PS: A Policy Simulator

Issam Aib and Raouf Boutaba
David R.Cheriton School of Computer Science,
University of Waterloo, Ontario, Canada
{iaib;rboutaba}@uwaterloo.ca

***Abstract*— This paper presents $\mathcal{PS}$, a *Policy Simulator* tool intended to serve in the validation and performance evaluation of policy-based management solutions. $\mathcal{PS}$ is a discrete process-based simulation environment which allows the specification of all major components required for a policy-based management solution. These include the ability to specify management policies, events, metrics, probes, service-level objectives, service-level agreements (SLAs), as well as business-level objectives. We describe the main components of $\mathcal{PS}$ and show how it has been used to evaluate the performance of a policy-based management solution for a web application hosting SLA.**

I. Introduction

Although research in policy-based management has been going on for more than a decade, it is still not easy to put into practice. This owes much to the theoretical and practical difficulties in proving not only the correctness but also the efficiency of policy-based solutions when it comes to the management of real scale systems with hundreds or even millions of policies interacting in a dynamic way.

A number of policy languages and architectures were proposed. However, effective techniques for refinement and consistency/completeness analysis remain to be developed. It is therefore reasonable that venturing into a full policy-based solution for managing one's enterprize infrastructure remains difficult to justify.

With the current state of art in policy-based management, it is possible to do some simple static analysis, mainly for the detection and resolution of conflicts between security policies. However, for the broader range of management policies, including quality assurance policies, there is no established theoretical basis for correctness analysis let alone efficiency analysis. Furthermore, no work has been done on modeling the dynamics of management policies at system runtime. A policy-based solution should provide, in addition to correct behavior, an adequate performance which justifies its adoption in real scale management systems.

In this regard, resorting to simulation as a low cost testing facility provides a sound alternative. Similar to network simulation tools, which were introduced to cope with the difficulties in modeling the dynamics of queueing systems, we developed the policy simulator $\mathcal{PS}$ to serve as a tool for the simulation and analysis of the dynamics of policy-based management solutions.

The paper first presents the architectural components of $\mathcal{PS}$ followed by more details on the implementation and usage of the $\mathcal{PS}$ package. Section V presents a policy-based service-level specification (SLS), which implements the SLA use case given in section IV. The use case shows the importance of both the refinement process and the efficient orchestration of policy execution at runtime. Section VI presents several runtime policy scheduling mechanisms which have been applied onto the generated SLS. Simulation results of the performance of each of those algorithms are then presented in section VIII. We finally conclude by discussing related works and pointing out possible enhancements to the $\mathcal{PS}$ functionality.

II. $\mathcal{PS}$ Architecture

$\mathcal{PS}$ is designed in a way to support a business-driven management that has policy support at its core. Figure 1 shows the architectural components of $\mathcal{PS}$. Policies are modeled after the Event-Condition-Action (ECA) paradigm and are conceptually stored into the policy repository (bottom-left of figure 1) constituting a fast-access knowledge base of actions to take at the occurrence of well specified events and system conditions. The event service (bottom-right of figure 1) allows event sources and event listeners to be registered and routes events to their appropriate listener(s). Any $\mathcal{PS}$ component can be an event source and/or listener. An *active* $\mathcal{PS}$ policy is triggered when both its event and condition parts are simultaneously satisfied. When this occurs, the policy state switches from *active* to *triggered* and the policy is sent to the triggered-policies queue (TPQ). There it will await for the policy decision point (PDP) to decide of the actual time of the execution of its *action* part.

The above cycle represents the typical behavior of a conventional policy-based system. It is illustrated by the *policy control loop* of figure 1. All the effectiveness and adaptiveness that policy-based management promises lies in the appropriate design of policies as well as the algorithms used by the PDP in order to properly orchestrate the queue of triggered policies.

At an upper level, the $\mathcal{PS}$ architecture supports SLAs and high-level business objectives. Low-level policies are generated, using the currently available refinement techniques and domain specific expertise, in a way so as to enforce the set of contracted SLAs and business-level objectives of the service provider.

The cycle of observing SLA-level and business-level states and deciding of what new low-level control actions or strategies to follow in order to maximize the business profit of the service provider is captured by the *policy business loop* at the upper-level of figure 1. This loop is not as straightforward to implement as the lower *policy control loop*.

