On Leveraging Policy-Based Management for Maximizing Business Profit

Issam Aib, *Student Member, IEEE*, and Raouf Boutaba, *Senior Member, IEEE*

***Abstract*— This paper presents a systematic approach to business and policy driven refinement. It also discusses an implementation of an application-hosting Service Level Agreement (SLA) use case. We make use of a simple application hosting SLA template, for which we derive a low-level policy-based Service Level Specification (SLS). The SLS policy set is then analyzed for static consistency and runtime efficiency. The *Static Analysis* phase involves several consistency tests introduced to detect and correct errors in the original SLS. The *Dynamic analysis* phase considers the runtime dynamics of policy execution as part of the policy refinement process. This latter phase aims at optimizing the business profit of the service provider. Through mathematical approximation, we derive three policy scheduling algorithms. The algorithms are then implemented and compared against random and First Come First Served (FCFS) scheduling. This paper shows, in addition to the systematic refinement process, the importance of analyzing the dynamics of a policy management solution before it is actually implemented. The simulations have been performed using the $\mathcal{PS}$ Policy Simulator tool.**

***Index Terms*— Policy-based management, policy optimization, policy refinement, Policy analysis.**

I. Introduction

BY separating the rules that govern the behavior of a system from the functionality it supports, policy-based management (PBM) promises to reduce IT costs while simultaneously improving quality of service (QoS) and dynamic adaptability to change. PBM is currently present at the heart of a multitude of management architectures and paradigms with such diversified prefixes as SLA-driven, business-driven, autonomous, adaptive, and self management, to name a few. In this paper we will only focus on QoS related policies. However, policies are also extensively used in the security arena.

Although research on PBM has been going on for more than a decade, there is still no strong existence of full fledged PBM success stories in real scale applications, especially when it comes to QoS management. This owes much to proving the correctness of policy based solutions when it comes to managing real scale systems with hundreds or even thousands of policies interacting in a dynamic way. A multitude of policy languages and architectures have been proposed but techniques for refinement, and consistency/completeness analysis remain both scarce and immature. It is then understandable why venturing into a full PBM solution for managing one's IT infrastructure remains difficult to justify.

System performance is equally interesting as well. While it is true that policy based solutions promise dynamism and flexibility, they often come with no guarantee of high performance. Verma [1] states that PBM solutions should not be considered a case of expert systems because of the strong weight of the performance parameter they have to carry with them. Work on the benchmarking of PBM solutions is also marked by great scarcity. It is necessary that a PBM solution not only works, but also must be as efficient as existing legacy solutions, if not better. In this regard, the maximization of business profit should represent the crucial goal that any QoS PBM solution should target.

In this paper we develop a use case for the business driven SLA refinement into low level management policies. In addition, we present an implementation which maximizes the business profit of the service provider. The case spans all of the business driven SLA management loop. We show the details of how the refinement process is conducted to produce a policy based SLS. Two analysis phases are then applied to the generated SLS. The *Static Analysis* phase checks the SLS' policy set for consistency and stability. The new *Dynamic Analysis* phase addresses the business-driven dynamic policy analysis in which we emphasize the need for incorporating business (and SLA) related data, encoded mainly within metrics generated during the refinement process, to handle the orchestration of policies at runtime. This analysis proves crucial in making the same set of generated policies (SLS) achieve the best performance at runtime.

The work described in this article is an extension of the one presented in [2] on several aspects: First, It develops two different SLA enforcement strategies and provides a more detailed refinement process. Second, it includes further mathematical analysis and proof of concepts. Third, it implements and evaluates a larger number of policy scheduling algorithms. Fourth, it provides a more thorough analysis and discussion of the simulation results. Finally, it includes a more comprehensive related work section.

This article proceeds as follows. Section II presents the generic SLA use case. Section III defines the business profit function we would like to optimize and section IV presents two strategies to enforce it. Section V derives a formal SLS for the generic SLA and shows the different stages of the proposed refinement process. After that, the static analysis is conducted on the generated SLS and shows how policy-

Manuscript received November 23, 2006; revised June 4, 2007. The associate editor coordinating the review of this paper and approving it for publication was M. Ulema. This work was presented in part at the 10^{th} IFIP/IEEE Integrated Network Management Symposium (IM), May 2007. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

I. Aib and R. Boutaba are with the David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada (e-mail: {iaib; rboutaba}@uwaterloo.ca).

Digital Object Identifier 10.1109/TNSM.2007.021104.

loop anomalies can be detected and fixed. Next, we present our dynamic analysis approach in which we provide an approximate solution of the transient state of a variant of an $M/M/C_t/C_t$ queue. We then use this solution to derive better runtime scheduling algorithms of triggered policies and hence a better generated business profit for the same low level policy set. Sections X and XI present the simulation environment we used, the conducted simulations and finally summarized results of the performance of our policy scheduling algorithms.

II. Generic $\mathcal{SP}$ SLA

Listing 1 Generic application hosting SLA: $\mathcal{SP}$ SLA

1) Customer $\mathcal{C}$ is provided a web application hosting service with schedule sc.
2) Maximum capacity is of cp_{max} simultaneous connections.
3) $\mathcal{C}$ is charged $\$ch = a \times cp_{max}$ monthly.
4) Monthly average availability of the hosted service $\geq av_{min}$.
   a) An i^{th}successive availability violation incurs a reward of $r_i \times ch$.
   b) At the 3^{rd}successive availability violation, the SLA is considered void.
5) Min average time to process end customer requests $= rt$ ms.
   a) Otherwise, $\mathcal{C}$ is rewarded $rt.ref \times rt$.

We consider an Application Hosting Service Provider $\mathcal{SP}$ which advertises a set of SLAs to its customers. The set of SLAs is derived from the simple generic SLA of listing 1, named $\mathcal{SP}$ SLA. $\mathcal{SP}$ operates in its IT infrastructure a pool of $sp.cp$ identical server units. Server units are allocated to each SLA instance to ensure its QoS requirements.

$\mathcal{SP}$ SLA states, in 5 clauses, that $\mathcal{SP}$ offers an application hosting service supporting a load (capacity) of cp_{max} simultaneous end client connections to the system, an availability average of av_{min}, an average response time rt, all with a monthly cost ch. The total set of parameters can be gathered in the tuple (sc, cp_{max}, a, av_{min}, r_1, r_2, r_3, rt, $rt.ref$). Each instance of this tuple generates an *SLA type* the $\mathcal{SP}$ can advertise to its potential customers. An SLA instance is a realization of an SLA type for a particular customer. In the following, sla_i denotes an SLA type and $sla_{i,j}$ denotes SLA instance j of SLA type i, all of which are derived from the generic SLA of listing 1.

Before going further into the SLA refinement process, the first step is to define the business profit function the $\mathcal{SP}$ intends to maximize.

III. Defining the Business Profit Function

Denoted in this paper as Ψ, the business profit function provides a measure of the profitability of the service provided by the $\mathcal{SP}$. Detailed modeling of business profit and business goals is beyond the scope of this paper. In the general case, Ψ should be a function of several service and business level parameters such as service operation cost, net financial revenue, customer satisfaction, and market share. To keep the use case as simple as possible, we define Ψ to be the sum of the net financial profit gained from each contracted SLA. This implicitly assumes that managing $\mathcal{SP}$'s IT infrastructure incurs a fixed cost which is independent of the number of contracted SLAs. Hence,

$$\Psi = \sum_{i \in \mathcal{SLA}} \left(\sum_{j \in \mathcal{SLA}_i} (NP(SLA_{i,j})) \right) \quad (1)$$

where $\mathcal{SLA}$ represents the set of all SLA types that $\mathcal{SP}$ supports; $\mathcal{SLA}_i$ the set of contracted instances of SLA type SLA_i; and $NP(SLA_{i,j})$ the net profit generated by SLA instance j of the SLA type i. The problem $\mathcal{SP}$ has to solve is to reach $Max(\Psi)$? We will elaborate further on this function in section VIII.

IV. Enforcement strategies

The clauses of listing 2 help identify the set of Service Level Objectives (SLOs) that the $\mathcal{SP}$ has to enforce. The identification we employ is semi-formal as it requires interpreting a textual specification and pouring it into a formal specification. Section V-A explains how this SLO set has been generated.

Using a policy-based approach, the $\mathcal{SP}$ has multiple choices as to how to enforce them. In the following we describe two of them: a guaranteed approach and a lazy one.

A. Guaranteed enforcement strategy

With this strategy, $\mathcal{SP}$ will pre-allocate for each new SLA instance the exact number of resources (server units) required to enforce its SLOs at maximum load.

We will assume that $\mathcal{SP}$ possesses a mapping function $su(rt)$ which gives the load (number of simultaneous connections) a single server unit (su) instance can handle while still respecting the response time constraint rt (for end customers) as specified in the SLA.

Let $|sla_{i,j}.sus|$ denote the number of server units allocated to $SLA_{i,j}$. The maximum number of server units required by each $SLA_{i,j}$ is then:

$$Max(|sla_{i,j}.sus|) = \lceil cp^i_{max} \times av^i_{min} / su(rt_i) \rceil \quad (2)$$

Let $|sla_i|$ be the number of contracted SLAs (instances) of type i. When using the guaranteed approach, we should always have:

$$\sum_{i \in \mathcal{SLA}} \left(|sla_i| \times \lceil cp^i_{max} \times av^i_{min} / su(rt_i) \rceil \right) \leq sp.cp \quad (3)$$

We assume that the set of contracted SLAs become activated at the same time. If the system runs with no unexpected failures, the guaranteed enforcement approach will produce business profit:

$$\Psi = \sum_{i \in \mathcal{SLA}} ch_i \left(\sum_{j \in \mathcal{SLA}_i} sc_{i,j}.duration() \right) \quad (4)$$

Given a set of defined SLA types, $\mathcal{SP}$ can find the number of instances to contract for each SLA type so as to maximize Ψ by solving the integer programming simplex formed by $Max(eq.3)$ and eq.4.

