Generic autonomic service management for component-based applications

Nabila Belhaj

université PARIS-SACLAY

NNT : 2018SACLL004

Thèse de doctorat

TELECOM SudParis

Generic Autonomic Service Management for Component-based Applications

Thèse de doctorat de l'Université Paris-Saclay
Préparée à Telecom SudParis

École doctorale n°580
Sciences et technologies de l'information et de la communication

Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Evry, le 25/09/2018, par

Nabila Belhaj

Composition du Jury :

Mme. Salima Benbernou Professeure, Université Paris Descartes, France	Rapporteur
M. Khalil Drira Directeur de Recherche CNRS, Université de Toulouse, France	Rapporteur
Mme. Karine Zeitouni Professeure, Université de Versailles-Saint-Quentin-en-Yvelines, France	Examinateur
M. Philippe Merle Chargé de Recherche INRIA, Université Lille I, France	Examinateur
M. Djamel Belaïd Professeur, Telecom SudParis, France	Co-encadrant
M. Samir Tata Professeur, Telecom SudParis, France	Directeur de thèse

Acknowledgements

This thesis has been an enriching and tough journey, full of ups and downs. It felt like chasing down different rabbit holes. Fortunately, I was surrounded by many supportive people, who have given me strength to go through the difficult times. So, it is a great pleasure to thank them all for their positive impact during this quest.

I would like to start by thanking the jury members for accepting to evaluate my research work. Professor Salima Benbernou and Professor Khalil Drira for being my thesis reviewers and for their thoughtful and encouraging comments. Professor Karine Zeitouni and Professor Philippe Merle for being my thesis examiners and for their relevant remarks.

I would like to thank my thesis advisor Djamel Belaïd for his patient guidance along these years. I am indebted to him for helping me grow as a researcher. Despite his busy schedule, he always managed to free himself to impart knowledge and provide advice on different aspects. I am glad that we continue working together to help me widen my research horizon and sharpen my sense of rigor. I would also like to express my appreciation to my thesis director Samir Tata. His valuable advice, constant and enthusiastic support helped me overcoming many difficulties.

I am thankful to my former office-mate Mohamed Mohamed who has been source of help, to Imen Ben Lahmar for her kind encouragement and to Hamid Mukhtar for our fruitful collaboration. Thank you all for granting me the opportunity to learn from each one of you and for spending a great deal of your time investing in me.

I owe my warmest affection to all the members of the computer science department of Telecom SudParis. Especially to Brigitte Houassine for her kind help and assistance. To many of my friends and fellows, for their encouragements, discussions and criticism. Thank you Monika for your kindness, thank you Mai, Leila, Aïcha, Emna, Souha, Nour, Hayet, Rania, Kunal, Nathan and Tuanir. Thank you all for the laughters and good memories.

I am forever grateful to my parents Ahmed, Mahjouba and khalatoutou Mina who were always there to provide unconditional love and support. Thank you for having faith in me even when doubting myself. I am deeply thankful to my best friend, soul-mate and loving husband Anas for his patience and understanding. Thank you for putting up with my temper and for helping me getting through the tough times.

My thoughts go always to my grandmother whose loss has been of great sorrow. They go to my grandfather, who passed away without being able to say goodbye. Your love, devotion and blessing allowed me to be who I am today. Thank you for raising me and for giving me a good education. I wished you were around to share this moment with you.

To the memory of my grandparents,
to my family, my parents and my husband,
in gratitude for their unconditional love and support.

Résumé

Au cours de la dernière décennie, la complexité des applications a considérablement évoluée afin de répondre aux besoins métiers émergeants. Leur conception implique une composition distribuée de composants logiciels. Ces applications fournissent des services à travers les interactions métiers maintenues par leurs composants. De telles applications sont intrinsèquement en évolution dynamique en raison de la dynamicité de leurs contextes. En effet, elles évoluent dans des environnements qui changent tout en présentant des conditions très dynamiques durant leur cycle de vie d'exécution (par exemple, en termes de leur charge, disponibilité, performance, etc.). De tels contextes représentent une lourde charge pour les développeurs d'applications aussi bien pour leurs tâches de conception que de gestion. Cela a motivé le besoin de renforcer l'autonomie de gestion des applications pour les rendre moins dépendantes de l'intervention humaine en utilisant les principes de l'Informatique Autonomique.

Les Systèmes Informatiques Autonomes (SIA) impliquent l'utilisation des boucles autonomiques, dédiées aux systèmes afin de les aider à accomplir leurs tâches de gestion. Ces boucles ont pour principal rôle d'adapter leurs systèmes à la dynamicité de leurs contextes, en se basant sur une logique d'adaptation intégrée. La plupart du temps, cette logique est donnée par des règles statiques codées manuellement, souvent spécifiques à un domaine et potentiellement sujettes à des erreurs. La construction de ces règles demande beaucoup de temps et d'effort tout en exigeant une bonne expertise. En fait, elles nécessitent une compréhension approfondie de la conception ainsi que de la dynamicité du système afin de prédire les adaptations précises à apporter à celui-ci. Par ailleurs, une telle logique ne peut envisager tous les scénarios d'adaptation possibles, donc, ne sera pas en mesure de prendre en compte des adaptations pour des situations précédemment inconnues. Les SIA devraient donc être assez sophistiqués afin de pouvoir faire face à la nature dynamique de leurs contextes et de pouvoir apprendre par eux-mêmes afin d'agir correctement dans des situations inconnues. Les SIA devraient également être capables d'apprendre de leur propre expérience passée afin de modifier leur logique d'adaptation en fonction de la dynamicité de leurs contextes.

Dans ce manuscrit de thèse, nous abordons les lacunes décrites en utilisant les techniques d'Apprentissage par Renforcement (AR) afin de construire notre logique d'adaptation. Cependant, les approches fondées sur l'AR sont connues pour leur mauvaise performance lors des premières phases d'apprentissage. Cette mauvaise performance entrave leur utilisation dans le monde réel des systèmes déployés. Par conséquent, nous avons amélioré cette logique d'adaptation avec des capacités d'apprentissage plus performantes avec une approche AR en multi-pas. Notre objectif est d'optimiser la performance de l'apprentissage et de le rendre plus efficace et plus rapide, en particulier durant les premières phases d'apprentissage. Nous avons aussi proposé un cadriciel générique visant à aider les développeurs d'applications dans la construction d'applications auto-adaptatives. Nous avons donc proposé de transformer des applications existantes en ajoutant des capacités d'autonomie et d'apprentissage à leurs composants. La transformation consiste en l'encapsulation des composants dans des conteneurs autonomiques pour les doter du comportement auto-adaptatif nécessaire. Notre objectif est d'alléger la charge des tâches de gestion des développeurs et de leur permettre de se concentrer

plus sur la logique métier de leurs applications. Les solutions proposées sont destinées à être génériques, granulaires et basées sur un standard connu, à savoir l'Architecture de Composant de Service. Enfin, nos propositions ont été évaluées et validées avec des résultats expérimentaux. Ils ont démontré leur efficacité en montrant un ajustement dynamique des applications transformées face aux dynamicités de leurs contextes en un temps beaucoup plus court comparé aux approches existantes.

Mots-clés: Informatique Autonomique, Gestion Autonomique, Conteneurs Autonomes, Application à base de Composants, Prise de Décision Auto-Adaptative, Apprentissage par Renforcement.

Abstract

During the past decade, the complexity of applications has significantly scaled to satisfy the emerging business needs. Their design entails a composition of distributed and interacting software components. They provide services by means of the business interactions maintained by their components. Such applications are inherently in a dynamic evolution due to their context dynamics. Indeed, they evolve in changing environments while exhibiting highly dynamic conditions during their execution life-cycle (e.g., their load, availability, performance, etc.). Such contexts have burdened the applications developers with their design and management tasks. Subsequently, motivated the need to enforce the autonomy of their management to be less dependent on human interventions with the Autonomic Computing principles.

Autonomic Computing Systems (ACS) implies the usage of autonomic loops, dedicated to help the system to achieve its management tasks. These loops main role is to adapt their associated systems to the dynamic of their contexts by acting upon an embedded adaptation logic. Most of time, this logic is given by static hand-coded rules, often concern-specific and potentially error-prone. It is undoubtedly time and effort-consuming while demanding a costly expertise. Actually, it requires a thorough understanding of the system design and dynamics to predict the accurate adaptations to bring to the system. Furthermore, such logic cannot envisage all the possible adaptation scenarios, hence, not able to take appropriate adaptations for previously unknown situations. ACS should be sophisticated enough to cope with the dynamic nature of their contexts and be able to learn on their own to properly act in unknown situations. They should also be able to learn from their past experiences and modify their adaptation logic according to their context dynamics.

In this thesis manuscript, we address the described shortcomings by using Reinforcement Learning (RL) techniques to build our adaptation logic. Nevertheless, RL-based approaches are known for their poor performance during the early stages of learning. This poor performance hinders their usage in real-world deployed systems. Accordingly, we enhanced the adaptation logic with sophisticated and better-performing learning abilities with a multi-step RL approach. Our main objective is to optimize the learning performance and render it timely-efficient which considerably improves the ACS performance even during the beginning of learning phase. Thereafter, we pushed further our work by proposing a generic framework aimed to support the application developers in building self-adaptive applications. We proposed to transform existing applications by adding autonomic and learning abilities to their components. The transformation entails the encapsulation of components into autonomic containers to provide them with the needed self-adaptive behavior. The objective is to alleviate the burden of management tasks on the developers and let them focus on the business logic of their applications. The proposed solutions are intended to be generic, granular and based on a well known standard (i.e., Service Component Architecture). Finally, our proposals were evaluated and validated with experimental results. They demonstrated their effectiveness by showing a dynamic adjustment to their context changes in a shorter time as compared to existing approaches.

Keywords: Autonomic Computing, Autonomic Management, Autonomic Containers, Component-based Applications, Self-Adaptive Decision Making, Reinforcement Learning.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Research Context

With the popularization of internet in the late 90's, there have been a surge in Information Technologies (IT) to cater to the increasing business requirements of service users. Recently, service providers are in constant fervor to propose online and 24/7 available services. Nowadays services are provided and consumed on several emerging and distributed platforms, ranging from a pay-as-you-go fashion in Cloud contexts to a ubiquitous and pervasive fashion in Internet of Things contexts. As a consequence of this growth, an explosion in the number of software resources providing services has been noticed during the last decade.

Software resources are distributed and interacting software components that compose applications to fulfill their business (i.e., functional) requirements to cater to the users expectations in terms of Quality of Service (QoS). One particular paradigm that allows building applications upon a set of services is the Service-Oriented Architecture (SOA) [18]. SOA is a software architecture that enables the design of applications based on composition of loosely-coupled services. Developping such applications can be done by using the Service Component Architecture (SCA) [19]. SCA is a technology agnostic standard that allows the development of SOA applications by assembling existing software components. An SCA component can be either an atomic component or composite holding an assembly of components. A component-based application is then represented by a composite that holds several interacting components or composites. Each component implements the business logic (i.e., functional logic) of its provided services. The application components

maintain business interactions with each others by providing and/or requiring services to/from one another to accomplish the application's business logic.

SCA enables assembling component-based applications from existing and reusable components. Such applications are in perpetual expansion since they also enable plug-in new components due to the emerging business needs to which application developers have to quickly respond. Consequently, the application's size continue growing by being composed of multitude of interacting components to form complex structures. Moreover, the application components can be executed in a variety of distributed deployment environments. All these factors have progressively contributed in elaborating complex applications.

Such applications are intrinsically in dynamic evolution due to their context dynamics. Actually, an application component faces internal dynamics during its execution life-cycle. The internal dynamics may involve the component's availability (e.g., the component is stopped for some reason), performance (e.g., meeting or not QoS requirements) or load (e.g, the component is overloaded due to augmented service demands), etc. The application component is also exposed to external dynamics during its execution life-cycle. The external dynamics concern the component's hosting environment where it evolves. An environment may expose its hosted components to dynamic changes such as in CPU or memory resources usage in a hosting Virtual Machine (VM) in the Cloud or changes in network signal of a hosting Mobile device in pervasive environment. These internal and external dynamics are what define the context dynamics of the application components.

The dynamic contexts of these applications and the continuous growth of their complexity result in increasing management (i.e., non-functional) tasks that burdened the application developers. As a consequence, the applications started requiring more and more human interventions to handle non-functional aspects along with the efforts needed for the development of applications' business logic. This trend was faced by introducing Autonomic Computing to reduce as much as possible the human intervention and manual reconfigurations in favor of self-management and autonomic behavior.

Autonomic Computing [20] has emerged to provide systems with the needed autonomic behavior to deal with their context dynamics. This paradigm came to address non-functional aspects of management in order to equip systems with self-management capabilities. Autonomic Computing Systems (ACS) could then be endowed with self-* properties such as self-reconfiguring by adapting "on the fly" to context changes or self-optimizing their performance or efficiently optimize their resource utilization.

The key element of the autonomic behavior of ACS is the autonomic MAPE-K loop. The MAPE-K stands for Monitoring, Analysis, Planning, Execution and Knowledge. Depending on the management requirements, this loop is associated with one or several system components to provide them with self-* properties and then be qualified as Managed components. To do so, this loop operates by harvesting monitoring data from the Managed component (or several Managed components) as well as its (or their) deployment environment, analyzing the monitoring data and in case of anomalies (e.g., Service Level Agreement (SLA) violation), planning for corrective actions and then executing them. This loop holds information about the Managed component, its context and the other MAPE parts. These information are contained in the Knowledge and are necessary to the MAPE parts to perform their non-functional operations. This loop is intended to adapt its associated Managed component not just to its internal changes but also to its environment dynamics. Thereby, autonomic MAPE-K loops are used in ACS to reduce human intervention by providing these systems with the ability to autonomously manage themselves. ACS are then able to respond to their management requirements and then meet their established management objectives (e.g., SLA management).

1.2 Motivation and Research Problems

Autonomic MAPE-K loops are supposed to provide ACS with the autonomic behavior that meets the management needs. This behavior is driven by the adaptation logic (i.e., non-functional logic) held by the autonomic loop and upon which it acts in its deployed context. In most cases, this adaptation logic is built based on adaptation rules written by the application developers. These rules give predefined adaptation scenarios that are coded into ECA (Event Condition Action) rules. ECA rules describe, for an event containing information about some attributes (e.g., response time of a service) verifying certain conditions (e.g., value exceeding a threshold) than the corresponding corrective adaptation action (e.g., scale-out) should be executed.

ECA rules are basically a decision policy in which an event describes the system state. This state is mapped to a decision (i.e., adaptation) action which gives a reactive behavior to the system. The mapping of states and actions is what describes the decision making process that derives the decision policy. Thereby, ECA rules describe static and hand-coded decision policy that constitutes the adaptation logic of the autonomic loop. To develop such logic, the developers make use of their limited view of context dynamic. They attempt to predict the right adaptations to perform if the system encounters anomalies (i.e., the verified conditions).

In fact, when deploying ACS, developers may not have exhaustive knowledge about the context dynamics. They can only make use of their limited view of few system states and actions for hand-coding decision policies. However, achieving good performance with an adaptation logic based on fixed and hand-coded policies may be very difficult and inflexible [21]. It is hardly feasible to predict and envisage all the possible adaptation scenarios to hand-code these policies especially for complex contexts. It is also difficult to be accurate enough to reach the best possible adaptations (i.e., optimal actions) that lead the system to meet its management objectives. As a consequence, these policies are not able to adapt to changing contexts. They are not able to take the appropriate adaptations for previously unknown situations for which no adaptation scenarios were predefined.

Such adaptation logic entails developing accurate models of context dynamics which is typically a difficult and time-consuming task, hence, requiring a costly expertise. It requires a detailed understanding of the system design to be able to predict how changes brought to resources may affect the system's performance and ability to achieve its management goals. Such adaptation logic is perfectly feasible for small contexts, but not suitable for nowadays complex contexts. Therefore, it is difficult to engineer accurate knowledge models with acceptable performance for complex contexts. Indeed, the development of these models with such accuracy is effort-consuming, often concern-specific and potentially error-prone. These difficulties present a major obstacle to the widespread use of autonomic computing in deployed and complex contexts.

Ideally, ACS are supposed to autonomously adapt themselves in dynamic contexts and learn to solve problems based on their past experiences [22]. We consider that they should be sophisticated enough to deal with the dynamic phenomena of their contexts by themselves. They also have to be able to learn on their own to properly act in previously unknown situations. ACS should be able to learn adaptation logic by dynamically building their decision policies. However, ACS are self-adaptive based on their predefined adaptation logic which requires building accurate knowledge model [23]. Consequently, we consider that ACS should be equipped with learning capabilities to dynamically learn, compute and change their adaptation logic to fit to their dynamic contexts.

Reinforcement Learning (RL) [24] brings great promises in overcoming the gap in creating accurate

knowledge models. RL consists of developing agents that learn a desired behavior through trial-and-error interactions with their dynamic contexts in order to perform tasks without the need to specify the nature of these tasks. The agent's behavior is usually guided by a reward function that rewards good action selection and punishes bad action selection leading to good or bad performance, respectively. From our point of view, RL offers several key advantages from ACS perspectives. First, it provides the possibility to develop optimal policies without requiring an explicit model or a thorough understanding of the context dynamics. Furthermore, by its foundation in Markov Decision Processes [25], RL's underlying theory is fundamentally a sequential decision theory [26] that deals with the dynamic phenomena of an environment. Therefore, RL potentially offers benefits for ACS by being part of the building blocks that compose the MAPE-K loop's adaptation logic.

Often, RL techniques develop learning algorithms that derive the agents learning behavior. These algorithms have for aim learning an optimal decision policy by searching for the best combinations of states and actions. It is a search problem in which the best combinations correspond to the optimal action that should be selected for a given state, hence, finding the optimal policy. To do so, the agent should have several and repetitive interactions with it context until it acquires enough knowledge about its dynamics to finally converge. It converges when it finally infers the optimal decision policy that it was progressively computing through its interactions with its environment. However, commonly used RL techniques known as one-step approaches [27] are known for their poor performance induced during the early stages of learning [21]. They are known for converging very slowly which increases the time the system might poorly perform. The system has poor performance when its does not meet its management requirements (e.g., spending longer time in SLA violations). In fact, at the early stages of learning, a RL-based agent has not acquired enough experience on its context yet. The agent may accumulate the selection of random and potentially bad decision actions.

Furthermore, the convergence takes a longer time when the agent acts in a complex context where the combinations of states and actions to learn about are large. Actually, to find a good quality policy, or at best cases the optimal policy, the agent has to experiment all the combinations of states and actions. Although, in complex contexts, the time taken by the agent to compute its decision policy is potentially larger and then would certainly reflect badly on the system performance. Such bad performance represents a major obstacle to delivering services with acceptable quality. Especially in systems where SLA requirements established for applications should be respected to guarantee good performance for service delivery. The existing one-step approaches do not cope with the complexity of nowadays systems to scale to their large state and action spaces. Subsequently, constituting an obstacle to the widespread usage of RL techniques in real-world, complex and deployed systems.

We think that RL techniques should perform better in order to be coupled with ACS. They should provide better performing capabilities to deal with the quality requirements of nowadays complex applications. We consider that the usage of enhanced RL techniques will enable us to propose an approach with a RL-based adaptation logic that provides a better performance. Such approach will allow us to dynamically build and modify an optimal decision policy that better suits the context dynamics. Thus, we will not necessarily have to acquire a knowledge about the context dynamics to hand-code large ECA rules. Subsequently, reducing the development cost in terms of time, effort and expertise. It is important to note that such approach does not exclude the usage of ECA rules, rather it completes them by enabling the system to learn on its own to optimally act in unknown situations where no adaptation contexts were initially defined.

When digging farther to grasp the labor of building ACS, we find another major problem faced by application developers. Driven by their concern of reducing as much as possible human intervention, they increased their development labor to render autonomous the management of their applications. In fact, in addition to developing the business logic of their application components, they also have to develop the adaptation logic of their non-functional components, necessary to the autonomic management of their functional components. The developers should also take into consideration the development induced by the integration of the non-functional components to the functional ones by managing their interactions.

The development efforts intended for the non-functional aspects are built from scratch which may potentially be error-prone. Furthermore, the developers have to take into account the differences of each functional component context. Each functional component has different and specific management requirements and so as its deployment environment. By considering each of these specific requirements, the developers may end-up building problem-specific management solutions. Indeed, their built solutions may be tied to the business-logic of the applications. These solutions may not be reused for other management concerns, not to mention that they are not equipped with learning capabilities to reduce even more the development labor.

We think that the application developers should focus more on their business requirements rather than the adaptation aspects. The developers should be provided with facilities to support them in building the autonomic behavior of their functional components. They should be supported with generic and dynamic tools to alleviate the burden of building the non-functional aspects. The objective is to facilitate the integration of non-functional aspects into their existing functional components at minimum cost. We also think that the proposed tools should be strengthened by being conform to existing standard. Also, the adaptation logic should not be tied to the problem-specific parts of the business logic. The adaptation logic should be provided in separation from the business logic to add more flexibility and reusability to the proposed solution for other management concerns. The adaptation logic should also be provided in separated non-functional components to add more flexibility and reusability to them. We also think that the developers should be provided with a solution that enables the autonomic behavior on different granularity levels (i.e., fine-grained and coarse-grained). The solution should also be extensible to encompass different management requirements for different contexts. It should integrate the specific needs of the applications as well as their deployment environments.

1.3 Related Work

When looking into the literature, we find research work that addressed self-management and adaptivity in ACS in variety of contexts. We noticed that, very often, the adaptation logic is built upon reactive and predefined rules [10, 28, 7, 29, 8, 9, 30, 31, 32, 33, 34, 11]. Even when some of these work have the advantage of either proposing standardization work [28, 33, 30] or frameworks using standards [29, 11, 7, 8, 9, 31]. This remark remains the same in terms of reactive rules usage when researchers provide generic frameworks without using any standard such as [34, 10, 32].

We also had a look into approaches that used adaptive control techniques [35, 13, 36, 14, 37, 38, 39] to build ACS. They are knowledge model-based approaches that require intensive knowledge building to develop controllers that control the system behavior. They seem domain-specific and also built upon static rules. Other techniques known as discrete control synthesis [13, 39, 14, 38] that also aimed at generating

controllers to control the system behavior. They built their reactive rules in behavioral contracts to generate controllers as finite state machines. The behavioral contracts seem inflexible and not subject to reuse when the context changes.

We also looked into some learning-based approaches [40, 41] that showed promises in dynamic task automation (e.g., service composition). RL-based solutions were also proposed to equip ACS with the needed autonomic behavior to address some management problems. Almost all of these work have used the one-step learning approaches [42, 43, 16, 40, 17] which showed longer time for convergence, hence, poor performance during the learning stage. Some of the learning approaches proposed to enhance the learning such as work [21, 15] that used Neural networks or in work [41] by distributing the task among several agents to converge faster. However, these learning-based work propose concern-specific solutions that are not generic, hence, not subject for reuse.

1.4 Thesis Objectives

The aim of our research work is to provide solutions that respond to the described problems. The proposed solutions should comply with characteristics of genericity, standardization and granularity while being endowed with adaptation flexibility and performance optimization. We described these characteristics in the following:

- **Genericity:** the description of the provided solution should be agnostic to the application components regardless of their used technologies or implementations. The solution should also be generic to encompass different management concerns and not be specific to a particular management concern. Therefore, the solution should be reusable and extensible to be applied to a variety of functional components without implying major modifications on the solution.
- **Standardization:** the description of our proposed solution should be consolidated by being conform to existing standards (such as SCA, REST, HTTP, etc.). The solution will then be strengthened by making it more accessible for usage and more generic.
- **Granularity:** the solution should be applicable at different granularity levels, from fine-grained to coarse-grained levels, depending on the management requirements. The solution should be used for atomic functional components and composites of different levels and also for the application composite.
- **Adaptation Flexibility:** the solution provided for the adaptation logic should be flexible by enabling its change. Depending on the needs, it can be based on ECA rules for a reactive behavior and also based on learning capabilities so that the behavior can change depending on the context dynamics. The solution should enable a dynamic learning of an optimal decision policy. It should also allow adding new learning capabilities to the adaptation logic to adopt different and potentially better learning behaviors.
- **Performance Optimization:** not only the proposed solution is learning-based, it should also optimize the learning performance in order to optimize the system's performance. The objective is to render a solution that has the potential to scale for complex applications.

1.5 Thesis Contributions

In the light of the aforementioned shortcomings, the main contributions that will be thoroughly detailed in this manuscript are the following:

Our first contribution is dedicated to bringing solutions to the adaptation logic problems in ACS. We plan on improving the adaptation logic of a MAPE-K loop. Instead of using static hand-coded adaptation logic, we plan on equipping this loop with a learning-based adaptation logic. Accordingly, we enhance the Analysis component of the MAPE-K loop with sophisticated Reinforcement Learning capabilities to dynamically and autonomously self-adapt the behavior of component-based applications. Correspondingly, we model the decision making process of the Analysis component as a Markov Decision Problem. Afterwards, we solve this problem by defining the Markov Decision Process's concepts that are related to our approach. We also introduce an approach of decision learning algorithm that helps conducting the Analysis behavior in learning dynamically its optimal decision policy. To guide the Analysis component in its decision and learning behavior, we propose a reward function algorithm. This function rewards good action selection that leads to good performance of the system. It also punishes bad action selection that leads to bad performance. The proposed algorithm is supposed to learn from its past experiences and compute dynamically the optimal decision policy at execution time. The RL algorithm that we use for our adaptation logic is based on a multi-step learning approach. It is based on a combination of Temporal Difference methods [27] and Monte Carlo methods [44] in a multi-step fashion which outperforms the one-step learning approaches used in the literature. The multi-step learning algorithm converges faster compared to existing learning approaches. Subsequently, reducing the time the system might spend in SLA violations as demonstrated in our experiments.

Our second contribution in intended to support the application developers in building the autonomic behavior of their applications. The purpose is to help them focus more on their business requirements rather than the adaptation aspects which will largely reduce their development labor. In this contribution, we propose a generic framework for building self-adaptive component-based applications. The self-adaptive applications are not built from scratch, rather they are transformed to equip them with the needed autonomic behavior. The framework offers reusable and extensible functionalities intended to support the application developers in transforming their existing applications. We designed this framework to accompany the developers in defining certain aspects that are specific to their non-functional requirements. We also provide it with an abstract design to enable extending and overriding the autonomic behavior and also enable its integration with different deployment environments. Furthermore, to allow its integration with different applications, we designed this framework by using the Service Component Architecture standard. Actually, we designed the framework as an autonomic container that holds the non-functional components providing the self-adaptive abilities to functional components. The transformation of the application components happens by encapsulating them in autonomic containers to render them self-adaptive. The transformed components are then self-adaptive and can dynamically learn optimal policies based on a multi-step learning algorithm at execution time. The framework enables the integration of autonomic management on different granularity levels depending on the management needs. We proved the effectiveness of our proposals with a case study of an application that we transformed. Accordingly, we used the framework by defining the elements to be provided, extended and implemented to customize the framework behavior and render it adequate to the developer management needs. We validated our proposals with experiments demonstrating that the framework is lightweight by showing a

negligible overhead during and after the application transformation.

1.6 Thesis Outline

The remainder of this thesis manuscript is organized as follows:

- **Chapter 2: Background** presents an overview of some concepts and paradigms that are necessary to grasp the remainder of this manuscript. In this chapter, we provide definitions of the architectures behind the design of service-based and component-based applications. We also define the Autonomic Computing paradigm as well as the Markov Decision Processes formalism behind the learning approach that we adopted. Accordingly, we present the different Reinforcement Learning techniques that we use in our approach, namely, the combination of Temporal Difference methods and Monte Carlo methods.

- **Chapter 3: State of the Art** positions our research work in the literature. In this chapter, we survey existing work that aimed to provide solutions to the autonomic management problems. We review work addressing these problems either by proposing or using standards. We also review work that adopted control techniques to provide systems with an adaptive controlled behavior. We also present work that used RL techniques to provide systems with the desired autonomic behavior. We conclude with a discussion around the advantages and drawbacks of each work and compared them to our proposals.

