-99.5}{99.0}) + \frac{1-1}{1} + \frac{20-20}{20.0}}{6} = 0.002 \quad (5)$$

The similarity of the cases is evaluated by the Euclidean distance, which for two cases takes the square root of the sum of the squared differences of each of the parameters. Of course, as for the utility, one could also weigh these parameters, which we chose to renounce for the beginning.

Furthermore, for the retrieval of similar cases, we implemented two methods. Each method seeks some cases, from which it chooses the one with the highest utility. The first method, which we call *t-neighborhood method*, looks for the case with the highest match percentage and takes all cases into consideration that have a distance of $t\%$ to the case with the highest match percentage. The second method, the *clustering method*, uses a k-means clustering algorithm [15] to group the cases into k clusters, from which we choose the one that includes the case with the highest match percentage. We try the clustering for several ks, and finally choose the k that has the lowest variance among all clusters.

VII. Evaluation

In this section we compare the outcomes of the test case using CBR with what we had expected a rational administrator to do. Thus, e.g., if storage for an application is extremely scarce, but all other values are in normal range, we expect the administrator to add allocated storage by the highest possible percentage – we will refer to this as the *intensity* of an action –, and not to increase any other parameter, do nothing or even decrease storage.

After feeding the knowledge base with 9 different cases, we test it against the SLA defined in our use case with 6 new cases and evaluate the results. The initial cases are displayed in Table III, where each column holds one of the cases 1-9. The upper part of the Table (parameters with subscript *b*), shows values as they were measured *before* any action took place. The row *Action* indicates the triggered actions in the specific cases followed by the measured parameters *after* the suggested action (parameters with subscript a). The Row *Utility* shows the utilities gained by these actions.

The 6 test cases that are stored one after the other into the knowledge base are presented in Table IV. The columns depict the cases 1-6, whereas the rows show the parameters at the beginning of the CBR cycle.

The result, i.e., what action was triggered, can be seen in Tables V and VI for the *clustering* and the *neighborhood method*, respectively. In Table V, the *expected action* column shows what action one could expect to be triggered in the test case (the same column is valid for Table VI and is not repeated therein). The *recommended action* columns in Tables V and VI define which action was actually recommended by the CBR mechanism. The variance column of Table V gives us an insight on how compact these clusters are. A low variance signifies high coherence (the points of one cluster have a small distance to each other), whereas high variance signifies the opposite. Additionally, in Table VI, where we present results for $t = 3\%$ and $t = 5\%$, we show the number of cases in the t-neighborhood of the case with the highest match percentage. This shows how large the set of cases was to choose the one with the highest utility. The more cases there are, the higher the chance to catch a case with a higher utility, but at the same time the smaller the similarity is to the original one.

Based on the evaluation results presented in Table V and VI we conclude that the actions are pretty much the same for both algorithms and relate to the expected action. Only the intensity of the action is always higher than one would expect to be necessary, because greater improvements always have higher utility values (cf. Equation (4)). This could be ameliorated by modifying the utility function to allow for *more moderate* actions to have higher utilities. Nevertheless, the problematic SLA parameter, i.e., the parameter whose resources were scarce, is always identified correctly. With the exception of case 5, which has excellent SLA parameter values and does not require any action to be executed, all methods recommend an action to trigger except the neighborhood method for $t = 3\%$. This is explained by the same argument why higher intensities have always been chosen: Doing more than is necessary always achieves a higher utility than doing

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
IB_b	15.0	11.0	10.5	15.0	15.0	15.0	15.0	15.0	15.0
OB_b	20.0	20.0	20.0	13.0	12.5	20.0	20.0	20.0	20.0
St_b	1200	1200	1200	1200	1200	1050	1000	1000	1200
Av_b	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.45	99.4
$RPMs_b$	1	1	1	1	1	1	1	1	1
PMs_b	20	20	20	20	20	20	20	20	20
Action	Do noth-ing.	IBW + 5%	IBW + 10%	OBW + 5%	OBW + 10%	St + 5%	St + 10%	M + 5%	M + 10%
IB_a	15.0	11.55	11.55	15.0	15.0	15.0	15.0	15.0	15.0
OB_a	20.0	20.0	20.0	13.65	13.75	20.0	20.0	20.0	20.0
St_a	1200	1200	1200	1200	1200	1103	1100	1200	1200
Av_a	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5
$RPMs_a$	1	1	1	1	1	1	1	1	1
PMs_a	20	20	20	20	20	20	20	20	20
Utility	0.0	0.009	0.0175	0.009	0.017	0.009	0.016	$8.41 \cdot 10^{-5}$	$1.68 \cdot 10^{-4}$

Table III
Initial Cases for CBR

	1	**2**	**3**	**4**	**5**	**6**
IB	15.0	11.0	10.5	15.0	20.0	10.0
OB	20.0	20.0	20.0	13.0	25.0	18.0
St	1200	1200	1200	1200	1250	1450
Av	99.5	99.5	99.5	99.5	99.6	99.5
$RPMs$	1	1	1	1	1	1
PMs	20	20	20	20	20	20

Table IV
Test cases for CBR

Case	Expected Action	Recommended Action	Variance
1	IBW + 5%	IBW + 10%	23
2	OBW + 5%	OBW +10%	18
3	St + 5%	St + 10%	208
4	St + 10%	St + 10%	14
5	None	St + 10%	96
6	IBW + 10%	IBW + 10%	40

Table V
EVALUATION RESULTS USING THE CLUSTERING ALGORITHM

	$t = 5\%$		$t = 3\%$	
Case	Recomm. Action	Cases in t-neighborhood	Recomm. Action	Cases in t-neighborhood
1	IBW + 10%	2	IBW + 10%	2
2	OBW + 10%	4	OBW +10%	4
3	St + 10%	2	St + 10%	2
4	St + 10%	3	St + 10%	2
5	M + 5%	2	None	1
6	IBW + 10%	8	IBW + 10%	5

Table VI
EVALUATION RESULTS USING THE NEIGHBORHOOD ALGORITHM

less or nothing. Thus, the value of doing nothing could also be appreciated more in the definition of the utilities.