However, such approaches prove inefficient in practice where the running SLAs are actually not fully loaded at all times, which is the case for most web applications.

B. Lazy enforcement strategy

In this approach, $\mathcal{SP}$ allocates server units to each SLA on a per need basis. Conversely, it removes server units from an SLA as soon as this latter starts experiencing low load. Similar to the statistical multiplexing of traffic crossing a shared physical network cable, this technique allows $\mathcal{SP}$ to contract a number of SLAs with a total maximum sever pool capacity beyond the actual capacity it possesses.

At instantiation time, an initial number of $n_0(\geq 1)$ server units is allocated to the SLA. When the connection intensity (number of simultaneous end customer connections) reaches a certain threshold th_a of the current SLA capacity, a request is made to the server pool to obtain an additional server unit. Conversely, if a low threshold th_r is reached, an action is triggered to release a server unit to the free server units pool. With this technique, $\mathcal{SP}$ aims at a higher business revenue than promised by the guaranteed enforcement approach (eq.3-4). Notice that at this stage, three new parameters have been added to the SLA type, which are the thresholds th_a and th_r and the initial minimal number of server units n_0.

V. Refinement of the generic $\mathcal{SP}$ SLS

The refinement will be done in a series of phases starting by the formal specification of SLOs and high-level policy rules. Following this, the Enforcement strategy along with available system resources functionality are used to guide an iterative refinement process until a fixed point is reached where all SLS statements are directly supported by the resources at hand. Hence, depending on which enforcement strategy is used and/or which system resources are available different SLSs can be generated.

A. SLO set specification

Service Level Objectives (SLOs) represent logical constraints over SLA parameters that the $\mathcal{SP}$ has to respect. Using a straightforward formulation, the SLOs of listing 2 correspond respectively to clauses one to five of the $\mathcal{SP}$ SLA (listing 1), with the sub-conditions 4-a/b and 5-a excluded. These sub-conditions will be dealt with in the next iteration. As the title of listing 2 indicates, this SLO set corresponds to the guaranteed enforcement approach as it exactly translates (excluding sub-conditions) $\mathcal{SP}$ SLA into formal terms.

Listing 2 SLO set for the Guaranteed enforcement of $\mathcal{SP}$ SLA

sloSet sloSetG = {
  slo slo_{sc} = $(schedule == sc)$;
  slo slo_{cp} = $(ws.cp == cp_{max})$;
  slo slo_{ch} = $(payment.sum(month) == ch)$;
  slo slo_{av} = $(ws.av \geq av_{min})$;
  slo slo_{rt} = $(ws.rt \leq rt)$;}

The capacity SLO slo_{cp} of listing 2 states that the total capacity of allocated server units for an $\mathcal{SP}$ SLA instance should be equal to cp_{max}. For the lazy enforcement, this SLO is a requirement that is stronger than what is actually needed. This is because, in this approach, $\mathcal{SP}$ intends to allocate server units to each SLA on a per need basis. So slo_{cp} needs to be weakened to the new expression of listing 3. With this expression the runtime capacity of an SLA, in terms of the sum of capacities of allocated server units, is allowed to be less than the value contracted in the SLA.

The passage from sloSetG to sloSetL is an example of SLO refinement which is refinement strategy dependent.

Listing 3 SLO set for the Lazy enforcement of $\mathcal{SP}$ SLA

sloSet sloSetL **extends** sloSetG = {
  slo slo_{cp} = $(ws.cp \leq cp_{max})$; }

The next iteration will deal with the tracking of SLO states and how state changes/violations are signaled.

B. Metrics and SLO constraints for SLO state tracking

In our refinement approach we make use of events as the means to signal SLO violations. When an SLO is violated, an event is generated to inform about the parameters related to the violation, such as the time of its occurrence and/or some log information that might be useful for its processing. To evaluate the constraint defined by the SLO, there is a need to have the values of each of its parameters at hand. Therefore, these parameters need to be defined as metrics that are computed at SLA runtime and serve as input to a constraint tracking object which periodically evaluates and informs about that SLO's state. Such a dependency hierarchy is illustrated in figure 1.

The required runtime SLO constraints and metrics and their relationships can hence be identified in a top down fashion with the prior knowledge of what resource level (leaf) metrics are supported by the underlying IT infrastructure of the service provider.

The output set of required high level metrics and SLO state constraints is listed in listing 4. This set shows additional parameters that need to be filled for an actual SLS instance. For example, there is a need to define specific values for service deployment and un-deployment times. In addition, the capacity SLO slo_{cp} generates the need to compute a new metric m_{cp}. This metric is set to collect the actual runtime load $ws.cp$ of the server units. Each metric is then used at runtime as an input to the SLO evaluation. Similarly, metrics are also generated to track the states of the payment, availability, and response time SLOs.

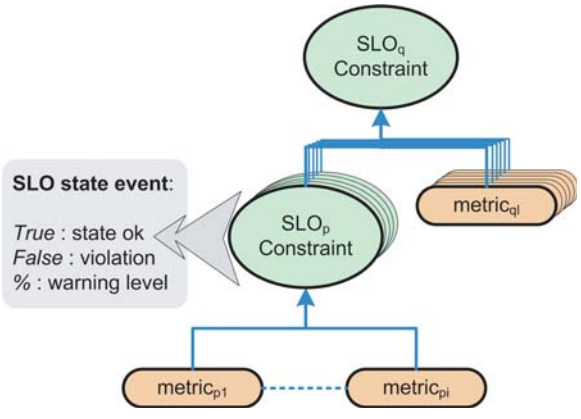


Fig. 1. SLO state constraints and related metrics.

Listing 4 Metrics identification based on available high level system metrics

- $slo_{sc} \Rightarrow$
  define schedule=sc,
  schedule.deployTime=<? >,
  schedule.undeployTime =<? >, ...
- $slo_{cp} \Rightarrow$ **metric** $m_{cp} = ws.cp$
- $slo_{ch} \Rightarrow$ **metric** $m_{ch} = payment.sum(month)$
- $slo_{av} \Rightarrow$ **metric** $m_{av} = ws.av$
- $slo_{rt} \Rightarrow$ **metric** $m_{rt} = ws.rt$

C. Policies for SLO violation events

The consequences of violating an SLO need to be specified in the SLA. At this phase, a set of policies of the form *on not(slo) do action* is generated based on the generated SLO set (listing 2, 3). Since SLO metrics are now defined (listing 4), this set can be automatically generated as is shown in listing 5[1]. In this set, $not(slo)$ is replaced by an event type $\overline{slo}$ which is intended to convey the violation information to all those SLS policies that need it.

Listing 5 Automatic SLO violation policies

- $slo_{cp} \Rightarrow$
  eventType $\overline{slo_{cp}}$;
  policy pm_{cp} = {
  on $(m_{cp} > cp_{max})$ **do** $generate(\overline{slo_{cp}})$}
- $slo_{ch} \Rightarrow$
  eventType $\overline{slo_{ch}}$;
  policy pm_{ch} = {
  on $(m_{ch} < ch)$ **do** $generate(\overline{slo_{ch}})$}
- $slo_{av} \Rightarrow$
  eventType $\overline{slo_{av}}$;
  policy pm_{av} = {
  on $(m_{av} < av_{min})$ **do** $generate(\overline{slo_{av}})$}
- $slo_{rt} \Rightarrow$
  eventType $\overline{slo_{rt}}$;
  policy pm_{rt} = {
  on $(m_{rt} < rt)$ **do** $generate(\overline{slo_{rt}})$}

D. SLO violation policies

Each time an SLO is violated an action is required in order to bring the SLA back to a normal state. SLO violations can also trigger actions that are specified in the SLA. In this iteration we generate policies related to SLO violations that are directly deductable from the SLA. Listing 6 shows the policies generated for each SLO violation clause. This set of policies is required by both enforcement strategies.

Availability is computed as a monthly average as specified in the generic SLA. However, the $\mathcal{SP}$ can compute it in several ways. A similar case exists with the semantics of "successive". We will not elaborate on the possible options here in order to keep our discussion focused. We assume that $\mathcal{SP}$ computes availability as a sliding window average of length av_{win} (= one month in the SLA). For "successive", the $\mathcal{SP}$ refers to those violations which occur within at least one availability window interval (1 month).

Policies p^1_{av}, p^2_{av}, and p^3_{av} implement availability violation penalties as specified in clause 4 of the generic $\mathcal{SP}$ SLA of listing 1. p^1_{av} states that on the occurrence of an event e_1 of type $\overline{slo_{av}}$, meaning a violation of slo_{av} at runtime, the action that needs to be carried out is to credit the customer account with r_1 monetary units. p^2_{av} and p^3_{av} implement respectively the penalty clauses for the 2^{nd} and 3^{rd} successive violations. The where conditions enforce the one month "memory" on successive violations and ensure that only one of p^1_{av}, p^2_{av} or p^3_{av} can be triggered at a time.

Listing 6 SLA specific SLO violation policies

double av_w = 1 month; // availability window
event $\overline{slo_{av}}$ e_1, e_2, e_3; // events of type $\overline{slo_{av}}$

- Clause 4-a $\Rightarrow$
  policy p^1_{av} = {
  on e_1
  do $c.credit(r_1)$
  where $not(p^2_{av} \vee p^3_{av})$}
  policy p^2_{av} = {
  on $(e_1 \rightarrow e_2)$ // e_1 followed by e_2
  do $c.credit(r_2)$
  where $((time(e_2) - time(e_1) < av_w) \wedge not(p^3_{av}))$}
  policy p^3_{av} = {
  on $(e_1 \rightarrow e_2 \rightarrow e_3)$
  do {$c.credit(r_3)$
  where $((time(e_3) - time(e_1) < av_w))$}
- Clause 4-b $\Rightarrow$
  policy p^{3b}_{av} = {
  on $(e_1 \rightarrow e_2 \rightarrow e_3)$
  do {SLA.terminate()
  where $((time(e_3) - time(e_1) < av_w))$}
- Clause 5-a $\Rightarrow$
  policy p_{rtv} = {
  on $\overline{slo_{rt}}$
  do $c.credit(rt.ref)$

Listing 7 Policy to enforce at customer side

- Clause 3 $\Rightarrow$
  policy pC1 = {
  on every month
  do $\mathcal{SP}$.credit(ch)
  start at sc.activationTime }

E. Enforcement strategy dependent policies

For the guaranteed enforcement approach, $\mathcal{SP}$ has to enforce policy p_{g1}. This policy allocates the number of server units (equation 2) that are needed to guarantee availability and response time SLOs at all times, that is, even during maximum load. These policies also make all necessary initializations required by the specific web application of the SLA instance.

