A Formal Checklist for Self-Optimizing Strategy-Driven Priority Classification System

Patrizio Dazzi[1,2], Antonio Panciatici[1,3], and
Marco Pasquali[1,2]

[1] Institute of Information Science and Technologies (ISTI) – CNR, Pisa, Italy
[2] IMT Lucca Institute for Advanced Studies, Lucca, Italy
[3] Engineering PhD School “Leonardo da Vinci”, Pisa, Italy

Abstract. In this paper we identify the fundamental elements needed to make self-optimizing a strategy-driven classification system. The paper presents, using a formal notation, a checklist allowing classification-strategy designers to make autonomous their strategies. The proposed model is evaluated applying it to a real scenario in which a dynamic stream of elements is classified according to QoS constraints. The tests performed point out that the proposed approach can be fruitfully exploited.

1 Introduction

The *Autonomic Computing* is an initiative started by IBM in 2001 (aka ACI [19]). Its ultimate aim is to create self-managing computer systems to overcome their rapidly growing complexity and to enable their further growth. It takes inspiration from the autonomic nervous system of the human body. The nervous system controls important biological functions (e.g. respiration, heart beat, and blood pressure) without any conscious intervention.

The ACI focuses on the definition of foundations for autonomic systems and, in particular, on the definition of elements that are fundamental to make computing system autonomous. In a self-managing Autonomic System, the human operator acquires a new role: She does not control the system directly, instead she defines general policies and rules that are used as an input for the self-management process.

The *Autonomic Computing* paradigm is being considered for Grid Computing [15, 14] to address some issues that are typical of Computational Grids. For instance the management of a huge number of resources [29]. Moreover, an important issue of Grid Computing concerns its dynamism [4]. In general, the dynamism is considered from the resources point of view and many solutions have been provided [26, 10, 5]. In the present contribution, dynamism is studied from the point of view of Grid jobs: the computational requests of Grid users. Typically, a Grid job is an item consisting of a program code to be executed on a specific computer architecture coupled with the set of input data. A Grid job is characterized by computational requirements that need to be satisfied in order to execute it w.r.t. its QoS constraints. Generally, such requirements are specified in a XML file. This file is used by a Grid scheduler to decide where (on which computational resource) and when (in which order) each Grid job has to be computed.

There are several different design approach that can be exploited to implement a Grid scheduler [23, 25, 8]. A promising approach consists in multi-level schedulers (e.g. [18]). Their architecture consists of: a low-level scheduler, that deal with computational resources, and a meta-scheduler, that analyzes the Grid jobs hw/sw requirements and send them to the appropriate low-level scheduler. An interesting feature of the meta-scheduler is the classifier, that labels each job with a priority that define an order of execution among jobs. The low-level scheduler uses such a priority to order the jobs inside its local queue. In this way the first job to be scheduled is the one with the highest priority (w.r.t. the total order estabilished by the meta-scheduler).

During the last years the models and technologies derived from Grid Computing have been exploited to deploy Utility Computing [21] services. Utility Computing is a service provisioning model in which a service provider makes computing resources (and infrastructure) management available to the customer as needed, and charges them for specific usage rather than a flat rate. Like other types of on-demand computing, the Utility model seeks to maximize the efficient use of resources and/or minimize associated costs. At the main time, an important commitment for a utility-system service provider is to satisfy its user community needs. The provided service in our case is the execution of Grid jobs, hence correct job scheduling decisions must be taken in order to satisfy the maximum number of users. As more satisfied users trust the system, the more *trustworthy* the system becomes, and more users are attracted by it.

This work describes, using a formal notation, which are the fundamental elements needed to make autonomous a strategy-based [16] job classification system. The aim is to provide to the system mechanisms to recognize mismatches between actual and expected classifier behavior (with respect to a target behavior), and to provide to it the capability to adapt itself.

Our approach is not intended to be too abstract nor too formal. We describe with a high-level notation the elements that a strategy designer needs to care about designing an autonomous strategy. The challenge is to provide technique that enables software systems to evolve in order to remain useful [20], but to do so in a way that does not incur downtime as traditional maintenance processes do [28].