VIII. CONCLUSION AND OUTLOOK

In this paper we discussed several approaches for knowledge management in self-adaptable Clouds. We evaluated rule based systems, default logic, situation calculus, and case based reasoning. Furthermore, we adopted the *case based reasoning (CBR)* method for the interpretation of measurement data with the goal of preventing SLA violations by triggering appropriate actions. Additionally, we designed a CBR based mechanism for the automatic re-configuration or even avoidance of threat thresholds topped with the introduction of general utility functions, which we were able to design without any semantic knowledge of the SLA parameters.

Currently, the CBR approach has been evaluated only against one SLA. A big issue, however, is that concurring SLAs may prevent other applications from being executed, especially if resources are scarce. Also, we have only used predefined SLA parameters, which in the future we will extend to the generation of user defined SLA parameters including the development of appropriate DSLs. Furthermore, we want to validate this approach by generating an extensive simulation model of a cloud environment over several time steps - using that, we will be able to evaluate not only CBR, but also other knowledge management methods from a hands-on point of view.

Nevertheless, we provided a means of proactively gearing the cloud infrastructure against SLA violations regardless of the SLA parameters in use. We have presented the proof of concept for the realization of the CBR-based knowledge management systems for self-adaptable Clouds.

ACKNOWLEDGMENTS

The work described in this paper is supported by the Vienna Science and Technology Fund (WWTF) under grant agreement ICT08-018 Foundations of Self-Governing ICT Infrastructures (FoSII).

REFERENCES

[1] Drools, www.drools.org.
[2] (FOSII) - Foundations of Self-governing ICT Infrastructures, http://www.infosys.tuwien.ac.at/linksites/fosii.
[3] FreeCBR, http://freecbr.sourceforge.net/.
[4] IT-Tude: SLA monitoring and evaluation, http://www.it-tude.com/sla-monitoring-evaluation.html.
[5] Jess, www.jess.org.
[6] SLA@SOI, http://sla-at-soi.eu/.
[7] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches, 1994.
[8] Sheikh Iqbal Ahamed et al., editors. *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2009, Seattle, Washington, USA, 20-24 July 2009*. IEEE Computer Society, 2009.
[9] Grigoris Antoniou. A tutorial on default logics. *ACM Comput. Surv.*, 31(4):337–359, 1999.
[10] Raphael M. Bahati and Michael A. Bauer. Adapting to run-time changes in policies driving autonomic management. In *ICAS '08: Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*, pages 88–93, Washington, DC, USA, 2008. IEEE Computer Society.
[11] Ivona Brandic. Towards self-manageable cloud services. In Ahamed et al. [8], pages 128–133.
[12] Yuan Chen et al. Translating service level objectives to lower level policies for multi-tier services, 2008.
[13] V. C. Emeakaroha, I.Brandic, M. Maurer, and S. Dustdar. Low level metrics to high level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments. In *The 2010 High Performance Computing and Simulation Conference (HPCS 2010), in conjunction with The 6th International Wireless Communications and Mobile Computing Conference (IWCMC 2010) (to appear)*, Caen, France, 2010.
[14] Kaouthar Fakhfakh et al. A comprehensive ontology-based approach for SLA obligations monitoring. In *ADVCOMP '08: Proceedings of The Second International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 217–222, Washington, DC, USA, 2008. IEEE Computer Society.
[15] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.
[16] Peer Hasselmeyer, Bastian Koller, Lutz Schubert, and Philipp Wieder. Towards SLA-supported resource management. In *High Performance Computing and Communications*, pages 743–752. Springer, Berlin / Heidelberg, 2006.
[17] Thomas Heinis and Cesare Pautasso. Automatic configuration of an autonomic controller: An experimental study with zero-configuration policies. In Ahamed et al. [8], pages 67–7.
[18] Jeffrey O. Kephart and Rajarshi Das. Achieving self-management via utility functions, 2007.
[19] Bithika Khargharia, Salim Hariri, Ferenc Szidarovszky, Manal Houri, Hesham El-Rewini, Samee Ullah Khan, Ishfaq Ahmad, and Mazin S. Yousif. Autonomic power & performance management for large-scale data centers. In *IPDPS*, pages 1–8. IEEE, 2007.
[20] Giannis Koumoutsos, Spyros Denazis, and Kleanthis Thramboulidis. SLA e-negotiations, enforcement and management in an autonomic environment. *Modelling Autonomic Communications Environments*, pages 120–125, 2008.
[21] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Electronic Transactions on Artificial Intelligence*, 2:159–178, 1998.
[22] Vinod Muthusamy and Hans-Arno Jacobsen. SLA-driven distributed application development. In Ahamed et al. [8], pages 31–36.
[23] Adrian Paschke and Martin Bichler. Knowledge representation concepts for automated SLA management. *Decision Support Systems*, 46(1):187–205, 2008.
[24] Benny Rochwerger et al. The RESERVOIR model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4), 2009.
[25] Jing Xu, Ming Zhao, José Fortes, Robert Carpenter, and Mazin S. Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213–227, 2008.