[1]The notation for policy rules is inspired from [3], [4].

At the other end of the SLA life time, the un-deployment time policy $p_{undeploy}$ takes care of wrapping up actions which include resource liberation, deactivation of active SLA policies, and some reporting actions.

Listing 8 Guaranteed approach specific policies

- Clause 1 $\wedge$ Clause 2 $\wedge$ Clause 5 $\Rightarrow$
  policy p_{g1} = {
  at $sc.deployTime$
  do $ws.add(\lceil cp_{max} \times av_{min}/su(rt) \rceil)$
  }
- Clause 1 $\Rightarrow$
  policy $p_{undeploy}$ = {
  at $sc.undeployTime$
  do SLA.undeploy()}

For the lazy enforcement approach, policy rule p_{g1} gets replaced (or split) into the three rules of listing 9.

Listing 9 Lazy enforcement dependent policies

double th_a, th_r;
constraint $0 < th_r < th_a \leq 1$; // %
int n_0; **constraint** $1 \leq n_0$; // %
policy p_{dep} **overrides** p_{g1} = {
at $sc.deployTime$
do $ws.add(n_0)$}
policy p_{add} = {
on $(ws.load \geq th_a)$
do $ws.add(1)$
where $(ws.cp \leq cp_{max})$}
policy p_{rem} = {
on $(ws.load \leq th_r)$
do $ws.free(1)$
where $(|ws.su| > 1)$}

p_{dep} is a deployment time policy which initializes the new SLA by requesting an initial number n_0 of server units from the pool of free server units. p_{dep} is related to the lazy enforcement strategy and hence overrides policy p_{g1}. $p_{undeploy}$ is not in conflict with the lazy enforcement and is kept unchanged.

p_{add} and p_{rem} implement the lazy enforcement approach by tracking the load of the server units that are available to the SLA instance. They execute the necessary actions each time a threshold is crossed, p_{add} by requesting a new server unit at high load times and p_{rem} by releasing one at low load times.

The determination of the appropriate values of parameters th_a and th_r are SLS specific and tunable by the $\mathcal{SP}$. In the conducted simulations these two parameters had a strong impact on the generated profit.

Finally, SLO slo_{rt} (listing 3) is implicitly implemented in both strategies by limiting the maximum load on each server unit to $su(rt)$ (see section IV-A).

F. Another iteration for generating metrics/policies

The condition on policies p_{add} and p_{rem} need to be defined in terms of constraints over metric values. Hence, there is a need to detect $ws.load$ through a metric and have an event generated each time this metric crosses one of th_a and th_r thresholds. Listing 10 shows how p_{add} and p_{rem} are updated.

Listing 10 Additional metric generation and policy set update

- $p_{add} \Rightarrow$
  metric $mTh_{add} = ws.load$
  policy $pmTh_{add}$ = {
  on $(mTh_{add} \geq th_a)$
  do generate($evTh_{add}$) }
  update p_{add} = { **on** $evTh_{add}$ }
- $p_{rem} \Rightarrow$
  metric $mTh_{rem} = ws.load$
  policy $pmTh_{rem}$ = {
  on $(mTh_{rem} \geq th_r)$
  do generate($evTh_{rem}$) }
  update p_{rem} = { **on** $evTh_{rem}$ }

G. Final iteration and Complete SLS specification

So far, this section described the $\mathcal{SP}$ SLS generation process for both the guaranteed and lazy enforcement policies. Each output SLS is the completed service level specification (SLS) of the generic $\mathcal{SP}$ SLA. Listing 11 summarizes the output of this refinement process. The generation of this SLS involved several sub processes, which in the general case are expected to be iterative with a fixed point property.

Note that a first part of the SLS needs to be enforced at $\mathcal{SP}$ side. The $\mathcal{SP}$ responsibilities have been grouped under one *role structure* named $\mathcal{SP}$. The second part of the SLS is related to customer payment policy (pC1 in listing 7) and needs to be enforced at customer side, under role $\mathcal{C}$. The $\mathcal{SP}$ role contains a total of thirteen policies, five metrics, and three constraints. Policies p^3_{av} and p^{3b}_{av} have been blended into one policy p^3_{av}. Metrics mTh_{add} and mTh_{rem}, being identical, have been blended into the metric mLoad. Note that we assumed that the $\mathcal{SP}$ takes care of all metric computations and not the customer or a third party.

The inclusion of third parties and the distribution of metric computations can also be included in the SLS specification but was not considered here in order to keep the use case simple.

Finally, we would like to emphasize that the final SLS does not contain any SLO definition. All necessary logic has been specified in the sets of policies, metrics, and events. Hence, an SLO can be a component of an SLA or an intermediary SLS, but not the final SLS.

VI. Static analysis

The second phase after the generic SLS is generated conducts a static analysis in order to test the consistency of the generated policy set. For the static analysis phase we identified and conducted four types of tests: action conflicts, deadlocks, loops, unreachable states (dead code (i.e policies)), erratic behavior.

$\mathcal{SP}$ needs to test the generated policies to make sure they are free from any of these defects.

Listing 11 $\mathcal{SP}$ SLS for the lazy enforcement strategy

sls $\mathcal{SLS}$ = {

- // Service Provider Role
  role $\mathcal{SP}$ = {
  schedule sch;
  int n_0;//
  double th_a, th_r, av_w = 1 month;
  constraint $0 < th_r < th_a$, $0 < th_a \leq 1$, $1 \leq n_0$;
  metric $m_{cp} = ws.cp$, $m_{av} = ws.av$
  metric $m_{ch} = payment.sum(month)$
  metric $mLoad = ws.load$, $m_{rt} = ws.rt$;
  eventType $\overline{slo_{cp}}$, $\overline{slo_{ch}}$, $\overline{slo_{av}}$, $\overline{slo_{rt}}$;
  event $\overline{slo_{av}}$ e_1, e_2, e_3; // events of type $\overline{slo_{av}}$

- // SLO violation notification policies
  policy pm_{cp} = {
  on $(m_{cp} > cp_{max})$ **do** $generate(\overline{slo_{cp}})$}
  policy pm_{ch} = {
  on $(m_{ch} < ch)$ **do** $generate(\overline{slo_{ch}})$}
  policy pm_{av} = {
  on $(m_{av} < av_{min})$ **do** $generate(\overline{slo_{av}})$}
  policy pm_{rt} = {
  on $(m_{rt} < rt)$ **do** $generate(\overline{slo_{rt}})$}

- // Lazy enforcement specific policies
  policy p_{dep} = {
  at $sc.deployTime$ **do** $ws.add(n_0)$}
  policy $pmTh_{add}$ = {
  on $(mLoad \geq th_a)$ **do** generate(evLoad) }
  policy p_{add} = {
  on $evLoad$ **do** $ws.add(1)$
  where $(ws.cp \leq cp_{max})$}
  policy $pmTh_{rem}$ = {
  on $(mTh_{rem} \geq th_r)$ **do** generate(evLoad) }
  policy p_{rem} = {
  on $evLoad$ **do** $ws.free(1)$
  where $(|ws.su| > 1\}$ }

- // SLA specific SLO violation policies
  policy p^1_{av} = {
  on e_1 **do** $c.credit(r_1)$
  where $not(p^2_{av} \vee p^3_{av})$}
  policy p^2_{av} = {
  on $(e_1 \rightarrow e_2)$ // e_1 followed by e_2
  do $c.credit(r_2)$
  where $((time(e_2) - time(e_1) < av_w) \wedge not(p^3_{av}))$}
  policy p^3_{av} = {
  on $(e_1 \rightarrow e_2 \rightarrow e_3)$ **do** $\{c.credit(r_3); SLA.terminate()\}$
  where $((time(e_3) - time(e_1) < av_w))$}

- // response time violation policy
  policy p_{rtv} = {
  on $\overline{slo_{rt}}$ **do** $c.credit(rt.ref)$ }

}

- // Customer Role
  role $\mathcal{C}$ = {
  do $\mathcal{SP}$.credit(ch)
  start at sc.activationTime
  }
  }

A. Static conflicts Analysis

Action *conflicts analysis* relates to those policies which have conflicting actions and which can execute at the same time on the same system object. By observing the generated policy set, we notice that p_{dep} and p_{add} request additional server units. However, p_{dep} executes only once at the SLA deployment time while p_{add} becomes active only after p_{dep} has executed correctly. p_{rem} releases one server unit which is an action expected to be always successful. p^1_{av}, p^2_{av} and p^3_{av} cannot execute at the same time (even though this does not cause trouble). In the implementation, this translates to making method c.credit() synchronized. When several SLA instances are running, policies of type p_{dep} and p_{add} can conflict due to lack of sufficient server units. For example, in case only one free server unit is available, only one policy can be executed while the others need to wait until enough resources become available. However, this is a *runtime conflict* that the lazy enforcement strategy, for example, accepts. So, from the static analysis point of view, the policy set is actions conflict free.

B. Deadlock Analysis

For *deadlock analysis*, it is straightforward that the policy set is deadlock free as there is only one possible blocking action ws.add(). This action requests a number of server units from the pool of free server units. So a deadlock situation at runtime is not possible.