- **Chapter 4: Self-Adaptive Decision Making Process based on Multi-step Reinforcement Learning** describes our first contribution. In this chapter, we present our proposals of enhancing the adaptation logic of a classical autonomic MAPE-K loop. We model the Analysis component decision making process as a Markov Decision Problem that we solved with the Markov Decision Process concepts and the enhanced multi-step learning algorithm that we used. The objective is to allow for a dynamic computation of an optimal decision policy at execution time. Therefore, we provide the algorithmic details that guide the decision making process to compute the optimal policy.

- **Chapter 5: Framework for Building Self-Adaptive Component-based Applications** introduces the framework intended to support the application developers in rendering their applications self-adaptive. We present the framework design and generic functionalities that can be extended and reused for several management requirements. We also show, with a component-based application case study, the usage of our framework to render self-adaptive the application components. To do so, we describe the elements to be provided, extended or implemented in order to customize the framework for our specific management needs.

- **Chapter 6: Evaluation and Validation** presents the different aspects related to the evaluation and validation of our proposals that are presented in Chapters 4 and 5. We made use of the case study to perform the different experiments. We introduce the implementation aspects and then the experiments that we used to validate our contributions. The results demonstrate the effectiveness of our proposals by showing a self-adaptive behavior of the transformed application and better learning performance of the used multi-step learning. The results also show that the framework is lightweight.

- **Chapter 7: Conclusions and Open Research Challenges** concludes this thesis by summarizing the contributions and discussing new research challenges and future extensions. We present in this chapter our research perspectives that we already started and plan to accomplish in short terms. We also present research perspectives in the middle and long terms.

Chapter 2

Background

Contents

2.1 Introduction

We dedicate this chapter to providing an overview of the different concepts and paradigms that we used in our contributions. We present then the basis background concepts necessary to understand the remainder of this manuscript. We start by introducing, in Section 2.2, the Service Oriented Architecture (SOA) and Service Component Architecture (SCA). SOA and SCA mainly represent the targeted application architectures and also the SCA-based design of the non-functional management aspects of our contributions. Afterwards, we introduce the Autonomic Computing Systems in Section 2.3 by introducing the Autonomic Computing paradigm used by these systems. Subsequently, we present the underlying theory of Markov Decision Processes that we use in our learning approach in Section 2.4. Finally, we give an overview on Reinforcement Learning techniques in Section 2.5, namely Monte Carlo and Temporal Difference methods.

2.2 Service Oriented and Service Component Architectures

Service-Oriented Architecture (SOA) is the result of standards and developments of Web services in support of automated business (i.e., functional) integration [45, 46]. In [18], SOA is defined as *"a logical way of designing a software system to provide services to either end user applications or other services distributed in a network through published and discoverable interfaces."*. SOA allows the definition of loose-coupling architectures independently of standards, implementations or protocols. The main key elements of this architecture are services which describe some needed business logic. A service is seen as an open and self-describing element intended for rapid, low-cost development and deployment of distributed applications. SOA allows building applications upon a composition of reusable and loosely-coupled software services that fulfill the application business logic.

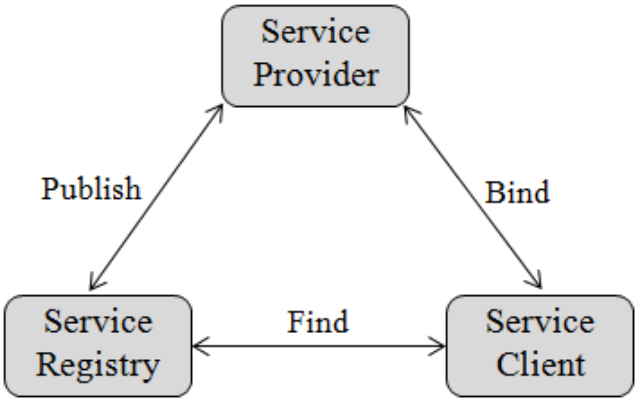


Figure 2.1: SOA architecture

Figure 2.1 depicts the three basic actors operating in a SOA: Service Provider, Service Client and Service Registry that are interacting through operations of publish, find and bind. The Service Provider corresponds to a role of a software entity that offers services while the Service Client represents a requestor entity that consumes a specific service. The Service Registry constitutes an entity that holds information on the available services and the way to access them.

The key advantage of this approach lies in the loose-coupling of the services composing an application. Services are provided by means of platform independent parts which indicates that they could be consumed by using any computational platform, operating system or a programming language. To implement and access the business functionality, Service Providers and Service Clients (respectively) can make use of different technologies. Despite the technologies differences, the functionalities representation of these services on a higher level remains the same. Thereby, describing applications as composition of services regardless of their implementations is very interesting. It allows the separation of the services implementations from their business functionalities. Therefore, an application is executed by composing various services offered by heterogeneous parts w.r.t. their services descriptions.

Building applications based on SOA can be done with the well known programming model Service Component Architecture (SCA) [19]. As shown in Figure 2.2, SCA has introduced a software component as its

basic artifact to design applications. OASIS [47] has defined a component as *"a configured instance of a piece of code providing business functions. It offers its functions through service-oriented interfaces and may require functions offered by other components through service-oriented interfaces as well. SCA components can be implemented in Java, C++, and COBOL or as BPEL processes. Independent of whatever technology is used, every component relies on a common set of abstractions including services, references, properties and bindings"*.

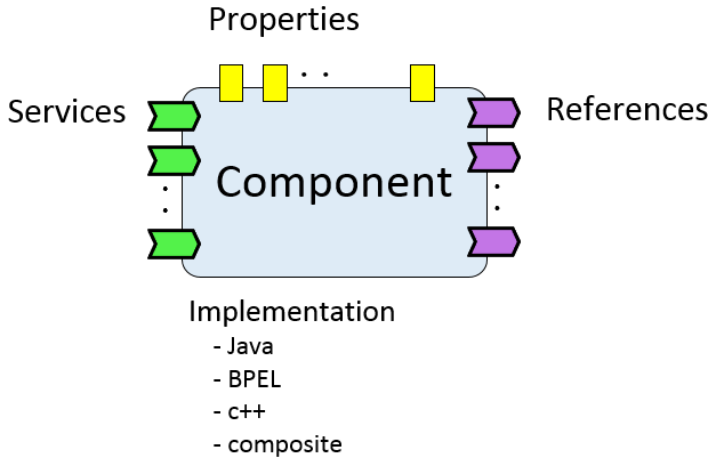


Figure 2.2: SCA Component Diagram (adapted from [1])

As depicted in Figure 2.2, a provided service indicates the business function offered by the component. A reference or required service describes the business requirement or need from other components. Services and references are linked by wires. The depicted component may (or not) define one or several properties that may (or not) allow for their reconfiguration. An SCA component can be either an atomic component or composite holding an assembly of components as shown in Figure 2.3.

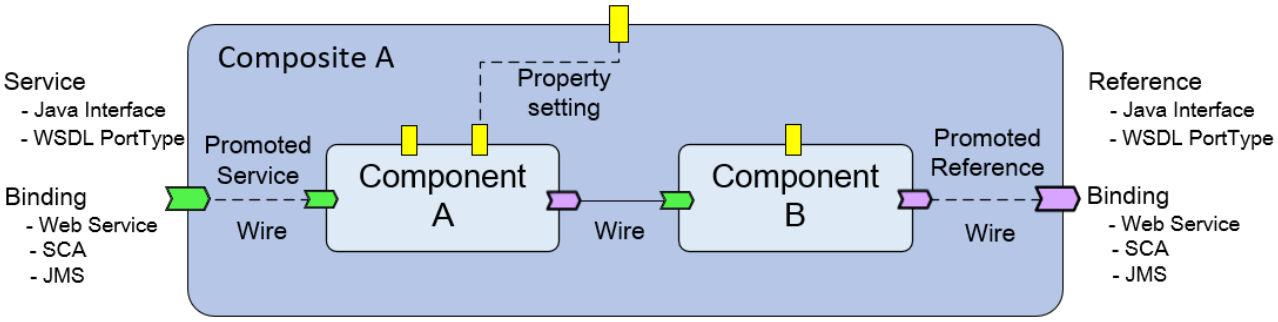


Figure 2.3: SCA Composite Diagram (adapted from [1])

The internal components maintain business interactions to fulfill their business logic as well as their containing composite's. The internal interactions are described with wires to indicate internal and direct service

calls among components. The external interactions are described with bindings to depict remote service calls among remote components. As shown in this figure, an SCA composite may promote one or several services, references or properties of its internal components.

2.3 Autonomic Computing Systems

Autonomic Computing [20] was introduced by IBM researchers after drawing inspiration from the human nervous system that handles unconscious reflexes (e.g., digestive functions). In 2001, IBM launched the autonomic computing initiative [2] as an attempt to intervene in computing systems in a similar fashion as its biological counterpart. The essence of this initiative is to develop systems endowed with autonomic behavior, capable of self-governance and self-organization without human intervention. The ultimate aim is to render computing systems completely autonomous by means of autonomic management (non-functional) whereby the system's conditions remain in accordance with the established high-level management objectives. An autonomic system should comply to four major self-properties [48]. Self-reconfiguration for an "on the fly" adaptation to the context changes. Self-optimization for an optimal performance and resource utilization. Self-healing for failure discovering to promote service delivery and diagnosing to prevent disruptions. Self-protection for identification and anticipation of unauthorized accesses and attacks protection. We are interested in all these properties except self-protection, since this property is more relevant to system security and protection and these aspects do not meet our research scope.

Among these properties, we focus on self-reconfiguration and self-optimization, as self-healing and self-protection properties are not relevant in the current context of our work.

Autonomic Computing Systems (ACS) imply the usage of the so-called autonomic MAPE-K loop [6]: Monitoring, Analysis, Planning, Execution and Knowledge. As shown in Figure 2.4, the autonomic loop is associated with one or several resources and are called Managed resources. As it is described by IBM, the MAPE-K loop should operate as follows:

- The Monitoring offers mechanisms that collect monitoring data from the Managed resource then aggregate, filter, and report them,
- The Analysis offers mechanisms that model complex situations, interpret the current state of the system and predict future situations,
- The Planning offers mechanisms that provide the actions needed to accomplish certain objectives according to some guiding policy rules or strategy,
- The Execution offers the mechanisms that handle the execution of the planned actions over the Managed resource,
- The Knowledge offers the mechanisms that are used by the other MAPE elements to retrieve necessary information for their functioning.

Despite this description, in most cases, this loop is built upon the classical behavior that consists of collecting monitoring data, analyzing them and in case of anomalies, generating corrective adaptation actions then

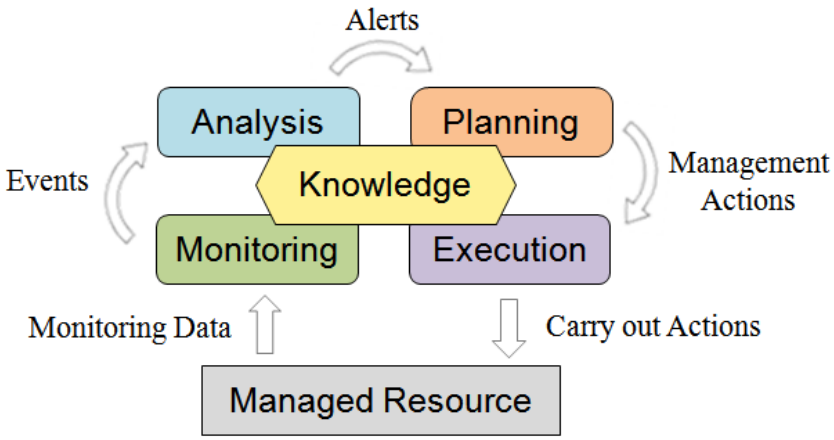


Figure 2.4: Autonomic MAPE-K loop reference model (adapted from [2])

finally executing them. In most cases, this loop acts upon some static hand-coded adaptation logic, provided by application developers. Its main limitation resides in its inability to learn on its own to fit to the unexpected dynamics of its context.

Actually, we find it very interesting to use learning abilities in conjunction with ACS. The purpose of this conjunction is to create systems that reason about their decision making (i.e., action selection) to provide better decision policies that fit to changing contexts. It is worth noting that we raise similar issues that the Artificial Intelligence [49] field is dealing with. Such issues include autonomic decision making, learning and reasoning and are tackled by Reinforcement Learning [24] techniques. More formally, these technique draw their grounding from Markov Decision Processes [25]. Thus it is interesting to explore such processes to grasp the theoretical basis of Reinforcement Learning techniques.

2.4 Markov Decision Processes

Markov Decision Processes (MDPs) [25] (or Markov Decision Problems) are mathematical formalism of decision problems commonly named sequential decision problems. As sequential may suggest, the main feature of these problems is the relationship between the current decision and the future decisions. These problems consist of several decision problems that are sequentially presented. At each sequence step, a decision is made by a decision maker called agent. The decision is made by the agent by selecting an action while taking into account its impact on the solution of the future problems. The sequential nature of these problems is typically seen as planning problems in Artificial Intelligence field [49]. This sequantial nature is also very often related to the shortest path approaches in stochastic environments [3].

More formally, MDPs are described by the five-tuple concepts (S, A, T, P, R) and defined as:

- S is the set of state space that summarizes all the agent situations which describe the agent's received observations of its environment,

- A is the set of action space containing all the possible actions needed to control the state dynamics,
- T is the set of time steps or transitions where decisions take place,
- $P()$ is the transition probability function that describes the cognition of the states dynamics,
- $R()$ is the reward function that associates a reward signal (i.e., real value) to each state transition.

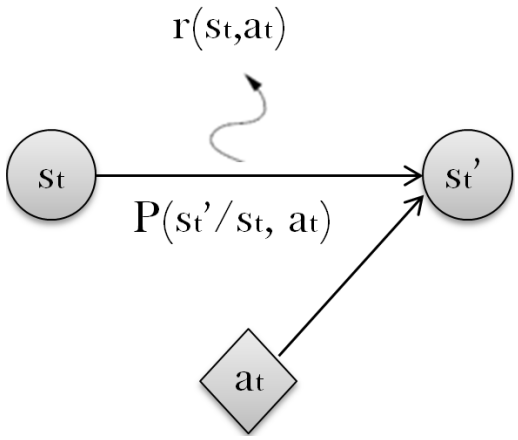


Figure 2.5: Markov Decision Process [3]

Figure 2.5 depicts the MDP diagram in which at every time step or transition t in T, an action is executed in a current state s_t. This action impacts the process by transiting to a new state s'_t (note that $s'_t = s_{t+1}$ as for the next transition $t+1$). A reward r_t is then received by the agent for this transition to indicate the utility for choosing action a_t when the state of the system is s_t. The transition probability $P(s'_t/s_t,\ a_t)$ indicates the probability to transit from state s_t to state s'_t knowing that action a_t is executed. Therefore, the transitions sequences of the states represent MDPs as Markov chains that visit states, controlled by actions and valued by rewards. This representation describes a model of the agent's behavior and environment dynamics and the cognition of all the introduced MDP concepts.

By choosing actions (i.e., making decisions), the agent controls its environment evolution. The process of choosing actions is what describes the agent behavior and is referred to as its decision policy $\pi(s,a)$ or simply π. The policy is a function that maps a state s to an action a either deterministically or with a probability distribution over actions, $\pi(s,a) = Pr(a_t = a|s_t = s)$.

When decision making problems are addressed in systems, they can be modeled as MDP problems in order to be solved. Solving an MDP is controlling the agent's behavior so that it acts optimally in order to maximize its incomes and finally optimize the system's performance [44]. Therefore, solving a MDP implies searching for an optimal policy (at least in our case). In a given set of state and action spaces, we search for the optimal couples $\{(s,a)^*\}$ that provide the best sequences of rewards. More formally, this corresponds to assessing a decision policy on the basis of computed expected cumulative sum of received rewards along a trajectory (i.e., sequence of several transitions). The expected cumulative sum of rewards stands for the estimation of

the utility of choosing actions in certain states and it should be maximized on the long run sum. It is called the value function and its optimization (i.e., maximization) resembles searching for the optimal policy $\pi*$. It is a function that is computed in most techniques that are based on MDP theory, namely Reinforcement Learning.

2.5 Reinforcement Learning

Reinforcement Learning (RL) [50] draws its grounding from MDPs. It is defined as a way of programming an agent in order to make it learn a desired behavior through trial-and-error interactions with its dynamic environment. Rewards and punishments (i.e., reward values) are distributed to the agent to guide its decisions without explicitly stipulating how its task should be achieved. On each interaction step with its environment, the agent perceives an input s that indicates the current state of the environment. The agent acts by choosing an action a that changes the environment state to s'. A reward value r is communicated to the agent to indicate the utility of using the action a in the state s and is associated to the transition $(s \rightarrow s')$. The agent should behave in a way that increments the long-run sum of reward values [51].

The sequence of actions, states transitions and rewards are what define the agent experiences with its environment. Actually, along the transitions, the agent actively accumulates experience about the possible states, actions, rewards and transitions of the environment which strengthen its ability to act optimally in the future. The agent's main objective is to find a decision policy π that maximizes the long-run sum of reward values, namely the optimal policy π^*. Recall that a policy π represents a function that maps states to actions whereas an optimal policy π^* maps states to the best possible actions (i.e., optimal actions).

To compute the optimal policy through its interactions with its environment, the agent computes and updates its expected value function with the accumulated rewards along the transitions until convergence of the learning process. The convergence occurs when the agent learns enough on its environment, through its trial-end-error interactions, by experimenting several times all the combinations of its state and action spaces. Upon convergence, the agent is finally able to directly choose the best possible action to execute by deducing it from its expected value function computations. We distinguish two ways of computing and updating this function either upon performing several transitions on long trajectory (Monte Carlo methods) or upon a single transition in both incremental and dynamic programming fashion (Temporal Difference methods).

2.5.1 Monte Carlo Methods

Monte Carlo methods [44] are MDP-based methods that consist of performing a large set of transitions on a long trajectory before updating incrementally the value function. Actually, large number of trajectories are performed from all the states s in the state space S. In each trajectory, the agent saves the received rewards and the performed transitions from some starting state s_0 to a final state s_T. Let $(s_0, s_1, .., \ s_{T-1}, s_T)$ be the trajectory with the accumulated rewards $(r_0, r_2, .., \ r_{T-2}, r_{T-1})$. Note that there is no reward r_T since s_T is the final state and no transition is performed after this state. After performing this trajectory, we go back to each visited state on the trajectory to update the estimation of its value function. This update is made in a forward fashion as shown in Figure 2.6. Accordingly, the considered rewards of a state s_t, where $t \in (0, 1, ..., T-2, T-1)$, are $(r_t, r_{t+1}, .., \ r_{T-2}, r_{T-1})$ which is technically the sequence of future observed rewards for the state s_t while performing the trajectory, hence, the forward view shown in the Figure 2.6.

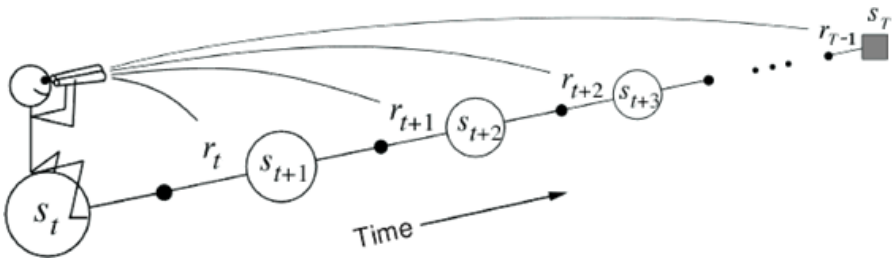


Figure 2.6: Forward view of the agent to update the estimations of its value functions (adapted from [4])

For Monte Carlo methods, it is mandatory to decompose the learning process into several successive episodes of finite trajectories [44]. Otherwise, computing and updating the value function wont take place. Furthermore, the learning process should wait until a trajectory of large number of transitions is performed no matter how long it takes. In addition, all the information collected on transitions should be recorded which may require massive memory storage. The learning process can be improved to perform updates after each transition with an incremental and dynamic fashion with Temporal Difference methods.

2.5.2 Temporal Difference Methods

Temporal Difference methods [27] are MDP-based methods that came to perform an incremental and dynamic update of the value function estimation. We do not need to wait until the end of a trajectory to compute an update of the value function estimation. Rather the update is performed after each interaction of the agent with its environment. Indeed, after each transition t: $s_t \rightarrow s'_t$, the received reward r_t is used to compute the value function estimation. Note that we can also refer to transition t as t: $s_t \rightarrow s_{t+1}$ since $s'_t = s_{t+1}$. Thereby, Temporal Difference methods need less memory and less laborious computations than the conventional Monte Carlo methods. They also produce more accurate computations after each transition to update the value function estimation.

The convergence of Temporal Difference methods implies experimenting, several times, all the possible transitions at each couple (s, a) of the state and action spaces. However, when interacting with its environment, the agent realizes a single update to the value function on the visited couple (s, a). It is known as one-step approach since a single update is performed at each transition. This update process is particularly slow, since an agent at the beginning is deprived of any information concerning the value function. Indeed, when interacting with its environment during the early stages of learning, the agent has not enough experience and may choose bad actions. This may be poorly reflected on the system performance. This may take a long time before these one-step methods converge to an optimal policy, especially when the search space is large (i.e., state and action spaces are large).

In our approach we use a combination of Temporal Difference methods and Monte Carlo methods called multi-step methods [50]. We are persuaded that this combination holds the potential of converging faster while preserving the system performance. In fact, we use the incremental and dynamic updates of Temporal Difference methods with Monte Carlo methods but in a backward fashion as shown in Figure 2.7.

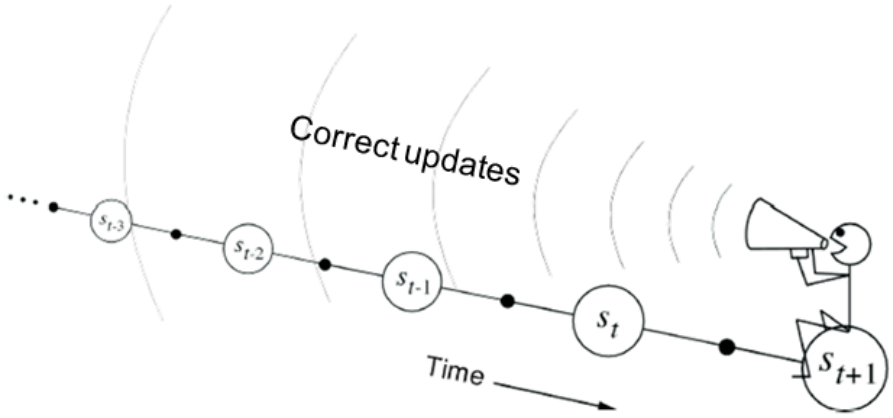


Figure 2.7: Backward view of the agent to correct previousely updated estimations of its value functions (adapted from [4])

Instead of the forward updates of Monte Carlo, we go back on the previously visited couples to correct their previously updated estimations of value functions with the currently received reward. These couples are called eligibility traces [50] and they represent the list of previously and freshly visited couples that are eligible for update of their value functions estimations. That is to say, when the transition from state s_t to s_{t+1} is performed, the reward r_t is used to recompute then correct the estimations of the previously visited states. It is like when the agent looks back into its previous estimations to provide more accurate corrections as it moves forward in its transitions. This way of updating and correcting helps the agent to quickly have as accurate as possible its estimations which speeds up the computation of its optimal decision policy. This approach, that we adopt, improves the learning performance along with preserving the system performance, especially in real-world deployed systems where performance requirements are very pointy.

2.6 Conclusion

In this chapter, we presented the basic concepts related to our research work which helps positioning and understanding our contributions and so as the state of the art. We introduced the Service Oriented Architecture and the Service Component Architecture standard that we used both as targeted application architecture and in the design of our framework approach. Moreover, we give an overview of the autonomic MAPE-K loop behind the autonomic behavior of Autonomic Computing Systems. We also presented Markov Decision Processes that we use as background theory to model the adaptation logic of the MAPE-K loop. Reinforcement Learning techniques were also presented to position our choice of the learning approach. Accordingly, we introduced Monte Carlo methods and Temporal Difference Methods that we used in combination in our learning approach. The reviewed concepts serve as starting points throughout this manuscript for building our approaches. In the next chapter, we investigate research work that were devoted to autonomic management in the literature.

Chapter 3

State of the Art

Contents

3.1 Introduction

Over the last years, there has been a surge in research work aiming to equip distributed system resources with self-management and adaptive capabilities. Many research work have devoted to provide these capabilities in different environments (e.g., Cloud, Internet of Things, data centers, etc.) and addressed different management concerns (e.g, resource provisioning, optimization of power consumption, Service Level Agreement management, etc.).

In this chapter, we survey the state of the art in order to better contextualize our contributions regarding existing work. We provide an overview of existing solutions in the literature that aimed to cater to the self-management problems. We start by presenting the generic approaches that focused on providing standardization solutions and frameworks in Section 3.2. Secondly, we review some interesting work that modeled

the management and adaptation problems as control problems and proposed solutions stemming from control techniques in Section 3.3. Afterwards, in Section 3.4, we examine some work that modeled the management and adaptation problems as decision making problems and suggested learning techniques to solve them. Finally, we conclude this chapter with a discussion synthesizing the presented work. We also provide, in Section 3.5, a comparison table of the studied work and ours on the basis of the characteristics that we established for our contributions.

3.2 Standardization Efforts and Frameworks for Autonomic Management

There are different research work around self-management and adaptivity in Autonomic Computing Systems (ACS). Some of these work were devoted to standardization and others were dedicated to providing frameworks for variety of contexts. In the following we provide an overview on these work.

3.2.1 Standardization Efforts

Some research work focused on providing standardization solutions to support the management of self-* properties in ACS. For instance, the well-known specification Java Management Extensions (JMX) [28] is a Java technology that is intended for monitoring and management of system resources such as applications or devices. It provides an instrumentation mechanism to monitor the system state and invoke the reconfiguration commands. This specification defines the concept of the JMX API (Application Programming Interface), known as MBeans or managed beans. A MBean represents a managed Java object that is Java class and implements some interfaces set forth in the JMX specification. The objective of a MBean is to instrument the system resources by exposing a management interface that consist of a set of readable and/or writable attributes, invokable operations or descriptions. The Mbeans should be registered in a core-managed object server, namely the MBean server. This server offers monitoring and management access to registered Mbeans and invoke operations on them. Moreover, the specification defines standard connectors used to enable remote management access to the MBean server.

Another standardization effort focusing on the autonomic management of applications exist. Topology and Orchestration Specification for Cloud Applications (TOSCA) [33] is an emerging standard that targets the specification of portable Cloud applications, their management and the automation of their deployment tasks. In this specification, the structure of a Cloud application is formalized as a topology graph by describing it as a set of nodes with defined relationships. It also describes the management and deployment tasks as management plans to be executed. This specification provides an XML-based language to describe the nodes and relationships in a Service Template document. More specifically, an application is represented as a Service Template as shown in Figure 3.1. The Service Template holds a Topology Template of the application and management plans. The Topology Template is defined as a directed graph that describes the structure of a composite application. The nodes of the application, namely Node Templates, represent the application components. Whereas Relationship Templates are the relations between these components. The Node Type and Relationship Type define respectively, the types of existing Node Templates and Relationship Templates. For an application component, the Node Type defines its observable properties, its interfaces for management operations, its requirements necessary to operate it and the capabilities it provides to meet other component's

requirements. For a given relationship, the Relationship Type defines its observable properties and the interfaces for management operations. The Service Template defines also the Plans to allow describing the aspects related to the management and/or deployment of the application. A Plan represents the adaptive behavior that is written as a workflow to orchestrate the adaptation operations. For instance, the Plan may describe the life-cycle management of the application by following a BPEL [52] or BPMN [53] description. The Plan is executed by the suitable engine, namely OpenTOSCA runtime [54], to manage the different nodes and their relationships. However, we believe that the TOSCA Plan describes a fixed management behavior that could have been improved with learning capabilities. Furthermore, this specification seems to be dedicated solely to applications in Cloud contexts.