Metrics and metric probes are used in $\mathcal{PS}$ as a means to track the states of SLAs and business objectives. They

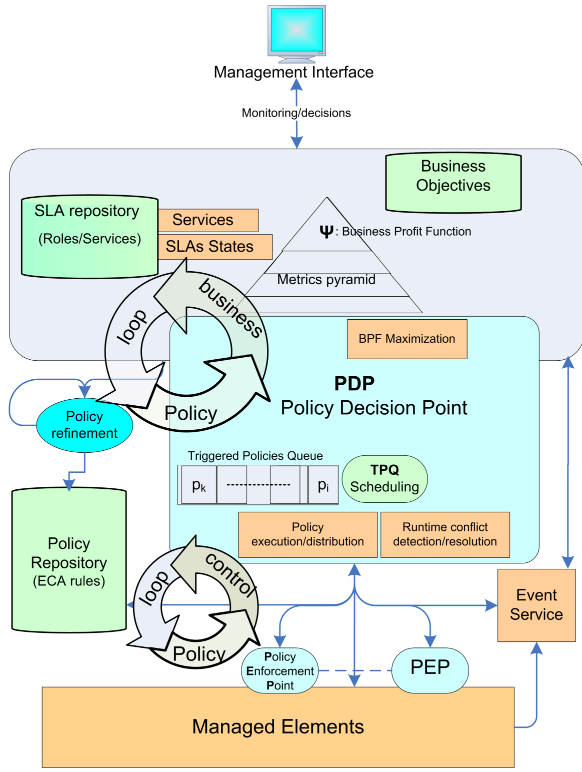


Fig. 1. Illustration of the $\mathcal{PS}$ architecture

represent a central building block for monitoring activities. For this, $\mathcal{PS}$ provides support for the definition of simple as well as compound (high-level) metrics. Several business-level and service-level metrics can be defined. These metrics are generally computed bottom-up from low-level resource metrics. At the top of the *metrics pyramid* lies the *business-profit function* Ψ. This metric reflects the measure of profitability of the business run by the service provider. In the general case, Ψ is a function of several service and business-level parameters including: service profitability, raw financial benefit, customer satisfaction, and market share. If carefully designed, the maximization of this metric should be the ultimate goal of the service provider. In the use-case presented in this paper, we consider a very simplistic Ψ which accumulates the raw monetary revenue gained from the running SLAs.

Although the goal of maximizing the business profit is clear to state, it is actually domain and even use-case specific to implement. The decoupling of the policy business and control loops offers a high-level of adaptiveness and efficiency to the system without loosing the critical *reactive* property of a conventional policy-based solution.

The next section expands on the usage and implementation of $\mathcal{PS}$ components.

III. $\mathcal{PS}$ IMPLEMENTATION AND USAGE

$\mathcal{PS}$ offers a discrete simulation environment based on the process interaction world view. It builds on the base of the open source package *javaSimulation* [6], which follows very closely the SIMULA programming language.

Each component of $\mathcal{PS}$ is implemented as a JAVA class and follows the life cycle automaton of figure 2-a. Life cycle state transitions occur through primitive calls triggered by components with the proper access rights. These primitives are implemented as overridden JAVA methods and are represented by the transition arrows in figure 2-a.

Right after it is constructed, a $\mathcal{PS}$ component is in an *idle* state. This means that it does not reserve/consume any system resources and has no responsiveness to events. The *compile()* primitive refers to the subcomponent generation process. The *deployed* state is that of a component which has been completely installed into the system and is only awaiting the green flag to start responding to events and interacting with its environment. A component can only be terminated if it is in the *idle* state. For example, an SLA instance can only be terminated when all of its the low-level policies, metrics, and other associated objects have been terminated.

The *compile()* primitive in $\mathcal{PS}$ is equivalent to the refinement process of high-level SLAs, service-level objectives (SLOs), or business-level goals into intermediate or low-level policy rules. In practice, this can be done online, off-line, or in a hybrid way. In the *online refinement* case, an automated or interactive refinement process is triggered at the time the *compile()* method of the $\mathcal{PS}$ object is called. In the *off-line* case, the refinement is done over the class (SLA, SLO, or other high-level class) of the $\mathcal{PS}$ object. This results in the generation of a hierarchy of components representing together the low-level implementation of the initial class, in addition to mapping code within each *compile()* method which indicates how each sub-component of that hierarchy will be generated at runtime. In the *hybrid* case, first an off-line refinement is carried out to generate intermediate-level components. These components are subsequently refined online whenever this is needed at system runtime. At present, $\mathcal{PS}$ only supports the off-line manual refinement because of the lack of existing off-the-shelf refinement tools.

