CObAPAS: Combinatorial Optimization based Approach for Autonomic Systems

Pedro F. do Prado, Luis Nakamura, Marcos Santana, Regina Santana
Institute of Mathematics and Computer Science - University of São Paulo
São Carlos, SP, Brazil
Email: {pfprado,nakamura,mjs,rcs}@icmc.usp.br

Omar A. C. Cortes
Federal Institute of Maranhão
São Luis, MA, Brazil
Email: omar@ifma.edu.br

Abstract—**This paper proposes a new approach to develop autonomic systems or transform traditional systems into autonomic ones. This approach is based on defining the autonomic module of the system as a combinatorial optimization problem. After that, a wide range of different techniques can be used to implement the autonomic module of the system. This study addresses two major problems: autonomic system specification and autonomic system evaluation. The former helps the developer to understand the system goals, constraints and scope, the latter, helps the developer quantitatively evaluate the efficiency of different techniques of implementing the autonomic module of the system. A case study demonstrates the viability and effectiveness of the proposed approach.**

***Keywords*–Autonomic systems; combinatorial optimization based approach for autonomic systems; QoS-aware service selection; combinatorial optimization problems; performance evaluation.**

I. Introduction

The concept of self-adaptation is presented in many research areas like: biology, chemistry, logistics, etc.. Self-adaptivity in computer-based systems is relatively newer. Some of the first references to self-adaptive computer systems are from the late 1990s. The term self-adaptation covers multiple aspects of how a system reacts: Self-Awareness, Context-Awareness, Self-Configuring, Self-Optimizing, Self-Healing and Self-Protecting. There are two approaches for creating self-adaptive systems: centralized and decentralized. In the centralized one, the analysis and planning are concentrated in one single entity. Furthermore, this form of self-adaptation has the advantage of cohesiveness and low communication overhead if compared with a decentralized mechanism [1]. An **Autonomic System** (AS) is an example of centralized self-adaptive system. On the other hand, decentralized self-adaptation, distributes the analysis, planning, or the feedback mechanism among different parts of the self-adaptive system. **Autonomic computing** (AC) is the computing paradigm behind an AS. The general idea is to mimic the autonomous nervous system of humans, which concentrate itself on higher-level objectives, instead of more specific and detailed aspects. For example, a person can concentrate on writing a letter instead of actively controlling the heartbeat, blood pressure, level of insulin on the blood and so on.

AC constitutes an important computing paradigm to automate complex systems management and reduce the need of human intervention. It can be applied to modern and widely used commercial solutions. One of the most used Cloud Computing services provider, the Amazon Elastic Computing Cloud (EC2), provides some tools for self-managing the users systems, by means of increasing or decreasing the number of Virtual Machines (VMs), according to the users demand and previous defined policies. Companies like: Netflix and the Jet Propulsion Laboratory/NASA uses EC2 solution [2].

In [3] Affonso et al. proposed a reference architecture for self-adaptive software. They present an adapted control loop based on **M**onitor, **A**nalyze, **P**lan and **E**xecute, based on **K**nowledge (MAPE-K) and define the modules that must be implemented in order to achieve this reference model. However, they are focusing on how to solve a problem (how to implement an autonomic control loop) and not on how to define the problem that must be solved by this autonomic control loop. The authors in [4] proposed a benchmarking framework for distributed autonomic systems. They also do not focus on how to define the problem that the autonomic control loop must solve. In other cases, frameworks were proposed to help the development of autonomic systems. Although, these frameworks are useful, they usually focus on some specific paradigm or architecture, i.e., Service-Oriented Architecture (SOA), sensor networks or cloud computing [5][6]. Other related works focused on creating detailed and domain-specific performance models of systems, using queuing network models or Petri nets that can be used by an AM to implement aspects like self-configuring and self-optimization. These models are mostly domain-specific, complex to create and validate, and cannot be easily adapted in cases of changes in the system [7][8]. Further, it is important to point out that some related works give qualitative and general information about the performance of different techniques to implement an AM, like Artificial Neural Networks (ANN), linear feedback control, performance model based adaptive control, decision tree and so on [9][10]. Finally, most of the related works deal with one or two aspects of AC, like self-healing and/or self-configuring [5][7]. In this paper, we focus on developing an approach to define the problem that must be solved by an autonomic system. Firstly, we present the steps required to define the problem that an autonomic system must solve. Secondly, we present a simple case study to demonstrate the viability of the proposed approach. Finally, we provide some ideas to future works that can improve the proposed approach and other possible applications.