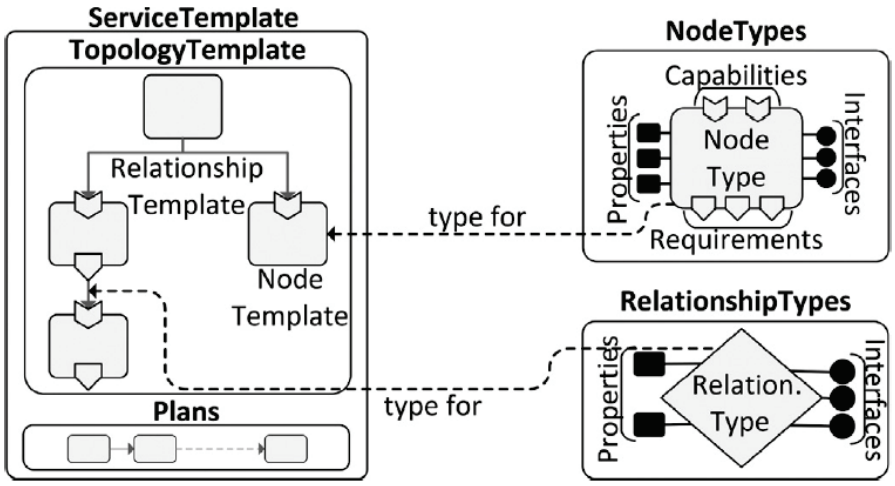


Figure 3.1: Structural elements of a Service Template, nodes and their relations [5]

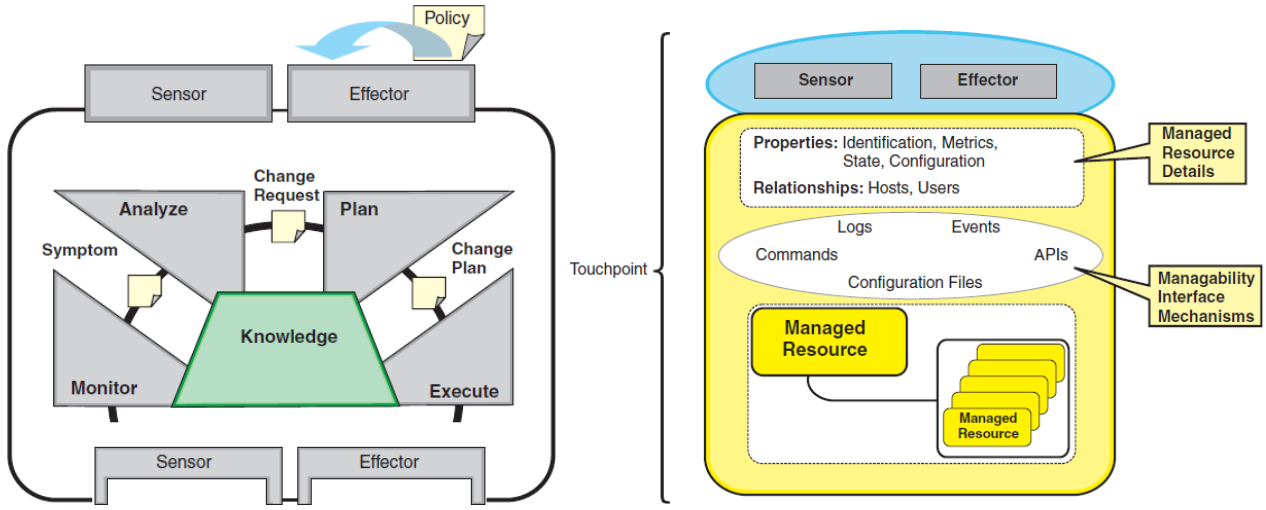
Similarly, Cloud Application Management for Platforms (CAMP) [30] is a specification for the monitoring, the management and the deployment of applications. It targets applications in Cloud environments at PaaS (Platform as a Service) level. CAMP allows application developers to manage their applications based on the REST API [1]. The management strategies that are created are then REST-based which fosters interoperability between Cloud environments and eases distributed application management. CAMP represents PaaS as a set of connected application components called Assembly resource by using Platform components representation. In this specification, the Platform is described as a set of Platform components offering a list of capabilities services (e.g., create a message queue) to be used by application components according to their requirements. Thus, a Platform component is related to an application component if it has the needed capabilities associated to the requirements of the application component. The standard defines also resources of Sensors and Operations which provide means to interact with the application. An Operation resource is an action that can be executed on some resource. A Sensor resource represents a dynamic monitored data that can be exposed or fetched from another system. A Sensor resource can also provide Operations in order to adjust the frequency of monitoring or resetting some values of monitored metrics. CAMP ensures also the

[1] http://restlet.org/

portability of CAMP applications by using a Platform Deployment Package (PDP) containing all the descriptions and dependencies of the application (i.e., deployment plan, certificates, source code, etc.). However, the strategies do not propose any learning mechanisms to foster a dynamic computation of adaptation policies. Further, the specification seems to be Cloud-specific.

3.2.2 Frameworks, Toolkits and Methodologies

Pioneer in Autonomic Computing field, IBM proposed an Autonomic Computing Toolkit [32] that provides guiding steps to adding autonomic abilities to system resources. In this toolkit, a collection of tools and technologies were introduced to enable users to build the autonomic behavior of their systems. The main tool is the Autonomic Management Engine that is an implementation of the Autonomic Manager as shown in Figure 3.2a. It holds the MAPE-K parts (Monitor, Analyze, Plan, Execute and Knowledge) that work together to ensure the autonomic functionalities of the Managed Resource. The toolkit provides Sensor and Effector Touchpoints as depicted in Figure 3.2b to control the Managed Resource. The sensor provides mechanisms to sense the managed resources state and generate log messages using XML (eXtensible Markup Language) format. The effector offers configuration mechanisms to carry out adaptations on the managed resource. Regarding the adaptation logic, the toolkit provides an Adapter Rule Builder that is used to create adaptation rules. Accordingly, it provides a Generic Log Adapter configuration file XML format that allows defining adaptation scenarios as rules for detected events. In our point of view, this toolkit could have been enhanced with learning mechanisms to dynamically learn a policy instead of fixed adaptation rules.



(a) Functional details of the Autonomic Manager [6]

(b) Managed resource Touchpoint [6]

Figure 3.2: Autonomic Manager [6]

The JMX specification is adopted by other research efforts such as [7, 8] to provide their frameworks and build their management concerns. Adamczyk et al. [7] introduced the Autonomic Management Toolkit (AMT) framework intended for constructing adaptive systems. The architecture of AMT is presented in Figure 3.3

and provides an Autonomic Manager that is composed of three main modules accessed though the Autonomic Manager Interface. The three modules are: Policy Management Module (PMM), Policy Evaluator Module (PEM) and Resource Access Module (RAM). The PMM contains hand-coded policy rules. The PEM holds a list of reasoners that instantiate and evaluate these rules. The RAM defines the resources and the interactions with the MBeans. Actually, the AMT framework used JMX for sensing and effecting purposes and combined it with a rule engine-based decision making module. As rule engine, authors used Drools [2] to constitute their rules knowledge base system. Drools is a Business Rules Management System solution that supports the creation of ECA (Event Condition Action) rules. Drools also provides a core Business Rules Engine to execute these rules. However, this framework is limited to hard-coded and customized rules for the adaptation logic and does not support learning mechanisms.

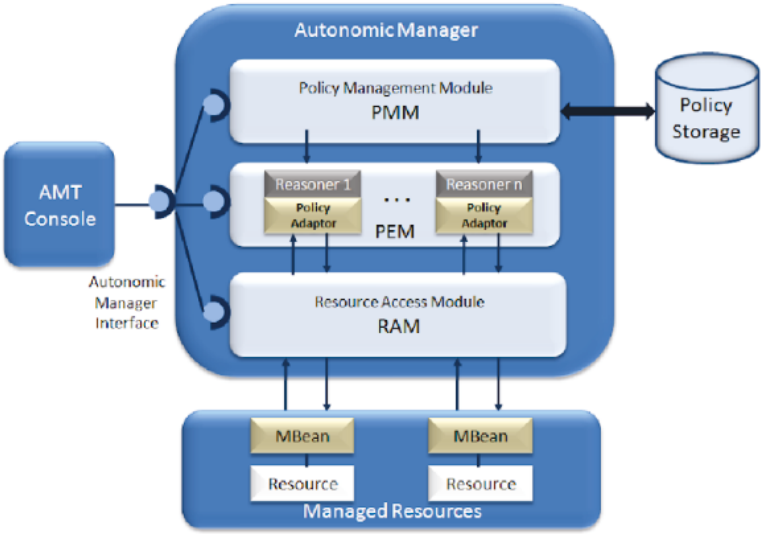


Figure 3.3: Key components of AMT architecture [7]

Likewise, Ruz et al. [8] used JMX to introduce a framework supporting SLA (Service Component Agreement) management for component-based applications. In this paper, authors used the SCA (Service Component Architecture) standard to design their framework. As depicted in Figure 3.4, they implemented the Monitoring, Analysis, Decision and Execution concerns in separated components and then attached them to a component to govern it. The Monitoring component uses JMX to collect metrics values from the managed component. The SLA Analysis consumes monitoring data and compares them against the SLOs (Service Level Objectives) then alerts the Decision component in case of infraction. The Decision component implements strategies to react to the infractions. According to the predefined strategies, the Decision component outputs the corresponding action then sends it to the Execution component to carry it out. In this paper, we noticed that the framework did not provide a Knowledge component that we believe is necessary to the functioning of the other components. We also noticed that the designed adaptation logic remains rigid when it

[2] https://www.drools.org/

comes to building their strategies. These JMX-based work [7, 8] present the disadvantage of relying solely on frozen strategies neither adequate for changing contexts nor flexible for future reuse.

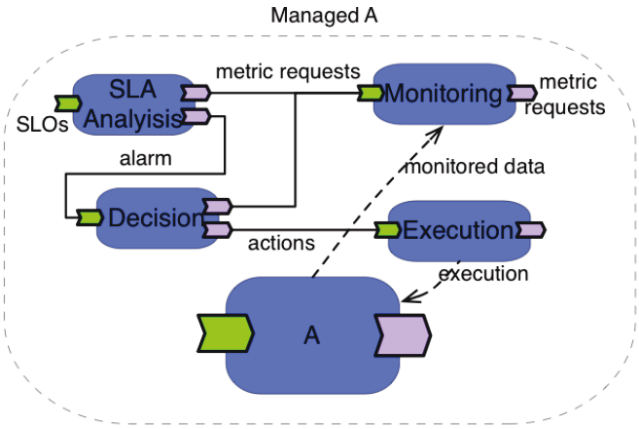


Figure 3.4: SCA component "A" with all its attached monitoring and management components [8]

Some research work used/extended TOSCA for their application management needs in Cloud environments. For instance, the authors of [29] extended TOSCA in the modeling of management operations that define the behavior of application components. More specifically, they extended the XML description of the Node Type to add management protocols. For a given application component, a management protocol is described as a finite state machine with a set of states and transitions. The finite state machines are associated to the application components and are expressed as conditions on the components' requirements and capabilities. However, the TOSCA Plan remains based on fixed strategies which could have been enhanced with learning abilities.

FRAMESELF [9, 31], is a generic autonomic framework that addressed self-management and self-awareness (i.e., context-aware and profile-aware) of machine to machine networks (M2M) in highly distributed systems. This framework described M2M and their relationships by using multi-model representation based on graphs and ontologies. The M2M domain is modeled by using the Web Ontology Language (OWL) [55] for semantic knowledge modeling, publishing and sharing ontologies. As described in Figure 3.5, FRAMESELF is based on IBM MAPE-K autonomic loop to manage M2M entities. The monitor offers mechanisms that process and correlate the events received from the sensors then generates reports as symptoms. The analyzer has for activity correlating and modeling complex situations. It receives symptoms then checks if a request for change should be generated as response. The planner receives the analyzer requests to construct the plans needed to achieve its goals by acting upon a policy embedded in the knowledge. The executor controls the execution of the plans by using effectors. The executor can be coupled with different technologies such as REST and RMI [3]. The knowledge contains semantic rules according to which the MAPE components achieve their tasks. The knowledge is modeled based on three layers of representation: the M2M application layer that is ontology-based and uses the European Telecommunications Standards Institute (ETSI) [56] standard

[3] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

to describe the functional architecture of M2M applications. The communication layer that is based on the Event-Driven Architecture (EDA) [57] and Service-Oriented Architecture (SOA) [58] for an asynchronous publish/subscribe pattern and communication decoupling among system entities. The deployment and configuration layer that is based on graph plans using the GraphML standard [59] and OSGi deployment plateform [60]. However, FRAMESELF does not provide any learning-based mechanisms. Its adaptation logic is based solely on reactive rules to self-manage its M2M entities.

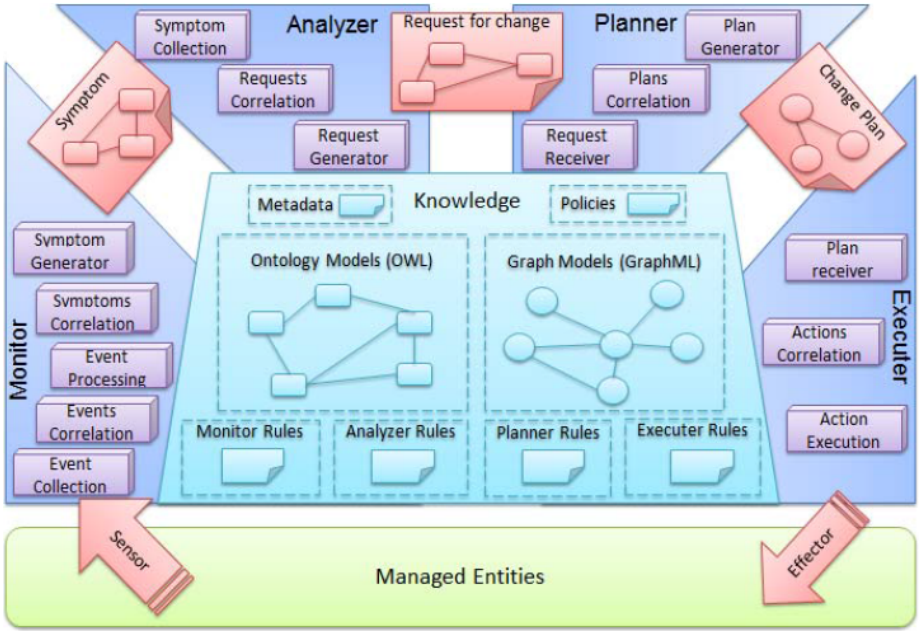


Figure 3.5: FRAMESELF architecture overview [9]

ACS have found their place in Internet of Things (IoT) [61] environments as well. A methodology and generic architecture where introduced in [34, 10] to govern manageable resources (e.g., heaters, lamps, household machines, etc.). As shown in Figure 3.6, an autonomic controller is proposed to perform management tasks for a house in a smart grid. The controller manages some control elements (CE) that govern the manageable resources. The controller is under two authorities: the electricity provider that imposes power saving goals and the smart house owner that gives comfort goals. The controller should conciliate both goals that could be conflicting since the provider imposes certain load management objectives to the grid for power saving. The owner may decide to specify some comfort goals like maintaining temperature or lighting ambiance. Three types of control layers are defined in Figure 3.6 depending on the type of required behavior. The control layers are: the base control layer for a reactive behavior in response to stimuli by following a predefined strategy. The adaptation layer that defines a self-adaptive behavior to adjust the controller reactions. The goal management layer that is used for an agent-like behavior to negotiate and accept or refuse goals. However, the described self-adaptive behavior of the adaptive layer is not adjusted dynamically by using learning-based approach, it is also based on a predefined strategy.

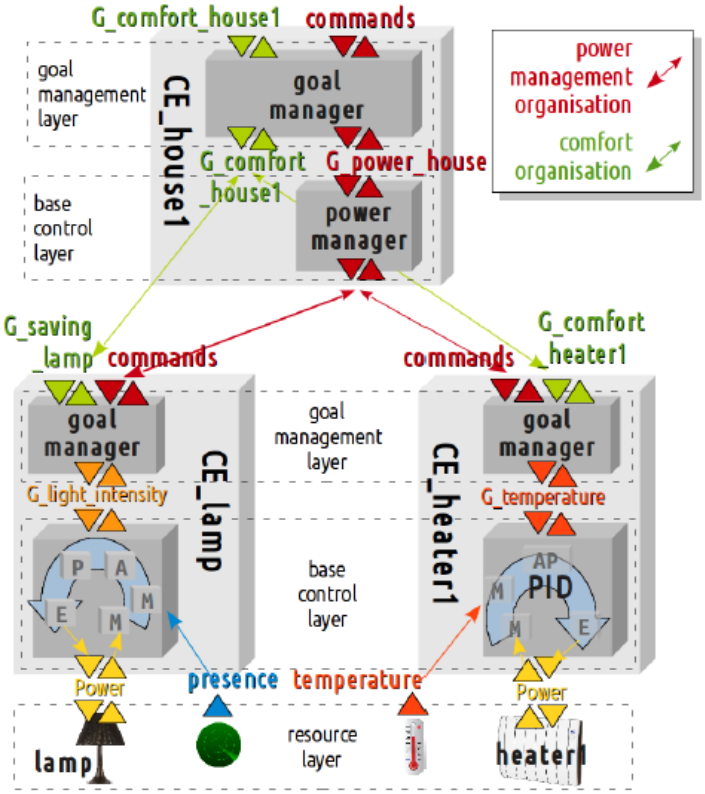


Figure 3.6: Design detail for the house control [10]

soCloud [11], is a service-oriented and component-based PaaS for the management of distributed and large scale applications across multiple Clouds. The management is mainly focused on the portability, availability, elasticity and provisioning of applications. soCloud provides a model based on extended SCA standard to build multi-Cloud SaaS applications. Its architecture is split in two parts: soCloud agent, and soCloud master as depicted in Figure 3.7. The soCloud agent hosts, monitors and executes soCloud applications. It also mediates interactions between the soCloud master and the deployed applications. The soCloud master holds the components: Workload Manager, Controller, Service Deployer, Constraints Validator, Node Provisioning, PaaS Deployment, SaaS Deployment and a Load Balancer. The Workload Manager filters and aggregates the events (i.e., metrics values) received from the Monitoring. It also analyses the events according to some rules then sends reports to the Controller. This latter constructs actions necessary to achieve management objectives. The Service Deployer is responsible for the management of services (e.g., bindings and locations) based on rules and constraints specified by the service developer. Furthermore, this component selects the cloud providers that fulfill the management constrains and rules and provide the lowest prices. These constraints and rules should be validated by the Constraint Validator so that the application deployment can be performed by the Service Deployer that executes the suited operations of Node Provisioning, PaaS Deployment and SaaS Deployment. The Load Balancer is used for the high availability to balance service requests

among application instances that are deployed on multiple Cloud providers. Regarding the adaptation logic, the authors annotated SCA artifacts to express their management rules and constraints. Thus, they used annotations to ensure elasticity by increasing/decreasing instances of application components. They also annotated location constraints to map components from a physical host to another and annotated computing resource constraints for the provisioning of VMs. Regarding the availability management, they defined annotations for replication rules to specify the number of instances for each component. They also used the Load Balancer, in case of node failure, to switch from an application instance to another. This work has the advantage of using the SCA standard. However, the adaptation logic is based solely on rules and constraints which could have been improved with learning abilities.

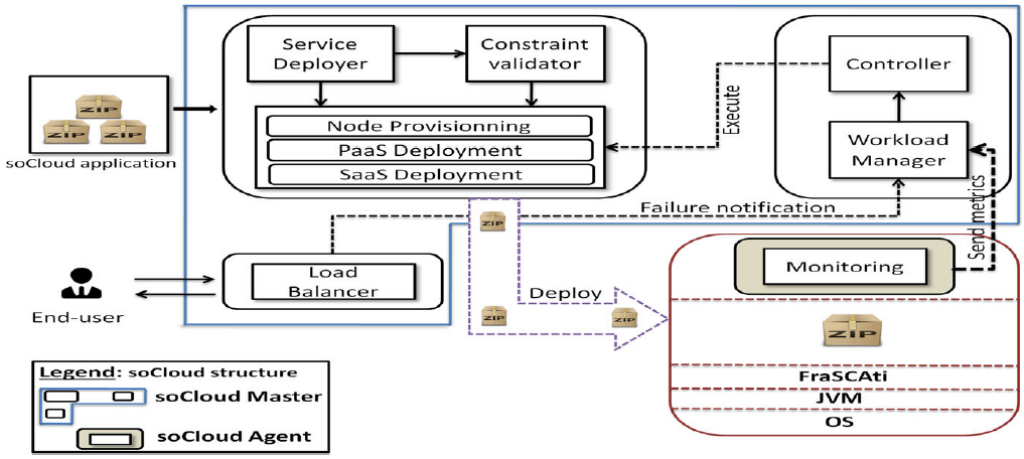


Figure 3.7: Overview of the soCloud Architecture [11]

3.3 Control-based Approaches for Autonomic Management

In this Section, we investigate approaches that considered the autonomic management in computing systems as control problems. In the following, we provide an overview on solutions stemming from adaptive control theory and discrete control synthesis techniques.

3.3.1 Adaptive Control Theory Techniques

Control theory [62] is a formal and well established discipline that initially emerged to build robust systems (e.g., electrical, mechanical, aeronautical systems, etc.) that behave as expected. Control theory provides mathematically grounded techniques that allow designing controllers for industrial plants. Recently, there are research work that started using control theory modern principles to build adaptive and autonomic systems with such robustness [63].

As shown in Figure 3.8, a managed resource is managed by a controller by computing input and output information signals overtime. The objective is to have the managed resource dynamic characteristics and

systematically perform corrections on detected errors overtime. The system acts according to a feedback control loop. In adaptive systems, this loop makes decisions and then autonomously modifies in the controller parameters to adapt the control signals to the changes of the managed resource. The feedback control analyses the results of the implemented decisions and compares them with the required results to look for possible errors. Afterwards, the feedback control formulates corrective measures.

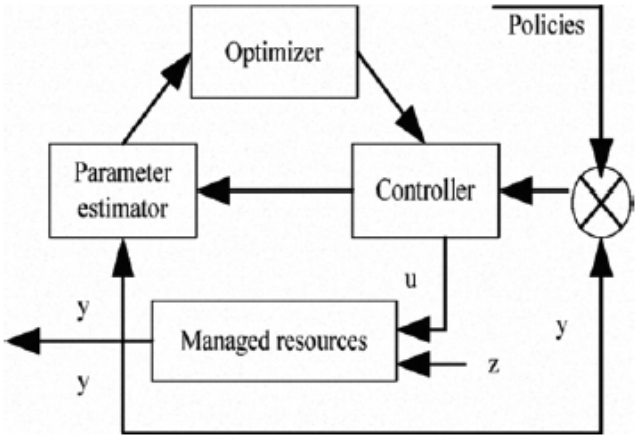


Figure 3.8: Autonomic clement based on adaptive control theory [12]

An early work that addressed autonomic management by using adaptive control theory principles is introduced in [37]. In their work, the authors explored control theory principles to build self-managing computing systems. They considered a cluster of Apache HTTP servers and intended to optimize their response time. The output information signal is the CPU utilization and the control input is a parameter that describes the number of maximum connections allowed by a server. The parameter is adjusted to adapt the CPU utilization. The controller should adapt to system dynamics that are determined in terms of arrival rates and shifting from static to dynamic pages in terms of page requests. Driven by the need to ease research in autonomic computing, the authors introduced DTAC, a deployable testbed enriched with pluggable components (e.g., control algorithms, sensors, effectors). Authors proposed the SASO properties (Stability, Accuracy, Settling time and Overshoot) of control systems to evaluate an autonomic computing system after an adaptation occurs. Similarly, control theory principles were also used in [36] in database systems for their self-management. The used feedback control loop showed its efficiency when evaluated according to the SASO properties. However, the focus area of these approaches [37, 36] is solely on autonomic features of their systems, they did not propose any learning mechanisms to learn for previously unseen situations.

Another work [35] targeted the optimization of power consumption, cooling and performance management over different resource levels (e.g., virtual machines, servers, etc.) in a data center. The proposed approach leveraged feedback control mechanisms to govern and provide multiple power consumption management solutions. The authors used controllers for their resources and established a set of constraints on their power consumption. The controllers expose APIs control interfaces to allow for external tuning. For instance, the authors introduced a VM Controller (VMC) that maintains constraints such as budget violations or VM power

utilization. The VMC has then a constrained optimization problem in which it should select the decision actions that optimize its constraints. The decision actions can be the number of VMs to be mapped to servers or migration actions. The main purpose is to minimize an objective function that encompasses the migration overhead and the total power consumption of the VMCs. To search for the near optimal VMs placement, a greedy bin-packing algorithm [64] is used to satisfy all the constraints. Nevertheless, the authors constructed fixed strategies for their adaptation logic.

A recent research work [65] that presented formal grounding on the effectiveness of control theory principles to equip distributed systems with adaptive mechanisms. In this paper, different classes of controllers designs were introduced: time-based, event-based and knowledge-based controllers. Time-based controllers are divided into two classes continuous-time and discrete-time controllers depending on the input signals of the system whether they are continuous or discrete. In both controllers, the controller acts depending on a pre-specified time intervals. The authors argued that the discrete-time controller fits better to control software systems given that these systems signals are not continuous [66]. When these time intervals are not pre-specified but dependent on external inputs of the system, then the controller is event-based. The authors presented the knowledge-based controller as a rule-based controller and stated that it is best suited to build self-adaptive software systems. The authors also argued that the lack of knowledge about the system may limit the effectiveness of this controller. In this case, learning techniques could be coupled with a knowledge-based controller to infer new rules for the system control.

3.3.2 Discrete Control Synthesis Techniques

We reviewed the approaches that built controllers for the autonomic management in computing systems. The following approaches synthesized controllers based on reactive models and discrete control techniques [67]. These techniques provide tools to generate controllers embedding the control logic (i.e., adaptation logic) by using a specification of a system resource and a description of the desired behavior. The behavior description is often modeled as an automaton with a set of states and transitions from a state to another.

For instance, research work [13, 39] adopted discrete control synthesis techniques and showed their effectiveness for the autonomic management of their systems. The authors addressed resource optimization of replicated and load balanced servers in cluster-based systems. They targeted the properties of self-sizing by dynamically optimizing the replication degree of servers and self-repairing by managing the failure of the machines hosting the servers. In their approach, authors used synchronized languages and discrete control synthesis to manage their application composite. As shown in Figure 3.9, they defined an Autonomic Manager (AM) as an observable and controllable component and supposed that it is equipped with a model of its behavior. Their solution consisted of using the Heptagon/BZR reactive language [68] to program an AM into a node. In a node, the behavior model of an AM is translated into a finite state machine (i.e., automaton). The behavior of an AM is controlled through controlling its state transitions by using some equations that define the reaction steps. The reaction steps represent the policy or the behavioral contract associated to the node. We believe that an adaptation logic based on a behavioral contract is inflexible for dynamic contexts. It seems not being subject to reuse without recoding from scratch the behavioral contract to fit to another context even for a small change of this context. Moreover, this adaptation logic requires an intensive knowledge about all the possible context changes to predict the transitions of the states and the impacting decisions.

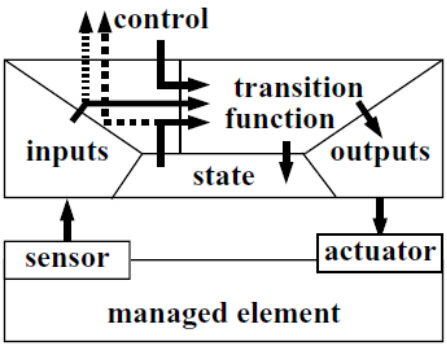


Figure 3.9: Heptagon/BZR code detailed model [13]

Another work using similar techniques is presented in [14]. In this paper, the autonomic management of smart devices is addressed in Internet of Things (IoT) environments based on ECA rules and their coordination. The proposed solution consisted of performing a model transformation from a given description of the ECA rules to a synchronous programming language. Accordingly, a set of boolean expressions are defined to specify the set of inputs (i.e., actions) and outputs (i.e., device states) signals then the boolean expressions are mapped to the ECA events. Afterwards, the ECA rules are translated into a description grammar and then into a Heptagon/BZR program code. This code is defined in the body of a main node divided into three sub-nodes: events, rules and devices as shown in Figure 3.10. The events node processes the received signals to determine if they occur according to the ECA definition. The rules node triggers the actions according to the events received from the events node. The triggered actions are sent to the devices node to be processed before sending them to the actual devices.