C. Erratic behavior Analysis

The constraints on the definition of th_a and th_r (listing 9) have been set following the intuition that the threshold to request a new server unit should be strictly greater than the one which frees an acquired one. Without these constraints and if th_r was fixed, possibly due to a mistake of the system operator, to a value greater than th_a, the concerned SLA instance might never free any acquired server unit until it is terminated, or on the other extreme, it might show *erratic behavior* in case of a shortage in server units.

For the first case, $ws.load = th_a \Rightarrow p_{add}$ gets triggered $\Rightarrow$ new $ws.load < th_a < th_r$. This implies that when the number of connections decreases, no action will be taken by the SLA and it will continue to hold resources that it is actually not using!

The latter case, however, is more harmful. It occurs when p_{add} is triggered while no resources are available in the system and $ws.load$ continues to grow until reaching th_r. At this moment p_{rem} is triggered reducing $su.size$ by one.

Since we have

$$ws.load = \frac{|connections|}{(su.cp \times su.size())} \tag{5}$$

This implies that $ws.load$ increases. Hence, p_{rem} gets triggered again, and so on, until all but one of the server units are freed (because of the **where** clause in p_{rem}). It is then expected that availability violations occur leading possibly to the SLA termination (policy p^3_{av}). With a slight chance, the SLA can still survive if before p^3_{av} is triggered the load on the unique left server unit diminishes. A way to detect this type

of erratic behavior is to specify a rule for the static analyzer which states:

```
//fsupl = free server units pool
policy pErraticTest1 = {
  on (triggered(p^3_av) ∧ (fsupl.size > 0))
  do signal(erraticBehavior);}
```

D. Unreachable code (policies)

Based on our assumption that one server unit instance is loaded only up to $su(rt)$ simultaneous end customer sessions, the system can deduce that policy p_{rtv} cannot be triggered at runtime. So, theoretically, these policies can be safely removed from the output SLS. However, in practice server machines can become congested due to various causes and, although the constraint on the number of simultaneous sessions is respected, the response time might still be violated. The final decision to remove or keep p_{rtv} should be left to the discretion of the SLS designer.

E. Policy-loop Analysis

This step checks for potential *static policy loops*. We say of a policy-based system to contain a static policy loop if there exists at least one system state $\mathcal{S}_l$, different from the final state $\mathcal{S}_{final}$, which when reached the system keeps on always coming back to it after a finite number of state transitions. The policy loop is dynamic if it is not static and occurs at runtime due to runtime conditions.

In this section, the runtime state of an SLA is defined by the tuple $(ws.cp, np)$; where $ws.cp$ is the capacity in number of allocated server units and np the accumulated net profit. np is affected by operations $SP.credit()$ of policy pC1 (listing 7) and $C.credit()$ of policies p^1_{av} to p^3_{av} (listing 9).

We also consider that the state of $\mathcal{SP}$'s system to be the sum of the states of all the contracted SLAs augmented by the state of the free server units pool and all the business metrics the $\mathcal{SP}$ maintains.

For policies p^1_{av} to p^2_{av}, an oscillation case is impossible because any of them cannot occur more than once during any availability interval (1 month). Also, if all of them occur during the same availability interval, the corresponding SLA is terminated. Termination is still a safer "state" than a "looping"!

For policies p_{dep}, p_{add} and p_{rem} there is a potential existence of a static loop. This is because p_{dep} and p_{add} request additional server units while p_{rem} requests an operation which nullifies their actions by freeing one server unit. Thus, further analysis is required on these policies.

With this semi-formal analysis, the $\mathcal{SP}$ should be relatively assured that the generated SLS will achieve the goal of the input SLA of listing 1. However, there is still a subtle error which was not discovered until after observing the execution of a batch of randomly generated simulations.

For some randomly generated input parameters which respect all the above stated constraints, the system still enters an infinite loop oscillating between policies p_{add} and p_{rem}. By analyzing closely this case we found that the constraint $0 < th_r < th_a \leq 1$ is not sufficient. This leads us to consider the case when p_{rem} can be automatically triggered once p_{add} is triggered and vice versa.

First, let's recall that:

- $ws.load = |connections|/ws.cp$, where
  - $|connections|$ is the current number of end customer connections.
  - $ws.cp = su(rt) * su.size$, i.e web server capacity = capacity of one server unit times the number of allocated server units.
- p_{add} increments $su.size$ by 1 while p_{rem} decrements it.

Just before p_{add} is triggered, i.e. an end customer is terminating a session, we define: $z = su.size$, $n = |connections|$, number of connections, $s = su(rt)$, $c = n/su(rt)$, and $z_{max} = cp_{max}/su(rt)$.

For p_{add} to be triggered right after an end customer exits we should have:

$$\frac{n-1}{s \times z} < th_a \leq \frac{n}{s \times z} \tag{6}$$

The successful execution of p_{add} increments z to $z+1$ to translate the adjunction of a new server unit to the SLA instance.

For p_{rem} to be triggered right after p_{add} has executed (creating a loop), we should then have:

$$\frac{n-1}{s \times z} > th_r \geq \frac{n-1}{s \times (z+1)} \tag{7}$$

By putting these two inequations together and simplifying we obtain the condition:

$$\frac{th_a}{th_r} \leq 1 + \frac{1}{z}, \forall z | 1 \leq z \leq z_{max} \tag{8}$$

Hence, to avoid a static loop starting from p_{add} we prove that it is necessary and sufficient to have ($\rceil$ is the not operator):

$$\rceil (p_{add} \rightarrow p_{rem}) \Leftrightarrow th_a > 2 \times th_r \tag{9}$$

The second case is to get the condition to avoid having p_{add} automatically triggered right after p_{rem} has executed. Following similar reasoning we obtain :

$$\frac{th_r}{th_a} < \left(1 - \frac{1}{z_{max}}\right), \forall z | 2 \leq z \leq z_{max} \tag{10}$$

From eq.9,10 we prove that the necessary and sufficient condition for the policy set of listing 9 to be loop free is :

$$\mathcal{SP} \text{ SLS is loop free} \Leftrightarrow th_a > 2 \times th_r \tag{11}$$

□

At this point the static analysis phase ends. The output of a static analyzer tool, if it existed, should be a recommendation to change the constraint on th_r to become compliant with eq.11.

The next step explores aspects of what we call a business-driven dynamic analysis of an SLS [5].

VII. Business-driven dynamic analysis

At a normal SLA operation, all triggered policies should execute properly. Conceptually, a triggered policy needs the approval of the policy decision point (PDP) before it can execute [1]. This implies the existence of a conceptual queue, or waiting room, for triggered policies which the PDP serves by scheduling them according to some predefined scheme. In practice, the PDP can be implemented as a hierarchy of PDPs distributed within the IT infrastructure. Our analysis will be based on the conceptual PDP of $\mathcal{SP}$'s IT infrastructure.

In this paper we assume two possible default services which the PDP offers to the the Triggered Policies Queue (TPQ):

1) FCFS: This is actually a variant of the traditional First Come First Served (FCFS) algorithm in which triggered policies are serviced in FCFS as long as there are enough available resources for their actions part. In the case where a triggered policy cannot execute because of unavailable system resources, the policy in question retains its order in the TPQ but the PDP skips it each time it processes the TPQ until resources become available for its execution. For $\mathcal{SP}$ SLS this case can occur for policies p_{dep} and p_{add}.

2) RND (for RaNdoM): In this algorithm, the PDP picks the next policy to run at random from the set of runnable triggered policies.

If $\mathcal{SP}$ chooses to contract a number of SLAs with a total maximum capacity exceeding the actual capacity it has in its servers pool, it can be proven that by taking additional concern at the TPQ scheduling level better overall business profit can be achieved.

VIII. Business driven TPQ scheduling

In this section, we introduce a new TPQ scheduling technique that is intended to provide a better handling of peak utilization times for $\mathcal{SP}$'s resources leading to better overall business revenue. This technique takes into consideration the runtime states of instantiated SLAs in servicing the TPQ.

Since the only policies which may incur delay are p_{add} and p_{dep}, we will only consider them for this analysis. The other policies can hence be serviced according to any default discipline (FCFS or RND) without loss of performance or business value.

The decision problem that the PDP has to solve when faced with a number of policies in TPQ requesting more resources than available (here server units) is to determine which policies to grant resources to, i.e. allow to execute, and which policies it will delay hoping that enough resources will be freed. Delaying a triggered policy can lead to a violation of SLOs and violating an SLO can cause penalties paid to the $\mathcal{SP}$ customers. The aim of $\mathcal{SP}$ is to configure its PDP's decision algorithm so as to reflect the goal of maximizing the business profit function Ψ defined in eq.1.

A. The Business aware TPQ scheduling decision problem

Delaying p_{dep} or p_{add} can lead to the violation of the availability SLO (slo_{av} in listing 3). slo_{av} is defined over the monthly average availability of the web application to end customers. Based on listing 1, the availability ($= ws.av$ in listing 3) of each SLA instance is defined as the fraction of successful service requests to the fraction of total service requests of end customers computed over a month time window.

Definition*: monthly availability of a web service*

$$ws.av = \frac{|\text{processed requests}|_{\text{in}av_w}}{|\text{total number of requests}|_{\text{in}av_w}} \tag{12}$$

When confronted with a sequence of p_{dep} and p_{add} policies in the TPQ, the PDP can utilize the information on the availability metric for the SLA to which each policy is associated in order to predict the *impact time* for each delayed policy. The impact time in this case is the time at which a violation of the availability SLO occurs.

We will make use of a *greedy approach* to the maximization of business profit. We approximate the maximization of Ψ to the minimization of the impact (i.e. loss) on Ψ for each decision cycle the PDP performs onto the TPQ.