A. Policy rules

Policies are modeled according to the Event-Condition-Action paradigm. As a $\mathcal{PS}$ component, a $\mathcal{PS}$ *policy rule* follows the life cycle defined in figure 2-a. In addition to this, when in the *active* state, the policy rule evolves within the sub-automaton of figure 2-b so as to reflect the behavior of an ECA rule.

At the interception of an event, and when the *condition* of the policy is met, a *policy_triggered* event is generated. By default, this event is intercepted by the policy decision point PDP (core component of figure 1). After that, the policy enters the *triggered* state whereby it "sleeps" waiting for the green light to execute its actions part. The PDP proceeds by queuing it (actually, a reference to it) into the TPQ. As long as the policy is in the triggered policies queue it remains in the *triggered* state. When the PDP decides to allow the policy to run, its state changes to *running* and an asynchronous call to method *policy.policyActions()* is made. This allows both the PDP and the policy to evolve independently. Synchronization facilities can be used in case the user wants to give more control to the PDP over policy execution. Once the *policyActions()* method returns, the policy returns back to the *active* state where it gets

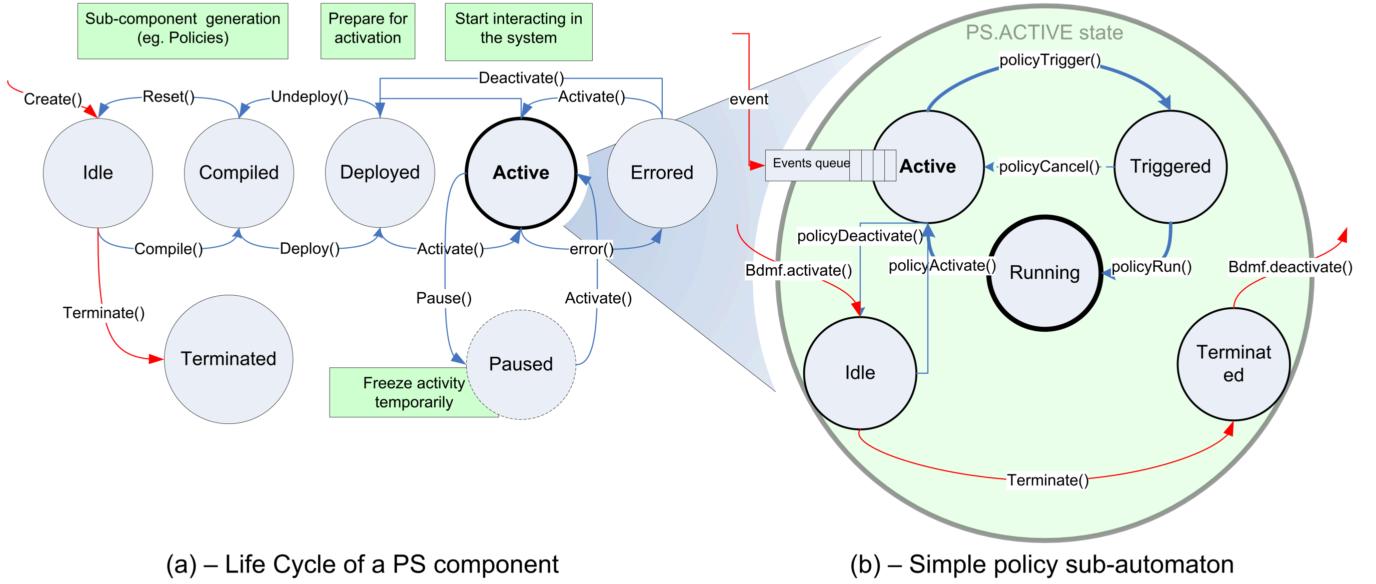


Fig. 2. Life cycle automaton of a $\mathcal{PS}$ component

to process the next available event, if any, in its event queue. It also generates an event to notify whether its *actions* part has executed correctly or encountered problems.

Some policies execute only once, such as the policy rule p_1 in figure 4. Other policies can be designed to execute a limited or even an unlimited number of times. For example, the policy rule p_2 (figure 4) runs each time its event and condition parts are met. The explanation of what p_1 and p_2 actually do is given in section V-B. When a policy reaches the maximum number of executions, it switches to the *idle* (figure 2-b) state.

A $\mathcal{PS}$ user should derive his own policy class from *ps.PolicyRule* then override the base methods *event()*, *condition()*, and *policyActions()*. By default, *event()* returns *true* and *condition()* returns *false*. The default dynamics of policy activation are taken care of by $\mathcal{PS}$.