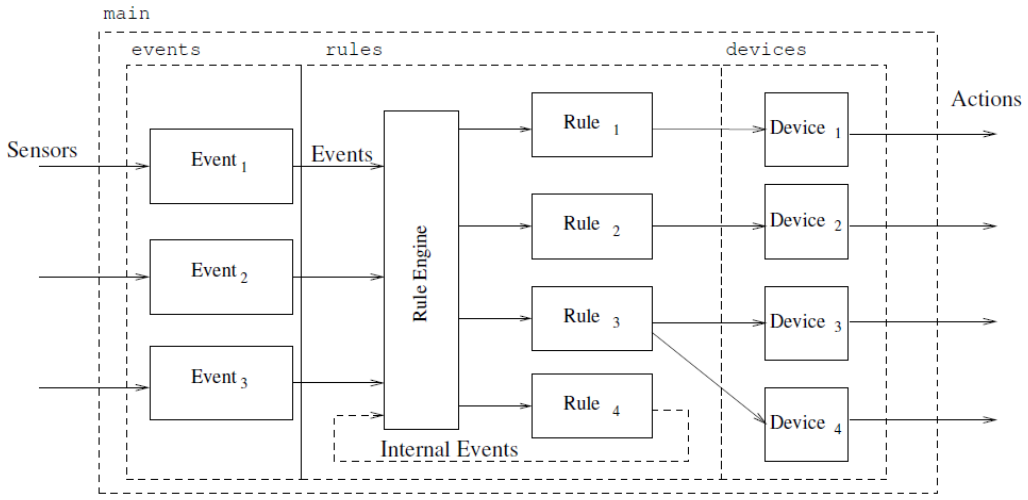


Figure 3.10: Controllable AM: A case of a finite state machine manager [14]

The authors targeted also problems related to ECA rules management such as: redundancy when a rule is replicated; inconsistency when contradictory rules are applied and circularity when rules are activated without a finished execution to reach a stable state. The authors proposed the usage of verification and control techniques on their ECA rules. They used verification to detect syntax errors with the Sigali synthesis tool [69] for model checking and filtering of rules to solve the redundancy problem. They also used the Heptagon/BZR code to detect inconsistencies in these rules. Moreover, the authors defined circularity problems as a causality error and detected them by using the heptagon/BZR. The circularity is solved by using an automaton that observes the oscillation states. After verification and control of these rules, the authors generate the controller (that is a finite state machine) to manage the smart devices. However, the usage of these techniques involves the generation of a controller with a behavioral contract for the management of multiple devices in the system. In such techniques, any additional ECA rule would enforce redoing the whole process to integrate the new rule and then later generate the new controller.

Similarly, in [38], control and synthesis techniques were used to autonomously manage applications in Cloud systems. The system is composed of a self-sizing and self-repairing autonomic managers (AMs). The first AM handles the servers' replications while the second detects and repairs failures. An AM is represented by an abstract model, namely Labeled Transition System (LTS), that corresponds to all the possible executions of the manager. An LTS AM is modeled with a sequential behavior and described as a graphical model with a set of states and transitions of the states. The authors needed to generate the model of a global LTS AM controller for their self-sizing and self-repairing AMs with synthesis techniques. To do so, they used a specification language, namely LNT [70] (LOTOS Translator), to build a set of regular expressions with reactive rules to specify how both LTS AMs should manage the system. The regular expressions are encoded as alphabets of actions (e.g., messages like add server message) and states. Validation techniques were also used to validate the regular expressions by using the CADP [71]. CADP takes as input the controller model and some defined logic properties as reactive rules (e.g., checking for deadlocks in the controller). However, the generation of the global LTS AM controller centralizes the management at its level and deprives the other controllers (i.e., self-sizing and self-repairing) from their autonomic abilities. Such approach represents source of bottleneck which is not suitable for complex distributed environments. Furthermore, the adaptation logic is hand-coded and model-based which renders inflexible the deployment in a different context without thoroughly changing this logic. This approach does not take into consideration the possibility that the system might encounter unknown states.

3.4 Learning-based Approaches

There have been research work undertaking studies on how to use Reinforcement Learning (RL) techniques to achieve automatically some tasks in computing systems (e.g., dynamic service composition). There are also other research work that targeted autonomic management objectives. Thus, they explored the feasibility of RL techniques in combination with autonomic computing principles in different contexts and for different management concerns. In the following, we provide an overview on some of these approaches.

3.4.1 Learning for Task Automation in Computing Systems

RL techniques have shown their efficiency in computing systems other than in robotic field. There have been research work using RL algorithms for adaptivity and tasks automation in software systems. For instance, research work presented in [40] addressed adaptive service composition by using RL techniques. This work targeted the dynamic adjustment of service composition for web services based on QoS (Quality of Service) attributes to ensure user satisfaction. In their approach, the authors modeled service composition as a Markov Decision Process (MDP) in which no prior knowledge about service composition in required. The learning algorithm that they proposed dynamically adjusts itself to fit a varying environment, where the properties of the component services continue to change. For each web service, some QoS attributes are defined and should be considered during the service composition. An action consists of a web service call and a learning episode consists of series of web services calls in order to constitute the optimal workflow for the needed service composition. A simulated environment of web services was used to prove the efficiency of the proposed approach. Nevertheless, the authors did not propose to enhance the learning approach to improve the learning performance of their one-step algorithm. This could have potentially made their approach more suitable for real-world service compositions to avoid possible initial poor performance of the algorithm.

Similarly, a recent work [41] targeted dynamic and optimal web service composition by assembling existing services to alleviate cost and deployment time. The learning performance is enhanced by proposing a distributed version of the learning algorithm. A service composition is defined as a task and split into sub-tasks, each one is handled separatly by a RL agent. The authors used the theory of articulation state to decompose a service composition into multiple service compositions and then assign them to the agents before the learning phase actually starts. Each sub-task represents a sub-goal for an agent which reduces the search space and then accelerates the algorithm convergence. Agents having similar sub-tasks can share their gained strategies which improves their efficiency. The authors reduced the communication cost among agents by using an agent that supervises and takes in charge the communications between the other agents. This agent supervises their learning and synchronizes their computations. It also retrieves information about the participating agents then attributes the sub-tasks to agents that can satisfy some QoS constraints. This approach is tested in a simulated web service environment and showed its effectiveness. Nevertheless, the supervisor may be source of bottleneck in case of synchronizing several agents to handle a large service composition. In comparison with their approach, they used the one-step learning algorithm that they enhanced with the theory of articulation state for sub-tasks splitting. In contrast, we use a multi-step learning approach that is a combination of Temporal Difference methods [27] and Monte Carlo methods [44] to enhance the learning performance.

3.4.2 Learning for Autonomic Computing Management

There have been research work undertaking studies on how to use learning techniques in ACS for different management concerns and in different contexts. The following gives an overview on the proposed learning-based solutions.

Early usage of RL techniques in ACS is shown in [42] to address dynamic servers allocation to applications in data centers. The main goal is to perform optimal resource allocation of a number of servers among a number of applications to dynamically build an optimal allocation policy. A utility function was defined to

express the service level delivered to the clients at certain demand level and used to compute reward values. As shown in Figure 3.11, the allocation decisions is made by a resource arbiter that reallocates servers among the applications. The arbiter's decision is taken on the basis of a resource utility curve V(n) that it requests from each application manager and uses it to compute its value function. This curve is computed according to the monetary benefit generated by each application and defined in the SLA. The arbiter chooses a joint action for all its managed applications and it should maximize its value function.

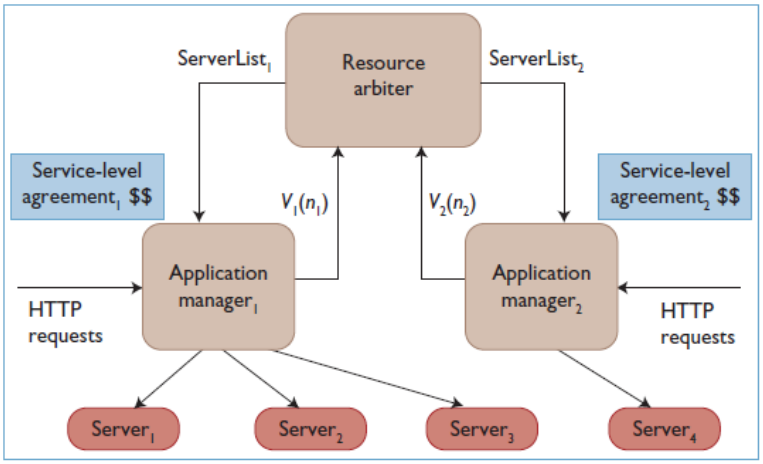


Figure 3.11: Resource allocation scenario in prototype data center [15]

Authors had to reduce the state space by making a strict approximation to represent the applications state. Pre-learned initial value function was also claimed to address poor performance and convergence slowness of the algorithm. However, the decision process is centralized at the arbiter's level which is source of bottleneck when the system scales. This approach remains less efficient since the algorithm performs poorly if the state and actions representations are not compacted. The proposed algorithm converges slowly which increases the SLA violations frequency during the learning phase. The authors enhanced the performance of their algorithm in [21, 15] with Neural Networks [72] to improve the performance of their learning approach. Thus, they used the data as they collected it to train their Neural Network. However, Neural Networks are known for requiring the availability of big data set to train on (which is not our case) in order to provide a noticeable improved performance. Furthermore, their approach seems solely coarse-grained, the arbiter should take a joint decision for several application managers which remain a major source of bottleneck.

iBalloon [43] is a framework that targets virtualized resource provisioning in clustered cloud environments. A RL mechanism is proposed to facilitate VMs resource allocations to optimize the performance of the hosted applications. Thus, a dynamic tuning of the VM resources such as CPU, memory and I/O bandwidth is performed. The cluster cloud management problem is decomposed into sub-problems of individual VM resource allocations. The cluster performance is considered optimized if the individual VMs meet their SLAs (i.e., regarding their hosted applications) along with high resource utilization. The framework functionality is decoupled into three components: Host-agent, App-agent (i.e. Application-agent) and Decision-maker. Once a VM appears on the system, iBalloon maintains its application profile and history records for the anal-

ysis of future capacity management. The VM submits its SLA profile to the App-agent and registers within the Host-agent and the Decision-maker. A physical machine holds a Host-agent that collects synchronously all the VMs resource requests and allocates resources to them then computes a feedback signal (i.e., reward value). The App-agent maintains the application SLA profile and reports on its performance. The Decision-maker hosts a learning agent that makes the allocation decisions. The learning approach relied on negative rewards to punish the VMs that request resources in order to force them to use their resources more efficiently and prevent them from being in resource contention. Nevertheless, the adaptation logic is coarse-grained and the one-step learning approach was adopted without improvement which may induce poor performance of the learning algorithm at the early stages of learning.

Another work using RL techniques in ACS is presented in [16] and applied to a simulated model of a web application in Matlab Simulink. A learning agent is added to the planning phase of the MAPE-K loop to learn a policy that should select the best adaptive actions. The proposed agent can be modified to make both reactive and deliberative decision making. Figure 3.12 depicts the decision maker that was proposed, it holds a state generator that discretizes the attributes observed values. It contains also a state mapper that aggregates the discretized attributes, a reward function that computes the reward values, a RL engine that both decides for the next action to execute and computes the value function. The proposed learning algorithm learns about its environment and manages to compute an optimal decision policy. However, the learning performance could have being enhanced if the authors have proposed an adequate mechanisms to render their approach suitable for real-world applications other than simulated ones.

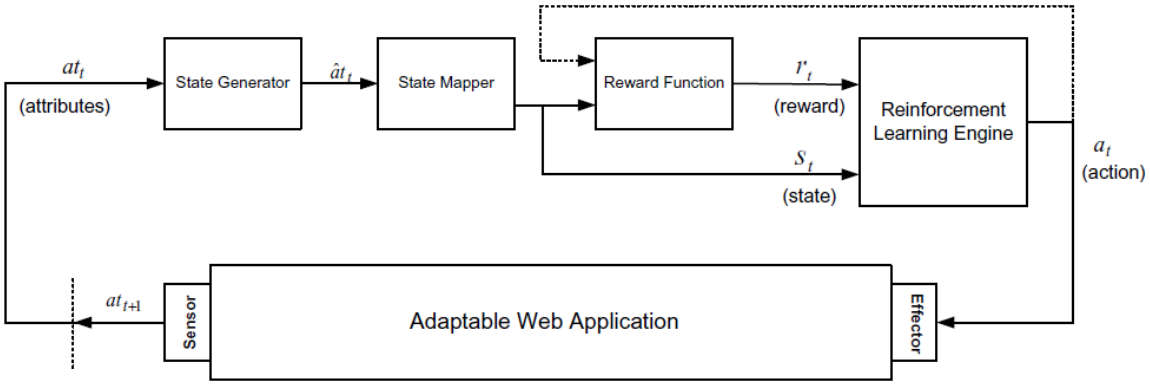


Figure 3.12: Process Model of the Proposed RL-based Decision Maker [16]

A recent work [17] addressed the problem of adaptive service management in Mobile Cloud Computing (MCC) [73]. The purpose is to perform an optimal location selection for service task execution from mobile devices to the Cloud in order to optimize the execution cost. The cost corresponds to the quality of experience that can consider parameters such as service execution time, battery life or user satisfaction. In this paper, the authors introduced an agent based on RL and supervised learning [74] algorithms. The former is dedicated to learn optimal task allocation policy while the second is intended to constitute and exploit a knowledge model of trained data. The proposed architecture of the agent is depicted in Figure 3.13.

The agent consists of four modules: Processing Module, Learning Module, Training Data and Generated

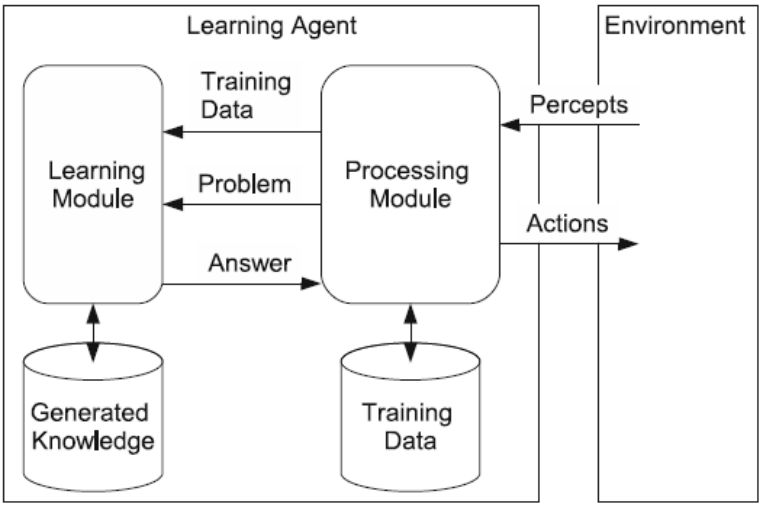


Figure 3.13: The service management agent architecture reflecting its learning abilities [17]

Knowledge. Processing Module receives and processes the percepts from the environment. It also stores the training data and carries out the actions (i.e., tasks migration). The Learning Module runs the learning algorithm and uses the knowledge learned to answer problems. The Training Data stores data samples to be used during learning. The Generated Knowledge stores models of the learned knowledge. The proposed agent uses some information (i.e., percepts) to decide which service task to migrate from the mobile device to the Cloud. These information are about the devices conditions in terms of memory, CPU and network. They can also be about potential costs such as data transmission charges. The proposed agent showed a dynamic learning of the migration decisions for video file processing. The RL algorithm that was used is one-step learning and it could have been enhanced to improve the learning performance.

3.5 Synthesis of the Related Work

When analyzing the aforementioned standardization efforts/usage [28, 7, 8, 33, 30, 29, 9, 31, 11], we find that they are advantageous for the autonomic management of the applications. The adoption of standards strengthens their solutions by making them more accessible for usage and more generic. In comparison to these approaches, our proposed framework is designed and implemented using the SCA standard which ensures its usage for applications regardless of their implementations. By being a technology agnostic standard, SCA allows the development and deployment of SOA applications by composing existing components (or services). Therefore, designing the framework based on this standard allows its usage for a variety of applications.

Furthermore, the functionalities of our framework are separated among different components to allow an isolated replacement or modification of them which ensures more flexibility and reusability to the framework. It is also designed to be independent of the deployment context, generic and extendable at the same time to meet the specific management needs of applications. Our approach enables a fine-grained as well as a

coarse-grained transformation of an application which is not provided by these frameworks [9, 31].

When looking closer into the proposed adaptation logics of these standardizations and frameworks [28, 7, 8, 33, 30, 29, 9, 31, 32, 34, 10, 11], we find that they are based solely on reactive and predefined ECA (Event Condition Action) rules. They do not propose any learning mechanisms for their adaptation logics in contrast to our approach. Unlike these work, the adaptation logic embedded in our framework has the advantage of allowing a dynamic computation of the decision policy. The computed policy can be modified at execution time according to the context dynamics in contrast to the previous approaches where the policy is fixed.

When analyzing control-based approaches [37, 36, 35, 13, 39, 14, 38], we find that they have the advantage of providing autonomic computing systems with the SASO properties. However, it is hard to propose generic and flexible approaches since they seem domain-specific. In most cases, these approaches are model-based, they require an intensive knowledge about the context dynamics to hand-code fixed strategies. Such knowledge may be costly to acquire in terms of time and effort and not necessarily available at deployment time. It involves predicting the transitions of the system states upon executing the impacting decisions.

Regarding these work [13, 39, 14, 38] that enabled the usage of discrete control synthesis to generate controllers. Their usage of behavioral contracts may be inflexible when the context changes. These behavioral contracts seem not being subject to reuse to fit to other contexts. In such techniques, any additional new ECA rules should be integrated to the system by following from scratch the process of generating the controller. These work did not propose any learning mechanism similarly to [65]. Except when the authors of this paper argued that control techniques can be coupled with learning techniques when the knowledge about system dynamics is not always available.

Succinctly, the aforementioned control and synthesis-based approaches tend to create controllers that use hard-coded policies known for their reactive behavior but deprived from reasoning abilities to self-adapt their behavior in highly dynamic real-world contexts. In contrast, our approach proposes a model-free and learning-based approach for self-adapting to previously unseen situations and to reason for dynamic optimal policy inference.

When reviewing the learning-based approaches [40, 41, 42, 21, 15, 43, 16, 17], we noticed that in contrast to them, our decision and learning logics are both embedded in the Analysis component (of the MAPE-K loop) and are dedicated to each component of the application. They are also separated among different components to allow an isolated replacement or removal to ensure more flexibility and reusability to the Analysis component.

In addition, we adopted a multi-step learning approach by combining Monte Carlo methods and Temporal Difference methods that show very promising results (see Chapter 6). Our approach enables the acceleration of the convergence time to spend less time in SLA violation during the learning phase which is best suited for real-world applications. Thus, the system can scale for larger state and action spaces which is hardly achievable with one-step approaches adopted in [42, 43, 16]. To improve the learning performance, in [41], the learning is decomposed among several one-step learning agents by using the theory of articulation state while in [21, 15] neural networks were combined with one-step learning. In [17], one-step learning were combined with supervised learning to construct classifiers and collect statistical data in order to build a knowledge model in contrast with out model-free approach.

In Table 3.1, we illustrate a synthesis of the studied research work. For each work, this table shows whether it verifies the characteristics that we have established for our framework. These characteristics are derived directly from our thesis objectives and are previously described in Section 1.4. When a work responds to one of these objectives, there is a check mark ✓ in the corresponding cell, otherwise, this cell is empty.

Table 3.1: Synthesis of the related work

Studied Solutions	Genericity	Standardization	Granularity	Adaptation Flexibility	Performance Optimization
Standardization Efforts and Frameworks for Autonomic Management					
IBM ToolKit [32], 2004 Frey et al. [34], 2012 Frey et al. [10], 2013	✓		✓		
JMX [28], 2006 Adamczyk et al. [7], 2008	✓	JMX	✓		
Ruz et al. [8], 2010	✓	JMX, SCA	✓		
TOSCA [33], 2013 Brogi et al. [29], 2015	✓	TOSCA, BPMN	✓		
CAMP [30], 2017	✓	CAMP, REST	✓		
FRAMESELF [9], 2012, [31], 2015	✓	REST, ETSI, GraphML			
soCloud [11], 2016	✓	SCA	✓		
Control-based Approaches for Autonomic Management					
Lightstone et al. [36], 2007					
Yixin et al. [37], 2005 Raghavendra et al. [35], 2008 Gueye et al. [39], 2012 Gueye et al. [13], 2013 Abid et al. [38], 2017			✓		
Cano et al. [14], 2014	✓				
Filieri [65], 2017				✓	
Learning-based Approaches					
Tesauro [42], 2005 Rao et al. [43], 2011	✓			✓	
Tesauro et al. [21], 2006 Tesauro [15], 2007	✓			✓	✓
Wang et al. [41], 2016				✓	✓
Amoui et al. [16], 2008				✓	
Wang et al. [40], 2010				✓	
Nawrocki et al. [17], 2018				✓	
Our Approach	✓	SCA	✓	✓	✓

3.6 Conclusion

In this chapter, we presented different existing and relevant work in different contexts and for different management concerns. We classified these work into three major categories depending on the types of solutions they provided. We started by reviewing existing generic approaches such as standardization efforts, frameworks, toolkits and methodologies. We followed up with control-based approaches that used adaptive control principles and discrete control synthesis. Afterwards, we presented learning-based techniques used for task automation and autonomy in computing systems. Finally, to have a clear position of our contributions regarding the existing work, we performed a comparative synthesis on the basis of our contributions characteristics. In the next Chapters 4 and 5, we will introduce our contributions that respond to the addressed characteristics. We will start in Chapter 4 by introducing the aspects related to our proposed adaptation logic. Later on, we will introduce in Chapter 5 the aspects related to the architecture of our framework and a case study demonstrating its usage.

Chapter 4

Self-Adaptive Decision Making Process based on Multi-step Reinforcement Learning

Contents

4.1 Introduction

Component-based applications entail a composition of distributed software components often running in different environments. The components maintain business interactions by providing and requiring services to each others to accomplish their business logic. The continuous growth of these applications' complexity and

the dynamic conditions of their contexts result in increasing management tasks that burden the application developers. Autonomic Computing [20] came to provide systems with the needed autonomic behavior to make them self-adaptive to their context changes. Autonomic Computing Systems (ACS) imply the usage of an autonomic loop that adapts the system not just to its internal changes but also to its environment dynamics. In most cases, this loop is designed based on static and hand-coded adaptation logic, known for being time and effort-consuming and potentially error-prone.

When deploying an application, the developer may not have an exhaustive knowledge about the context dynamics. They can only make use of their limited view for hand-coding some adaptation scenarios by mapping few system states to decision actions to constitute a decision policy. However, it is hardly feasible to envisage all the possible adaptation scenarios or be accurate enough to reach the adequate adaptations. Hence, such adaptation logic is not able to determine appropriate adaptations for previously unknown situations. In fact, this logic entails developing an accurate model of the context dynamics which is typically a difficult and time-consuming task, hence requiring a costly expertise. As such, this logic requires a detailed understanding of the system design to be able to predict how changes brought to the system may affect its performance to finally cater to the established Service Level Agreement (SLA) [75]. Such adaptation logic is perfectly feasible for small contexts, but not suitable for nowadays complex contexts.

ACS should be sophisticated to deal with the dynamic contexts of complex applications and learn to solve problems based on their past experiences [22]. However, as described, ACS are self-adaptive based on their predefined adaptation logic [23]. Consequently, we consider that ACS should be based on learning capabilities to learn to dynamically compute and change their adaptation logic to fit to their dynamic contexts.

Reinforcement Learning (RL) [24] offers advantages from ACS perspectives. RL provides the possibility to build an adaptation logic dynamically without requiring a detailed understanding of the context dynamics. Furthermore, by its grounding in Markov Decision Processes [25], RL's underlying theory is fundamentally a sequential decision theory [26] that deals with dynamic phenomena of a system. Therefore, RL may provide benefits for ACS by being part of the building blocks composing their adaptation logic.

However, commonly used RL techniques, namely one-step approaches, are known for their poor performance induced during the early stages of learning [21]. Used in some specific management concerns, the one-step learning approaches have shown a dynamic but very slow adaptation to the context changes. In fact, the existing one-step approaches do not cope with the complexity of nowadays applications to scale to their large state space. They converge very slowly, hence, increasing the time the system might poorly perform by spending longer time in SLA violations. Actually, at the early stages of learning, a RL-based learner has not acquired enough experience on its context yet, it may accumulate the selection of random and potentially bad adaptation actions. This represents a major hindrance to delivering an acceptable service quality that respects the SLA terms established for the application. Subsequently, constituting an obstacle to the widespread usage of RL techniques in real-world deployed system.

In this chapter, we aim to enhance the adaptation logic of ACS with sophisticated and better-performing learning abilities. Since the decision making process of the adaptation logic is driven by the Analysis component of the autonomic loop, we propose to enhance the Analysis with a multi-step learning approach to dynamically learn an optimal decision policy. The proposed approach adds more autonomy and more efficiency to the learning performance. The learning is timely-efficient by converging faster compared to one-step approaches. In addition, our approach reduces considerably the time the system might spend in SLA viola-

tions during the beginning of learning phase. Accordingly, we introduce the underlying multi-step RL theory used to enhance our adaptation logic. We also provide the algorithmic details that guide the decision making process of our learning approach.

The remaining chapter is structured as follows. In Section 4.2, we give an overview of our approach. We introduce the classical behavior of static adaptation logic and then the improvements intended to be brought to this logic. In Section 4.3, we model the decision making process of the Analysis component as a Markov Decision Process (MDP) problem. We solve the MDP problem in Section 4.4 by first arguing for our learning techniques choices. Then, introducing the learning algorithm that conducts the Analysis behavior to learn dynamically an optimal decision policy. Finally, we conclude this chapter by underlining the main aspects of our contributions and introducing for the next chapter.

4.2 Approach Overview

A component-based application consists of a composition of distributed components offering and requiring services from one another to fulfill their business logic. To endow the application with a self-adaptive behavior, we use Autonomic Computing principles to equip the application components with the needed autonomic abilities. In our approach, we propose to associate each of the application components with an autonomic MAPE-K loop (refer back to Figure 2.4) and call them Managed components. This loop implements five central components: Monitoring, Analysis, Planning, Execution and Knowledge [20]. These components act by collecting monitoring data, analyzing them and in case of anomalies, planning for corrective actions (also called adaptation or decision actions) then finally executing them. The Knowledge maintains necessary information for the functioning of the rest of the MAPE components. This loop is intended to adapt its associated Managed component to the dynamic of its context. It adapts the component not just to its internal changes but also to the dynamic of its hosting environment.

Classically, the MAPE-K loop behavior is built upon a static and hand-coded adaptation logic which provides this loop with a predefined and reactive behavior [76]. Indeed, the Analysis component examines the monitoring data and determines anomalies as adaptation contexts by means of predefined rules it holds. Whenever an adaptation context is confirmed, the Analysis component alerts the Planning component that triggers an adaptation action according to these rules. More specifically, they are called ECA (Event Condition Action) rules which represent a static decision policy. ECA-based decision policies react by triggering a decision action (i.e., adaptation action) for execution, upon reception of an event that verifies a described condition. As previously described, this adaptation logic is static and not able to envisage all the possible adaptation scenarios to hand-code them. We need to address its shortcomings by proposing a sophisticated adaptation logic that dynamically computes a decision policy.

In our approach, we intend to enhance the classical behavior of Autonomic Computing Systems (ACS) with a learning-based approach. We propose to improve the adaptation logic of the MAPE-K loop by equipping the Analysis component with Reinforcement Learning (RL) techniques. Instead of an adaptation logic based on static ECA policy, we build this logic based on a RL algorithm. The constructed adaptation logic will then be able to dynamically learn an optimal decision policy (i.e., best possible decision actions) based on its past experiences. However, we have studied RL techniques and know how poorly they may perform during

the first stages of learning. This disadvantage is not tolerated in real-world deployed systems. To preserve the system's performance, we propose to equip the adaptation logic with better performing RL techniques, namely Multi-step learning approaches. The enhanced adaptation logic efficiently learns an optimal decision policy at execution time.