The paper starts with the formalization of strategy-driven classifier systems (Section 2). In Section 3, we present a case study in which we transform a classification strategy in an autonomic one. Such process is led by the proposed formalization. In Section 4 we present the experiments conducted and the related results. In Section 5 related works are described. Finally, in Section 6 we draw our conclusions and the path of future work.

2 Strategy-driven classification systems

A priority classification system S can be formally described with the following higher-order function:

$$f_S : I \times F_{strategy} \longrightarrow P. \tag{1}$$

Where I is the set of all possible item that need to be classified, $F_{strategy}$ is the set of all the possible strategy-functions and $P \subset \mathbb{N}$ is a finite set of priority values. A strategy-function $f_{strategy} \in F_{strategy}$ is defined as:

$$f_{strategy} : I \rightarrow P \tag{2}$$

The system S applies $f_{strategy}$ to each input $i \in I$ in order to obtain a priority value $p \in P$.

A priority classifier system can be defined **self-optimizing** when it is able to adapt itself in order to maintain good classification performance whatever an input comes into the system. From our perspective, a priority classifier offers a good performance when it is able to satisfy two requirements:

- the priority value assigned to each input item is well-proportioned to the item relevance (e.g. very important item must have a very high priority)
- the priorities assigned by the system must be coherent with a specified target policy

There are two different approaches to design a classifier able to achieve good performance, hence address the reported requirements. The first one consists in using a $f_{strategy}$ designed with a deep knowledge about the items that the classifier has to classify: a strategy-function strictly tailored to the item that will be classified, able to manage every possible input stream. The other approach is to design a self-optimizing classifier able to change the strategy-function taking care of the priorities assigned to a finite portion of the past classified items. This partial information is used by the self-optimizing classifier to analyze the trend of the priorities distribution assigned. Such information drives the tuning of the strategy-function behavior. This last design approach is the one we examine in this paper.

To model the self-optimization nature of the classifier we need to enrich the definition previously introduced for the strategy-function based classification system. Namely, we need to define how the strategy function can be modified to enhance its performance. This requires a reconfiguration mechanism able to modify the strategy-function and an evaluation mechanism able to evaluate the historical data and to drive, through the reconfiguration mechanism, the changes in the strategy-function.

Formally, the reconfiguration and evaluation mechanisms RM can be modeled with the two following functions:

$$F_{eval} : H \times C \longrightarrow C \tag{3}$$

$$F_{reconf} : C \longrightarrow F_{strategy} \tag{4}$$

Where H is the set of all possible historical data. C is the configuration of the RM, that is the set of all possible function tuning parameters. $F_{strategy}$ is the set of all possible strategy functions.

Every time that the system completes the computation of a new item, F_{eval} evaluates the priority distribution $d_{current}$ obtained analyzing the current historical data $h_{current} \in H$. If the result of this evaluation highlight an incoherence between $d_{current}$ distribution and the target policy d_{target}, F_{eval} generates a new configuration $c \in C$. If F_{eval} does not recognize any incongruence, it simply returns as new configuration the last one used by the system.

Once a new configuration is available the F_{reconf} uses this configuration in order to select a novel and more appropriate $f_{strategy}$ in the $F_{strategy}$ set. If the new configuration coincides with the last one adopted by the system, the RM simply return the last used $f_{strategy}$.

From an operative point of view, the behavior of the RM is defined by the composition of the F_{eval} and F_{reconf} functions ($F_{reconf} \circ F_{eval}$).

2.1 Realistic self-optimizing priority classification system

In the previous section we described an *ideal* self-optimizing classification system, that can access to an unlimited set of historical data, and which is able to generate a new strategy function every time that F_{eval} recognized any minimum difference between the $d_{current}$ and d_{target} priority distributions.

More realistic systems are characterized by reconfiguration mechanism that can only access to a finite set of historical data and which are not able to generate a new strategy function every time that they recognized a minimum difference. These limitations are mainly due to performance and data-knowledge problems. In a real scenario classification systems trigger the choice of a new strategy function based on checking the belongings of the current distribution to a permissible working region of the target distribution.

In order to formalize these systems we need to introduce two concepts: the vector space D and the polytope $\mathcal{P}$. The vector space D with dimension P, where P is the number of possible priority values, is a convenient way to represent priority distribution of historical data. In particular each priority distribution, obtained from $h \in H$, is represented with a vector $d = (d_1, ..d_k, .., d_P)$.