We chose the domain of **QoS-aware Service Selection** (QSS) to develop our case study. This domain is suitable for AC because its environment is highly dynamic and must be able to deal with changes in workload, QoS preferences, fault tolerance and so on. We transformed a traditional QSS system into a self-configuring and self-optimizing one to demonstrate the viability of CObAPAS.

This paper is organized as follows: in Section II the concepts related to AC and QoS-aware service selection are

presented. Section III contains the approach to develop AS. Section IV presents a case study to validate the proposed approach. Finally, in Section V are presented the conclusions and future work.

II. Autonomic computing and QoS-aware service selection

A. Autonomic computing

The automation of computational resources management is not a new problem for computer scientists. For decades, software components have evolved to deal with the growing complexity of performing the control of systems, sharing resources and execution of operational management [11]. **Autonomic computing** is a computational paradigm based on biological systems that aim to deal with the management of complex systems, offering the possibility of self-management minimizing the need for human intervention [12]. Autonomic computing is based on four principal attributes, namely [11]:

- **Self-configuring:** dynamically configure itself, a system can adapt (with minimal intervention) to the deployment of new components or changes in the system.
- **Self-healing:** detect problematic operations and then initiate corrective actions without disrupting system applications.
- **Self-optimizing:** efficiently maximize resource allocation and usage to meet end users' needs with minimal intervention. It addresses the complexity of managing system performance.
- **Self-protection:** detect hostile or intrusive behavior as it occurs and take autonomous actions to make itself less vulnerable to unauthorized access and use, viruses, denial-of-services attack, and general failures.

Autonomic systems are composed of two parts: Autonomic Element(s) (AE) and Autonomic Manager (AM). An AE can be divided into: hardware (computers, printers, routers, etc.) and software (web service, application container, virtual machine, etc.). The communication between AM and AEs occurs using **Sensors** and **Effectors**. Sensors collect data about the AEs. On the other hand, Effectors have the function of performing the operations sent by the AM to the AEs. The AM implements a control loop based on four activities: **M**onitor, **A**nalyze, **P**lan and **E**xecute, based on **K**nowledge (**MAPE-K**).

Monitor: monitors and collects the relevant details of interest from the managed element. **Analyze:** analyzes information provided by the monitor activity to determine if it is necessary to take some action. If some action is required, it is passed to the plan activity. **Plan:** creates a plan (or a sequence of actions) by structuring actions to achieve system goals. **Execute:** performs the actual actions, hence changing the behavior of the managed element. The **K**nowledge Base contains information about the system, that must be monitored, the different available plans and so on [13].

B. QoS-aware service selection

Quality of Service (QoS) is a set of non-functional properties of Web services. Some of well known QoS attributes are: cost, response time, availability, security, and so on. QoS-aware web services composition (QWSC) is defined as an integration of different services aiming to attend complex business needs. For example, instead of manually accessing a service for buying an airplane ticket, and after that another service to reserve a hotel room, the user can access a composed service that performs both tasks. QWSC is divided into two parts: creation of the composition flow and QoS-aware service selection. In the former, the developer of the composed web service can use some business process modeling language, like Web Services Business Process and Execution Language (WS-BPEL). Using WS-BPEL the developer will define the order of execution of the services, the exchange of data between them and if some services will execute in sequential or parallel order [14]. Figure 1 shows the division between these activities.

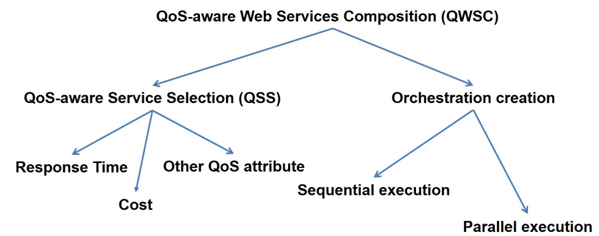

Figure 1. Different aspects of QoS-aware Web Services Composition.