Let P_{tpq} be the set of policies of type p_{dep} or p_{add} that are queued in TPQ at time t_0. The PDP constructs for each policy $p_i \in TPQ$ a tuple $(p_i, r_i, I(p_i))$. r_i is the time p_i was triggered. $I(p_i)$ is an impact probability function which gives for each $t \geq 0$ the expected impact on Ψ in the case p_i is delayed t time units after the current system time t_0. Based on this sets of tuples, the PDP has to make a decision in order to minimize the impact on Ψ.

B. Mathematical model of the $\mathcal{SP}$ SLA

Predicting the impact of delaying p_{dep} or p_{add} implies predicting the probability of violating slo_{av} in future time. This implies predicting the evolution of availability ws_{av} over time for each $p_i \in P_{tpq}$.

To do so we model the state of an SLA instance as a tuple $M/M/C_t/C_t\,|A_t|D_t$. $M/M/C_t/C_t$ models a variable capacity markovian queue where: λ =rate of end customer requests, μ =service rate for a single customer request, the number of available server slots at time t
$C_t = ws.cp = su(rt) \times su.size()$, and all requests arriving at full load time get rejected (no waiting queue).

A_t denotes for each t the number of granted end customer requests (sessions) during the last availability window $[t - av_w, t]$. D_t denotes the number of the denied ones. $T_t = A_t + D_t$ represents the total number of arrivals during the last availability window. Requests arrival is modeled as a poisson source as it reflects the most common type of arrivals. Exponential service times denote the time during which a customer remains connected to the web service. In the case of a web site this can model the time of one customer session in the web site. A similar case applies to other types of servers such as audio/video streaming servers. Note that this does not contradict with the response time SLO as the response time represents the responsiveness of the web service to end customer queries during one end customer session.

The servers' capacity, in terms of the number of end customer sessions, varies within the discrete set $\{su(rt), su(rt), 2su(rt), \ldots, cp_{max}\}$.

In what follows we will focus more on policy p_{add}. The study of p_{dep} follows a similar approach.

Let N_t denote the number of end customer requests being serviced at time t. We have at any time $0 \leq N_t \leq C_t$

Note that all of A_t, D_t, T_t, C_t, and N_t can be easily obtained at runtime by defining corresponding metrics at the SLS level.

Policy p_{add} is triggered when the SLA load exceeds th_a. Let t_0 denote this time. In the following, when t_0 is used as a subscript the t is omitted for clarity.

Because markovian processes are memoryless, the PDP can take as input to its Impact Prediction Algorithm (IPA) the tuple $(t_0, A_0, D_0, C_0, N_0)$. The output is the probability function I. As a simplification of I, the IPA can determine the time it will take starting from t_0 until av_t drops below av_{min}, hence triggering an availability violation.

In spite of all the simplifications done, still remains the difficult problem of predicting the evolution of a Markovian process at transient state.

On page 78 of his book *Queueing Systems, Vol. 1* [6], Leonard Kleinrock, commenting on the transient solution of an $M/M/1/\infty$ queue, says 'This last expression is most disheartening. What it has to say is that an appropriate model for the simplest interesting queueing system leads to an ugly expression for the time dependent behavior of its state probabilities.'

More recently, Sharma describes in his book *Markovian Queues* [7] a novel approach to the transient analysis problem. He was the first to provide the transient solution for the $M/M/\infty$, $M/M/N$ and $M/M/2/N$ queues. Sharma states that for higher order queues the problem becomes much more complicated to be handled by currently known Algebraic techniques. This problem is till unsolved.

In order to make the policy decision-making process business aware, we will attempt to find an acceptable solution to the transient analysis of a $M/M/C_t/C_t$ queue that still respects the short time requirement for the PDP' decision. The decision making problem is further complicated by the fact that the PDP is concerned not only with predicting when the servers become fully loaded, but also with the prediction of when after that time the SLA availability drops below its minimum contracted value.

The next subsections present two prediction functions and their corresponding impact minimization algorithms, which we have implemented and for which we provide performance results in section XI.

C. Predicting the First Time to Degradation / Violation

As a first approximation we divide the prediction process into two phases: the *fill up* phase and the *time to violation* phase. In the first phase, the system starts with configuration $(t_0, A_0, D_0, C_0, N_0)$ and evolves to the new configuration $(t_f, A_f, D_f = D_0, C_f = C_0, N_f = C_0)$, where t_f represents the time when the SLA server units reach full load ($N_f = C_0$). During interval $[t_0, t_f]$ it is expected that no service request denial occurs as the SLA can still handle more load. The time to violation phase starts at time t_f and lasts until reaching the violation of the availability SLO (slo_{av}). It is described by configuration $(t_v, A_v, D_v, C_v, N_v)$, where

$$\frac{A_v}{A_v + D_v} \stackrel{\simeq}{\leq} av_{min} \tag{13}$$

Given this information, the PDP can predict that if it delays that p_{add} instance the corresponding SLA is expected to experience a violation of its slo_{av} at time t_v. That is, after $t_v - t_0$ time units.

The problem is hence amenable to providing an approximate but realtime solution of the time to fill up and time to violation phases.

1) Fill up phase - predicting t_f: The computation of t_f is based on the average behavior of $M/M/C_t/C_t$.

The incoming flow λ of end customer requests is subdivided into two sub flows. A first sub flow of rate $\mu \times N_0$ keeps busy the N_0 occupied server slots. This is because their aggregate average service rate is $\mu \times N_0$. The remaining sub flow is then

$$\lambda' = \lambda - \mu \times N_0 \tag{14}$$

λ' constitutes the new flow of incoming connections for the servers of rank $> N_0$ (after a simple reordering of server slots).

We now consider this new set of server slots separately. At time t the number of connected customers is N'_t. At time $t + dt$ this number will increase by dN'_t where:

$$dN'_t = \lambda' dt - \mu N'_t dt$$

$$\Rightarrow \frac{dN'_t}{dt} + \mu\, N'_t = \lambda' \tag{15}$$

We define $\rho = \frac{\lambda}{\mu}$ and $\rho' = \frac{\lambda'}{\mu} = \rho - N_0$. With the condition $N'_0 = 0$ we get:

$$\Rightarrow N'_t = \rho'(1 - e^{-\mu t}) \tag{16}$$

By putting:

$$t_f = t_0 + t'_f \tag{17}$$

t'_f is then the solution in t of $N_t = C_0$. Hence,

$t'_f = \frac{-1}{\mu} ln\left(1 - \frac{C_0}{\rho'}\right) \Rightarrow$

$$t'_f = \frac{1}{\mu} ln\left(\frac{\rho - N_0}{\rho - N_0 - C_0}\right) \tag{18}$$

Interestingly, this function is independent of t_0 and is related only to N_0, C_0, λ, and μ.

As a side effect of the approximation, this function returns a positive result only if

$$1 - \frac{C_0}{\rho'} > 0 \Rightarrow \rho > C_0 + N_0 \tag{19}$$

This indicates that the prediction function we just developed is expected to work when the corresponding SLA instance is using only a small percentage of the maximum number of server units that can be allocated to it.

In the following we will refer to the formula of equation 18 as the *FTD function*, for First Time to Degradation.

2) Time to violation phase - predicting t_v: Following the same reasoning, we assume that λ gets divided into two sub flows. The first sub flow of rate $\mu \times C_0$ works on keeping all server units busy. The second sub flow represents the loss flow and serves to count down towards availability violation.

As in the first case, we define our modified incoming flow as:

$$\lambda'' = \lambda - \mu \times C_0 \text{ , and } \rho'' = \frac{\lambda''}{\mu} = \rho - C_0 \qquad (20)$$

All customers in the poisson flow of rate λ'' get rejected. Hence, on average availability is expected to be violated at $t_v = t_f + t'_v$ where:

$$av_{min} = \frac{A_f + \mu \times C_0 t'_v}{T_f + \lambda \times t'_v} \Rightarrow t'_v = \frac{T_f \times av_{min} - A_f}{\mu \times C_0 - \lambda av_{min}}$$

Where $T_f = T_0 + \lambda t'_f$ and $A_f = A_0 + \lambda t'_f$.

$$\Rightarrow t'_v = \frac{T_0(av_{min} - av_0) - \lambda(1 - av_{min})t'_f}{\mu \times C_0 - \lambda av_{min}} \qquad (21)$$

Let $\triangle av = (av_{min} - av_0)$. The expected absolute time t_v at which a violation of the availability SLO will occur is given by:

$$\begin{aligned} t_v &= t_f + t'_v = t_0 + t'_f + t'_v \\ &= t_0 + t'_f + \\ &\quad \frac{T_0(av_{min} - av_0) - \lambda(1 - av_{min})t'_f}{\mu \times C_0 - \lambda av_{min}} \\ &= t_0 + \frac{1}{(C_0 - \rho\, av_{min})} \times \\ &\quad \left(\frac{T_0}{\mu}(av_{min} - av_0) + (C_0 - \rho)\, t'_f \right) \end{aligned}$$

Using algebraic computations, we obtain:

$$\begin{aligned} t_v &= t_0 + \frac{(C_0 - \rho)}{\mu\,(C_0 - \rho\, av_{min})} \\ &\quad \times \left(\frac{T_0 \times \triangle av}{(C_0 - \rho)} - ln\left(1 - \frac{C_0}{\rho - N_0}\right)\right) \end{aligned} \qquad (22)$$

□

t_v represents, on average, the time at which a violation is expected to occur if meanwhile p_{add} is not granted the execution privilege. It does not necessarily reflect the actual time of first violation at runtime.

It is interesting to note that this formula can be computed in O(1) if the values of $\{C_0,\ T_0,\ N_0,\ av_0\}$ are available. Fortunately, the metrics instantiated from the developed $\mathcal{SP}$ SLS (listing 11) can provide this information instantly at runtime. This has a definite advantage at runtime over any formula which predicts the exact transient evolution of an $M/M/C_t/C_t\,|A_t|D_t$ queue, even though such a formula has not been discovered yet [7].

In the following we will refer to the formula of equation 22 as the *FTV function*, for First Time to Violation.