In this context each component d_k of d is a value grater than or equal to zero and less than or equal to one ($0 \leq d_k \leq 1 \; \forall_{k=1,..,P}$). It represents the percentage of jobs belongings to historical data h to which the system has assigned the priority $k \in P$. Since each component d_k is a percentage grater than zero and less than one, the sum of all d_k must be equal to one ($\sum_{k=1}^{P} d_k = 1$). We enforce this last constraints by using the Norm ($\|.\|_1$).

The polytope is defined as:

$$\mathcal{P} = \{d : \|d - d_{target}\|_2 \leq \delta\} \tag{5}$$

It characterizes the permissible working region as the set of all priority distributions that are "far" from the target distribution of a quantity less than or equal to a certain radius δ. In this context the measure of the distance is performed using the Euclidean Norm ($\|.\|_2$).

In other words, if $d_{current}$ belongs to the polytope $\mathcal{P}$ it means that the distance between $d_{current}$ and d_{target} is less than the fixed radius δ, hence $f_{strategy}$ does not need to be reconfigured, otherwise a new reconfiguration is performed.

From a formal point of view, in order to consider the polytope, the F_{eval} function must be changed. Indeed, it has to generate a new $f_{strategy}$ configuration only if $d_{current}$ does not belong to the polytope ($\mathcal{P}$). Formally:

$$F_{eval} : H \times C \times \mathcal{P} \longrightarrow C \tag{6}$$

3 Case study

In this section we describe how to apply the proposed formalization model to a real scenario concerning the scheduling of jobs on computing farms according to jobs QoS constraints. We show that the use of our approach would result in simplifying the turning of a non-autonomic strategy driven classification system in an autonomic one.

In general, the objective of a job scheduler is twofold: to optimize both the system throughput and the applications performance. This means that the scheduler tries to maximize the overall resource utilization guaranteeing the required level of QoS to applications.

To this end we propose an autonomic classification strategy that can be used in the online paradigm [24]. The job classifiers receive jobs from users, classify the jobs and eventually send them to a scheduling system. In our system, as depicted in Figure 1, the classifier is the front-end and the scheduler is the back-end: the first one is interfaced with users whereas the latter deals with physical resources.

The goal of our job classifier is to assign a priority value to each submitted job. The priority value defines a *job total-order of execution*. It is computed when a job is submitted to the front-end of the system. The job priority is a function of only job's parameters and it does not consider any system information, such as: the number and the type of machines, the software licenses availability, and the machines workload. The goal of our system is to exploit job attributes/characteristics and metrics to enable a job classification in an independent way with respect to the features of the computing platform used.

To validate our approach we consider a simple job classifier that use only a single job parameter.

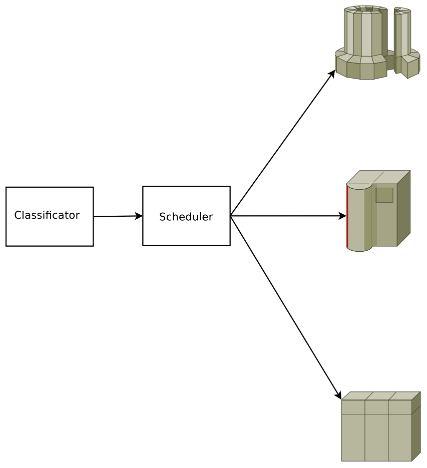


Fig. 1. System architecture

To this end, we introduce a strategy which is used to implement a job classifier, which exploits a job parameter, according to the description given in section 3.1. We also describe an autonomic version of the strategy showing how to turn the non-autonomic version into the autonomic one by using our formal approach.

3.1 Classifier Strategy

The aim of the classifier strategy we conceived is to classify input jobs according to a policy given by the system administrators. In our study, the policy is a target job priority distribution, and the aim of the classifier strategy is to carry out a job classification with respect to this distribution.

Each input job is characterized by a parameter X that can be an attribute of the job or a result of a function of different job's attributes. To compute the priority value of a job i, the classifier must know the *minimum* and *maximum* value that X can assume. In this way it is possible to set static thresholds (T_1, T_2 depicted in Figure 2) that identify subintervals in the [*minimum*, *maximum*] interval. These thresholds are specified according to the administrator policy.