B. SLA Model

$\mathcal{PS}$ offers an intuitive and generic SLA model (GSLA), whereby an SLA is made up of a service package and a set of parties, each of which plays a role in delivering or consuming part or all of the service package. A *Role* defines a set of duties of a party in the SLA. It can contain sets of SLOs and high-level policies. An SLS is the result of an SLA refinement process. This process is called using the SLA.*compile()* primitive, which recursively calls the compile primitives of its subcomponents. The result of this would be a set of high-level metrics, such as SLO and SLA-level metrics, in addition to a set of low-level policy rules which together represent the compiled SLS.

C. Metric probes and Graphs

$\mathcal{PS}$ provides support for defining simple as well as compound (high-level) metrics. Metrics are always enveloped within metric probes, which are objects able to listen and react to system events.

Basic data types are supported by the built-in *ps.Metric* class. Users can specialize this class to define their own metric types. A compound-metric probe listens to changes in sub-metrics probes as well as to other system events. At the top of the *metrics pyramid* (figure 1) lie business-level metrics which give the state of the system at the business level. Service-level metrics can also be defined and all of the Service, SLA, SLO, and Role classes support the adjunction of metrics that are needed to define their respective states. All possible data types are supported as metric values. The change of a metric value can be signaled to upper and lower metric probes so that they can update their respective values accordingly.

In order to control the propagation of metric updates, six propagation methods are supported: none, parents, children, parentsThenChildren, childrenThenParent, and generic. The generic method subsumes all of the previous ones and is to be used with extra care. Allowing metric updates to propagate in all possible directions is needed for the generic case, however update propagation loops have to be avoided. To enable the visualization, or simply the recording, of metrics evolution in time, $\mathcal{PS}$ supports graph objects as special metrics which hook on the top of other metrics and record all their new values, time stamps, and any other required data. Graph objects can also write all (time, value) pairs to persistent storage for future analysis.

IV. Web application hosting SLA use case

The following use case illustrates the importance of measuring the performance of a policy-based solution or, at least, having an understanding of how it would behave before it is actually implemented within a real scale infrastructure.

We consider a generic application hosting service provider $\mathcal{AP}$ which offers a set of SLA contracts. The $\mathcal{AP}$ has a pool of server units of size *sp.cp* with fixed CPU capacity and which he can allocate to different SLA instances. The advertised SLA types are derived from the simple generic SLA of figure 3.

1) Customer $\mathcal{C}$ is provided a web application hosting service with schedule sc
2) Max 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 $\geq av_{min}$
   - An i^{th}successive availability violation incurs a reward of $\$r_i \times ch$
   - At the 3^{rd}successive availability violation the SLA is considered void
5) Minimum average time to process customer service requests is rt ms
   - Otherwise, $\mathcal{C}$ is rewarded $\$rt.ref \times rt$

Fig. 3. Generic web application hosting SLA: $\mathcal{AP}$ SLA

This SLA states, in five clauses, that the $\mathcal{AP}$ offers a web 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* of ch monetary units.

Each instance of the tuple $(sc$,cp_{max},a,av_{min},r_1, r_2, r_3, rt, $rt.ref)$ can generate an SLA type which the $\mathcal{AP}$ can advertise to potential customers. 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 figure 3.

For simplicity, Ψ is defined as the sum of the raw financial profit gained from each contracted SLA. Consequently, the violation of any SLO incurs a penalty to Ψ. Hence, the $\mathcal{AP}$ needs to implement its policy-based solution so as to minimize the impact of SLA penalties while maximizing the usage of available system resources.

V. Refinement process and SLS generation

There is currently no standard methodology for the refinement of SLAs into low-level configuration and management actions. A semi-formal refinement methodology has been applied to the $\mathcal{AP}$ SLA in order to derive the SLS of figure 4 [2]. The generation of the SLS was based, in addition to the input SLA, on the following business-level policy.

A. Enforcement strategy

In this use case, we assume that the service provider follows a business strategy for resource allocation in which resources should be allocated only when needed and released otherwise. Applied to the $\mathcal{AP}$ SLA case, the SLS policies should request server units on a per need basis in order to maximize server resource utilization. At SLA instantiation time, a minimal number of server units are allocated. When the load on the allocated server units reaches a certain threshold thA of the current SLA capacity, a request is submitted to the server pool to get an additional server unit. Conversely, if a low threshold thR is reached, an action is triggered to release a server unit back to the pool of free server units. With this method, the service provider aims at obtaining a higher business profit than if it used a guaranteed enforcement approach.