Our enhanced Analysis component runs a RL algorithm to dynamically compute the optimal decision policy. Accordingly, we propose to equip the Analysis with two distinct behaviors before and after convergence of the learning algorithm. Before the convergence, it adopts both a decision and learning behaviors. We refer to these two behaviors as its ability to make decisions (i.e., select actions) then reason to learn about them in order to determine better decisions in the future. The convergence of the learning algorithm implies that the Analysis has concluded the computation of its optimal decision policy. Thus, after convergence, the Analysis component adopts a different behavior. It stops its learning process then sends the computed policy to the Planning component. After convergence, the Analysis component determines adaptation contexts according to what it has learned, then alerts the Planning component. The latter chooses the best possible decision action corresponding to the optimal policy computed earlier by the Analysis component.

To construct our adaptation logic, we formulate the decision making process of the Analysis component as a decision making problem then propose the adequate mechanisms to solve it. To do so, we explore the underlying theory behind RL algorithms, namely Markov Decision Processes (MDPs). We model our decision problem as a MDP then map its different concepts to our approach. We solve our decision problem with the necessary algorithms that allow an efficient guidance of the decision and learning behaviors of the Analysis. Therefore, we endow the Analysis with better performing learning techniques based on Multi-step learning that outperforms the classical one-step learning techniques.

4.3 Modeling the Decision Making Problem as a Markov Decision Process

In this section, we introduce the decision making process of the Analysis component as a decision making problem. Thus, we argue our choice of modeling this problem as a Markov Decision Process (MDP) in order to solve it. Afterwards, we formulate the different MDP concepts that are related to our approach.

As defined in [25], Markov Decision Problems are mathematical formalism of decision problems that generalizes the shortest path approaches in stochastic environments. Their grounding is fundamentally a sequential decision theory that deals with dynamic phenomena of an environment. Hence, representing the underlying theory of Reinforcement Learning (RL) algorithms. A RL-based agent that interacts with its environment, acts according to a Markov Decision Process (MDP). A MDP integrates the concepts of: state that summarizes the agent's (i.e., the Analysis component in our case) situation; the decision action that influences the state's dynamics; the reward signal that is a value associated with each of the state transitions; and the transition probability that indicates the cognition of the context dynamics. Therefore, MDPs are Markov's chains that visit states, controlled by actions and valued with rewards. Solving an MDP means controlling the agent's behavior so that it acts optimally by choosing optimal actions. Acting as such, the agent in order to maximize its rewards and finally optimize the system's performance [44].

The MDP grounding as well as concepts meet adequately our needs to model our decision making problem. Thereby, we consider the decision making problem of the Analysis component as a Markov Decision

Problem. However, in a full MDP environment, all these concepts are previously known by the agent, namely, the set of states and actions, the reward function and the probability transition function. That is to say, in a full MDP, the agent knows the significance of states, whether good or bad they might be. It knows the impact of actions it chooses in certain states and the reward values it will receive. The agent has also a knowledge model of its context dynamics that describes the probability to transit from a state to another upon executing an action.

Note that in our case, our RL-based Analysis does not know the significance of its states or actions nor has a previous knowledge of the reward values and the context dynamics. The Analysis will interact with its context then learns gradually about it. The Analysis context is represented both by its associated system (i.e., Managed component of the application) and its deployment environment. By interacting with its context, the Analysis progressively learns how to optimally behave by selecting optimal actions. An optimal behavior allows to maximize the long run sum of its rewards in purpose to optimize its system's performance.

In the following, we formulate our decision problem as a finite MDP with a finite set of states and actions. During each state transition from a previous to a current state, the Analysis component receives a reward value. This reward indicates whether it chose the suitable action for the observed state or not. In our approach, we define the relevant concepts of MDP as follows.

4.3.1 State Space

The state space represents the set of states that summarize all the Managed component context situations during its execution life-cycle. An observed state of the Managed component context is monitored by the Monitoring component of the MAPE-K loop and then sent to the Analysis component. To determine this state, we need to determine the relevant information to be monitored from the Managed component and its environment. In fact, we should find the state representation that reflects the system's performance. Accordingly, we make use of the Service Level Agreement (SLA) terms that are established for the application during its deployment. Actually, the SLA holds some guarantee terms as a list of Service Level Objectives (SLOs) defined for some performance metrics. For instance, a metric could be the response time of some provided business service of the Managed component. The SLO of this metric can be some maximum value to not exceed (e.g., response time must not exceed $1000ms$). The number of the established metrics represents the most relevant metrics to represent the state of the Managed component. Note that this does not prevent from adding other metrics not necessarily defined in the SLA but can be relevant to the state representation.

For simplicity reasons, we refer to the state of the Managed component context as the state of the Managed component. More formally, we define a monitored (i.e., observed) state of the Managed component as a multi-tuples vector of the monitored metric values. The metrics are the ones targeted in the service level agreement defined for the application. Thus, for a given state s_i, its monitored metrics values are $\{v_{m_{1i}}, v_{m_{2i}}, .., v_{m_{ki}}, .., v_{m_{ni}}\}$, where $v_{m_{ki}}$ is the monitored value of the metric m_k that composes the i^{th} state and $1 \leqslant k \leqslant n$. The state vector should comply with the SLOs described in the SLA. We define these SLOs as: $O = \{o_{m_1}, o_{m_2}, .., o_{m_k}, .., o_{m_n}\}$, where o_{m_k} denotes the objective value for metric m_k explicitly required in the SLA.

We determine the state space S as being a finite set of states $\{s_i\}$ that are generated by combining all the

possible values of the targeted metrics $\{m_1, m_2, .., m_k, .., m_n\}$. We define the state space as follows:

$$S = \{s_i \ / \ s_i = (v_{m_{1i}}, v_{m_{2i}}, .., v_{m_{ki}}, .., v_{m_{ni}})\} \tag{4.1}$$

The state space contains the combination of all the possible values $\{v_{m_{1i}}, v_{m_{2i}}, .., v_{m_{ki}}, .., v_{m_{ni}}\}$ of the metrics $\{m_1, m_2, .., m_k, .., m_n\}$ whether complying with the SLOs or violating them. We generate all the possible values of each metric by sampling its values according to some determined step interval real value.

4.3.2 Action Space

The action space represents the set of adaptation actions used to self-adapt the Managed component to its context dynamics. An adaptation action occurs by invoking one or several Management services that are provided by the Execution component of the MAPE-K loop. A Management service is a non-functional service providing a set of operations that constitute the adaptation actions. The overall Management services provided by the Execution component represent the desired self-adaptive behavior of the Managed component. In our case, the action space is determined by the following list of Management services and their operations. This list is not limited and can be extended according to the management needs.

- LifeCycle service: enables creating, starting, stopping, restarting and destroying the Managed component,
- Scaling service: allows scaling-out by inserting new instances of the Managed component. It also allows scaling-in by removing extra instances of the Managed component,
- Parametric service: adjusts the adaptable properties values of the Managed component,
- Binding service: handles the functional dependencies of the Managed component by binding or unbinding it with other components,
- Adapter service: inserts and removes Adapter components such as a load balancer, a compressor, a decompressor, etc.

More formally, we define an action a_j as a single or multiple Management services among the Management services provided by the Execution component. Thereby, we determine three types of actions:

- Elementary services A_{el}, they represent an elementary Management service,
- Service compositions A_{comp}, they are defined as a composition of elementary Management services,
- Orchestrated actions A_{BPM}, they are Business Process Management (e.g., Business Process Execution Language [52]) and represent an orchestration of elementary Management services.

We define the action space A as a finite set of actions and is a combination of the elementary services A_{el}, the service compositions A_{comp} and the orchestrated services A_{BPM}:

$$A = \{a_j \ / \ a_j \in A_{el} \cup \ A_{comp} \cup \ A_{BPM}\} \tag{4.2}$$

4.3.3 Reward Function

We define a reward function that produces real values as reward signals intended to guide the Analysis component to learn the desired optimal behavior (i.e., select the optimal action). Thus, it ought to explicitly punish a bad action selection that leads to degraded performance, meanwhile, encouraging good action selection in the future. It also rewards a good action selection that helps maintaining or reaching the SLOs described in the SLA.

In our approach, we define the reward function to enable gauging the performance of our system after each executed action a_j. More specifically, we determine this function to evaluate the system (i.e., Managed component) quality. We define the system quality as its ability to meet its SLOs. We determine the system quality by comparing the monitored values of the Managed component's state to the SLOs defined in the SLA. To evaluate the system quality, we define a compliance degree function that compares the system state to the established SLOs. The function verifies whether the monitored values of the metrics complies with (or violates) the SLOs to provide an indication on the improvement (or degradation) degree of the system quality. Accordingly, this function initially identifies five classes to which the system quality is attributed. We list the system quality classes in Table 4.1 according to their desirability order regarding the SLOs, from the most to the least desired: VERY-GOOD, GOOD, NEUTRAL, BAD, VERY-BAD.

Table 4.1: System quality classes according to system state satisfaction

System quality class	State satisfaction to SLOs
VERY-GOOD	Largely satisfies the SLOs
GOOD	Satisfies the SLOs
NEUTRAL	Hardly satisfies the SLOs
BAD	Violates the SLOs
VERY-BAD	Largely violates the SLOs

Therefore, the reward function described in Algorithm 1 computes the reward values based on the compliance or violation degree of the current state s'_t. The function is also based on the system quality improvement or degradation degree regarding the transition from the previous state s_t to the current one s'_t after executing the action a_t. Accordingly, the compliance degrees of both states are compared to determine whether the system state has improved or degraded.

As shown in this algorithm, for a given transition t: $(s_t \rightarrow s'_t)$, each metric value $v'_{m_{kt}}$ of the current state s'_t, where $1 \leqslant k \leqslant n$ and n is the number of metrics composing a state, is compared to its value objective o_{m_k} to compute a reward distance $r_{v'_{m_{kt}}}$ (line 3). $d_{v'_{m_{kt}},o_{m_k}}$ represents a function that computes a real value as a distance between the value objective o_{m_k} and the current monitored value $v'_{m_{kt}}$. In our Algorithm, we choose to consider the minimum reward $r_{v'_{m_{kt}}}$ of a given metric m_k as the most significant reward regarding the evaluation of the state s'_t compliance degree $D_{s'_t}$ (line 5). This choice is motivated by our need to strictly punish a metric violation even when the other metrics are compliant. This will force the Analysis component to select the adequate actions that satisfy all the metrics SLOs. Although, one may choose to consider either the average of the rewards $\{r_{v'_{m_{1t}}}, r_{v'_{m_{2t}}}, .., r_{v'_{m_{kt}}}, .., r_{v'_{m_{nt}}}\}$, or their weighted average if the

metrics are prioritized or even their conditional average.

As previously stated, the compliance degree function define five classes to which states are assigned. Thus, based on the minimum reward computed, the state s'_t might be: VERY-GOOD, GOOD, NEUTRAL, BAD or VERY-BAD (line 5). Subsequently, the reward $r_{s'_t}$ is computed according to the class belonging of state s'_t (line 6). We also estimate the compliance degree D_{s_t} for the previous state s_t (line 8) for comparison purposes. We examine the current state improvement or degradation compared to the previous state by comparing the compliance degrees of the transition $(s_t \rightarrow s'_t)$ to compute $r_{(s_t \rightarrow s'_t)}$ (line 9). The final computed reward r_t is assigned to the couple (s_t, a_t) (line 11) and then issued to the Analysis to use it (line 12).

Algorithm 1 : Reward Function

1: **Input:** s_t, s'_t
2: **for all** metrics values $v'_{m_{1t}}, v'_{m_{2t}}, .., v'_{m_{kt}}, .., v'_{m_{nt}}$ **do**
3: $$r_{v'_{m_{kt}}} = \begin{cases} 0 & \text{if } v'_{m_{kt}} = o_{m_k} \text{ (NEUTRAL)} \\ d_{v'_{m_{kt}}, o_{m_k}} & \text{if } v'_{m_{kt}} \text{ satisfies } o_{m_k} \text{ (either GOOD or VERY-GOOD)} \\ -d_{v'_{m_{kt}}, o_{m_k}} & \text{otherwise (either BAD or VERY-BAD)} \end{cases}$$
4: **end for**
5: $D_{s'_t} \leftarrow complianceDegree(min\{r_{v'_{m_{1t}}}, r_{v'_{m_{2t}}}, .., r_{v'_{m_{kt}}}, .., r_{v'_{m_{nt}}}\})$
6: $r_{s'_t} \leftarrow computeReward(D_{s'_t})$
7: do loop in Step 2 for metrics values $v_{m_{1t}}, v_{m_{2t}}, .., v_{m_{kt}}, .., v_{m_{nt}}$
8: $D_{s_t} \leftarrow complianceDegree(min\{r_{v_{m_{1t}}}, r_{v_{m_{2t}}}, .., r_{v_{m_{kt}}}, .., r_{v_{m_{nt}}}\})$
9: $r_{(s_t \rightarrow s'_t)} \leftarrow compareStates(D_{s_t}, D_{s'_t})$
10: $r_t \leftarrow r_{s'_t} + r_{(s_t \rightarrow s'_t)}$
11: $assign\ r_t\ to\ couple\ (s_t, a_t)$
12: **Output:** r_t

4.4 Online Multi-Step Reinforcement Learning Based Decision Making

To solve the MDP problem described in the previous section, we introduce the decision learning algorithm that conducts the Analysis component behavior in learning dynamically the optimal decision policy. Prior to introducing this algorithm, we need to present the different aspects related to our approach and important to enhancing the Analysis behavior. Thus, we present the online RL approach (Section 4.4.1) and the multi-step approach (Section 4.4.2) that we adopt instead of the one-step approach. We also describe the challenges that are encountered in these approaches (Section 4.4.3) then end up introducing the learning algorithm that conducts the decision learning process of the Analysis.

4.4.1 Online Reinforcement Learning

There is a distinction between two classes of learning problems: offline and online [3]. In an offline problem, the agent learner trains since the beginning over all the available situations of its system before its deployment. Basically, it is a matter of learning over an entire static data set. Thus, the agent has an explicit knowledge

model (i.e., full MDP concepts) of the system which implies the state and action spaces, the transition probability function and the reward function. That is to say, we have a previous knowledge about the system's dynamics and the rewards as well. Therefore, given a couple (s_t, a_t), before executing the action a_t, an agent can describe the exact reward r_t that it will receive for the execution of a_t. The agent can also tell what state s'_t its system transits to and at what transition probability $P_{a_t}(s_t, s'_t)$.

In contrast, in an online problem, we do not have an explicit knowledge model of the system (i.e., model-free in our case). The knowledge is limited to what the agent can discover by interacting with its environment. An online MDP problem is solved by controlling the agent's behavior to act optimally and then learn an optimal decision policy. To do so, the agent learns continuously as the data comes in to update its behavior which renders this type of learning subject to changes over time.

Due to the limited availability of past data at deployment time, we do not have an exhaustive knowledge about the context dynamics. One can make use of their limited view of some few ECA rules, if available, which may help at the beginning of learning phase. However, this small knowledge is not sufficient to assume having a prior knowledge about the context dynamics. Consequently, the Analysis has to make decisions at execution time with no prior knowledge about its context dynamics. Thereby, we adopt an online learning approach for a dynamic learning of the optimal behavior.

In our MDP problem, we can only provide the Analysis with the list of the state and action spaces. It is up to the Analysis to discover the significance of the states (i.e., whether bad or good) and the effectiveness of its decision actions in certain states. Indeed, we have defined a reward function, but the Analysis has not a previous knowledge about the reward values that it may receive by executing an action a_t in a state s_t nor the probability to transit to a future state s'_t.

4.4.2 Multi-step Reinforcement Learning

Reinforcement learning is based on iterated interactions of the Analysis with its context. An interaction is represented by the transition $t : (s_t \rightarrow s'_t)$ that occurs upon executing a_t and is valued with the reward r_t. We say that the couple (s_t, a_t) is visited by the Analysis. Based on this interaction, the decision policy of the Analysis is dynamically and gradually built and improved. In fact, most of the RL-based algorithms do not compute directly the decision policy. Rather they use an iterative approximation of what is called the Q value function. This function represents a weighted accumulation of the computed reward values for each visited couple. The Q value function is associated to each visited couple of the state and action space. It describes for a couple (s_t, a_t) the utility value of choosing the action a_t in the state s_t. We only compute the estimation of this function since we do not have the precise knowledge of the context dynamics at disposal. By computing the estimation of this utility and updating it (i.e., the estimation) at each visited couple, we compute and improve progressively its decision policy.

Thus, based on the transition $t : (s_t \rightarrow s'_t)$, the RL algorithm realizes an iterative update of the Q value function at the visited couple (s_t, a_t). This update resembles an incremental and dynamic estimation of the Q value function based on the received reward r_t [44]. Thus, after each transition, the Analysis would update its Q value function estimation only at the currently visited couple (s_t, a_t). This single update of the Q value function is known as one-step Reinforcement Learning [50].

However, the one-step methods present the lack of updating the Q value function estimation of the current

visited couple once per transition. Moreover, the algorithm needs to go through each combination (s, a) of the state and action spaces several times (supposed infinite time) before it actually learns a good quality policy. By doing so, the algorithm convergence remains very slow especially when the state and action spaces are large which expands the research on the space solutions. This represents a major obstacle to scale to large state and action spaces, which limits the use of these one-step methods in large and complex computing systems. In fact, the raised issues can potentially increase the time the system spend in SLA violation during the early stages of learning. The violations are badly reflected on the system's performance which is a major drawback in real-world deployed systems.

To tackle these issues, we propose to use a combination of one-step methods, known also as Temporal Difference methods, and Monte Carlo methods [44]. As previously described in Section 2.5.1, Monte Carlo methods are known for updating the value function estimation of a visited state s_t after performing a trajectory (i.e., an episode of T transitions) [25]. The update of the value function estimation of the visited state s_t occurs at the end of each observed trajectory $(s_t, s_{t+1}, .., s_T)$. The rewards $(r_t, r_{t+1}, .., r_{T-1})$ accumulated on the trajectory are used to update the value function of the state s_t. This update method for the state s_t shows the forward view of Monte Carlo methods on the next visited states of the T transitions trajectory. However, we need to wait till the end of the trajectory so that we can compute a single update of the value function of the state s_t.

The combination of one-step RL methods and Monte Carlo methods is interesting when we can perform the forward view in a backward fashion as previously described in Section 2.5.2. It is also interesting when we can keep the incremental and dynamic update of the Q value function after each transition instead of waiting for a whole trajectory to be performed. The backward view is when the Analysis looks back in the previous visited states to re-update (more to re-correct) their Q value function estimations. In fact, the Analysis learns gradually along the transitions and we consider that at each transition, it learns more than in the previous one. Therefore, at each performed transition, the Analysis can back-propagate what it learned (in transition t) back to the previous k transitions: $\{t-1, t-2, .., t-k\}$. More precisely, the received reward r_t at transition t is used not only to update the Q value function estimation of the visited couple (s_t, a_t) but also to update the previously visited couples of the k previous transitions. Consequently, the combination of both methods implies an update of the Q value function estimation both in incremental, dynamic and backward fashion, hence in multi-steps instead of one-step.

Our Analysis learns dynamically its decision policy based on a multi-step reinforcement learning approach. In a multi-step approach, we use the eligibility traces [50], also called eligible couples and represent the list of the previously and freshly visited couples that are eligible for Q value updates. We call them eligibility couples or traces, because their Q value functions are eligible for updates. The way these traces are determined is detailed in Section 4.4.4. This approach is better performing than the one-step approach because it helps accelerating the learning process. The multi-step algorithm learns efficiently to converge faster and then avoid spending a long time in SLA violations which enhances considerably the system's performance.

4.4.3 Exploitation vs. Exploration Dilemma

During the learning phase, the Analysis selects a decision action and then updates the Q value function estimation to learn about its decision. When selecting its decision actions, the Analysis is confronted to the need

to find a compromise between the exploitation and the exploration of its state and action spaces. The Analysis exploits when it follows the best policy (called greedy policy), supposed optimal, it learned so far. Accordingly, it chooses an action that has already used before and knows its effectiveness regarding the current state, called greedy action. Whereas, it explores when it chooses randomly an action that has never explored before. The exploration consists of going through a new combination of couple (s, a) in the search space (i.e., all the combinations of couples $\{(s, a)\}$ of the state and action spaces). The objective is to explore the entire search space to be certain that the policy it learned is actually the optimal one. The exploration is helpful to dynamically refine and enrich the learned policy. However, it is risky since it is performed at execution time and the chances the system encounters some performance degradation can be high.

We need to balance the exploration and exploitation processes in a way that preserves the system's performance. To do so, the Analysis selects its actions by following a $\epsilon - greedy$ policy to control the exploration degree ϵ where $0 < \epsilon < 1$. A greedy policy corresponds to the best policy leaned so far. The Analysis should alternate between the choice of greedy actions it learned so far and the choice of new and randomly selected actions that has never explored before. Thus, with the $\epsilon - greedy$ policy, the Analysis should select random actions with the probability ϵ and greedy actions with the probability $1 - \epsilon$. In case the probability ϵ is close to 0, the Analysis will barely explore new couples which slows down its convergence speed and potentially get it trapped in a sub-optimal policy. In case ϵ is close to 1, the Analysis will explore faster its search space but at the cost of performance degradation of the system. Thus, it is a balance to take into consideration to choose the right value of ϵ. In our approach, the Analysis picks up a random action with a small probability (but not too small) ϵ and follows the best policy known so far for the rest of time $1 - \epsilon$.

4.4.4 Learning the Q Value Function Estimation

As previously described, learning dynamically a policy noted π is similar to learning the Q value function. Thus, learning dynamically an optimal policy π^* is similar to learning the optimal Q^* value function. More formally, at each transition t, the Analysis chooses an action $a_t = a$ at a state $s_t = s$. The Managed component state transits from $s_t = s$ to $s'_t = s'$ with the unknown transition probability $P_a(s, s') = P(s'_t = s'/s_t = s, a_t = a)$ and collects the immediate reward value r_t. P_a represents the knowledge (that we do not have) of the context dynamics where $\sum_{s' \in S} P(s'/s, a) = 1$. The value function of choosing action a in state s is computed as:

$$Q(s, a) = E\Big\{\sum_{l=0}^{\infty} \gamma^l r_{t+l} / s_t = s,\ a_t = a)\Big\}, \tag{4.3}$$

which represents the expectation value of the accumulated rewards along the transitions. The expectation value describes the distribution probability of the rewards that might be received along the transitions. This expectation enables the Bellman optimality principles [77] that allow solving MDP problems [78]. The discount factor γ denotes a performance criterion that helps with the convergence of the Q value function to its optimal value Q^*, where $0 < \gamma < 1$. More precisely, γ represents the unit value of a received reward at transition t which means that r_t is discounted by γ. Although, when this reward is received late at transition $t + l$, the same unit value will cost γ^l. In other words, a reward is more valued when it is collected sooner than later which induces the algorithm to converge faster. This parameter should be close to 1 to consider the accumulated rewards on the long term.

In our approach, we formulate an optimization problem in which the action a selected at state s should always maximize the Q value function $Q(s, a)$. Identifying the optimal policy π^* is similar to deriving an estimation of Q^* for all the couples which approximates its real value. Note that the real value of $Q(s, a)$ is computed when the context dynamics are known which is not our case. For each visited couple (s_t, a_t), we estimate its Q value function along with the Q value functions of the eligible couples by the end of each transition. We use an enhanced multi-step version Q(λ) of the Q-learning [44] to compute the estimation of Q value function with the following equations:

$$Q_{t+1}(s, a) \leftarrow Q_t(s, a) + \alpha_t(s, a) z_t(s, a) \delta_t(s_t, a_t), \tag{4.4}$$

$$\delta_t(s_t, a_t) = r_t + \gamma \max_b Q_t(s'_t, b) - Q_t(s_t, a_t) \tag{4.5}$$

Note that the visited couple (s_t, a_t) belongs also to the eligible couples. For a given couple (s, a), α_t denotes the learning rate value at the transition t and is $0 < \alpha_t < 1$. α_t indicates to what extent the recently acquired information during transition t override the old ones. A learning rate that is too close to 0 will make the Analysis learn nothing. It will exclusively make the Analysis exploit its prior knowledge. When the learning rate is close to 1, the Analysis will consider mostly the recently acquired information and ignoring the prior knowledge which may potentially lead to getting trapped in a sub-optimal policy. The learning rate should decay asymptotically along the transitions to indicate that the Analysis has learned enough on a couple (s, a). The learning rate is related to each couple (s, a) and decreases at each time a couple (s, a) is visited to asymptotically tend toward 0 to ensure convergence.

The temporal difference error $\delta_t(s_t, a_t)$ [27] computes the error that is used to correct the Q value function estimation after visiting the couple (s_t, a_t). Actually, it indicates the difference between the current estimation $Q_t(s_t, a_t)$ and the corrected estimation $r_t + \gamma \max_b Q_t(s'_t, b)$ computed after receiving the reward r_t. The expression $\max_b Q_t(s'_t, b)$ identifies the maximum Q value function for the state s'_t which corresponds to some currently known optimal action b so far. It also emphasizes the goal of finding the optimal action that maximizes the expected cumulative rewards (i.e., estimation of the Q value function). This explains the necessity to make a compromise between exploration and exploitation, since the Q value function updates are performed in terms of the optimal actions known so far (i.e., greedy actions during exploitation). The temporal difference error computations are refined during the transitions until they asymptotically decay to converge to 0. Hence representing a sign of approaching the best possible actions $\{a^*\}$ that represent the optimal policy π^*.

When $\delta_t(s_t, a_t)$ is computed at transition t, it is used to update the Q value function estimation of the couples $\{(s, a)\}$ that are eligible for updates. For a given couple (s, a), z_t denotes its eligibility to Q value function update (i.e., back-propagation). The couples eligible for updates are the latest visited couples $\{(s, a)\}$ by the Analysis. Consequently, we propose to endow the Analysis with a memory dedicated to the propagation of updates, namely transition memory $M_{transitions}$. Indeed, at each transition the Analysis perceives a reward, it computes $\delta_t(s_t, a_t)$ then propagates it back to the previous visited couples in the memory to perform several updates on their Q value functions. It is necessary to maintain a trace in memory of the M previously realized transitions.

When the error $\delta_t(s_t, a_t)$ is computed, the Equation 4.4 shows that this error is more or less considered according to how long ago a couple (s, a) has been visited. Freshly visited couples are maintained in memory in order to correct their Q value function estimations with the current computed error $\delta_t(s_t, a_t)$. We note the eligibility traces $(S, A)_{M_{transitions}}$, the set of backed-up couples $\{(s, a)\}$ in the memory and are eligible for updates. As soon as the transition t: $(s_t \rightarrow s'_t)$ is performed and the error δ_t is computed. The eligibility trace is computed by its coefficient trace $z_t(s, a)$ as follows:

$$\begin{aligned} z_0(s, a) &= 0,\ \forall (s, a) \in (S, A) \\ z_t(s, a) &= Id_{ss_t}.Id_{aa_t} + \gamma\lambda z_{t-1}(s, a) \\ Id_{ss_t}.Id_{aa_t} &= \begin{cases} 1, & \text{if } (s, a) = (s_t, a_t) \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{4.6}$$

Id_{ss_t} and Id_{aa_t} are identity indicators functions used to increment a recently visited couple (s_t, a_t). The trace of the current visited couple (s_t, a_t) is incremented to indicate its recentness. The traces of the other couples different from the visited couple are decayed by $\gamma\lambda$ to indicate that their freshness has reduced for a future transition $t+1$. λ is an eligibility factor that helps determining the number of the eligible traces to keep in memory and is $0 < \lambda < 1$.

We initialize the values of all the coefficient traces $z_t(s, a)$ that verify the condition $z_t(s, a) \leq \epsilon_z$ to 0, where $0 < \epsilon_z < 1$ and ϵ_z is close to 0. The coefficients verifying this condition are not considered freshly visited and thus their Q value functions are not eligible for updates. Recently visited states with the following transition memory are maintained, beyond this value, traces are initialized and corresponding couples are removed from the memory. We determine the transition memory with the following equation:

$$M_{transitions} = \frac{log(\epsilon_z)}{log(\gamma\lambda)} \tag{4.7}$$

4.4.5 Policy Initialization

As previously described, one of the recurring issues of online learning is the poor performance induced during the early phases of learning. Indeed, the Analysis has not acquired enough experience on its context dynamics yet, it may accumulate the selection of random and potentially bad actions. However, in our approach, we do not assume a prior knowledge of the context dynamics at deployment time. Instead, we can provide, if available, an ECA (Event Condition Action) that can constitute an initial small knowledge about few couples $\{(s, a)\}_{ECA}$ on what action to execute in which state. This initial knowledge may be exploited to initialize the Q value functions of these few ECA couples. This approach appears to hold the potential of reducing, occasionally, the time the system might poorly perform during the first learning stages. Otherwise, the Q value function of all the state and action spaces are initialized to 0 prior to starting the learning phase.