The administrator of the computing farm can specify the target job priority distribution. The policy can be used to drive the strategy behavior [3]. The administrator can specify any priority distribution, according to specific requirements or/and the capacity of the computing farm administrated. In this particular case study there are three priority classes: *Low*, $Medium$ and $High$. Figure 2 shows an example where the distribution is 20% of jobs with *Low* priority, 60% of jobs with *Medium* priority and 20% of job with *High* priority.

It is important to point out that a strategy which uses static thresholds can be effectiveness only if the X parameter is uniformly distributed respect to its domain of definition.

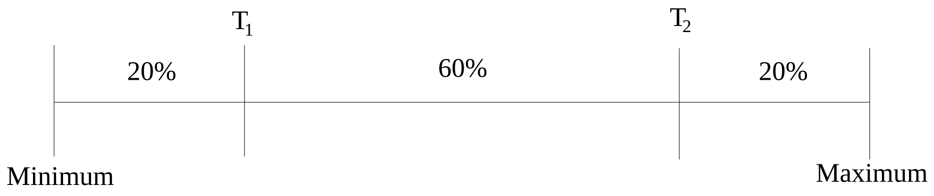


Fig. 2. Graphical representation of the thresholds given by the administrator policy.

Supposing for instance that the system has to classify a stream of jobs in two priority classes, $High$ and $Normal$. In particular it has to classify as high priority jobs the $q\%$ of the stream. Supposing also that the only threshold T is fixed to a given fraction $q\%$ of the X parameter domain ([*minimum*,*maximum*]). Jobs that have the X parameter less or equal to T are classified as $High$ priority jobs. Jobs that have the X parameter grater than T are classified as $Normal$ priority jobs.

If the distribution of X is uniform respect to its domain ([*minimum*, *maximum*]), only the $q\%$ of its possible values will be less or equal to T. Hence, only the $q\%$ of the jobs stream will be classified as $High$ priority jobs, while the remaining jobs of the stream will be classified as $Normal$ priority jobs. The target job classification policy is respected.

If the distribution of X is not uniform respect to its domain, the percentage of possible values of X that is less or equal to T will be different respect to $q\%$. Hence the percentage of jobs classified as high priority jobs will be different respect to $q\%$. The target job classification policy is not respected.

It is quite clear that this approach is still valid for an arbitrary number of thresholds.

3.2 Autonomic Classifier Strategy

In a real computing farm scenario it is not possible to assume that the stream of jobs has the X parameter uniformly distributed with respect to its range of values, e.g. if X represents the total amount of memory requested by a job, the total range value of X could be bound in [0, 4GByte] but a realistic distribution of requested memory is faraway from the uniform one. A such non-uniform distribution can cause an unprofitable priority values assignment. In order to avoid this situation, the system should be able to change the way in which it compute priorities, and to maintain a proper priority distribution. This behavior can be obtained by turning the non-autonomic classification strategy in an autonomic one. That consists in autonomic adjustment of the threshold (T_1, T_2) positions.

In order to better understand this concept, we suppose to have the scenario described in the previous section, the one with just one threshold T and with the X parameter not uniformly distributed. As already shown, due to the non uniform distribution of X, the observed percentage q' of jobs classified as $High$ priority jobs is different respect to q that is the desidered percentage. For instance, without loss of generality, let's suppose that q' is grater than q. This means that there are too much jobs classified as $High$ priority jobs, but also that the percentage q', of values of X that are less or equal to T, is grater than q.

In this case the autonomic behaviour consists in decreasing progressively the value of the T threshold. This strategy has the effect of reducing the percentage q' of the values of X that are less or equal to T, hence the number of jobs classified as $High$ priority jobs. This strategy is maintained until the observed percentage q' is not comparable with q, the desidered one. A such approach permits to respect quite well the target job classification policy also in case of high variability of the X parameter.

All the above observations can be done also for the specular scenario in which q' in less then q. In this case the threshold T is increased.

It is quite clear that this approach is still valid for an arbitrary number of thresholds.

From the formal point of view the turning process consists in providing a mapping between the formalization we present in Section 2 and the real scenario introduced in this case study.