B. $\mathcal{AP}$ SLS description

Figure 4 lists the $\mathcal{AP}$ SLS corresponding to the refinement of the $\mathcal{AP}$ SLA (figure 3) using the lazy-enforcement business policy. The details of this process do not fall within the focus of this paper and can be found in [2].

The SLS is made up of two roles, the service provider role $\mathcal{AP}$, and the customer role $\mathcal{C}$. The customer policy p_{C1} dictates to the customer to pay the $\mathcal{AP}$ on a monthly basis. The $\mathcal{AP}$ role defines sets of metrics, event types, events, and policies. Policies have been grouped into groups which share a common task. The SLO-violation notification policies generate an event when an SLO constraint is violated. These concern the capacity, customer payment, availability, and response time SLOs respectively.

The lazy resource allocation policies are composed of three policies. p_1 is a deployment-time policy which initializes the newly deployed SLA by requesting one server unit from the pool of server units. Policies p_2 and p_3 implement the lazy-enforcement approach by tracking the load of the server units available to the SLA instance and making the necessary actions each time a threshold is crossed. p_2 requests an additional server unit when the load exceeds thA while p_3 releases one server unit when the load goes bellow thR.

Policies p_4, p_5, and p_6 implement the availability violation penalties specified in clause 4 of figure 3. p_4 states that on the occurrence of an event e_1 of type $\overline{slo_{av}}$, that is, a violation of the availability SLO, the action that needs to be executed is to credit the customer account with $r1$. p_5 and p_6 implement the penalty clauses for the 2nd and 3rd successive violations respectively. The *where* clause enforces the one month "memory" on successive violations and makes sure that only one of p_4, p_5 or p_6 can be triggered at a time.

VI. Policy Scheduling mechanisms

When a policy is triggered, it is logically queued into the Triggered Policies Queue (TPQ) of the PDP. In order to study how the performance of the $\mathcal{AP}$ SLS changes based on the algorithm used by the PDP to schedule the TPQ, we implemented the following five scheduling algorithms.

A. First Time to Degradation First (FTDF)

By degradation, we mean the time at which service availability starts declining. This occurs, for example, when a triggered instance of policy p_2 (figure 4) is delayed by the PDP for a duration sufficient for the allocated server units to reach full capacity.

The FTDF algorithm selects among the TPQ policies, of type p_1 or p_2, the one with the nearest expected "first time to degradation".

B. First Time to Violation First (FTVF)

This algorithm is based on the prediction of the first time of violation of service availability. That is, the time at which availability drops bellow the minimum value specified in the SLA (av_{min} in figure 3).

The formulae of the prediction functions used by the FTDF and FTFV algorithms are given in [2].

sls $\mathcal{SLS}$ = {

- // SLS overall schedule
  schedule sch;
- // Service Provider Role
  role $\mathcal{AP}$ = {
  double thA, thR, av_w = 1 month;
  constraint $0 < thR < \frac{thA}{2}$, $0 < thA \leq 1$;
  metric $mWSCP = ws.cp$, $mAv = ws.av$
  metric $mMonthlyFee = payment.sum(month)$
  metric $mWSLD = ws.load$, $mRT = ws.rt$;
  eventType $\overline{slo_{cp}}$, $\overline{slo_{ch}}$, $\overline{slo_{av}}$, $\overline{slo_{rt}}$;
  event $\overline{slo_{av}}$ $e1, e2, e3$; // events of type $\overline{slo_{av}}$
- // SLO-violation notification policies
  policy pmWSCP = {
  on $(mWSCP > cp_{max})$ **do** $generate(\overline{slo_{cp}})$}
  policy pmMonthlyFee = {
  on $(mMonthlyFee < ch)$ **do** $generate(\overline{slo_{ch}})$}
  policy pmAv = {
  on $(mAv < av_{min})$ **do** $generate(\overline{slo_{av}})$}
  policy pmRT = {
  on $(mRT < rt)$ **do** $generate(\overline{slo_{rt}})$}
- // Lazy resource allocation policies
  policy p_1 = {
  at $sc.deployTime$ **do** $ws.add(1)$}
  policy pmP2ThAdd = {
  on $(mWSLD \geq thA)$ **do** generate(mWSLDEv) }
  policy p_2 = {
  on $mWSLDEv$ **do** $ws.add(1)$
  where $(ws.cp \leq cp_{max})$}
  policy pmP2ThRem = {
  on $(mP3ThRem \geq thR)$ **do** generate(mWSLDEv) }
  policy p_3 = {
  on $mWSLDEv$ **do** $ws.free(1)$
  where $(|ws.su| > 1\}$ }
- // Penalties for the violation of the availability SLO
  policy p_4 = {
  on $e1$ **do** $c.credit(r1)$
  where $not(p_5 \vee p_6)$}
  policy p_5 = {
  on $(e1 \rightarrow e2)$ // $e1$ followed by $e2$
  do $c.credit(r2)$
  where $((time(e2) - time(e1) < av_w) \wedge not(p_6))$}
  policy p_6 = {
  on $(e1 \rightarrow e2 \rightarrow e3)$ **do** $\{c.credit(r3); SLA.terminate()\}$
  where $((time(e3) - time(e1) < av_w))$}
- // Penalty for the violation of the response time SLO
  policy p_{rtv} = {
  on $\overline{slo_{rt}}$ **do** $c.credit(rt.ref)$ }
  }
- // Customer Role
  role $\mathcal{C}$ = {
  policy p_{C1} = {
  on every month **do** $\mathcal{SP}$.credit(ch)
  start at sc.activationTime
  }
  }