QoS-aware service selection is based on QoS attributes of services. It means that based on the QoS attributes the algorithm or other technique of service selection will decide which service will be included on the composite service. There are a wide range of different techniques to store and retrieve information about QoS attributes of services. They can be stored and retrieved in a relational database application [15] or using some semantic parallel approach [16]. QoS-aware service selection is a combinatorial optimization problem and is NP-Hard, thereby, many related works spent efforts developing and testing algorithms to solve it.

III. Development of autonomic systems based on combinatorial optimization problems

A. Motivation

This section will show our proposed approach to develop new autonomic systems or transform traditional systems into autonomic ones. The main idea of this approach is to provide a clear and easy form that can be used in a wide range of systems. The proposed approach is named CObAPAS: Combinatorial Optimization based Approach for Autonomic Systems. We define the problem that the AM must solve as a combinatorial optimization problem. CObAPAS has the following advantages:

- It is independent of architecture and/or technologies: it can be used from a simple web server to complex cloud environments.
- It can be used to create new autonomic systems or transform traditional systems into autonomic ones.
- It provides a formal and clear definition of the problem that must be solved.
- It allows to evaluate different solutions proposed for the stated problem quantitatively.
- It can address one or more aspects of AC, it can be constrained or unconstrained and it allows even

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-405-3

multiple constraints and/or multiple objectives to be minimized or maximized.

The required steps of CObAPAS are: **system definition**, **search-space definition**, **objective function definition** and **developing solutions, test scenarios and evaluation**. They will be shown in the next subsections. In Section IV, we present a case study to help illustrate our approach.

B. System definition

First, we must decide which system we aim to create or modify. For example, we can develop a web server from scratch or transform a traditional web server into an autonomic web server. The system could be composed of one single entity, i.e., a web server or can be composed of two or more entities, i.e., a system composed of a web server and a database application. In fact, a system could be very simple or composed of many entities that interacts with each other in different manners. Once we have defined the system, we can continue with the next steps.

C. Search-space definition

After choosing the system, it is necessary to define the search-space of our problem. The search-space is the set of attributes that should be modified to optimize the objective function that will be created on the next step. The search-space size is the number of all possible combinations of all defined attributes. For example, if we have ten attributes and each one can assume two values, the search-space size will be 2^{10}. It varies according to the system, and **the only restriction is that they all must be discrete**. Since we are dealing with combinatorial (or discrete) optimization problems, all attributes must be discrete. We have to define which attributes we want to consider in our system; in a system composed of a web server and a database application for example, there are some parameters that can be dynamically modified in execution time. Therefore, we can define that some of these parameters are our search-space and include them into our problem definition. Examples of such parameters are shows in Table I.

TABLE I. LIST OF PARAMETERS.

Web Server (IIS 5.0)	Database Server (SQL Server 7.0)
HTTP Keep Alive	Cursor Threshold
Connection Timeout	Locks
MemCacheSize	Priority Boost
MaxPoolThreads	Max Server Memory

D. Objective function definition

Now, we must decide which aspect(s) of AC we want to focus on, and other characteristics, such as if the problem will be single-objective or multi-objective, if it will be constrainted or unconstrainted and if the objective function must be minimized or maximized. Once we are dealing with the problem definition, there are no technological or architectural restrictions.

In [17], the authors present a wide range of combinatorial optimization problems, how to define them and some algorithms to solve them. In fact, this study did not focus on solutions for AS, but in the aspect of the problem formalization. In our point of view, any aspect of AC can be defined as an optimization problem. For example, suppose that it is required to develop a QoS-aware service selector (QSS) with Self-Healing capabilities. If some service is unavailable at execution time, the QSS should select an equal or similar service and execute it, instead of that one which is unavailable. We want that in all occurrences of unavailability, the QSS select other service as fast as possible. So, it can be defined as the minimization of average recovery time (time to select a new service and execute it) of the QSS.

Doing so, we can develop two or more solutions for the problem and quantitatively compare them. Therefore, after we have defined some test cases, instead of qualitative and generic affirmations, we can quantitatively compare the proposed solutions. In fact, this approach can be used to define the problem according to the AS developer's needs.

E. Developing the solution(s), creating test scenarios and evaluating the solutions

After we have defined the problem, we need to develop solutions for it. It is possible to use from simple static policies to heuristic algorithms or even complex and detailed queuing network models. Since the problem is formally defined, if we have two or more solutions, they can be quantitatively compared.