The equations 7 and 8 define such a mapping.

$$F_{eval} : H_{job-window} \times C_{T_1,T_2} \times \mathcal{P}_{d_{target},\delta_{tolerance}} \longrightarrow C_{T_1',T_2'} \tag{7}$$

$$F_{reconf} : C_{T_1',T_2'} \longrightarrow (Job \to (Low,\ Medium,\ High)) \tag{8}$$

The historical-data set consists in a job-window with a fixed size k that stores the priorities assigned to the last k jobs processed.

Each job is classified as Low, $Medium$ or $High$ priority job depending by its X parameter and the position of thresholds T_1 and T_2. The strategy configuration C_{T_1,T_2} is defined through the two values T_1 and T_2. The autonomic behavior of the strategy is obtained changing the thresholds values in a convenient way.

The polytope is defined by using the target distribution d_{target} and the radius $\delta_{tolerance}$.

F_{reconf} takes as input the new configuration $C_{T_1',T_2'}$ generated by F_{eval} and returns an $f_{strategy}$ which assigns as priority one of the three possible values to each job.

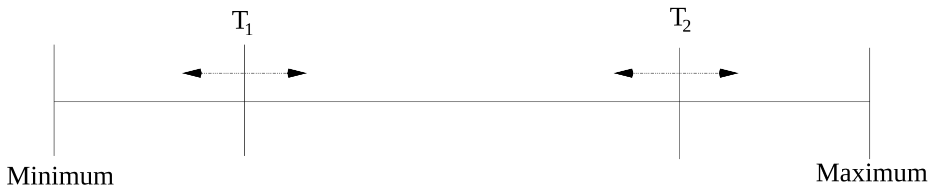


Fig. 3. Graphical representation of the sliding thresholds

4 Experiments

To evaluate the *Autonomic Classifier Strategy* (ACS) solution we conducted simulations applying the strategy to a stream of jobs characterized by a high variability of the X parameter. To compare ACS with *Classifier* (CS) we bind the X value in a predefined interval ([0, 5000]), but we assume that the distribution of X in this range can be not uniform.

To conduct evaluation we developed an ad-hoc event-driven simulator. For each simulation, a stream of 5000 jobs was randomly generated. Their X parameter was produced according to different distributions and described in each test. A simulation step includes: (1) selection and classification of new jobs, (2) check for correct behavior of the system, and eventually, perform the system adaptation (changing the thresholds values of the ACS strategy).

For each test, we fix two ACS's parameters: $\delta_{tolerance}$, which represents the accepted error in the resulting priority distribution (e.g. the distance between ACS results and the policy set by the administrators), and *job-window*, which stores the value of the last assigned priorities for a fixed number of jobs. $\delta_{tolerance}$ is set to 2% before which the thresholds are not changed, and *job-window* is set to 250, meaning that the instantaneous priority distribution is computed analyzing only the last 250 assigned priorities.

The aim of the experimentation phase was to carry out a priority distribution among jobs, according to a particular administrative policy. The policy constitutes the input of the proposed strategy: system administrators can define a relation among the number and the kind of jobs in the system.

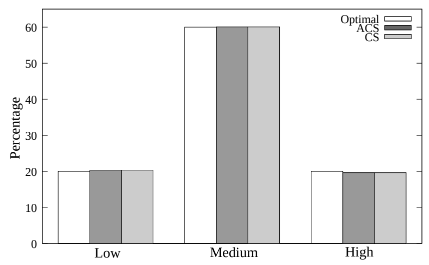
In our tests we have three administrator policies as described in Table 1.

Low	Medium	High
20%	60%	20%
70%	20%	10%
34%	33%	33%

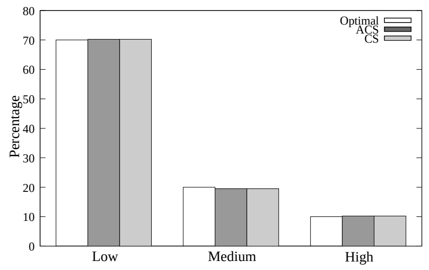
Table 1. Administrator priority distribution policies.

In our experiments we compare the ACS with the CS strategy and with the priority distribution given by the system administrators, which we refer as the optimal one in our case study (*Optimal*). For each test we measure the percentage of jobs that belong to each priority class for the three resulting classifications (*Optimal*, ACS, CS). Furthermore, we show how the thresholds change for the autonomic *Classifier* strategy.