Fig. 4. The $\mathcal{AP}$ SLS

C. Highest First Penalty First (HFPF)

This algorithm is based on prioritizing the policy which is expected to engender the highest first penalty. In the SLS of figure 4, runtime penalty values are computed based on policies p_4, p_5, and p_6.

D. First Come First Served (FCFS)

This is basically the classical FCFS algorithm which serves policies in the order of their arrival.

E. Random scheduling (RND)

In this algorithm, the PDP picks the next policy to run at random from the set of runnable triggered policies.

It is interesting to notice that, for the majority of simulation instances, the same set of input SLSs produced significant differences in the overall system performance based on the TPQ scheduling algorithm employed. The use of $\mathcal{PS}$ helped us in identifying the behavior of the different algorithms. In the next section we summarize the simulations conducted and the results obtained.

VII. The $\mathcal{AP}$ SLA simulation package

This package was built as a simulation instance which we run over $\mathcal{PS}$. The $\mathcal{AP}$ generic SLS of figure 4 was implemented as a single class descendent of $\mathcal{PS}$ class GSLA. The class contains two instances of class Role implementing the $\mathcal{AP}$ and $\mathcal{C}$ roles respectively. The same hierarchy is constructed for policies, metrics, and events. Each $\mathcal{AP}$ role has a serverGroup instance which manages a set of server units acquired 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. 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. The event service also allows other components to register as listeners to the same event type from that event source.

Finally, graph components have been hooked to several metrics to report their evolution in time. Matlab was used as a graph plotter because of the significantly large size of the generated graph files. For example, each SLA instance lasted for six months in simulation time units (seconds). With an availability probe every five minutes, a number of 51840 tuples were hence recorded for the availability metric alone.

VIII. Simulation results

We conducted a number of 200 simulations grouped into batches of five, for each of the five TPQ scheduling algorithms. Several Windows and Sun-Ultra machines were used. The simulations run in a total CPU time of ∼86 days.

The initial aim in conducting these simulations was to get tangible data concerning how the scheduling of runtime policy

actions can affect the performance of the same policy solution. This section shows summarized results of the conducted simulations, in addition to values related to the performance of $\mathcal{PS}$ itself.

Figure 5-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 times a TPQ scheduling algorithm performed best compared to the other ones. There is no algorithm which performed best at all times. HFPF performed best 43% of the time, which is a considerable percentage. Second in the rank was FTVF with 18% then FCFS with 14%, RND 13%, and finally FTDF performing best in 12% of the total number of conducted simulations.

Figure 5-b traces, for all simulation batches, the relative performance gain of the best scheduling algorithm compared to FCFS. A 0% value means an equal performance with FCFS, which occurs 13% of the time (5-a). The highest difference was in batch 39 in which FTVF performed best and produced 1900% better performance than FCFS. In batch 5 HFPF performed best and did 1000% better than FCFS. It is worth noting that graphs similar to 5-b were found when comparing the other scheduling algorithms. For example, in batch 39 FTVF performs 1900% better than HFPF, and in batch 33 HFPF performs 9000% better than FTVF!

Given the number of parameters to tune, even for this simple SLA case, it was computationally infeasible to determine a-priori the best parameters which yield the highest business profit. However, given a certain initial set of SLA types that the $\mathcal{AP}$ intends to advertise and a hardware configuration of the servers pool, it is possible to conduct extensive simulations to determine which scheduling algorithm is best and what maximum number of SLA instances it is advisable to accept for each SLA type.