4.4.6 Decision Learning Process of the Analysis component

In the previous sections, we presented an approach of enhanced reinforcement learning technique. Based on these sections, we describe the learning and decision making algorithm that determine the Analysis behavior.

Algorithm 2 depicts the Analysis behavior during the learning phase and after convergence. Prior to starting the learning phase, this algorithm takes the following RL parameters in input: the initial learning rate α_0, the discount factor γ, the exploration degree ϵ, the eligibility traces parameters λ and ϵ_z to compute the transition memory $M_{transitions}$ and the convergence criterion $\xi \rightarrow 0$ (line 1). In case an ECA is available, the Analysis uses it to initialize the Q value function (line 2). The Analysis matches the ECA couples $\{(s,a)\}_{ECA}$ to existing couples $\{(s,a)\}$ from the state and action spaces. Then, it assigns a positive value to their Q value functions. The Q value functions of the rest of couples are initialized to 0 to indicate that there is no prior knowledge about them. Afterwards, for all the couples $\{(s,a)\}$, the Analysis initializes their coefficient traces to 0 and their learning rates to the initial value α_0 (lines 3 and 4 respectively).

Algorithm 2 : Decision Learning Process

1: **Input:** $\alpha_0, \gamma, \epsilon, \lambda, \epsilon_z, \xi, ECA$
2: $Initialize(Q_0) \leftarrow Parse(ECA)$
3: $z_0(s,a) \leftarrow 0, \forall (s,a)$
4: $\alpha_0(s,a) \leftarrow \alpha_0, \forall (s,a)$
5: $t \leftarrow 0$
6: **Repeat:**
7: $s_t \leftarrow notifyCurrentState()$
8: $a_t \leftarrow selectAction(s_t)$
9: $executeAction(a_t)$
10: $s'_t \leftarrow notifyCurrentState()$
11: $r_t \leftarrow computeReward(s_t, s'_t)$
12: $\delta_t(s_t,a_t) \leftarrow r_t + \gamma \ \max_b Q_t(s'_t, b) - Q_t(s_t,a_t)$
13: $z_t(s_t,a_t) \leftarrow z_t(s_t,a_t) + 1$
14: Update $Q_t(s,a)$ and $z_t(s,a)$:
15: $(S,A)_{M_{transitions}} \leftarrow (S,A)_{M_{transitions}} \cup \{(s_t,a_t)\} \setminus \{\text{argmin}_{(s,a)}\ z_{(s,a)}\}$
16: **for all** $(s,a) \in (S,A)_{M_{transitions}}$ **do**
17: $\quad Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha_t(s,a) z_t(s,a) \delta_t(s_t,a_t)$
18: $\quad z_{t+1}(s,a) \leftarrow \gamma \lambda z_t(s,a)$
19: **end for**
20: $\alpha_{t+1}(s_t,a_t) \leftarrow decay(\alpha_t(s_t,a_t))$
21: $s_t \leftarrow s'_t$
22: $t \leftarrow t+1$
23: **Until:** All $\alpha\delta \leq \xi$
24: $\pi^* \leftarrow retrieveOptimalPolicy(\{Q^*(s,a)\})$
25: **Output:** π^*

At each transition t, the Analysis is notified (by the Monitoring component) about the monitored state s_t of the Managed component (line 7). Subsequently, the Analysis selects either an exploratory or a greedy action (line 8). The function $selectAction(s_t)$ uniformly picks a random action with a small probability ϵ and follows the best policy known so far for the rest of time $1-\epsilon$. The Analysis invokes the Management

services of the Execution component to execute the selected action a_t (line 9). Among the existing actions, we have the "empty" action that has no effect on the system. It is useful when for instance, the state satisfies or largely satisfies the SLOs, thus we do not need to execute an action with an actual effect. Once the system's state transits to a new state s'_t, the Analysis is notified (line 10). The Analysis can now compute the reward r_t based on the previous and the current states s_t and s'_t, respectively (line 11). It can also compute the temporal difference error δ_t of the visited couple (s_t, a_t) (line 12).

Once the error is computed, the coefficient of the couple (s_t, a_t) is incremented to indicate its recentness for the current transition t (line 13). The recently visited couple is added to the transition memory $M_{transitions}$ while the couple with the oldest trace of minimum value is removed (line 15). Likewise, oldest traces with traces $z_t(s, a) \leq \epsilon_z$ are initialized to 0 and also removed from the memory. Afterwards, the Q value functions of eligible couples are updated (line 17). Their traces are then decayed by $\gamma\lambda$, to indicate that their freshness has reduced for the next transition $t + 1$ (line 18). The learning rate α_t is decayed for the couple (s_t, a_t) to be used for a future visit of the same couple (s_t, a_t) (line 20). We decay the learning rate with the equation:

$$\alpha_{t+1} = \frac{n_{(s,a)}}{1 + n_{(s,a)}}\alpha_t \tag{4.8}$$

where $n_{(s,a)}$ is the number of visits of the couple (s, a). This equation ensures an asymptotic decrease of the learning rate for each visited couple to indicate that the Analysis has learned enough on them.

The Algorithm 2 reaches convergence when the correction $\alpha\delta$ brought to Q value function $Q(s, a)$ for all couples reaches some convergence criterion $\xi \rightarrow 0$ (line 23). After convergence, the Analysis retrieves the optimal policy π^* from all the optimal Q^* value functions of the couples (line 24) and returns it (line 25).

4.4.7 Convergence and Optimal Policy Retrieval

Theoretically, all convergence proofs generally require that the algorithm experiments all the transitions at each couple of the space [79]. Practically, a partial exploration of the search space may be sufficient to discover a satisfactory policy. In our case, we explore all the search space to find the optimal solution that is the optimal policy. Algorithm 2 needs to go through each combination (s, a) of the state and action spaces several times before it actually converges. To the best of our knowledge, there are convergence proofs established for the one-step version of Q-learning but not for its multi-step version Q(λ). In our case, we consider that the convergence happens when Q converges to Q^* which occurs under the following assumptions:

- The spaces S and A are finite,
- Each couple (s, a) is supposed visited an infinite number of time,
- $\sum_t \alpha_t(s, a) = \infty$ and $\sum_t \alpha_t^2(s, a) < \infty$,
- $\alpha\delta$ decays asymptotically to attain the convergence criterion ξ,
- The discount factor $\gamma \leq 1$

After convergence, the optimal policy π^* is computed according to:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s, a) \tag{4.9}$$

Thus, for each state s of the state space, the Analysis extracts the action a^* with the maximum Q^* value function. This action represents the optimal action to execute when the Analysis encounters the state s. The Analysis extracts then its optimal policy by associating each state s of the state space to the corresponding optimal action a^* from the action space.

4.5 Conclusion

Our main contribution in this chapter is to improve the adaptation logic of a classical MAPE-K loop. Instead of using static hand-coded logic, we equipped this loop with learning-based adaptation logic. Accordingly, we proposed to enhance the Analysis component with sophisticated Reinforcement Learning abilities for a dynamic adaptation of component-based applications. To do so, we modeled the decision making process of the Analysis component as a Markov Decision Problem. Then, we solved this problem by defining the MDP concepts related to our approach as well as the decision learning algorithm that conducts the Analysis behavior. The decision learning algorithm that we presented was guided by the reward function to learn from its past experiences and then be able to compute the optimal policy at execution time. We equipped the Analysis component with a multi-step learning approach that outperforms the one-step learning approaches. Our learning algorithm converges faster compared to existing learning approaches. It also reduces the time the system might spend in SLA violations as shown in Section 6.4.3.3.

Our research objectives are mainly focused on improving traditional ACS with sophisticated behaviors to be less dependent on human interventions. The purpose is to alleviate the burden of management tasks on application developers by dynamically rendering their applications self-adaptive. Towards this objective, we extend our current work to propose a generic framework for a dynamic building of self-adaptive component-based applications. We provide developers with a framework equipped with functionalities that facilitate endowing their existing applications with self-adaptive behavior for their self-management.

Chapter 5

Framework for Building Self-Adaptive Component-based Applications

Contents

5.1 Introduction

In the previous chapter, we presented an approach for enhancing the adaptation logic of classical Autonomic Computing Systems (ACS). We proposed to use a multi-step reinforcement learning approach to dynamically compute an optimal decision policy at execution time. In this chapter we push further our work to support

application developers by providing a generic framework that allows adding non-functional management facilities such as self-adaptation for the applications.

As previously described, component-based applications are intrinsically in dynamic evolution due to their context dynamics. They evolve in dynamic deployment environments while exposing dynamic internal conditions during their execution life-cycle (e.g., their availability, load, performance, etc.). Such contexts have burdened the application developers both with their business design and manual management tasks, hence, not being able to fully focus on the business (i.e., functional) logic of their applications. Subsequently, the Autonomic Computing principles were adopted to reduce as much as possible human intervention by rendering the applications autonomous.

However, reducing human intervention is made at the cost of increased development labor. Actually, in addition to developing the business logic of their application components, the developers have to develop the non-functional components that provide the adaptation logic necessary to the autonomic management of their functional components. Furthermore, they should integrate the non-functional components to the functional ones by managing their interactions. These development efforts intended for the adaptation aspects are built from scratch which may potentially be error-prone. Moreover, each functional component has different and specific management requirements itself and so as its deployment environment. By considering these specific requirements, the developers may end-up building problem-specific solutions, neither reusable for other management concerns, nor endowed with learning capabilities. Thereby, the non-functional aspects are undoubtedly time and effort-consuming in addition to demanding a costly expertise.

Therefore, dynamic and generic tools should be offered to developers to alleviate the burden of building non-functional aspects. Our objective is to provide them with the necessary support to facilitate the integration of non-functional aspects into their existing applications at minimum cost. Consequently, letting them focus more on the business logic of their applications. The adaptation logic should be provided in separation from the business logic to add more flexibility and reusability to the proposed solution.

In this chapter, we propose a generic framework that supports self-adaptation of applications based on generic functionalities and on a multi-step reinforcement learning algorithm. In addition, the framework design is based on the Service Component Architecture (SCA) [19] standard known for being implementation and technology-agnostic. This represents a key advantage by integrating the framework with existing applications regardless of their implementation or technology. It is worth noting that the application is not built from scratch, rather it is transformed by encapsulating its components in autonomic containers to render them self-adaptive. The business functionalities of the encapsulated composite/component are forwarded by the autonomic container in addition to be equipped with autonomic non-functional abilities. Depending on their needs, the developers can transform their application components on different granularities levels (i.e., fine-grained and coarse-grained transformations). The framework is designed to be extensible to integrate the specific needs of the applications and their deployment environments as well. The generic aspects of self-adaptation of our framework are designed to be separated from the problem-specific parts which constitutes a reusable solution for recurring management problems.

The remainder of this chapter is structured as follows. In Section 5.2, we give an overview of our approach. In Section 5.3, we introduce the internal structure of our autonomic container by providing a detailed description of its internal components. We describe the functionalities that are offered by the internal components and their maintained interactions. Afterwards, we provide an illustrative transformation example of

an application in order to observe the realized changes induced during the transformation. In Section 5.4, we demonstrate the usage of our framework with an illustrative case study of a component-based application in the Cloud. Accordingly, we define the different aspects that are required to the usage of the framework. Thereby, we identify the elements to be provided, extended or implemented to enable specifying the needed autonomic behavior of the application. We end up this section with an illustration of the performed transformation. Finally, we conclude this chapter by outlining the key aspects of our contributions and introducing for the upcoming chapter.

5.2 Approach Overview

In our work, we are proposing a generic framework that offers a set of generic facilities/functionalities to support non-functional management aspects of applications. We think that these aspects should not be tied to the business logic of the application. Developers should focus on the business aspects of their applications and still be able to reuse and plug-in the needed non-functional facilities. Thus, we advocate a separation of concerns [80] during the design of the framework and the transformation of the application.

Our work is mainly focused on providing the aspects related to the adaptation logic to application developers in order to support them in building self-adaptive applications. To do so, we developed the framework as a set of generic entities to facilitate its reuse for different management concerns such as some targeted self-* property (e.g., self-optimization, self-reconfiguration, etc.). The framework is provided with an initial self-adaptive behavior with some basic Management services and allows extending and overriding this behavior depending on the management needs. Its abstract design enables its reuse for different management concerns and intends for integration with different deployment environments. The framework enables also its integration with different applications regardless of their implementations. Actually, we designed the framework by using the Service Component Architecture (SCA) [19] programming model. This standard enables the integration of different and existing applications by being independent of any implementation, technology or protocol.

The framework provides means to equip existing applications with a self-adaptive behavior. To do so, the framework is developed in the form of an autonomic container that provides the self-adaptive behavior to the transformed components. The transformation occurs at deployment time and guaranties that the business logic of the application remains unchanged and separated from the added adaptation logic. The transformation entails the encapsulation of the application components in the autonomic container whenever they need autonomic management. The transformation process involves rendering the components monitorable and adaptable during their execution life-cycle. They are then enabled with the self-adaptive behavior and henceforward called *Managed components*. A Managed component is either atomic, composite or the overall application composite which allows the autonomic management on different granularity levels.

By encapsulating the Managed component, the autonomic container is able to autonomously manage its life-cycle, its functional dependencies, its adaptable properties, mapping it to other locations, etc, depending on the management needs. Since the Managed components of an application maintain functional interactions between them, the autonomic container promotes the Managed component's provided and required services as its functional services. It also provides non-functional services intended to self-adapt the Managed component

both to its internal (i.e., itself) and external (i.e, environment) dynamics.

5.3 Structure of the Autonomic Container

We designed our generic framework in the form of an autonomic container that behaves autonomously and encapsulates the application components whenever they need autonomic management. As shown in Figure 5.1, the autonomic container provides and requires the same functional services of its Managed component. It also provides a non-functional service of Notify to be notified by the external changes of its deployment environment.

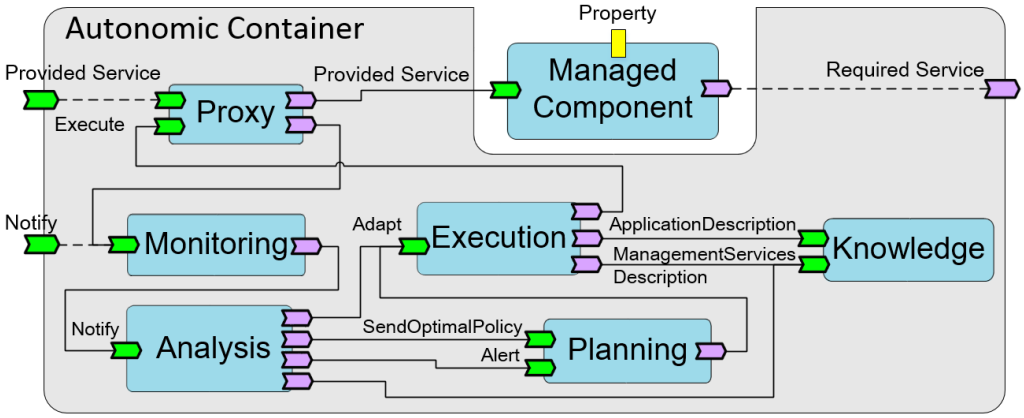


Figure 5.1: Architecture of the autonomic container

The container consists of an assembly of components allowing the functionalities of Proxy, Monitoring, Analysis, Planning, Execution and Knowledge. We separate these functionalities among different components to allow an isolated replacement of them which ensures more flexibility and reusability to the container. In the following we explain further the functionalities of the involved internal components and their interactions.

5.3.1 Proxy component

To add self-adaptive abilities to components, we need to render them monitorable and adaptable by transforming the application's components. The transformation entails the generation of the Proxy component which involves adding dynamically getter and setter operations. Getter operations allow the extraction of monitoring data from the Managed component. For instance, getting the current value of an adaptable property. Setter operations enable the execution of some actions on the Managed component. For instance, setting some value of an adaptable property. The setter and getter operations are illustrated in Figure 5.1 by the Execute service that is used by the Execution component to carry out actions and may be used to extract monitoring data. The generation of the Proxy component entails also providing the same functional services initially provided by the Managed component.

After its generation, the Proxy component acts like an intermediary component between the Managed component and the remote components that consume the services provided by the Managed component. Providing the same functional services initially provided by the Managed component allows the Proxy to intercept the service calls (i.e., service requests) that are intended for the Managed component. Afterwards, the Proxy extracts the monitoring data from the service call then forwards it (i.e., the call) to the Managed component. Once the Managed component delivers the response to the service call, the Proxy extracts the monitoring data for the second time. This allows the Proxy to check for any changes that happen in between the previous service call and the current on. It also checks for any changes during the current service call.

The extracted monitoring data concern mainly, the service call (e.g., computing the number of service calls), the service execution time and any noticed changes of the adaptable properties values. The monitoring data are sent to the Monitoring component through its Notify service.

5.3.2 Monitoring component

The Monitoring component is responsible for processing monitoring data to extract the information that are relevant to the autonomic management. The data are aggregated to compute the attributes that meet the management needs. Depending on the needs, the Monitoring component allows two types of monitoring, by polling and by subscription as shown in Figure 5.2. Monitoring by polling allows requesting the current value of a monitorable attribute such as a monitorable property. It is made by calling the getter method of this attribute by the Monitoring component as provided by the Polling entity.

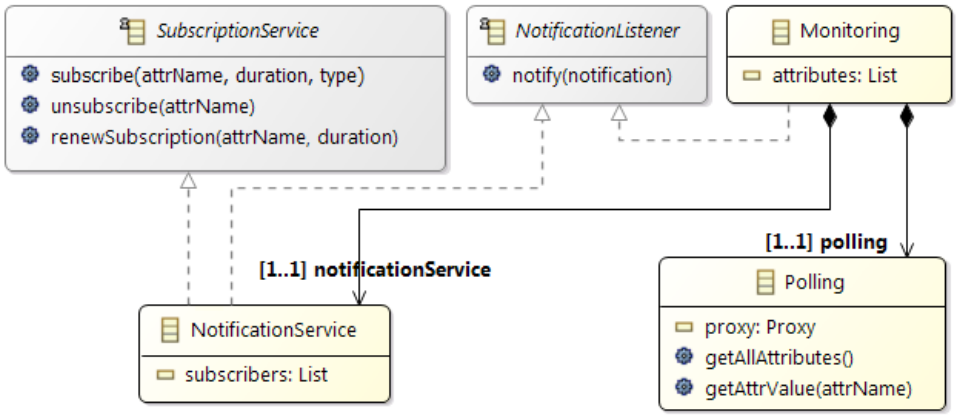


Figure 5.2: Monitoring diagram: overview on the main entities

Monitoring by subscription of an attribute can be performed by subscribing to the Notification service that is provided by the Monitoring component. The NotificationService entity is responsible for the management of clients (i.e., subscribers) subscriptions. The NotificationService entity maintains the list of subscribers to specific notifications. As soon as it receives, from the Proxy component, monitoring data that correspond to a subscription, the NotificationService produces the notification that matches the subscription and sends it to the subscriber component. The subscribers have to be notification listeners to receive notifications. They

should implement the NotificationListener entity (i.e., interface) to define the specific behavior adopted when receiving notifications.

The Notification service provides also a Subscription service that is represented by the SubscriptionService entity (i.e., interface). Accordingly, the subscribers can subscribe, unsubscribe or renew their subscriptions for notifications on some attributes. They can also define (or not) the subscription duration and indicate the subscription type. The type of subscription defines the monitoring mode that is either on interval or on change. In case the mode is on interval, the Notification service has to notify its subscribers at each time interval whether the attribute value has changed or not. Otherwise, the on change mode enables the Notification Service to notify the subscribers whenever a change occurs in the values of the attributes that match their subscriptions.

By default, the Analysis component appears in the subscribers list in order to be notified about the attributes that compose the Managed component's state.

5.3.3 Analysis component

Classically, the Analysis component determines adaptation context based on a hand-coded ECA (i.e., Event, Condition, Action) rules. It gauges the current situation (i.e., event) by comparing it against the rules it holds. When an adaptation context is confirmed, the Analysis generates an alert to the Planning component that selects the action corresponding to the ECA rules for execution.

As previously explained, the Analysis component is in charge of the adaptation logic. It is responsible for the decision and learning processes of the autonomic container. The Analysis makes decisions then reasons about them during the learning phase in order to make better decisions in the future. Recall that the decision making process of the Analysis component is designed as a Markov Decision Process (MDP). Accordingly, its design is based on the different concepts related to MDP and the adequate algorithm to solve it (i.e., MDP problem) efficiently (refer to Chapter 4). Therefore, Figure 5.3 shows the Analysis component that is mainly composed of the following entities: StateSpace, SLA, Metric, ActionSpace, DecisionModule, RLParameters, RewardFunction, RewardComponent, QvalueFunction and the ECAPolicy.

The Analysis component maintains a list of Reinforcement Learning (RL) algorithms (via its DecisionModule entity) that might be run during the learning phase. Accordingly, the Analysis provides the possibility to initialize the learning behavior with the RL algorithm needed to be run and its related RL parameters (refer to Section 4.4.6). As shown in Figure 5.3, the Analysis component needs the previous and current states of the Managed component in order to learn about the efficiency of the executed action. As previously introduced, a state represents the list of the observed metrics values that are relevant to the description of the Managed component's state. An action represents the adaptation decision that was executed in the previous state. The Analysis component provides a Notify service to receive notifications from the Monitoring component about the observed attributes values that match the Metrics targeted in the SLA (Service Level Agreement).

As shown in the same Figure, the StateSpace entity generates and maintains the list of the states that are encountered by the system. The StateSpace holds the SLA entity to recognize the metrics needed to generate the states and extract their SLOs (Service Level Objectives). The Metric entity provides means to generate all its possible values that may be encountered. It generates all the possible values by sampling them according to some step interval value that the framework user indicates. It also determines whether an observed metric value

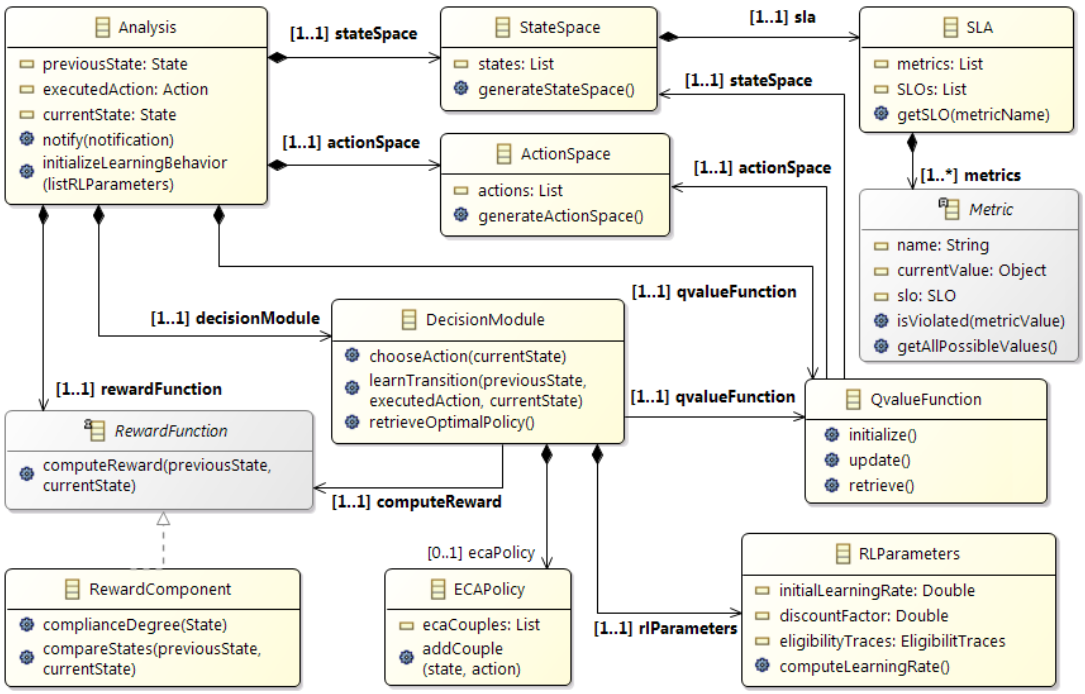


Figure 5.3: Analysis diagram: overview on the main entities

violates the SLO which allows identifying whether a state violates the SLOs. The Metric is an abstract entity that can be extended to define other Metrics types. The Analysis component initially provides a set of abstract Metric types: RealMetric, to process real values; BooleanMetric, to process boolean values; DiscreteMetric, to process discrete values.

The ActionSpace entity generates and maintains the list of the adaptation actions needed for the governance of the Managed component. To do so, the Analysis component consults the Knowledge component to retrieve the descriptions about the available Management services provided by the Execution component. An action represents a single, a composition or an orchestration of Management services to self-adapt the Managed component.

As depicted in Figure 5.3, the Analysis is also composed of the DecisionModule entity that is responsible for the decision and learning behaviors. Accordingly, the DecisionModule provides the list of implemented RL algorithms used to learn optimal decision policies. An optimal decision policy is a function that, for a given state of the Managed component, returns the best possible action for execution. Prior to learning the optimal policy, the RL algorithm should be selected and parameterized to be run by the DecisionModule during the learning phase. The parameters are handled by the RLParameters entity such as the initial learning rate α_0 and the discount factor γ that should be initialized by the framework user. Refer back to the full list of parameters in Section 4.4. The RLParameters containers also the EligibilityTraces component that maintains the eligibility traces. Once activated, EligibilityTraces component enhances the learning performance of the

DecisionModule to learn an optimal policy faster. The eligibility traces are thoroughly explained in Section 4.4.4.

The RewardFunction entity (i.e., interface) allows implementing the reward function to compute the reward values used to guide the DecisionModule in its decisions. The reward is computed by implementing the function that gauges the enhancement or degradation of the previous and current states of the Managed component. The Analysis is initially provided with a RewardComponent entity that computes the compliance degree of the states to the SLOs and then compares both previous and current states. The RewardComponent entity computes the reward values according to Algorithm 1 described in Section 4.3.3.

The QvalueFunction entity is used to maintain Q value function computations (refer to Section 4.4.4) for all the state and action couples of the space. This function represents the long sum of the accumulated rewards computed during the learning phase. Accordingly, this entity provides means to initialize and update the Q value function computations. Whenever the chosen RL algorithm reaches convergence, the QvalueFunction allows retrieving the Q value functions to compute the optimal decision policy.

Prior to the learning phase, the Analysis component may optionally contain the ECAPolicy entity. This entity represents an initial knowledge about some adaptation scenarios. In case available, this entity contains the ECA couples states and actions $\{(s, a)\}_{ECA}$. It may represent an initialization of the QvalueFunction entity with these couples to occasionally avoid bad performance and speed up the learning convergence.

During the learning phase, the DecisionModule chooses a decision action for the current state. Whenever the Managed component transits to a new state, the DecisionModule learns about the transition and the executed action by using the reward computations of the RewardFunction entity. Whenever the chosen algorithm reaches convergence, the DecisionModule retrieves the optimal decision policy and then the Analysis component sends it to the Planning component.

5.3.4 Planning component

After convergence of the selected RL algorithm, the Planning component receives the computed optimal policy from the Analysis component. The Planning component will then take in charge the action selection process according to the received optimal policy. Accordingly, as shown in Figure 5.4, the Planning component provides a service of sendOptimalPolicy to the Analysis component in order to receive the optimal policy. It also provides means to update this optimal policy in case the Analysis component restarts its learning process.

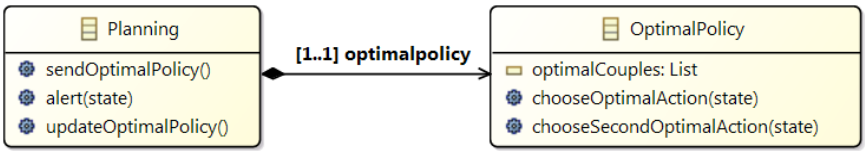


Figure 5.4: Planning diagram: overview on the main entities

The Planning component maintains the optimal policy in its OptimalPolicy entity that holds the list of the optimal state and action couples. For a given state, alerted by the Analysis component, the OptimalPolicy entity chooses the corresponding optimal decision action for execution by the Execution component. The OptimalPolicy entity maintains also second optimal decision actions that can send to the Execution component

for execution if needed. To carry out the actions, the Planning component requires the Adapt service of the Execution component as shown in Figure 5.1 which allows executing these actions.