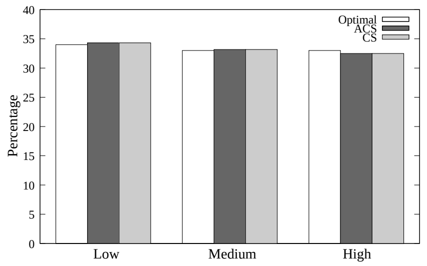
In the first test we consider a stream of jobs that has the X parameter uniformly distributed in the predefined interval. Figure 4 shows that CS and ACS behave in the same way and they are about optimal. Figures 4.a, 4.b, 4.c refer to (20-60-20), (70-20-10), (34-33-33) distributions in Table 1 respectively. We omitted the charts related to the thresholds variation because, in this case, their initial setting (i.e. $T_1 = 20\%$ and $T_2 = 80\%$ for the first distribution in Table 1) is never changed.



(a) Distribution 20-60-20.



(b) Distribution 70-20-10.



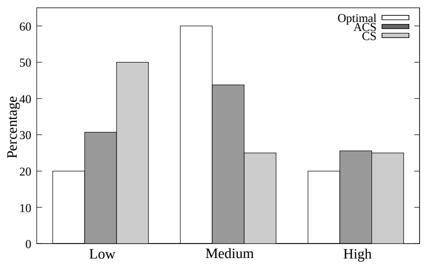
(c) Distribution 34-33-33.

Fig. 4. CS and ACS evaluation in the case of an uniform distribution of the X parameter

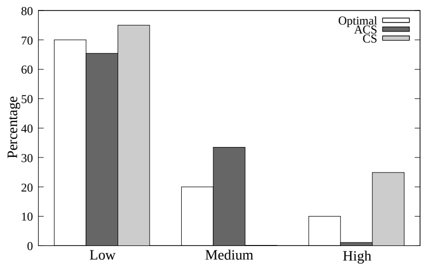
Figure 5 shows a test section in which the X parameter for each job is generated according to the Table 2. Each entry of that Table describes the number of generated jobs (first column), the minimum and the maximum value (second and third column respectively) that X can assume for those jobs, w.r.t. an uniform distribution of X between these limits.

#Jobs	Minimum	Maximum
1250	0	1000
1250	2500	3000
1250	0	500
1250	4500	5000

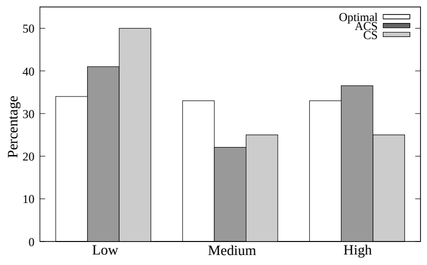
Table 2. Description of the job-stream with respect to the X parameter distribution.



(a) Distribution 20-60-20.



(b) Distribution 70-20-10.

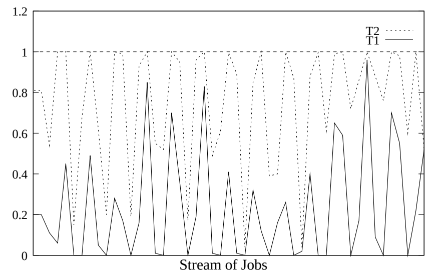


(c) Distribution 34-33-33.

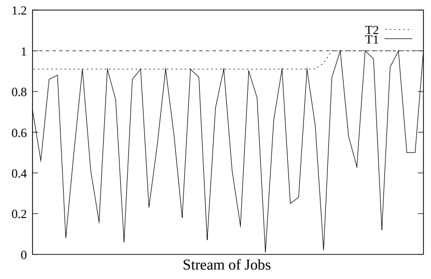
Fig. 5. CS and ACS evaluation in the case of an non-uniform distribution of the X parameter

As we can see in Figure 5, the priority distributions carried out by the ACS follow the trend of the *Optimal* ones. However, the CS distributions are distant from the optimum. This is because the CS classification depends on the input stream. As we can see in Figure 5.a and in Figure 5.c the distribution carried out by the CS is the same. On the other hand the ACS classifies jobs in a better way than the CS in respect to the given priority distribution. Moreover, for the $(70-20-10)$ distribution, CS classifies jobs as *low* priority class for 75% of the stream. This is because the jobs that in the other cases belong to the *medium* priority class here they belong to the left hand of the interval and they are classified as not important. Our ACS adapt its behavior to respect the administrator policy also in this case.