We conclude with a note on the number of events that were required for running a single simulation instance. Figure 5-c, column 1, shows that it is in the order of 10^5 to 10^6 with a maximum of 5.7×10^6 and an average of 2.2×10^6 events per simulation instance. Figure 5-c, colum 3, shows that the number of distinct event objects that were actually needed by each simulation has been lower by an order of magnitude. This was possible thanks to the event reuse and event sharing capabilities of $\mathcal{PS}$.

IX. Related Work

Although no tool was developed for the discrete simulation of policy-based management solutions, there has been important efforts related to the modeling, specification, and refinement of policies, all of which can be useful to combine with the functionality offered by $\mathcal{PS}$.

Policy Management for Autonomic Computing (PMAC) [1] is a generic middleware platform developed by 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 supports the system model adopted by the IBM autonomic computing architecture. 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 [1], PMAC is used to enhance configuration checking of storage area networks. However not too much is said about the implementation details.

First introduced in 1993, Cfengine [4] is perhaps one of the earliest policy-based configuration management solutions. It is fully specialized in the configuration of Unix-like and Windows networked computers. In Cfengine, a policy specifies what a *healthy* system state is using a declarative language. Cfengine then takes care of keeping the system always near to the healthy state by taking appropriate actions each time the system drifts to a *sick* state. Policies in Cfengine are similar to service-level objectives or high-level goals in the sense that they only specify the state to achieve but not how to actually do it. However, Cfengine is not common purpose and does not provide a generic policy-based platform as is the case, for example PMAC.

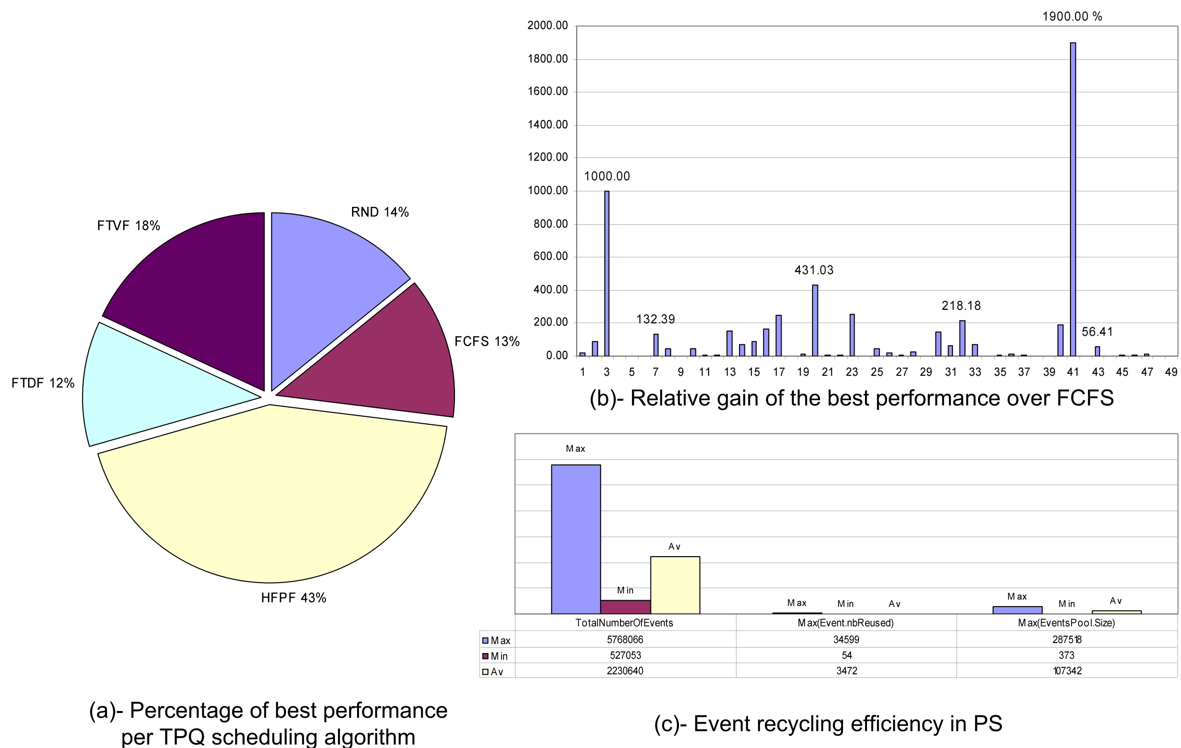
In another respect, policy refinement, although a difficult research problem, represents a key component in enabling policy-based management in the sense that it automates the implementation of high-level declarative specifications into low-level configurations and management rules. Recently, the refinement of management policies, using event calculus and abduction, has been addressed in [3] with a use case for quality-of-service management in differentiated services (DiffServ) networks. The paper stresses the need of 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 [5]. [7] considers a similar refinement use case and present a new methodological approach to the policy refinement problem by addressing the temporal execution of goals instead of using event calculus.