In order to achive an effective and a properly evaluation of the proposed solutions, it is mandatory to define some experiments which reflect possible real scenarios that the AS will face with. The authors in [18] explains in many details how to define a set of experiments, workloads, how to use statistical tools and so on.

After all these steps, the AS system is formally defined, with its solutions quantitatively compared. If more solutions arise, they can also be compared with the old ones. If something change after some period (for example, a new constraint must be added to the objective function), the objective function must be updated and the solutions must be re-evaluated.

IV. SELF-CONFIGURING AND SELF-OPTIMIZING QOS-AWARE SERVICE SELECTOR: A CASE STUDY

A. Motivation

Developing large-scale distributed systems presents the challenge of providing a way for software to adapt to changes in a computational environment. In response, the system must be able to handle all changes in the workload, failures, changes in QoS preferences, and so forth [19]. Furthermore, the need of developing systems that are capable of self-adapting is becoming greater [20].

The context of QoS-aware service selection is highly dynamic and susceptive to changes. For that reason, it is recommend that the system responsible for the service selection should be autonomic, instead of manually controlled by humans [21][22]. For example, if a service provider is overloaded, the average response time of its services can be unsatisfactory, so it should not be selected until its average response time returns to an acceptable level.

B. Problem definition

This case study will be as simple as possible, with the objective of showing how following the steps mentioned in Section III can lead to a well-defined combinatorial optimization problem, which helps to change a traditional system into

an autonomic one. The selected system is a QoS-aware web service selector, proposed by the authors in [15]. Five different algorithms were implemented and a performance evaluation was made. The QoS attributes considered were: availability, cost, response time, reputation and confidentiality.

Considering that each Web Service has its own QoS attributes, it is necessary to use aggregate functions for computing the QoS of the composition plan as a whole [15]. For example, Table II, described in [15], shows an example of aggregation of these attributes:

TABLE II. QUALITY OF SERVICE ATTRIBUTES.

Availability	$\prod_{i=1}^{i=n} availability(WSi)$
Cost	$\sum_{i=1}^{i=n} cost(WSi)$
Response Time	$\sum_{i=1}^{i=n} responseTime(WSi)$
Reputation	$\sum_{i=1}^{i=n} reputation(WSi) * 1/n$
Confidentiality	$\sum_{i=1}^{i=n} confidentiality(WSi) * 1/n$

The Web Services composition plan could be described as a sequence of tasks (abstract Web Services) with an initial and a final task. For any abstract WS, it could have some candidate services (concrete Web Services) with same or similar functionality but different QoS attributes. Thus, there are various composition plans for each execution path of composite service. For example, if there is one execution path, with 10 abstract WS and 15 (concrete Web Services) per abstract Web Service, then the number of composition plans should be about 15^{10} [15]. Table II presents the aggregate functions of QoS attributes considered in this paper. However, it is also necessary a form to assess the QoS of the composition as a whole, taking into account the QoS attributes defined. The function to be **maximized** in the experiments is shown in (1), considering A (**Availability**), C (**Cost**), RT (**Response Time**), R (**Reputation**) and Con (**Confidentiality**).

$$F(x) = A + C + RT + R + Con \quad (1)$$

Given that the QoS attributes were normalized in a form that 0 is the worst result and 1 is the best result possible, simply add up all the attributes of QoS, regardless if they have to be either minimized or maximized. The Equation 2 and Equation 3, presented in [23], represents respectively, the equation used for attributes that must be minimized and the equation used for the attributes that must be maximized. Then, for each QoS attribute, the aggregated QoS is calculated using the formulas presented in Table II. Thereafter, the composition aggregated QoS is computed using the formula shown in (2). Finally, this number is normalized between 0 and 1 and called Normalized Composition Aggregated QoS (NCAQ).

By doing that, we already accomplished **step one** and defined the system. The **step two** is to define the search-space. In this case study, only one attribute will be considered: static policy of the system. Static policies can be any fixed rule or algorithm that is used to implement the AM of the system. For example, a static policy can define that an autonomic web server must decline any request if its capacity is above 90%. The static policies are two algorithms developed in [15] and they will be explained in the subsection named Implemented algorithms.