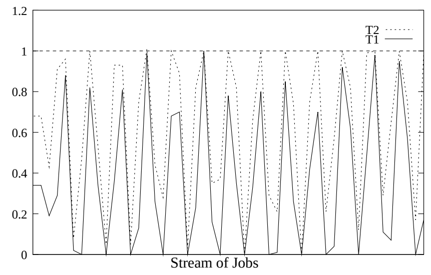
Finally, in Figure 6, we show how the thresholds of the ACS change at runtime (w.r.t. the three priority distribution defined in Table 1). On the X axis there are the elements of the input stream, on the Y axis there are the thresholds values. The thresholds define three interval for the three priority classes, the low priority belongs to the inteval $[0,\ T_1]$, the medium belongs to $[T_1,\ T_2]$ and the high belongs to $[T_2,\ 1]$. Obviously, The charts concern to a percentual rappresentation of the X parameter distribution interval. As we can see all the conducted experiments are characterized by a high variability of the thresholds values, only in the case shown in Figure 6.b the T_1 threshold is stable almost all the time.



(a) Distribution 20-60-20.



(b) Distribution 70-20-10.



(c) Distribution 34-33-33.

Fig. 6. Variations of the thresholds in the Autonomic Classifier strategy.

5 Related work

Adaptive performance tuning has only recently become conceivable, so only few papers address it directly.

Diao et al. [11] analyze how to choose certain parameters of the Apache web server in order to keep CPU and memory usage near a pre-set parameter. The authors make the assumption that there is an optimal setting for those parameters, and make no claim that the parameters impact the performance of the web server in a known way.

Raphael M. Bahati et al. define a policy as a notation to express required or desired behavior of systems and applications. In [3], they describe how policies are exploited and how they are realized as actions driving autonomic management in the context of managing the performance of an Apache web server. Their aim is to address the problem to express policies appropriate for an autonomic computing system and then map them to executable elements of the autonomic system.

In [28], Warren e al. describe mechanisms used to realize dynamic reconfiguration that must respect a number of fundamental issues when making run-time changes to a system. They suggest that such mechanisms have to behave according to: (1) the dynamic reconfiguration capability should not compromise applications *integrity/correctness*, (2) the run-time *overhead* introduced by a reconfiguration management facility should be acceptable, (3) the dynamic reconfiguration should be *transparent* to application developers. Their work are particularly concerned with preserving an applications integrity during periods of runtime change. They have extended OpenRec (a framework for managing reconfiguration of component-based applications [17]) with functionality which automatically verifies the structure of an application during periods of dynamic reconfiguration.

Other approaches, like ours, optimize performance maintaining a fixed level of service. For example Abdelzaher et al. [1] outline a system that maintains multiple complete content trees, each with a different quality setting. As workload increases, quality can be decreased in order to satisfy the maximum number of users. Additionally, Cohen et al. [9] use Tree-Augmented Naive Bayesian Networks to correlate system statistics to a high-level performance metric (compliance or non-compliance with required service levels). Unlike our work, this work relies on a specialized instrumentation layer.

Other work within the field of autonomic computing focuses on failure diagnosis [30, 7], file system organization [22], adaptive branch prediction [13], autonomous network creation [6], installation and configuration analysis [2] and utility function optimization [27].

6 Conclusions and Future Work

The complex nature of grid systems and distributed applications implies the need to automate their management in order to meet the operational and behavioral requirements. This paper highlights and outlines the fundamental elements that are needed to make autonomous a strategy-based classification system. The description is given using a formal notation.

We show that this formalization ease the design of autonomic classifiers. To show it, we present an interesting case study in which we describe how to exploit the formalization introduced. It is used to drive, in a real scenario, the process of making autonomous a strategy-based classifier used to annotate with a priority value the elements belonging to a stream of Grid jobs. We compare the performance results of the autonomic and non-autonomic systems. We point out that the system that exploits an autonomic behavior provides a priority distribution more adherent to the one required by the classifier administrator.