D. Impact Minimization Scheduling Algorithms

Provided with an approximation for t_f and t_v, the PDP can be configured to use several possible algorithms for the runtime minimization of impact on the business profit function Ψ. In this work we developed three different algorithms which are presented in this section. The performance of these algorithms is evaluated through simulations in section XI.

For each triggered policy $p_i \in SLA_{p_i}$, the PDP will create a tuple $(p_i, SLA_{p_i}, tt_{p_i}, td_{p_i}, tv_{p_i}, Pn_{pi})$, where: tt_{p_i} corresponds to the triggering time of p_i, td_{p_i} is the time of the next service degradation phase (corresponds to t'_f in eq.17), tv_{p_i} is the expected time of availability SLO violation (t_v in eq.22) in case p_i is delayed; and Pn_{p_i} is the penalty incurred based on the rules defined in SLA_{p_i} (one of $\{p^1_{av}, p^2_{av}, p^3_{av}\}$ depending on the runtime state of SLA_{p_i}).

The PDP has, among other possibilities, the following set of different scheduling algorithms to select from the TPQ the next policy to execute:

- Select the one with the first(lowest) time to violation, $Min(tv_{p_i})$. We call this algorithm FTVF, for First Time to Violation First.
- Select the one with the first(lowest) time to degradation, $Min(td_{p_i})$. We call this algorithm FTDF, for First Time to Degradation First.
- Select the one with highest penalty first, $Max(Pn_{pi})$. We call this algorithm HFPF, for Highest First Penalty First.

This selection is applied to those policies whose action part can be satisfied in terms of resource availability. Policies whose actions require more resources than available are delayed for the next TPQ iteration.

In the remainder of this paper we will use simulations to evaluate the performance of the proposed algorithms and study how they compare to the default FCFS and RND scheduling.

IX. The $\mathcal{PS}$ Simulation environment

The first problem faced when implementing the $\mathcal{SP}$ generic SLA use case is the lack of a simulation environment for testing the performance, correctness, and other properties of policy based solutions. We therefore developed a full fledged policy simulation tool $\mathcal{PS}$ [8] to use for the $\mathcal{SP}$ generic SLA use case. However, we have designed $\mathcal{PS}$ in such a way that it can be used by the wider SLA and Policy-based management research community. In the following, we will describe briefly $\mathcal{PS}$ main characteristics.

The policy simulator $\mathcal{PS}$ implements a discrete event simulation system based on the process interaction world view. The design of $\mathcal{PS}$ follows that of the business driven management framework we proposed in [5]. This includes an extended architecture for enabling business driven management that has policy support at its core foundation. $\mathcal{PS}$ offers the means to specify SLAs/SLSs, roles, policies, refinement/mapping rules, constraints, low level and high level metrics, events, as well as the functional details of a PDP.

$\mathcal{PS}$ builds on the open source package *javaSimulation* [9], which is a Java package for process-based discrete event simulation. The design of javaSimulation follows closely the design of the SIMULA programming language.

X. Generic $\mathcal{SP}$ SLA simulation package

This package (figure 2) was built as a simulation instance which we run over $\mathcal{PS}$. The $\mathcal{SP}$ generic SLS of listing 11 was implemented as a single class descendent of the generic SLA class. The class contains two instances of the class Role implementing the service provider role ($\mathcal{SP}$) and the customer role ($\mathcal{C}$) respectively. The Role class abstracts a set of commitments of a party in an SLA. The same hierarchy is constructed for policy groups, policies, metrics, and events. Each $\mathcal{SP}$ role has a serverGroup instance which manages a set of server units obtained from a serverPool component. A Poisson traffic source is attached to each serverGroup and is used to simulate session requests of end users. At the reception of each session request, the severGroup object tosses an exponential random number to simulate an exponential service time.

Almost all communications between the simulation components are done via events (ref. refrules for SLO violations). The event service allows any component to register as a source of a given event type. Time events (timeout counters) are also supported as a special event type. Another component can register as a listener to the same event type from that event source and the event service manages this relationship. The server pool, for example, generates an event each time it receives, accepts, denies or terminates a session. Leaf metrics propagate information they receive from server pool events and other components up to higher level metrics ($A_t, D_t, T_t, Av_t, NP_t, N_t, C_t, etc.$) until reaching the overall business profit function Ψ.

Finally, graph components have been hooked to several metrics to report their evolution in time. MATLAB has been used as a graph plotter because of the significant large size of the generated graph files.

XI. Simulation Results

In order to validate the policy based implementation of the generic $\mathcal{SP}$ SLS solution and, more importantly, to test the performance of each of the impact minimization scheduling algorithms (HFPF, FTDF, and FTVF) against the basic FCFS and RND, we generated an acceptable number of different simulation instances and analyzed their outputs.

We conducted a number of 330 simulations grouped into batches of five for the five scheduling algorithms (RND, FCFS, HFPF, FTDF, and FTVF), making a total 66 batches. We used three Windows machines (Intel Pentium-4, 1.6GHz), two Sun OS machines (SUNW-Ultra-4, 300MHz), and two Solaris 8 machines (Sun Ultra 60, 450 MHz). The simulations run in a total cpu time of ∼245 days with a median runtime per simulation of 12.11 hours.

The simulations were selected from a spectrum of 960 inputs generated by varying a subset of the $\mathcal{SP}$ SLS parameters through ranges of values. Although we selected our use case to be as simple as possible, we still had to deal with more than two dozen parameters for each simulation. These included the simulation life time (three and six months were used), time granularity (one time unit was used to equal one second), TPQ scheduling algorithm (RND, FCFS, HFPF, FTDF, FTVF), number of SLA types (with each type determined by tuple

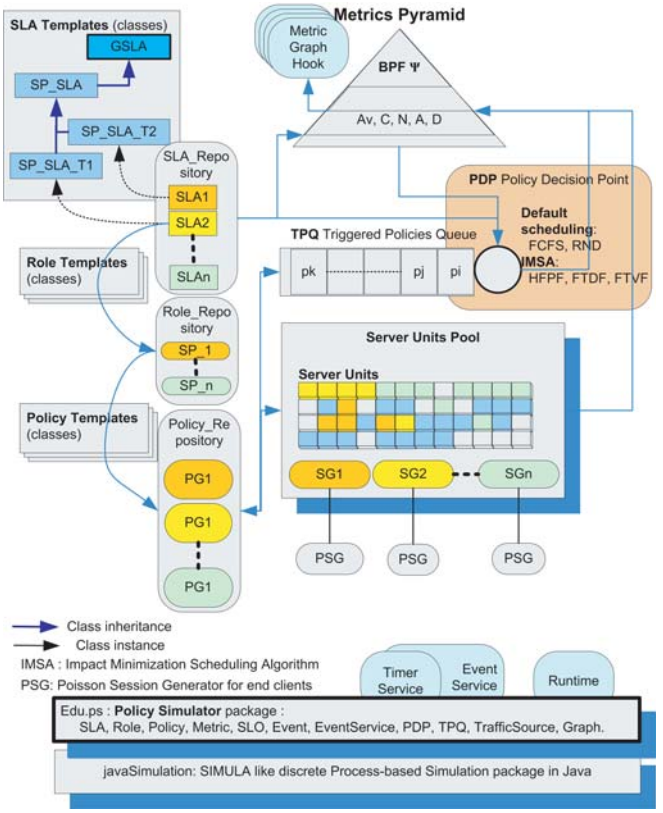


Fig. 2. $\mathcal{SP}$ testbed over $\mathcal{PS}$.

(cp_{max}, a, rt, $rt.ref$, av_{min}, av_w, λ, μ, $su(rt)$, penalties $\{r_1, r_2, r_3\}$, th_a, th_r and availability probe interval)), number of instances of each SLA type, and server pool capacity.

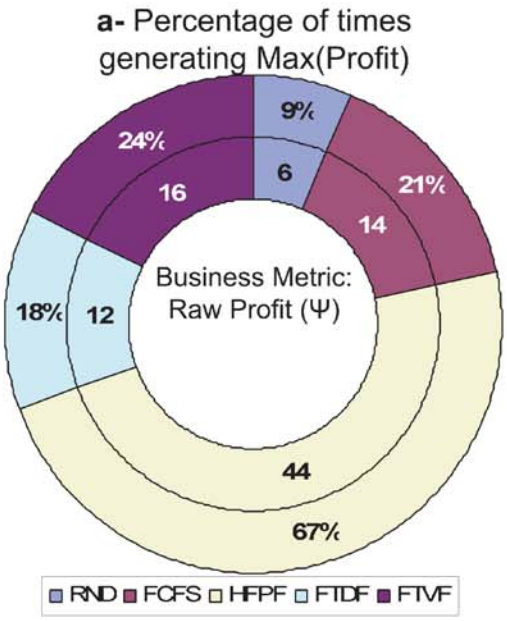
Compared to how simple the $\mathcal{SP}$ SLA is, this study gives a practical example of how complex testing and optimizing a policy based management solution can be.

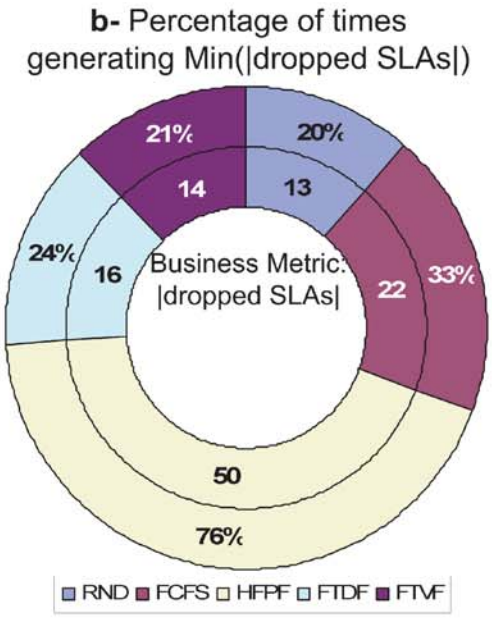
Figure 3-a summarizes the relative performance of each of the studied TPQ scheduling algorithms. The performance of an algorithm is equal to the business profit Ψ it generates. Each slice gives the percentage of time each algorithm performed best compared to the other ones. The inner doughnut slices give the actual number of batches where this happened. As a first observation, it appears that none of the algorithms performed best all the time. HFPF performed best 67% of the time, which is a considerable percentage. Second in the rank is FTVF with 24% , followed by FTDF and FCFS with 18% , and finally RND performing best in 9% of the total number of conducted simulations. The sum of these percentages is greater than 100% because there are cases where different algorithms produced the same highest business profit.