In **step three**, we must define the objective function. In order to define that function,we must consider which aspect(s) of the AC in the system we want to focus on. In our case study, we chose **self-configuring** and **self-optimizing**. After that, we must define if the objective function will be single-objective or multi-objective. If we choose multi-objective, it is necessary to guarantee that two or more objectives are in conflict, otherwise the global solution would be a single point in the search space. For instance, those functions can be something such as minimize the average response time and maximize the average QoS obtained. In our case study, we defined that the function would be **single-objective** and we must **minimize** the **average response time** of the attended requests. Finally, we must define if the objective function would be constrained or unconstrained, we chose **unconstrained**. So, the defined objective function is shown in (2):

$$Minimize \frac{\sum_{i=1}^{i=n} ResponseTime(R_i)}{n} \quad (2)$$

where *n* is the number of requests and $ResponseTime(R_i)$ is the response time of request R_i.

A wide range of different techniques can be used: ANN, heuristic algorithms, static policies, adaptive performance models and so on. Since in this paper we do not focus on the solutions, we chose static policies.

Step four is divided into three phases: developing the solution(s), creating test scenarios and evaluating the solutions. The solutions used in this experiments are described in subsection Implemented algorithms. The test scenario is described in the subsection Experiment design and the evaluation of the solutions is described in subsection Result analysis.

C. Implemented algorithms

Exhaustive Search (ES): This algorithm, also known as “brute force”, analyses all points in the search space. In the case of the QWSC problem, it compares the QoS obtained by all possible combinations of composite plans and returns the best one (with higher QoS). So, the obviously advantage of this algorithm is that the global optima are always guaranteed. The disadvantage is related to their computational complexity, because it is exponential. For instance, suppose a composite flow has ten abstract WS and one hundred concrete Web Services per abstract Web Service, the number of points in the search space will be 100^{10}, which will probably take hundreds of years to be calculated. Because of that, this algorithm could be used only in small search-space sizes, because of the soft real-time characteristic of the QWSC problem.

Greedy Heuristic (GH): This algorithm was an original idea proposed by the authors in [14]. For each abstract WS in the composite flow, the algorithm evaluates all concrete Web Services available for that abstract WS and selects the one with higher aggregate QoS. Due to all QoS attributes are normalized between 0 and 1 (and the highest is always the best one), it is necessary to calculate the sum of all QoS attributes of all concrete Web Services. The one with higher aggregate QoS is selected to its respective abstract WS. Suppose *j* is the current WS to be evaluated, *k* is the number of QoS attributes and *q* is the current QoS attribute, (3) represents the algorithm:

$$GH(WSj) = \sum_{i=1}^{i=k} q_i \quad (3)$$

The advantage of this algorithm is that it is very fast because it is directly related to the number of total concrete Web Services, i.e., suppose a composite flow with four abstract Web Services and one hundred concrete Web Services per abstract Web Service, the number of total concrete Web Services will be four hundred. So, the algorithm should calculate the aggregate QoS function of four hundred concrete Web Services; instead of calculating 100^4 composite plans like the ES algorithm does. The disadvantage of this algorithm is that it could not benefit from a larger deadline, because it is a deterministic algorithm.

D. Experiment design

The main goal of this study is to evaluate different policies to solve (2). Thus, the test environment is composed of three machines: one representing a client, another a service provider and a third one executes a MySQL server with the data about the QoS attributes of the Web services. In the considered environment, the three machines are in the same network and are linked by a gigabit network switch. The machines used are heterogeneous and their configuration is presented in Table III.

The experiments were conducted varying three factors in order to verify the performance of the policies and different number of abstract Web Services and concrete Web Services per abstract Web Service. The parameterization of these factors can be observed in Table IV. All experiments were executed ten times and the average response time was colected and presented in Figure 2.

TABLE III. ENVIRONMENT CONFIGURATION.

Machine	CPU	Clock	Cache	RAM
Service provider	Intel® Core™2 Quad	2.66 GHz	3 MB	8 GB
MySQL server	Intel® Core™i3	3.10 GHz	3 MB	4 GB
Client	Intel® Core™2 Quad	2.4 GHz	4 MB	4 GB

TABLE IV. LIST OF EXPERIMENTS.

Exp. number	abstract WS	concrete WS	Algorithm
1	2	100	ES
2	2	200	ES
3	3	100	ES
4	3	200	ES
5	2	100	GH
6	2	200	GH
7	3	100	GH
8	3	200	GH

E. Result analysis

The objective of these experiments was to discover which policy is most effective in optimizing the defined objective function. For this purpose, eight experiments were conducted, varying the number of abstract Web services and the number of concrete Web services for each abstract Web service.