We believe that $\mathcal{PS}$ is complementary to the above efforts and is useful whenever simulations are appropriate to validate a policy-based solution before its implementation and or deployment. It would also be beneficial to extend the $\mathcal{PS}$' front end with some of the already existing specification and refinement solutions.

X. Conclusion and Future work

This paper presented the architectural components of $\mathcal{PS}$, a discrete event-based Policy Simulator tool which we developed for benchmarking policy-based solutions. All the ingredients required for simulating a policy-based management solution are supported by $\mathcal{PS}$. These include the specification of SLAs, SLOs, roles, metrics, events, as well as policy rules.

The presented use case served two purposes with different levels of emphasis. First, it showed how $\mathcal{PS}$ can be used to implement a detailed policy-based SLS specification and validate its correctness through simulation. The relative complexity of the output SLS compared to the simplicity of the input SLA shows how important simulations can be in providing factual data about the feasibility (runnability, validity) of the resulting



	TotalNumberOfEvents	Max(Event.nbReused)	Max(EventsPool.Size)
Max	5768066	34599	287518
Min	527053	54	373
Av	2230640	3472	107342

(a)- Percentage of best performance per TPQ scheduling algorithm

(b)- Relative gain of the best performance over FCFS

(c)- Event recycling efficiency in PS

Fig. 5. Summarized performance results of the $\mathcal{AP}$ use case over $\mathcal{PS}$

SLS. Second, it showed the significant impact different policy selection mechanisms can have on the overall performance of a policy-based solution.

Several features can be added to $\mathcal{PS}$ in order to enhance its operation and usability. For instance, there is a need for a policy editor front end to facilitate the specification of $\mathcal{PS}$ components: SLSs, SLOs, metrics, events, and policy rules. The development of add-on tools to assist in the refinement process would be of great value. Also, providing $\mathcal{PS}$ with a more sophisticated event service, which supports more than simple event reuse, would allow the specification of a wider range of policy rules. Finally, the current $\mathcal{PS}$ implementation supports only one PDP. A possible extension to $\mathcal{PS}$ is to include other PDP architectures with distributed policy decision and policy enforcement functions.

References

[1] D. Agrawal, K.-W. Lee, and J. Lobo. Policy-based management of networked computing systems. *IEEE Communications Magazine*, 43(10), October 2005.

[2] I. Aib and R. Boutaba. Business-driven optimization of policy-based management solutions. In *IFIP/IEEE* 10th *International Symposium on Integrated Network Management (IM 2007)*, Munich, Germany, May 16-19 2007.

[3] 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)*, 3(2):12, 2nd quarter 2006.

[4] M. Burgess. A tiny overview of cfengine: Convergent maintenance agent. In *the 1st International Workshop on Multi-Agent and Robotic Systems (MARS/ICINCO)*, Barcelona, Spain, Sept 13-14 2005.

[5] 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*, pages 18–38, London, UK, 2001. Springer-Verlag.

[6] K. Helsgaun. Discrete event simulation in Java. Technical Report 1-1, Department of Computer Science, Roskilde University, DK-4000 Roskilde, Denmark, March 2004.

[7] 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*, 44(10), October 2006.

XI. Biographies

Issam Aib is a PhD candidate at the University of Paris 6 in France. He is currently a visiting researcher at the University of Waterloo in Canada. He received his M.Sc degree in Networking from the University of Paris 6 in 2002; and Ingnieur d'Etat and Magister degrees from the University of Constantine, Algeria, in 1999 and 2001 respectively. His research focuses on policy-based and business-driven management of networks and distributed systems.

Raouf Boutaba is an Associate Professor of Computer Science at the University of Waterloo. His research interests include network, resource and service management in wired and wireless networks. He 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 Committee on Information Infrastructure and the IFIP Working Group 6.6 on Network and Distributed Systems Management. He has received several best paper awards and other recognitions such as the Premier's research excellence award.