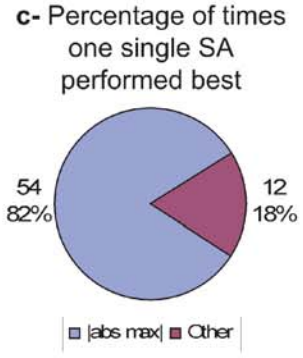
Figure 3-f traces, for all simulations batches, the relative performance gain of the best Scheduling Algorithm (SA) compared to FCFS. The gain is computed as:

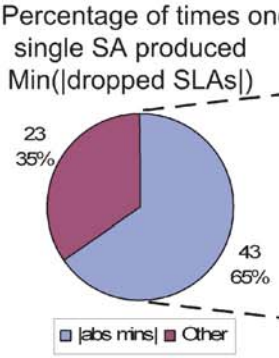
$$\frac{Max(\Psi_{SA}) - \Psi_{SA}}{|\Psi_{SA|}}\% \tag{23}$$

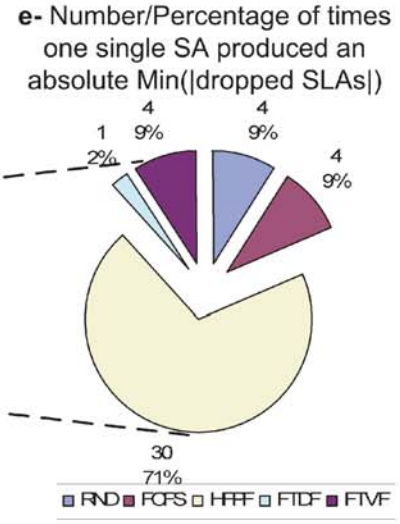
In this figure, a 0% value means an equal performance with FCFS, which occurs 18% of the time (3-a). The highest difference was in batch 3 in which HFPF performed best and produced 1000% better than FCFS. In batch 59 FTVF

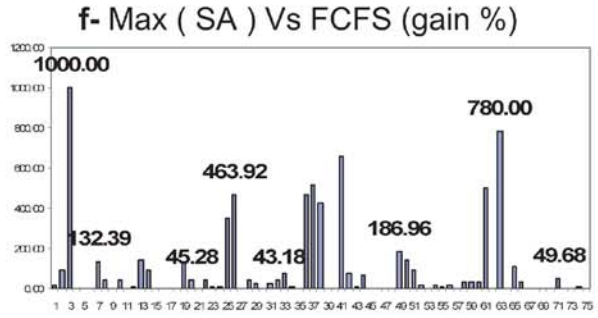


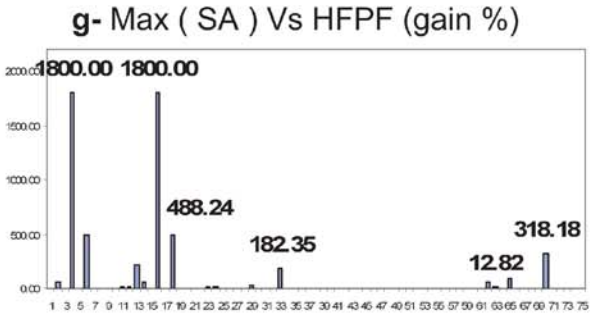


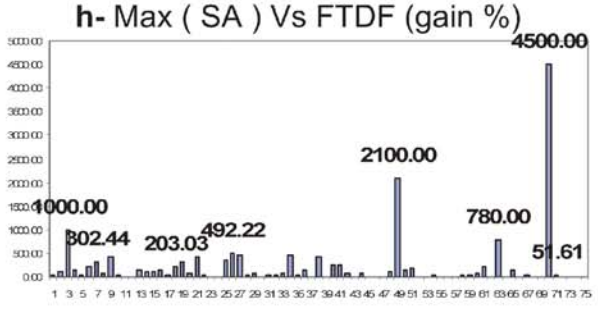












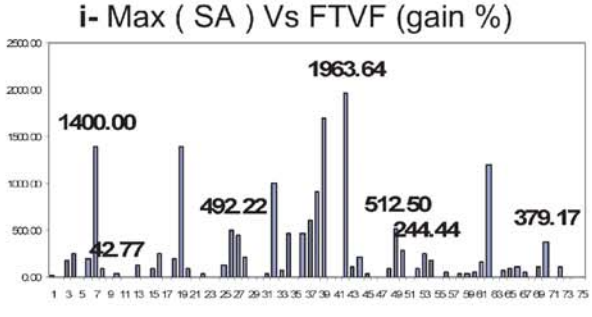


Fig. 3. Performance results of $\mathcal{SP}$ simulations over $\mathcal{PS}$.

generated the best net profit with a value 780% higher than the one generated by FCFS. The average gain is 119% with a median value of 34%. It is interesting to note that graphs similar to 3-a were obtained when comparing the other scheduling algorithms. For example, in batch 18 FTVF performs 1800% better than HFPF, and in batch 55 HFPF performs 1964% times better than FTVF! All these results are summarized in figures 3-[g-i]. Figure 3-c tells that the percentage of times only one single SA scored best is 82% while in 18% of the times only more than one SA scored the best performance. This shows the importance of collecting knowledge about which algorithm is expected to perform best before actually using it.

Another way to compare the performance of each SA is to track the number of SLAs which failed to continue their execution due to the occurrence of three successive availability violations (clause 4-b of the $\mathcal{SP}$ SLA). The termination of an SLA represents a considerable loss as it implies a cut in customer periodic payments. Figure 3-b gives the percentage of times an SA has generated the least loss in terms of the number of dropped SLAs. The service provider can use this business level metric to decide which SA to use instead of using the net profit metric.

Interestingly, figures 3-a and 3-b, however different, still have a strong similarity. An SLA is lost when it experiences three successive availability violations within the same availability window, i.e one month (listing 1, clause 4-b). Algorithm FCFS scores the lowest number of dropped SLAs 33% of the time while it achieves best business profit only 21% of the time. RND, HFPF, and FTDF also saw their score reduced with RND witnessing the highest relative decrease by going down from 20% to only 9%. Only FTVF generated the smallest loss in SLAs 21% of the time while it achieved best profit 24% of the time which represents an increase of 3%. This property distinguishes FTVF from the other algorithms and allows us to deduce that when FTVF manages to keep a higher number of SLAs alive it also manages to keep the number of experienced violations to the minimum. The random algorithm RND on the other hand performed worst in this regard. In all the 11% cases ($= 20 - 9$) where RND generated a lowest number of dropped SLAs that is shared with at least another SA, RND failed to give a better net profit. This suggests that the random scheduling has to be avoided whenever possible as it fails to manage the TPQ better than the other algorithms.

The analysis of the output graphs for the business profit metric shows that although some algorithms managed to keep a higher number of SLAs alive they still failed to provide best net profit because the remaining SLAs experienced recurrent availability violations causing penalties on the overall business profit.

Figure 3-d shows that in 65% of the time, i.e. in 43 batches out of the 66, only one single SA generated the lowest loss in SLAs. Figure 3-e shows that 71% of this number, i.e. 30 batches, has been scored by HFPF while FTVF scored a unique lowest SLA loss in only 4 batches. A closer analysis showed that when an algorithm generates a distinctive low loss in SLAs it automatically generates a distinctive highest net profit. This means that the batches related to figure 3-e do all belong to the |abs max| area of figure 3-c (the 82% pie).

RND and FCFS together performed best 30% of the time. This result shows that there is still room for better scheduling algorithms that are yet to be developed. This case was theoretically predictable as all of HFPF, FTDF, and FTVF make use of a greedy technique that seeks to maximize the business profit Ψ by minimizing the local impact on it by individual policy actions. It is also known that greedy techniques in general have no assurance as to the generation of global optima. Hence, the performance of RND and FCFS supports this theoretical prediction by practical numbers.

Using these two business-level metrics, the $\mathcal{SP}$ can decide which algorithm to use based on whether it gives greater importance to the mere net profit or rather to keeping the maximum number of SLAs running. Additional business metrics can be taken into consideration, such as the number of experienced penalties, and the selection process can grow more complex than the study presented in here.

The results obtained clearly demonstrate the importance of conducting a simulation before deciding which scheduling algorithm to use. Given the number of parameters to tune, even for this simple SLA case, it was computationally infeasible to determine beforehand the best parameters which lead the best business profit. However, given a certain initial set of SLA types that the $\mathcal{SP}$ intends to advertise and a hardware configuration of the server units pool, it is possible to conduct extensive simulations to determine which scheduling algorithm is best and what SLA admission control policy to use for each SLA type.

XII. Related Work

This work intersects with many related efforts in the area of business, SLA, and policy based management.

Policy refinement is still a difficult research problem. Recently, QoS policy refinement, using event calculus and abduction, has been addressed by Arosha B. et al. in [10] with a use case for QoS management in differentiated services networks. The paper stresses the need for application specific policy refinement patterns and presents a tool that is being developed for that purpose. The refinement tool is proposed as an add on to the Ponder policy toolkit [3]. Javier R. et al. [11] consider a similar use case and apply on it on a generic policy refinement process that is based on policy hierarchies.