In all experiments, the GH policy was more effective, since the average response time was considerably lower. In some cases, the average response time of the ES policy was more than two times the GH average response time. The lines inside the columns represents the calculated confidence interval (CI) (it was defined a 95% degree of confidence) and it is related to the variability of the results. In all experiments, the CI was lower in the GH policy. Considering that, GH is not just faster but also more stable than ES.

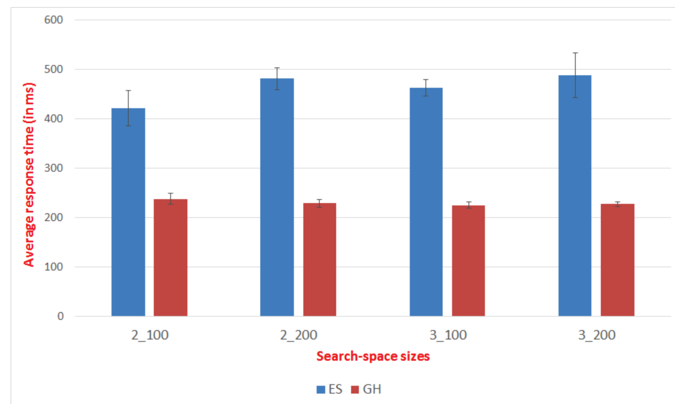

Figure 2. Average response times of ES (blue) and GH (red) in milliseconds.

V. CONCLUSIONS AND FUTURE WORK

This paper presented CObAPAS, a new approach to develop new autonomic systems or transform traditional systems into autonomic ones. It was discussed the importance of autonomic computing and the motivation for developing an approach that helps the problem formalization of an AS and has not architectural and/or technological limitations.

Compared to related works, our paper focuses on the problem formalization instead of proposing solutions for specific autonomic systems. Our approach can fit into the AS developer's needs since all attributes in the defined search-space are discrete.

A simple study case was presented, to validate our approach. In fact, we believe that many different AS can be created using CObAPAS. The experiments showed that it is possible to quantitatively compare different solutions for the AM, after the objective function was defined. CObAPAS provides two major benefits: guidelines for developing an AS and a way to quantitatively measure the quality of different solutions for the defined problem.

In future works, we plan to develop more sophisticated case studies to validate our approach, with multiple aspects of AC and/or multiple constraints. One example of case study is an autonomic Virtual Machines (VMs) manager. We will use the Famav tool, presented in [24]. Famav is a command line tool for managing VMs. Compared to Virsh (another command line tool) Famav presents a lower performance, but its ease and practicality minimizes this difference. We also plan to create another two approaches to develop new AS or transforming traditional systems into autonomic ones: one for AS systems based on continuous optimization problems and one for systems based on both continuous and combinatorial optimization problems. These new approaches also need some case studies to be validated and to show some applications in real-world problems.

ACKNOWLEDGMENT

The authors would like to thank the financial support of CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) and CAPES (Coordenação de Pessoal de Nível Superior).

REFERENCES

[1] V. Nallur and R. Bahsoon, "A decentralized self-adaptation mechanism for service-based applications in the cloud," IEEE Transactions on Software Engineering, vol. 39, no. 5, 2013, pp. 591 – 612.

[2] Amazon elastic compute cloud (amazon ec2). website. [Online]. Available: http://aws.amazon.com/ec2/ [retrieved: April, 2015]

[3] F. J. Affonso and E. Y. Nakagawa, "A reference architecture based on reflection for self-adaptive software," in VII Brazilian Symposium on Software Components, Architectures and Reuse, 2013, pp. 129 – 138.

[4] A. Vilenica and W. Lamersdorf, "Benchmarking and evaluation support for self-adaptive distributed systems," in Sixth International Conference on Complex, Intelligent, and Software Intensive Systems, 2012, pp. 20 – 27.

[5] W. Li, P. Zhang, and Z. Yang, "A framework for self-healing service compositions in cloud computing environments," in IEEE 19th International Conference on Web Services (ICWS), 2012, pp. 690 – 691.