5.3.5 Execution component

This component is responsible for the overall self-adaptive behaviors adopted by the autonomic container. The Execution component holds a list of Management services that define these behaviors. A Management service is a non-functional service providing a set of operations (i.e., methods) that constitute the adaptation actions. The Execution component is then responsible for executing the chosen adaptation actions depicted in the Execution entity in Figure 5.5. An action is executed by calling (i.e., invoking) either elementary, composition or orchestration of Management services. To carry out the actions, the Execution component may need to consult: the provided and required services, the adaptable properties and the reference of the Managed component, etc, held in the Knowledge component.

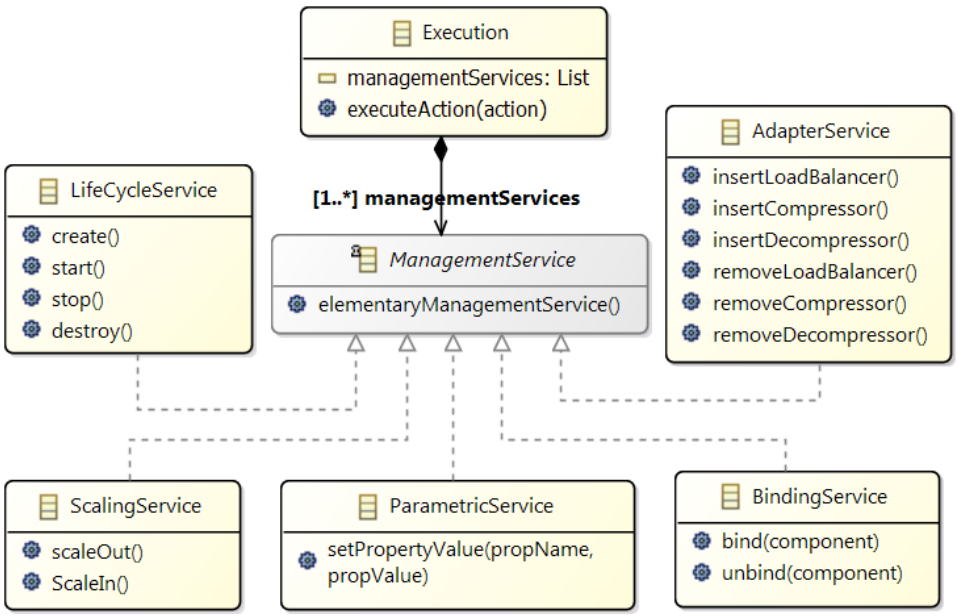


Figure 5.5: Execution diagram: overview on the main entities

As shown in Figure 5.5, the Execution component maintains the list of the ManagementServices entities that define the desired adaptation behavior. Since the application management requirements and deployment context may change, the framework allows the extension of this list by implementing the ManagementService entity (i.e., interface) with the specific adaptation behavior. The ManagementService entity allows implementing at least an elementary Management service. Depending on their management needs, the framework user can implement additional elementary, composition or orchestration Management services. The Execution component is initially provided with some basic and generic Management services entities as depicted in this same figure.

- LifeCycle service: enables creating, starting, stopping, restarting and destroying the Managed component,
- Scaling service: allows scaling-out by inserting new instances of the Managed component. It also allows scaling-in by removing extra instances of the Managed component,
- Parametric service: adjusts the adaptable properties values of the Managed component,
- Binding service: handles the functional dependencies of the Managed component by binding or unbinding it with other components,
- Adapter service: inserts and removes Adapter components such as a load balancer, a compressor, a decompressor, etc.

5.3.6 Knowledge component

The internal components of the autonomic container require a set of information necessary to perform their functionalities. The Knowledge is a key component that provides these information. As illustrated in Figure 5.6, the Knowledge contains the Managed component's reference that is used by the Execution component in order to carry out the adaptation actions. Each time an adaptation occurs, the Knowledge component provides the possibility to establish a historical memory about the modifications brought to the Managed component. Accordingly, the Knowledge enables its update whenever the information held about the Managed component changes. For instance, the changes may involve the deployment environment (e.g., changing the hosting environment) of the Managed component or the values of its monitorable and adaptable properties, etc. As depicted in the same Figure, the Knowledge component is mainly composed of the following entities: ApplicationDescription, Properties, SLADescription, ManagementServicesDescription, Environment and ECA.

The ApplicationDescription entity holds a file that contains a full description of the application. This entity parses this file to retrieve the list of the provided and required services of the Managed component to be aware of its bindings. It also extracts the list of the components that are embedded in the Managed component (in case it is a composite) and its used implementation. The Properties entity maintains the list of the Managed component's properties and all their possible values if they are monitorable and adaptable. These values are consulted by the Analysis component during the generation of the adaptation actions related to the Parametric service (i.e., adapt properties values) and provided by the Execution component. Accordingly, the Properties entity holds a table that contains all the possible values for all the monitorable and adaptable properties.

The SLADescription entity maintains the SLA file of the application. It parses this file in order to extract the list of the targeted metrics and their SLOs. The parsed information are used by the Analysis component for the creation of its SLA entity (refer to Figure 5.3) and then the generation of the state space as described in Section 5.3.3. The ManagementServicesDescription entity contains a description file about the existing Management services. The description entails information about the Management services components, their provided and required services, their implementations, etc. These information are consulted by the Analysis component along with the application description to enable the generation of its action space. The described Management services are both the ones initially offered by the Execution component and the ones that can be extended by the framework user.

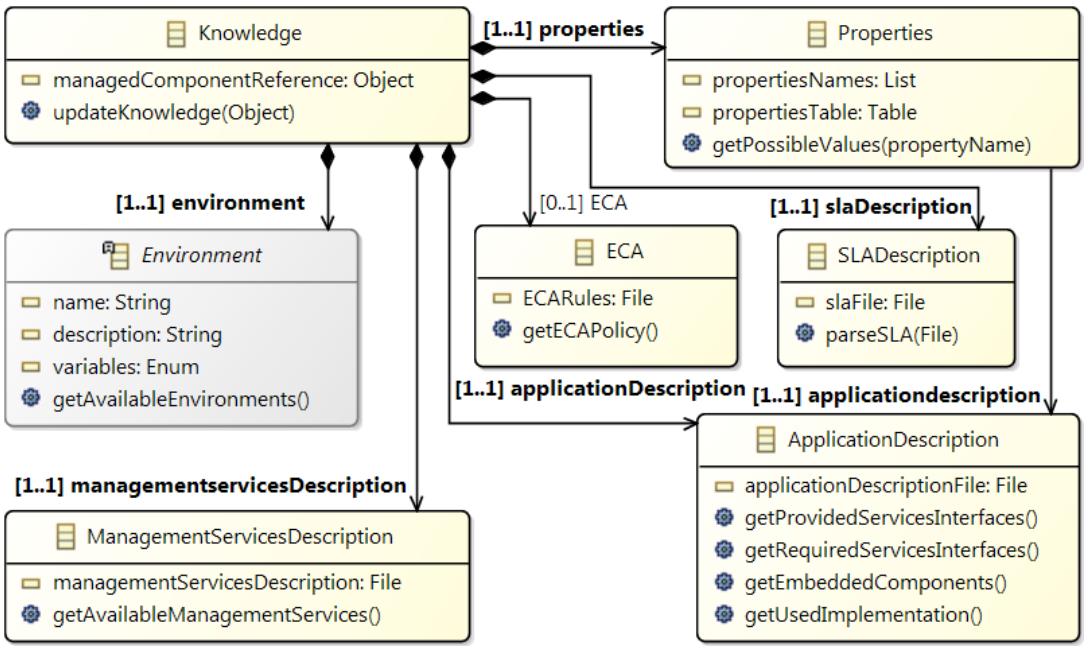


Figure 5.6: Knowledge diagram: overview on the main entities

The Knowledge contains also the abstract entity Environment that provides information about the deployment environment of the Managed component. This entity can be extended to encompass the specific hosting environment characteristics. It also allows checking for other available environments for this Managed component, if any, in case the user needs to map its Managed component to other location environments. For instance, in a Cloud environment, VM (virtual machine) environments can be defined. The user can then identify several VM instances and assign them as available environments that the Managed component can be migrated to.

In case the user provides an ECA table to the framework, the Knowledge may include an ECA entity that defines the ECA rules in file. The rules are parsed to get the ECA policy embedded in these rules. This entity, if available, can be used by the Analysis component to constitute its ECAPolicy entity and initialize the QvalueFunction entity depicted in Figure 5.3.

5.4 Framework Usage through a Case Study

We developed the framework with an abstract design to facilitate its reuse for different application management concerns. To be able to utilize the framework for their own specific needs, the users have to provide a list of requirements to the framework to endow their applications with the desired self-adaptive behavior. In this section, we demonstrate the usage of our framework through a case study of a component-based application that we implemented. Accordingly, we introduce the list of requirements that we provided to the framework.

We also describe the different extensions that we performed in order to customize the framework behavior to meet our management needs. After providing the requirements and performing the needed extensions, we illustrate the realized transformation of the application.

5.4.1 Case Study Description

To illustrate the framework usage, we implemented an application that allows users (i.e., clients) to consult products on the Internet. The application is intended to be deployed in the Cloud and is represented by the Products Composite depicted in Figure 5.7. This application allows web users to fetch and consult products through an online Catalog that brings products information from multiple inventories. Once called, the Consult service loads descriptions and pictures of the fetched products. To fulfill service execution, the Catalog component runs a search algorithm that is represented by its property SearchAlgorithm. The users have also the ability to order their favorite products as well. Whenever a user calls the Order service, logging to their account becomes mandatory in order to retrieve their personal information. The Products Composite invokes then remote components that are responsible for retrieving the user's information and carrying out the products order.

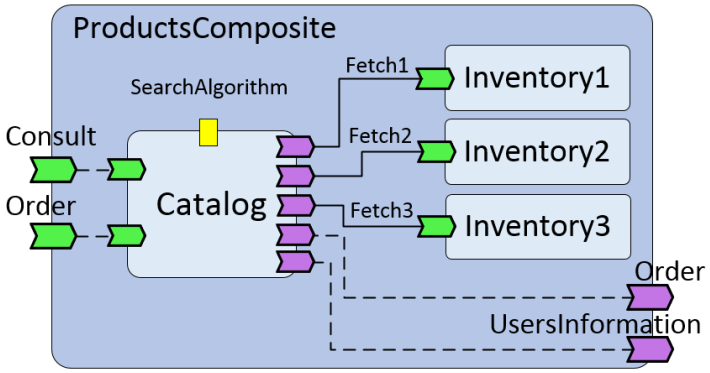


Figure 5.7: Component-based description of Products Composite

Application Description

To transform the described application and render it self-adaptive, we first start by defining its description file and provide it to the framework. The application description is a key element for the framework and it describes the application architecture. Therefore, we need to be able to recognize the application components structure. The purpose is to be able to transform them by rendering monitorable and adaptable their services and properties with the generation of the Proxy component and then its insertion along with the rest of the MAPE components.

Listing 5.1 depicts the description file of the Products Composite application. As depicted in this listing, the file contains a full description of the application's internal components, their properties, their provided and required services and their implementations. The description file is held by the ApplicationDescription entity

within the Knowledge component as previously described in Section 5.3.6. This file will be processed by our custom parser to extract the described information. We used the Architecture Description Language (ADL) of the Service Component Architecture (SCA) to write the description file.

Listing 5.1: Description file of the Products Composite application using SCA ADL

```
<composite name="ProductsComposite">
 <service name="Consult" promote="Catalog/Consult"/>
 <service name="Order" promote="Catalog/Order"/>
 <reference name="Order" promote="Catalog/Order"/>
 <reference name="UsersInformation" promote="Catalog/UsersInformation"/>

 <component name="Catalog">
  <property name="SearchAlgorithm"/>
  <service name="Consult">
   <interface.java interface="ConsultInterface"/>
  </service>
  <service name="Order">
   <interface.java interface="OrderInterface"/>
  </service>
  <reference name="Fetch1" target="Inventory1/Fetch1"/>
  <reference name="Fetch2" target="Inventory2/Fetch2"/>
  <reference name="Fetch3" target="Inventory3/Fetch3"/>
  <reference name="Order" target="OrderComponent/Order"/>
  <reference name="UsersInformation" target="UsersInformationComponent/UsersInformation"/>
  <implementation.java class="impl.CatalogImpl"/>
 </component>

 <component name="Inventory1">
  <service name="Fetch1">
   <interface.java interface="Fetch1Interface"/>
  </service>
  <implementation.java class="impl.Inventory1Impl"/>
 </component>

 <component name="Inventory2">
  <service name="Fetch2">
   <interface.java interface="Fetch2Interface"/>
  </service>
  <implementation.java class="impl.Inventory2Impl"/>
 </component>

 <component name="Inventory3">
  <service name="Fetch3">
   <interface.java interface="Fetch3Interface"/>
  </service>
  <implementation.java class="impl.Inventory3Impl"/>
 </component>
</composite>
```

Service Level Agreement Definition

We define the Service Level Agreement (SLA) for this application to deliver a good quality of service to web users. The guarantee terms are established in the SLA file, part of which is shown in Listing 5.2. The file is fed to the Knowledge component to constitute its SLADescription entity (refer to Figure 5.6). Since the Consult service is supposed to be heavily requested, the SLA requirements concern particularly this service.

Listing 5.2: SLA extract guarantee terms for the Products Composite application

```
<?xml version="1.0" encoding="UTF-8"?>
<wsag:AgreementOffer AgreementId="ab97c513">
<wsag:Name>xs:ProductsCompositeAgreement</wsag:Name>
<wsag:AgreementContext> ... </wsag:AgreementContext>
 <wsag:Terms>
  <wsag:All>
   <wsag:GuaranteeTerm wsag:Name="g1" wsag:Obligated="ServiceProvider">
    <wsag:ServiceLevelObjective>
     <wsag:KPITarget>
      <wsag:KPIName>Availability_UpTimeRatio</wsag:KPIName>
       <wsag:CustomServiceLevel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:exp="http://www.
           telecom-sudparis.com/exp">
        <exp:Greater><exp:Variable>minthreshold</exp:Variable> <exp:Value>0.9</exp:Value></exp:Greater>
       </wsag:CustomServiceLevel>
      <wsag:KPIName>AverageResponseTime</wsag:KPIName>
       <wsag:CustomServiceLevel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:exp="http://www.
           telecom-sudparis.com/exp">
        <exp:Less><exp:Variable>maxthreshold</exp:Variable> <exp:Value>1200</exp:Value></exp:Less>
       </wsag:CustomServiceLevel>
      <wsag:KPIName>ServiceCall</wsag:KPIName>
       <wsag:CustomServiceLevel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:exp="http://www.
           telecom-sudparis.com/exp">
        <exp:Less><exp:Variable>maxthreshold</exp:Variable><exp:Value>1000</exp:Value></exp:Less>
       </wsag:CustomServiceLevel>
     </wsag:KPITarget>
    </wsag:ServiceLevelObjective>
    ...
   </wsag:GuaranteeTerm>
  </wsag:All>
 </wsag:Terms>
</wsag:AgreementOffer>
```

As shown in this listing, we define three metrics for the Catalog component. Service availability v_{avai}, average response time v_{resp} and the number of simultaneous service calls v_{calls}. The described Service Level Objectives (SLOs) stipulate that:

- $v_{avai} = true$; the Consult service must be available at least 90% of time,
- $v_{resp} \leqslant 1200ms$; the average response time must not exceed 1200ms,
- $v_{calls} \leqslant 1000$; 1000 simultaneous service calls per second at most can be handled by the Catalog component without service degradation.

These metrics need to be monitored as they represent the most significant targeted metrics to compose the state of the Catalog component. Thus, we determine the state of the Catalog component as a 3-tuples vector $(v_{avai}, v_{resp}, v_{calls})$. Accordingly, these metrics should be extended in order to compose the state and then generate the state space.

5.4.2 Framework Extension and Instantiation

In this section, we present the different extensions and instantiations that we performed in order to use the framework for our own management concerns.

Metrics Extension

We need to extend the list of abstract Metrics initially provided by the framework. This extension allows as to compose the state of the Catalog component and later on generate the state space. As previously introduced in Section 5.3.3, the Analysis component has a set of abstract Metric types: RealMetric, to process real values; BooleanMetric, to process boolean values and DiscreteMetric, to process discrete values.

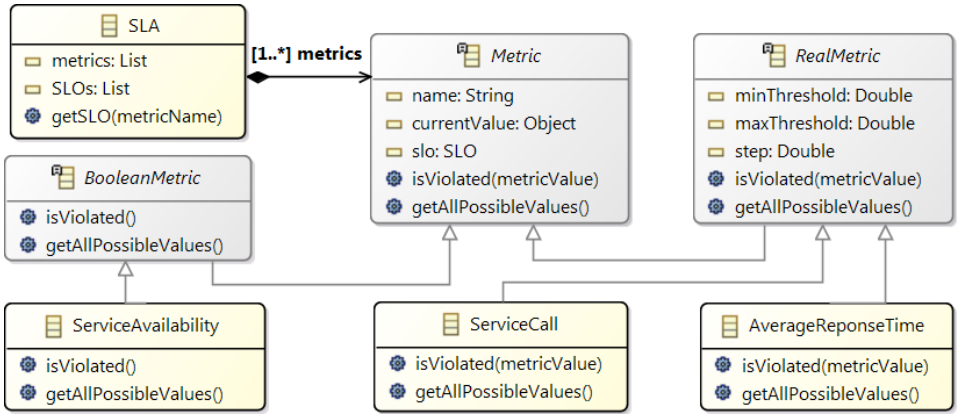


Figure 5.8: Extension of the Analysis component with the Metrics of the SLA

As shown in Figure 5.8, we extend the Metrics abstract types: BooleanMetric and RealMetric for our specific needs. We identify the service availability as a boolean metric while we define the average response time and the service call metrics as real metrics. Their values constitute the state vector $(v_{avai}, v_{resp}, v_{calls})$ and all their possible values should be determined to compose the state space. Thus, the boolean metric values are: $\{true, false\}$ whereas the real metrics values are discretized by our custom SLA parser according to a sampling interval step. Parsing the SLA helps to determine all the possible values for a given metric. Subsequently, it allows the generation of state space by realizing all possible combinations of metrics values.

Management Services Extension

The application and its deployment context are subject to dynamic changes and may require specific management needs. As such, the framework allows its extension by implementing additional Management services

with the needed specific adaptation behavior. Regarding our current management needs, we believe that the list of Management services initially provided by the framework (refer to Figure 5.5) is barely sufficient to satisfy them. Since the application is intended to be deployed in the Cloud, we need to extend this list in order to include another Management service: Migration service as depicted in Figure 5.9. The Migration service manages the migration of a component from its current Virtual Machine (VM) to another available VM in the system.

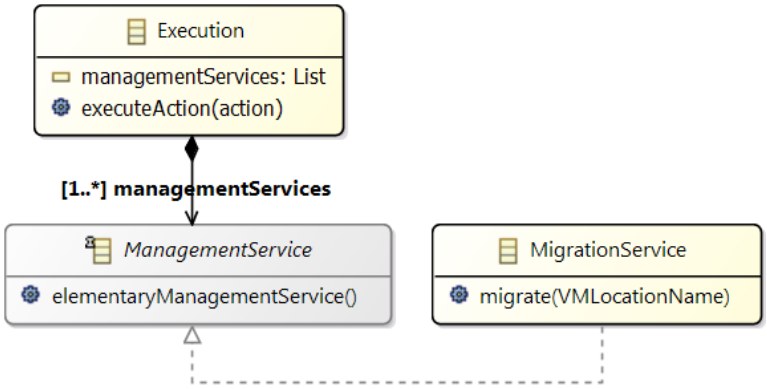


Figure 5.9: Extension of the Execution component with a Migration Service

We also provide the SCA description of this service to the Knowledge component. The Knowledge already contains a description file of the initially existing Management services provided by the Execution component (refer to Figure 5.5). As shown in Listing 5.3, the description of the additional Management service: MigrationService is provided to the Knowledge component in order to be added to the description file of the Execution component. This description entails information about the provided (and required if any) service, its implementation, etc.

Listing 5.3: Description file of the added MigrationService using SCA ADL

```
<composite name="MigrationComposite">
 <service name="MigrationService" promote="MigrationComponent/MigrationService"/>
 <component name="MigrationComponent">
  <service name="MigrationService">
   <interface.java interface="MigrationServiceInterface"/>
  </service>
  <implementation.java class="impl.MigrationComponentImpl"/>
 </component>
</composite>
```

As previously described, the Management services are necessary to the Analysis component to generate its action space. Accordingly, a list of actions will be generated for each available Management service provided by the Execution component, previously introduced in Figure 5.5. If we take the example of Scaling service, the actions generated from this Management service are service compositions and are: Scale-out and Scale-in. For instance, the Scale-out action is composed of the following sequence of elementary actions, executed in

this order: create second instance of Catalog → start second instance → create load balancer → bind load balancer to Catalog → bind load balancer to second instance of Catalog → bind Proxy to load balancer. Similarly, for the Migration service that we implemented, we generate the actions related to this service. But first, we need information about the hosting Cloud environment. More specifically, we need information about the available VMs in the system, hence, the Environment entity should be extended. Likewise, for the Parametric service (refer back to Figure 5.5) used to adapt the values of monitorable and adaptable properties values, we need to provide the list of all possible values of these properties to be able to generate parametric actions.

Environment Extension

Since the hosting environment may be different from an application component to another, we should provide information about the environment of the Managed components. Accordingly, the framework allows extending the Environment entity (refer to Figure 5.6) in order to implement the specific characteristics of the deployment environment. Extending the Environment entity is required so as the Knowledge component can be aware of the changes occurring in the Managed component environment and then behave correspondingly.

Therefore, we extend the abstract Environment entity as shown in Figure 5.10 by implementing the CloudEnvironment and VM entities. The CloudEnvironment allows recognizing the list of VM profiles that are available to host the Catalog component. We define the VMs characteristics such as their status (e.g., running, stopping), memory usage, cpu usage, etc. This extension allows performing migration actions with the implemented Migration service to map the Catalog component to other available VMs.

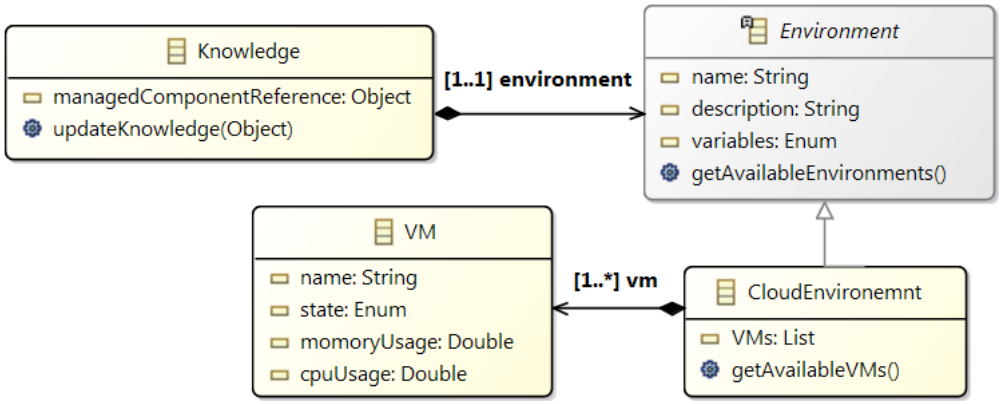


Figure 5.10: Extension of the Knowledge component with Cloud environment information

Properties Values

The properties of the Managed component can be monitorable and adaptable or not. In case they are, the possible values of these properties should be provided to the framework. In our case study, the Catalog component delivers its Consult service by fetching the products descriptions using its SearchAlgorithm. We implemented two simple search algorithms that can be used by the Catalog. The first algorithm performs a

simple search based on the exact keywords typed by a web user. Thus, the algorithm searches for the products that correspond exactly to these keywords, otherwise it may return a null result. The second algorithm fetches for similar keywords if the initial ones are incorrect or none existent in the inventories. This second algorithm performs extra processing regarding the first one which may require more time to deliver the Consult service. We refer to the first algorithm as SearchAlgorithm01 and the second one as SearchAlgorithm02.

We define an adaptable property for the Catalog component that specifies which SearchAlgorithm is to be used. Accordingly, we intend to dynamically insert getter and setter methods for an autonomic management of this property during the transformation of the Catalog component (with the Proxy generation). We are more interested by the setter method since we will use it to execute parametric actions with the Parametric service (refer to Figure 5.5) to adjust this property either on SearchAlgorithm01 or SearchAlgorithm02. Therefore, we provide these values to the Properties entity of the Knowledge component (refer to Figure 5.6). Later on, during the action space generation, the Analysis component uses these values to generate Parametric service actions.

Writing the ECA Table

We assume that a framework user may not have any knowledge about its context dynamics. Consequently, we consider that the ECA rules are an optional requirement of the framework. If available, ECA rules may be provided to the Knowledge component in order to use it as an initial knowledge about the actions to apply in certain states. This initial knowledge is used to initialize the QvalueFunction entity (refer to Figure 5.3) with the ECA couples $\{(s,a)\}_{ECA}$ to occasionally avoid bad performance induced during the first stages of learning and speed up the learning convergence. The ECA rules can be written as a table and integrate it to the Knowledge component. The table is parsed to match the described conditions with existing generated states in the state space and the mentioned actions with the generated actions in the action space.

In our case study, we do not have an exhaustive knowledge about the application context dynamics at deployment time. We can only make use of our limited view of adaptation scenarios to hand-code some ECA rules. Prior to the application transformation, we write the ECA table as depicted in Table 5.1 and supply it to the framework. This table stipulates that the Catalog component should be restarted if for some reason it is stopped or unavailable. It also indicates that whenever the Catalog is overloaded, that is to say, its simultaneous service calls and its average response time exceed the SLOs than insert a new instance of the component (i.e., scale-out). This table indicates also that if the Catalog has a slow response time even though the number of service calls is below the maximum value, then it should be migrated to a better performing VM if available. We make use of our custom parser to identify the ECA couples $\{(s,a)\}_{ECA}$ that match the provided rules. These couples are used to initialize the Q value function of the QvalueFunction entity.

Table 5.1: ECA table of Catalog component

Events with conditions	Action
$v_{avai} = false$	Restart
$v_{calls} > 1000$ and $v_{resp} > 1200$	Scale-out
$v_{calls} < 1000$ and $v_{resp} > 1200$	Migrate to VM with better performance

RL Parameters and Reward Function

As previously introduced, the Reinforcement Learning (RL) parameters are values to provide to the Analysis component to initialize its learning algorithm. Actually, the RL parameters are empirical values that impact the learning behavior and should be determined by the framework user. Indeed, these parameters are specific to each application transformation and should be determined empirically. The framework defines the interval values to guide the selection of these RL parameters as previously described in Section 4.4. We provide the empirical details on how we selected the RL parameters in Chapter 6 during the evaluation of the decision learning performance of the framework.

Regarding the reward function, the one that is initially provided by the framework and previously introduced in Algorithm 1 is satisfactory for our current management needs. As such, we choose not to implement another reward function.

5.4.3 Application Transformation

After providing the list of requirements to the framework, we perform the transformation illustrated in Figure 5.11. We choose to encapsulate the Catalog component into the autonomic container to render it self-adaptive. Actually, we implement a fine-grained transformation of the application as shown in this same figure. We solely encapsulate the Catalog component in the container, since the SLA stipulates terms targeting its Consult service.