Our work complements these efforts by providing a refinement process which clearly sets the goal and details the SLS solution. In addition, it provides a proof of correctness of the SLS as well as an evaluation of its efficiency through simulation. The static analysis introduces new tests, such as the oscillation test, which helped detect and correct anomalies in the output SLS. Moreover, the dynamic analysis represents a completely new "post" refinement phase which demonstrated the difficulty of proving the efficiency of policy based solutions. We have shown how a policy simulation tool can assist in this regard.

The issue of policy stability has been raised recently in [12]. The paper uses control theory for studying the stability of feedback regulation on SLA-like policies in bilateral provider agreements. In the SLS static process, different aspects of

stability tests are considered: deadlocks, loops, and erratic behavior. The erratic behavior test aims precisely at detecting system "instability". The work studies specific aspects of policy stability. We believe that more research effort is still needed in this area.

On the monitoring loop side, the automated generation of metrics for SLA monitoring has been addressed in [13] and [14] for the special case of web service SLAs. In [13], Alexander K. and Heiko L. stress the need for customer side monitoring and provides a comprehensive XML based notation for the specification, but not for the automatic generation, of composite metrics for web services. In our work we have shown through a detailed example, but do not develop a full theory of, how metrics are identified and generated from a Service Level Specification (SLS) (section V-F).

For the business driven decision making during SLA execution, Buco [15] discusses a technique, independent of any PBM solution, to automate and/or assist service personnel to prioritize the processing of action-demanding quality management alerts as per provider's service level management objective. Our work is similar to this effort in that it also considers the maximization of profit by working on the dynamic prioritization (scheduling) of policy actions. The difference our work holds, in addition to targeting policy based solutions, is that we consider this profit maximization schemes right from the beginning of the SLA refinement process and all the way down to the dynamic (runtime) analysis phase.

Policy Management for Autonomic Computing (PMAC) [16] is a generic middleware platform developed at IBM to provide software components that can be embedded in software applications to reduce the cost of writing applications capable of taking input from a policy based management system. PMAC uses a policy information model inspired from the Common Information Model (CIM) [17]. It supports the system model adopted by the IBM Autonomic Computing architecture. At the highest level, it provides four main components: a policy definition tool, a policy editor storage for policy deployment and persistence, an autonomic manager for policy evaluation, and a managed resource-side component for policy enforcement. It is implemented in JAVA and offers the right balance in the specification of policies by providing two different policy languages: ACEL which is based on XML, hence verbose, and SPL which is concise and human friendly, making it easily editable in a text editor. In [16] PMAC is used to enhance configuration checking of storage area networks. However not too much is said about the implementation details. In the paper a set of static analysis techniques are proposed: Dominance check, conflict check (two policies specifying different configuration parameters for the same component), coverage check, as well as conflict resolution using dynamic relative priorities. The first three tests are only defined in generic terms but no mechanism is actually provided in order to implement them. Relative priorities between policies can be defined at specification time using meta policies as shown in other works as well. At runtime, the policy scheduling algorithms we proposed allow to set dynamic priorities between triggered policies. The challenge is how one can set these priorities in advance and yet maximize business level metrics during system life time. The scheduling algorithms we have introduced address precisely this concern.

Finally, the use of policy specification and management tools such as Ponder [3], PMAC [16], PDL [4], CFENGINE [18], and Cauldron [19] is helpful in implementing real scale policy based solutions. In many cases, however, full scale implementations are costly and policy simulation tools can provide a valuable alternative. Simulations are helpful in evaluating the validity, consistency and efficiency of a PBM solution prior to its deployment. The $\mathcal{PS}$ tool [8] used to implement the $\mathcal{SP}$ use case is to our knowledge the first policy simulator developed specifically for the evaluation of policy-based solutions.

XIII. Conclusion

This paper presented a new approach for business driven policy enforcement and its use in application hosting environments. The refinement process involved an iterative approach the output of which is a set of metrics and low level QoS policies structured into roles. The *static analysis* phase served in detecting static anomalies (deadlocks, loops, unreachable states, and erratic behaviors) in the generated SLS as well as discovering additional constraints important for the runtime stability of the SLS. In the *dynamic analysis* phase, we attempted to bridge the gap between low-level management actions and the high-level business profit of the service provider. This faced us with the difficult problem of the realtime prediction of the transient state of a variant of an $M/M/C_t/C_t$ queue, which we denoted as $M/M/C_t/C_t\,|A_t|D_t$. We solved this problem through mathematical approximation and used it to derive the policy scheduling algorithms FTDF and FTVF. We also proposed a third algorithm named HFPF which uses runtime SLO states to decide the runtime prioritization of the triggered policies.

Using $\mathcal{PS}$, the policy simulator tool we developed for the simulation of policy management solutions, all three algorithms were implemented along with two other default ones (FCFS and RND). For statistical significance, we ran a considerable number of simulations to benchmark the performance of these algorithms. The simulations showed interesting results perhaps the most important of them is that no single algorithm outperformed the others at all times. This confirms, at least for the $\mathcal{SP}$ use case, the importance of conducting simulations before choosing a runtime policy management mechanism for a particular SLA type.

Formal policy based refinement of SLAs as well as the bridging between high level business goals and low-level management actions are challenging research issues. We believe that this work has contributed to both areas. Most importantly, this work has shown the value, from the service provider perspective, in considering policy runtime dynamics.

References

[1] D. C. Verma, *Policy-Based Networking: Architecture and Algorithms*, N. Riders, Ed. Sams, Nov 14 2000.

[2] I. Aib and R. Boutaba, "Business-Driven optimization of Policy-Based Management Solutions," in *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*. Munich, Germany: IEEE, May 21-25 2007, pp. 254–263.

[3] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder policy specification language," in *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*. London, UK: Springer-Verlag, 2001, pp. 18–38.

[4] J. Lobo, R. Bhatia, and S. Naqvi, "A policy description language," in *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence artificial intelligence conference*, 1999, pp. 291–298.

[5] I. Aib, M. Salle, C. Bartolini, and A. Boulmakoul, "A business-driven management framework for utility computing environments," in *the Ninth IFIP/IEEE International Symposium on Integrated Network Management (IM 2005), short paper*. Nice, France: IEEE, May 16-19 2005.

[6] L. Kleinrock, *Queueing Systems, Vol. I: Theory*. Wiley Interscience, Jan 1975.

[7] O. Sharma, *Markovian Queues*, ser. Mathematics and its applications, E. Horwood, Ed. Ellis Horwood, 1990.

[8] I. Aib and R. Boutaba, "PS: A policy simulator," *IEEE Communications Magazine - Network & Service Management Series*, vol. 45, no. 4, pp. 130–137, Apr 2007.

[9] K. Helsgaun, "Discrete event simulation in java," Department of Computer Science, Roskilde University, Denmark, Tech. Rep. 1-1, Mar 2004.

[10] A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegkas, M. Charalambides, and G. Pavlou, "Policy refinement for DiffServ quality of service management," *IEEE eTransactions on Network and Service Management (eTNSM)*, vol. 3, no. 2, p. 12, 2^{nd} quarter 2006.

[11] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou, "A methodological approach toward the refinement problem in policy-based management systems," *IEEE Communications Magazine*, vol. 44, no. 10, pp. 60–68, Oct 2006.

[12] K. Begnum, M. Burgess, T. M. Jonassen, and S. Fagernes, "On the stability of adaptive service level agreements," *eTransactions on Network and Service Management (eTNSM)*, vol. 2, no. 1, pp. 13–21, Jan 2006.

[13] A. Keller and H. Ludwig, "The WSLA framework: Specifying and monitoring service level agreements for web services." *Journal of Networks and Systems Management*, vol. 11, no. 1, 2003.

[14] A. Sahai, V. Machiraju, M. Sayal, A. P. A. van Moorsel, and F. Casati, "Automated SLA monitoring for web services," in *DSOM*, ser. Lecture Notes in Computer Science, M. Feridun, P. G. Kropf, and G. Babin, Eds., vol. 2506. Springer, Nov. 16 2002, pp. 28–41.

[15] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu, "Utility computing SLA management based upon business objectives," *IBM Systems Journal*, vol. 43, no. 1, pp. 159–178, 2004.

[16] D. Agrawal, K.-W. Lee, and J. Lobo, "Policy-based management of networked computing systems," *IEEE Communications Magazine*, vol. 43, no. 10, Oct 2005.

[17] DMTF, "CIM policy model v.2.8," Distributed Management Task Force DMTF, Tech. Rep., Jan 2004.

[18] M. Burgess, "Cfengine concepts, version 2.1.10," Faculty of Engineering, Oslo University College, Norway, Tech. Rep., 2004.

[19] L. Ramshaw, A. Sahai, J. Saxe, and S. Singhal, "Cauldron: A policy-based design tool," in *Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*. IEEE Computer Society, Jun 2006, pp. 113–122.

Issam Aib received the MSc. and PhD. degrees in Computer Science from the University of Pierre & Marie Curie, Paris, France, in 2002 and 2007 respectively. He is currently a Postdoctoral fellow at the school of computer science of the University of Waterloo (Canada) where he is conducting research on policy-based and business-driven management of networks and distributed systems since 2005. He is the recipient of the best student-paper award of the tenth IFIP/IEEE International Symposium on Integrated Network Management (IM 2007) for his work on the optimization of policy-based management solutions [2].

Raouf Boutaba received the MSc. and PhD. Degrees in Computer Science from the University Pierre & Marie Curie, Paris, in 1990 and 1994 respectively. He is currently a Professor of Computer Science at the University of Waterloo. His research interests include network, resource and service management in multimedia wired and wireless networks. Dr. Boutaba is the founder and Editor-in-Chief of the IEEE Transactions on Network and Service Management and on the editorial boards of several other journals. He is currently a distinguished lecturer of the IEEE Communications Society, the chairman of the IEEE Technical Committees on Information Infrastructure and on autonomic communications. He has received several best paper awards and other recognitions such as the Premier's research excellence award.