[6] K. Zielinski, T. Szydlo, R. Szymacha, J. Kosinski, J. Kosinska, and M. Jarzab, "Adaptive soa solution stack," IEEE Transactions on Services Computing, vol. 5, no. 2, 2012, pp. 149 – 163.

[7] D. Menascé, D. Barbará, and R. Dodge, "Preserving qos of e-commerce sites through self-tuning: A performance model approach," in ACM Conference on e-commerce, 2001, pp. 1 – 11.

[8] J. M. Ewing and D. A. Menascé, "Business-oriented autonomic load balancing for multitiered web sites," in IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009, pp. 1 – 10.

[9] L. Shen, J. Wang, K. Wang, and H. Zhang, "The design of intelligent security defensive software based on autonomic computing," in Second International Conference on Intelligent Computation Technology and Automation, 2009, pp. 489 – 491.

[10] L. Checiu, B. Solomon, D. Ionescu, M. Litoui, and G. Iszlai, "Observability and controllability of autonomic computing systems for composed web services," in IEEE International Symposium on Applied Computational Intelligence and Informatics, 2011, pp. 269 – 274.

[11] S. O. Schimidt, P. F. do Prado, and A. Silva, Fundamentals of Informations Systems - Fundamentos de Sistemas de Informação. Campus Elsevier, 2014, ch. IT infrastructure and emerging technologies - Infraestrutura de TI e tecnologias emergentes, pp. 77 – 91.

[12] A. Khalid, M. Haye, M. Khan, and S. Shamail, "Survey of frameworks, architectures and techniques in autonomic computing," in Fifth International Conference on Autonomous and Autonomic Systems (ICAS), 2009, pp. 220 – 225.

[13] P. T. Endo, M. S. Batista, G. E. Gonalves, M. Rodrigues, D. Sadok, J. Kelner, A. Sefidcon, and F. Wuhib, "Self-organizing strategies for resource management in cloud computing: state-of-the-art and challenges," in verificar, 2013, pp. 13 – 18.

[14] P. F. do Prado, L. H. V. Nakamura, J. Estrella, M. Santana, and R. Santana, "Different approaches for qos-aware web services composition focused on e-commerce systems," in 13th Symposium on Computing Systems, 2012, pp. 179 – 186.

[15] P. F. do Prado, L. Nakamura, J. Estrella, M. Santana, and R. Santana, "A performance evaluation study for qos-aware web services composition using heuristic algorithms," in The Seventh International Conference on Digital Society (ICDS), 2013, pp. 53 – 58.

[16] L. H. V. Nakamura, P. F. do Prado, R. Libardi, L. Nunes, J. Estrella, R. Santana, M. Santana, and S. Reiff-Marganiec, "Fast selection ofweb services with qos using a distributed parallel semantic approach," in IEEE International Conference on Web Services, 2014, pp. 680 – 681.

[17] H. Kellerer, U. Pferschy, and D. Pisinger, Knapsack Problems, Springer, Ed. Springer, 2004.

[18] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, Wiley-Interscience, Ed. Wiley-Interscience, 1991.

[19] D. A. Menascé, H. Gomma, S. Malek, and J. P. Sousa, "Sassy: A framework for self-architecting service-oriented systems," The Journal of IEEE Software, 2011, pp. 78 – 85.

[20] A. J. Ramirez, D. B. Knoester, B. H. C. Cheng, and P. K. Mckinley, "Applying genetic algorithms to decision making in autonomic computing systems," in ACM International Conference on Autonomic Computing, 2009, pp. 97 – 106.

[21] A. Charfi, T. Dinkelaker, and M. Mezini, "A plug-in architecture for self-adaptative web service compositions," in IEEE International Conference on Web Services (ICWS), 2009, pp. 35 – 42.

[22] G. H. Alferez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz, "Dynamic adaptation of service compositions with variability models," The Journal of Systems and Software, vol. 91, 2013, pp. 1 – 24.

[23] P. F. do Prado, "Desenvolvimento e avaliação de algoritmos para composição dinamica de web services baseada em qos," Master's thesis, Universidade de São Paulo (USP), 2012.

[24] Y. Neves, L. H. V. Nakamura, P. F. do Prado, and M. Santana, "Famav: Analise comparativa entre ferramentas de gerenciamento de maquinas virtuais," in Proceedings of XV Simposio em Sistemas Computacionais (WSCAD-WIC)., 2014, pp. 1 – 6.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-405-3