For the near future research we plan to improve the formalization outlined in this paper, both addressing more complex class of applications and describing the operational semantics of the elements required to make autonomous a classifier system. For the next future we plan to conceive a autonomic programming model based on an extended version of the autonomic formalization presented here.

References

1. Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 2002.

2. Gagan Aggarwal. On identifying stable ways to configure systems. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 148–153, Washington, DC, USA, 2004. IEEE Computer Society.
3. Raphael M. Bahati, Michael A. Bauer, and Elvis M. Vieira. Mapping policies into autonomic management actions. *icas*, 0:38, 2006.
4. Mark Baker, Rajkumar Buyya, and Domenico Laforenza. Grids and Grid Technologies for Wide-Area Distributed Computing. *Software – Practice and Experience*, 32(15):1437–1466, 2002.
5. Francine D. Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. pages ??–??, 1996.
6. Yu-Han Chang, Tracey Ho, and Leslie Pack Kaelbling. Mobilized ad-hoc networks: A reinforcement learning approach. *icac*, 00:240–247, 2004.
7. Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.
8. D. Clark. Scheduling of parallel jobs on dynamic heterogenous networks, 1995.
9. Ira Cohen, Jeffrey S. Chase, Moisés Goldszmidt, Terence Kelly, and Julie Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, pages 231–244, 2004.
10. Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459:62–??, 1998.
11. Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing web server performance with autotune agents. *IBM Syst. J.*, 42(1):136–149, 2003.
12. Joseph Hellerstein Fan. Characterizing normal operation of a web server: Application to workload forecasting and problem detection.
13. Alan Fern, Robert Givan, Babak Falsafi, and T. N. Vijaykumar. Dynamic feature selection for hardware prediction. *J. Syst. Archit.*, 52(4):213–234, 2006.
14. I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* MorganKaufmann, 1999.
15. Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–??, 2001.
16. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.
17. J. Hillman and I. Warren. An open framework for dynamic reconfiguration. *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, page 594603, 2004.
18. E. Huedo, R.S. Montero, and I.M. Llorente. A framework for adaptive execution on grids. *Journal of Software - Practice and Experience*, 2004.
19. IBM. Autonomic Computing Initiative. www.ibm.com/autonomic.
20. M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change.* Academic Press Professional, Inc., San Diego, CA, USA, 1985.
21. Ignacio M. Llorente, Ruben S. Montero, Eduardo Huedo, and Katia Leal. A grid infrastructure for utility computing. In *15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'06)*, pages 163–168, 2006.
22. Michael Mesnier, Eno Thereska, Gregory R. Ganger, and Daniel Ellard. File classification in self-* storage systems. In *ICAC '04: Proceedings of the First Interna-*

tional Conference on Autonomic Computing (ICAC'04), pages 44–51, Washington, DC, USA, 2004. IEEE Computer Society.

23. Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, pages 140–, 1998.
24. J. Sgall. *Online Algorithms*, chapter On-line scheduling, pages 196–231. Book Series Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Monday, April 10, 2006.
25. S. VADHIYAR and J. DONGARRA. A metascheduler for the grid, 2002.
26. J. L. Vázquez-Poletti, E. Huedo, Ruben S. Montero, and Ignacio M. Llorente. A Comparative Analysis between EGEE and GridWay Workload Management Systems. In *Proc. International Conference on Grid computing, high-performAnce and Distributed Applications on the Move Federated Conferences (GADA) 2006*, volume 4276 of *Lecture Notes in Computer Science*, pages 1143–1151, 2006.
27. W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *In Proceedings of the 1st International Conference on Autonomic Computing*, May 2004.
28. Ian Warren, Jing Sun, Sanjev Krishnamohan, and Thiranjith Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 37–46, Washington, DC, USA, 2006. IEEE Computer Society.
29. Jonathan Wildstrom, Peter Stone, Emmett Witchel, Raymond J. Mooney, and Michael Dahlin. Towards self-configuring hardware for distributed computer systems. In *ICAC*, pages 241–249, 2005.
30. Alice X. Zheng, Jim Lloyd, and Eric Brewer. Failure diagnosis using decision trees. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 36–43, Washington, DC, USA, 2004. IEEE Computer Society.