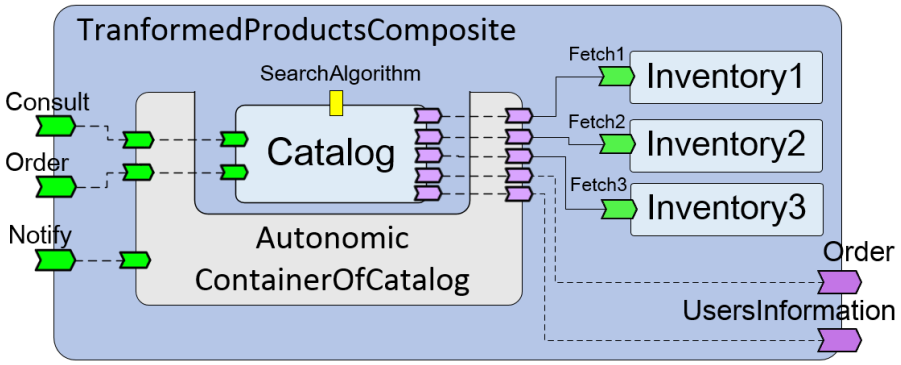


Figure 5.11: Products Composite application after transformation

The new description of the transformed application is shown in Listings 5.4 and 5.5 by using the SCA ADL description. These listings contain a full description of the transformed application internal components, their properties, their provided and required services and their implementations. This description is conform to the standard SCA description and it can be passed to an SCA runtime (e.g., FraSCAti [81]) that instantiates the components and binds them according to the TransformedProductsComposite description shown in Listing 5.4.

Listing 5.5 contains the description of the newly created composite: CatalogComposite. This composite is represented by the AutonomicContainerOfCatalog shown in Figure 5.11 and described in Listing 5.5.

As described in this listing, the CatalogComposite provides the functional services initially provided by the Catalog component. It also provides a non-functional service of Notify to be notified both by internal and external changes. The created CatalogComposite is equipped with self-adaptive abilities offered by the internal components: Proxy and MAPE-K of its autonomic container as shown in Listing 5.5.

Listing 5.4: Description file of the Products Composite application after its transformation using SCA ADL

```
<composite name="TransformedProductsComposite">
 <service name="Consult" promote="CatalogComposite/Consult"/>
 <service name="Order" promote="CatalogComposite/Order"/>
 <service name="Notify" promote="CatalogComposite/Notify"/>
 <reference name="Order" promote="CatalogComposite/Order"/>
 <reference name="UsersInformation" promote="CatalogComposite/UsersInformation"/>

 <component name="CatalogComposite">
  <service name="Consult" promote="ProxyComponent/Consult"/>
  <service name="Order" promote="ProxyComponent/Order"/>
  <service name="Notify" promote="MonitoringComponent/Notify"/>
  <reference name="Fetch1" promote="Catalog/Fetch1"/>
  <reference name="Fetch2" promote="Catalog/Fetch2"/>
  <reference name="Fetch3" promote="Catalog/Fetch3"/>
  <reference name="Order" promote="Catalog/Order"/>
  <reference name="UsersInformation" promote="Catalog/UsersInformation"/>
  <implementation.composite name="composite.CatalogComposite"/>
 </component>

 <component name="Inventory1">
  <service name="Fetch1">
   <interface.java interface="Fetch1Interface"/>
  </service>
  <implementation.java class="impl.Inventory1Impl"/>
 </component>

 <component name="Inventory2">
  <service name="Fetch2">
   <interface.java interface="Fetch2Interface"/>
  </service>
  <implementation.java class="impl.Inventory2Impl"/>
 </component>

 <component name="Inventory3">
  <service name="Fetch3">
   <interface.java interface="Fetch3Interface"/>
  </service>
  <implementation.java class="impl.Inventory3Impl"/>
 </component>
 </composite>
```

Listing 5.5: Description file of the Catalog Composite after its transformation using SCA ADL

```
<composite name="CatalogComposite">
 <service name="Consult" promote="ProxyComponent/Consult"/>
 <service name="Order" promote="ProxyComponent/Order"/>
 <service name="Notify" promote="MonitoringComponent/Notify"/>
 <reference name="Fetch1" promote="Catalog/Fetch1"/>
 <reference name="Fetch2" promote="Catalog/Fetch2"/>
 <reference name="Fetch3" promote="Catalog/Fetch3"/>
 <reference name="Order" promote="Catalog/Order"/>
 <reference name="UsersInformation" promote="Catalog/UsersInformation"/>

 <component name="ProxyComponent">
  <service name="Consult" promote="Catalog/Consult"/>
  <service name="Order" promote="Catalog/Order"/>
  <service name="Execute">
   <interface.java interface="ExecuteInterface"/>
  </service>
  <reference name="Consult" target="Catalog/Consult"/>
  <reference name="Order" target="Catalog/Order"/>
  <reference name="Notify" target="MonitoringComponent/Notify"/>
  <implementation.java class="impl.ProxyImpl"/>
 </component>

 <component name="Catalog">
  <property name="SearchAlgorithm"/>
  <service name="Consult">
   <interface.java interface="ConsultInterface"/>
  </service>
  <service name="Order">
   <interface.java interface="OrderInterface"/>
  </service>
  <reference name="Fetch1" target="Inventory1/Fetch1"/>
  <reference name="Fetch2" target="Inventory2/Fetch2"/>
  <reference name="Fetch3" target="Inventory3/Fetch3"/>
  <reference name="Order" target="OrderComponent/Order"/>
  <reference name="UsersInformation" target="UsersInformationComponent/UsersInformation"/>
  <implementation.java class="impl.CatalogImpl"/>
 </component>

 <component name="MonitoringComponent">
  <service name="Notify">
   <interface.java interface="NotifyInterface"/>
  </service>
  <reference name="Notify" target="AnalysisComponent/Notify"/>
  <implementation.java class="impl.MonitoringComponentImpl"/>
 </component>

 <component name="AnalysisComponent">
  <service name="Notify">
   <interface.java interface="NotifyInterface"/>
  </service>
```

```
51   <reference name="Adapt" target="ExecutionComponent/Adapt"/>
52   <reference name="SendOptimalPolicy" target="PlanningComponent/SendOptimalPolicy"/>
53   <reference name="Alert" target="PlanningComponent/Alert"/>
54   <reference name="ManagementServicesDescription" target="KnowledgeComponent/ManagementServicesDescription"/>
55   <implementation.java class="impl.AnalysisComponentImpl"/>
56  </component>
57
58  <component name="PlanningComponent">
59   <service name="SendOptimalPolicy">
60    <interface.java interface="SendOptimalPolicyInterface"/>
61   </service>
62   <service name="Alert">
63    <interface.java interface="AlertInterface"/>
64   </service>
65   <reference name="Adapt" target="Execution/Adapt"/>
66   <implementation.java class="impl.PlanningComponentImpl"/>
67  </component>
68
69  <component name="ExecutionComponent">
70   <service name="Adapt">
71    <interface.java interface="AdaptInterface"/>
72   </service>
73   <reference name="Execute" target="ProxyComponent/Execute"/>
74   <implementation.java class="impl.ExecutionComponentImpl"/>
75  </component>
76
77  <component name="KnowledgeComponent">
78   <service name="ApplicationDescription">
79    <interface.java interface="ApplicationDescriptionInterface"/>
80   </service>
81   <service name="ManagementServicesDescription">
82    <interface.java interface="ManagementServicesDescriptionInterface"/>
83   </service>
84   <implementation.java class="impl.KnowledgeComponentImpl"/>
85  </component>
86 </composite>
```

5.5 Conclusion

In this chapter, we introduced a generic framework for building self-adaptive component-based applications. The framework offers reusable and extensible functionalities intended to support the application developers in transforming their applications. It is also designed to accompany the users in defining certain aspects that are specific to their non-functional needs. The framework is provided with an abstract design which enables extending and overriding the autonomic behavior as well as its integration with different deployment environments. It also allows its integration with different applications by being designed using the SCA standard.

The transformation of the application occurs by encapsulating its components in autonomic containers to render them self-adaptive. The transformed components are self-adaptive and can dynamically learn optimal policies based on a RL algorithm. The framework allows the transformation on different granularity levels depending on the management needs. Instead of a static predefined policy, the autonomic container is equipped with an adaptation logic based on learning abilities to learn a decision policy dynamically at execution time.

Along with describing the framework offered functionalities, we presented a case study of a Product Composite application deployed in the Cloud. We demonstrated the usage of our framework with an illustrative transformation of this application. Accordingly, we described the different aspects that are required to the usage of the framework. We identified the elements to be provided, extended or implemented to customize the framework behavior and render it suitable to our application management needs.

In the next chapter, we intend to validate our contributions presented in Chapters 4 and 5. We employ the case study of the Products Composite application as a basis for our validations. We introduce the implementation details and the experimentations that we performed on this particular case study to assess our proposals.

Chapter 6

Evaluation and Validation

Contents

6.1 Introduction

We dedicate this chapter to the evaluation and validation aspects of our contributions presented in Chapters 4 and 5. We make use of the Products Composite application case study, previously described in Section 5.4, to implement and run the experiments. We start by providing implementation details of our contributions in Section 6.2. Afterwards, we introduce the performed evaluations in two steps. In the first step in Section 6.3.1,

we present experiments aiming to evaluate the framework usage for the application described in the case study. In the second step in Section 6.4, we introduce the experiments that were conducted to evaluate the learning abilities of the adaptation logic provided by the autonomic container. Accordingly, we describe the context dynamics of the application. Thereafter, we determine the Reinforcement Learning parameters necessary to run efficiently our multi-step learning algorithm. Finally we deploy the application in a Cloud context with and without transformation to compare their before and after behaviors. We use different versions of learning algorithms, one-step and multi-step in order to observe the application performance with both versions.

6.2 Implementation Details

We have implemented our framework using the Service Component Architecture (SCA) standard [82]. One of the main features of SCA is its independence of particular technology, protocol, or implementation. Henceforward, the choice of SCA will allow an easy integration of existing off-the-shelf applications to the framework, despite their heterogeneous implementations (e.g., JAVA, BPEL, C, etc.). Likewise, in our case study, we implemented the components of the Products Composite application using the SCA standard and in Java.

Regarding our framework, we have implemented the internal components of the autonomic container in Java and as separated SCA components to allow more flexibility in adding new functionalities to them. This enables future enhancement of our container, by facilitating updates and future changes in order to allow plugging or unplugging specific modules to customize and extend the container's default behavior.

We have implemented the mechanisms that allow the transformation of application components to render them monitorable and adaptable by generating the byte-code of the Proxy component. For this required byte-code manipulation we used the Java reflection API [4] and the Java programming ASSISTant (Javassist) library [5]. The Java reflection API provides classes and interfaces for retrieving reflective information about classes and objects. This API provides means to gather metadata of a particular class. It enables accessing information such as constructors, fields, methods of the loaded classes. The purpose is to enable analyzing and operating on their objects counterparts. Javassist is a class library that is intended for editing Java byte-codes by enabling the generation of new Java class files at runtime. Actually, it allows Java programs to create new classes and modify class files when the Java Virtual Machine (JVM) loads it.

The rest of the MAPE-K components are implemented as SCA components that contain the needed functionalities to enable autonomic management and learning abilities. As described in Section 5.3, some of the internal entities of the Monitoring, Analysis, Execution and Knowledge components are abstract designed. Since these internal entities are domain specific, the framework user needs to customize and instantiate them with the desired specific behavior. To this end, we have defined them either as abstract classes to be extended or interfaces to be implemented. This allows extending the needed specific functionalities without reinventing their hosting components (i.e., Monitoring, Analysis, Execution and Knowledge). The Planning component is implemented as SCA component that contains the functionalities described in Section 5.3.

Regarding the Migration service, we made use of Java Serializable API [6] which provides mechanisms for

[4] https://docs.oracle.com/javase/tutorial/reflect/
[5] www.javassist.org/
[6] https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

serialization and deserialization. These mechanisms allow serializing objects to save their state by representing them as bytes sequences in a file which includes all the object's information. In case the Managed component is a statefull component, then all its information as well as its autonomic component's are saved in a file (i.e., serialized). Afterwards, the file is transferred using SSH to the new VM location. After running the autonomic container in the new VM, the file is deserialized to reload the saved state.

6.3 Evaluation of the Framework Approach for Building Self-Adaptive Component-based Applications

In this section, we evaluate our proposed framework for building self-adaptive applications. To do so, we refer back to the case study of the Product Composite application that we introduced in Section 5.4 to perform the following experiments on our autonomic container. It is worthy to note that these experiments were executed on a 64-bit windows PC, equipped with 8GB of RAM and Intel core i7 with 2.70GHz frequency.

6.3.1 Transformation Overhead of the Framework

We are interested in evaluating the overhead induced by the utilization of the autonomic container during the application transformation. We are concerned that this time might be long which may affect the deployment phase. The overhead describes the time required to transform the application. It includes the times needed for the generation of the Proxy component and the instantiation of the MAPE-K components.

Referring back to the case study presented in Section 5.4, the average time required to perform the described transformation is 303 ms. This time includes 296 ms for the Proxy generation and 7 ms for the instantiation of the MAPE-K components. The transformation time is computed for a single autonomic container having a single Managed component (i.e., Catalog) and providing two functional services (i.e., Consult and Order services). During the experiments, we noticed that the times needed for the instantiation of the Monitoring, Analysis, Planning, Knowledge components are constant, very negligible and independent of any parameter. The Execution component holds the list of Management services that can be extended by the framework user. Thus, its instantiation time is dependent on their number, but it remains very negligible and does not have a noteworthy impact on the overhead.

As noticed during the transformation of the Catalog component, the Proxy generation has the most impact on the transformation time. To gauge the accuracy of this observation, we conducted the experiments shown in Figure 6.1. To determine which aspect has the most impact on the time required for the Proxy component generation, we varied the number of interfaces from 10 to 35 and methods from 10 to 100 of the Catalog component and computed the time required for its Proxy generation. Then, we aggregated the results to the time required for the instantiation of the MAPE-K components to compute the overhead.

When we fixed the number of interfaces to 10 and varied the number of methods from 10 to 100, we noticed that the growth slope of the curve is small. When we fix the number of interfaces to 35 and varied the number of methods from 10 to 100, we noticed that the growth slope of the curve is large. We deduce from Figure 6.1 that the growth slope of the generation time is largely small when the Proxy component implements 10 interfaces and 100 methods, than when it implements 35 interfaces and 100 methods. This

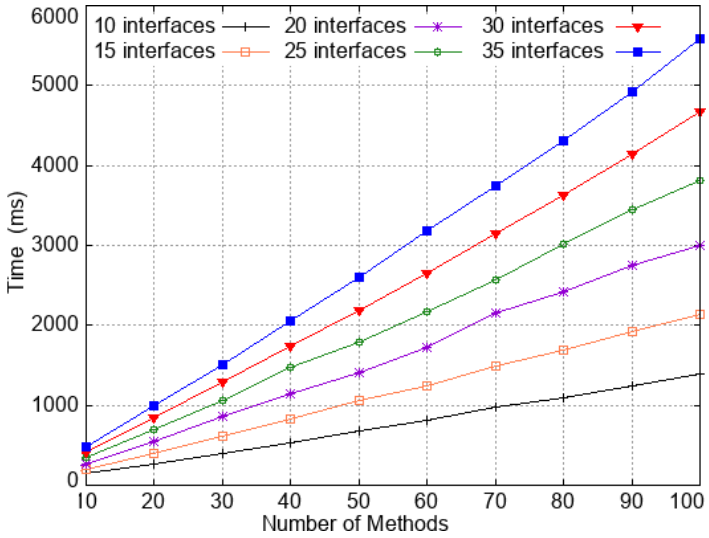


Figure 6.1: Time needed for the Proxy generation

deduction proves that the number of interfaces influences more than the number of methods on the time required for the generation of the Proxy's byte-code. We deduce that the time required for the generation of the byte-code of the Proxy component has more impact on the overhead.

Actually, the generation of a Proxy includes the time for loading interfaces classes from the disk memory and the time of byte-code class generation. In fact, when dynamically loading interfaces classes from the disk memory, we need to go through each interface class file, load it from disk memory, get all its methods and then byte-code generate the Proxy class. Technically, the time is more consumed in the interface class files loading from the disk memory than in methods loading since the methods are already defined in the class file. Indeed, the generation of a proxy consists of the time for loading interfaces classes from memory and the time of byte code class generation. Therefore, increasing the number of interfaces implies the augmentation of the time of proxy's generation. We conclude that the overhead strongly depends on the number of services interfaces provided by the transformed component. Nonetheless, it remains negligible and of the order of few seconds at most.

6.3.2 Post-Transformation Overhead of the Analysis prior to Decision Learning Process

After the transformation of the Products Composite application, the autonomic container took $6,7\ ms$ to start its decision learning process. Actually, after instantiating the Analysis component which includes instantiating its StateSpace and ActionSpace components. The Analysis needs to generate its state and action spaces prior to triggering its decision learning process. The state space generation depends on the number of metrics needed to be monitored. The action space generation depends on the number of Management services that are provided by the Execution component.

To study the impact of the metrics number on the time required to generate the state space, we conducted

the experiments shown in Figure 6.2. We varied the number of metrics defined in the SLA file from 3 to 100 metrics and generated state spaces with different interval steps to vary the number of generated states from 100 to 2000 states. The states are generated by parsing the SLA file to extract the metrics and the conditions on their required values and then sampling all the combinations of the metrics values according to the interval step. Figure 6.2 shows clearly that the number of targeted metrics has more impact than the number of generated states on the time of state space generation. As depicted in this figure, to generate the same number 2000 of states, it takes less time with 3 metrics (with smaller interval step value) than it takes with 100 metrics (with bigger interval step). It is related to the number of combinations (i.e., 100 metrics) needed to be performed to compose the state vectors that impacts more this time. We deduce that the time required for the generation of the state space depends on the number of metrics defined in the SLA. Nonetheless, this time remains negligible and of the order of 100 ms at most.

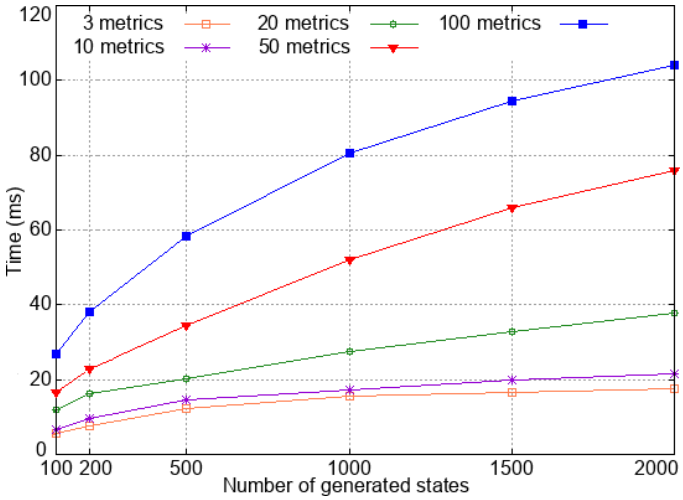


Figure 6.2: Time needed for state space generation

To study the impact of the Management services number on the time required to generate the action space, we conducted the experiments shown in Figure 6.3. We varied the number of Management services from 1 to 30 and the operations (i.e., methods) implemented per Management service from 1 to 30 to measure their impact on the time of action space generation. Recall that the actions are generated by consulting the Management services available for the Managed component and the information provided by the Knowledge component for these services (e.g., available VMs to generate Migration actions and properties values to generate Parametric actions, etc.). For instance, the Migration service has the migrate operation (see Figure 5.9) and in case there are five VMs available for the Catalog component, then the number of generated actions is 5. When we fixed the number of operations to 1 and changed the number of Management services, we noticed that the growth slope remains small. Whereas, when we fixed the number of operations to 30 and changed the number of Management services, we noticed that the growth slope increases faster. More specifically, Figure 6.3 shows that to generate the same number 300 of actions, its takes more time with 30 Management services and 10 operations than with 10 Management services and 30 operations. The number of Management services has more effect on the generation time than the number of implemented operations. This observation

is logical since going through the Management services interfaces classes takes more time than going through their methods. Yet, this time remains negligible and of the order of few milliseconds at most.

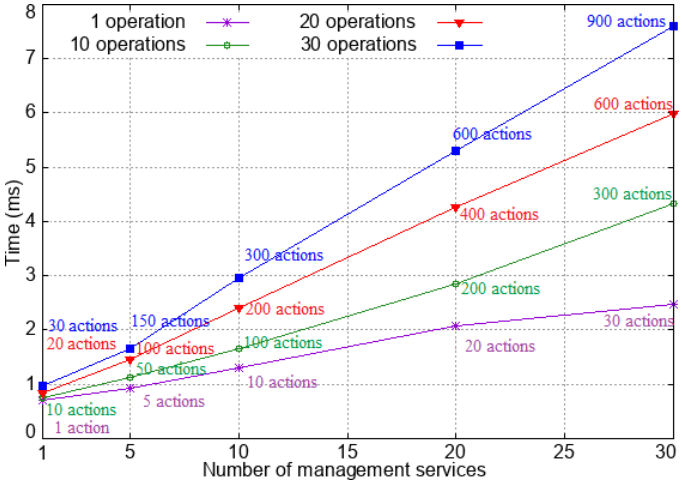


Figure 6.3: Time needed for action space generation

6.3.3 Synthesis

Based on the performed experiments, we conclude that the overhead induced by the framework during the transformation of the application depends strongly on the business functionalities of the application. This time is directly impacted by the number of service interfaces that are provided by the application components to generate the Proxy component. The transformation overhead remains negligible and of few seconds at most which demonstrated that our proposed container is lightweight and able to host an SCA component and equip it with self-adaptive capabilities at a minimal cost. We conclude that this time does not slow down the deployment phase of the transformed application. After the transformation and prior to starting its decision learning process, the Analysis component needed time to generate its state and action spaces. This time remains also very negligible and of the order of milliseconds. This time does not slow down the beginning of the learning phase either.

6.4 Evaluation of the Self-adaptive Decision Making Process approach based on Multi-step Reinforcement learning

The aim of this section is to present the experiments that we conducted to mainly evaluate the self-adaptivity, performance optimization and the SLA guarantee of the transformed application using our proposed multi-step learning. To do so, we started by determining the context dynamics of the application. Afterwards, we had to determine the best suited RL parameters to our application to ensure a efficient learning phase. Thereafter, we first deployed the application without transformation and later on with transformation. We compared the application behaviors before and after its transformation. The comparison purpose is to measure

the impact of using the autonomic container on the application performance with its compliance to the SLA terms. During the transformation, we used the two learning versions of one-step and multi-step learning that we implemented. We needed to compare the behavior of the transformed application of both learning versions in terms of self-adaptivity, performance optimization and the SLA guarantee.

6.4.1 Context Dynamics Description

During the experiments, we determined the context dynamics by the workload variations. We defined the workload as the number of simultaneous clients calls (i.e., service calls to the Catalog component). We implemented a workload generator and determined three intensities of workload: light, medium and heavy. A light workload implies service calls number that remains below the capacity (i.e., SLO) of the Catalog component. A medium workload concerns service calls number that is around the capacity of the Catalog component which may exceed or not this capacity. A heavy workload is about service calls number that largely exceeds this capacity.

These intensities were randomly generated for alternated interval times to observe how the application reacts to these dynamics before and after its transformation. We also occasionally triggered random breakdowns with service unavailability of the transformed Catalog component to observe the application behavior before and after its transformation. The comparison of the application behavior before and after its transformation is necessary to measure the impact of using the autonomic container on the application performance under these context dynamics.

6.4.2 Reinforcement Learning Parameters Selection for the Learning Efficiency

Prior to evaluating the application performance, we need to define the Reinforcement Learning (RL) parameters that ensure an efficient learning phase to our Analysis component. In this first stage of evaluation, we determined these parameters and studied their impact on the learning efficiency and reaching convergence. We previously provided an ECA policy (refer to Section 5.4.2) but we decided for these experiments to not include it. Therefore, we run these experiments without any prior knowledge of the context dynamics. We needed to efficiently gauge the parameters choice effect on the learning efficiency without possibly an external impact of the ECA policy.

To run the multi-step learning algorithm depicted in Algorithm 4.4.6, we first determined the discount factor γ value. According to Section 4.4.4, the discount factor should be close to 1 to consider the accumulated rewards on the long term. In fact, γ indicates to the algorithm how far ahead in time the algorithm should look in terms of collected rewards. Actually, a discount factor value that is closer to 0 indicates to the algorithm that only the immediately received rewards are being considered (refer to Equation 4.5). This implies a shallow look-ahead of the accumulated rewards. A discount factor value that is closer to 1 makes the algorithm consider not only the recently accumulated reward but also the long run sum of the reward values. Furthermore, we need to slightly discount the received rewards in order to encourage the algorithm to accumulate rewards sooner than later which helps speed up the convergence [44]. Thus, we decided to set the discount factor to $\gamma = 0.9$ for the rest of experiments to slightly discount the computed reward values and consider their long run sum at the same time. It seems a naive method to set this parameter value, but it showed its effectiveness

in several RL-based approaches in the literature [3].

The following RL parameters $\alpha, \epsilon, \lambda$ and ϵ_z directly affect the learning efficiency. Thus, we determined them empirically with the following experiments. As previously introduced in Chapter 4, these parameters range in an interval from 0 to 1. During the experiments, we set different values in this interval for all these RL parameters and experimented their combinations, although, we selected the most significant values for illustration. In the following, we temporarily set the exploration degree to $\epsilon = 0.3$, the eligibility factor to $\lambda = 0.8$ and the parameter $\epsilon_z = 10^{-4}$ for the following experiments.

In the first set of experiments, we needed to determine the suitable learning rate α to efficiently learn an optimal decision policy. Thus, we run the multi-step algorithm with different learning rates α from 0.1 to 0.6 to observe its impact on the learning efficiency as depicted in Figure 6.4. For each learning rate we run the experiments for 1000 transitions to sufficiently observe the evolution of the mean Q value function. In this figure, we computed the mean Q value function along the transitions of a set of states and actions of the search space with different learning rates.

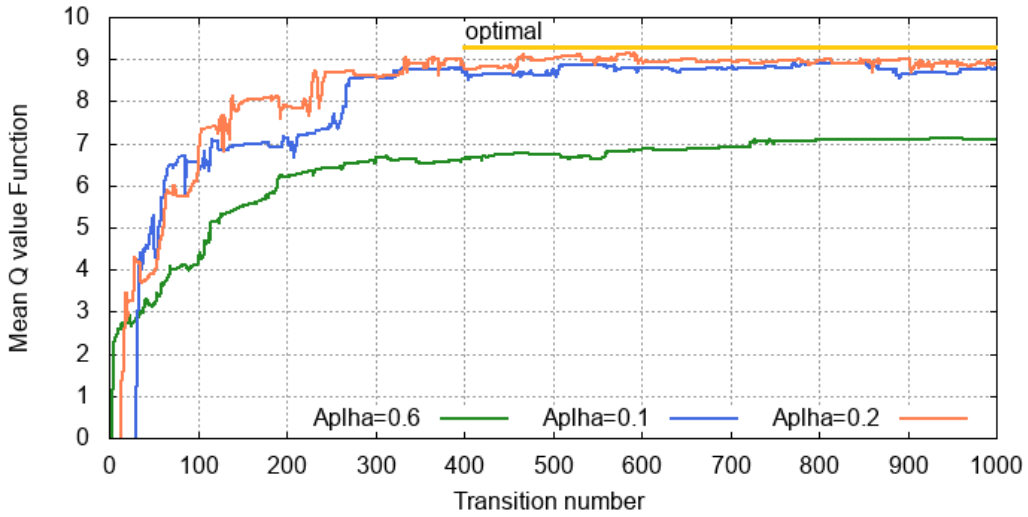


Figure 6.4: Efficiency of Multi-Step Learning with Different Learning Rates

As depicted in this figure, the mean Q value function started converging around the 300^{th} transition for all the learning rates. We notice that when $\alpha = 0.6$, the mean Q value function increases very fast at the very beginning of learning but it quickly slows down its growth slope and then fails achieving optimal values. We can see that a higher learning rate may get trapped into a sub-optimal policy. When $\alpha = 0.1$, the mean Q value function increases slowly and guarantees reaching an optimal policy. When $\alpha = 0.2$, we notice that the mean Q value function increases quickly while reaching an optimal policy which shows it effectiveness. We deduce that a higher learning rate may accelerate the learning at the very beginning while a small one helps avoiding sub-optimalily. The last value $\alpha = 0.2$ seems to have a good tradeoff between speed and effectiveness. For the rest of experiments we set the learning rate to $\alpha = 0.2$.

In the second set of experiments, we analyzed the impact of exploration degree ϵ on the learning speed to decide which value to keep for the rest of experiments. We run again the multi-step learning algorithm with

different parameters for the $\epsilon-greedy$ policy from 0.1 to 0.8 and computed the number of transitions needed to the algorithm to converge to the optimal policy. We conducted the experiments shown in Figure 6.5 to find the right balance between exploration and exploitation for our case study. As shown in this figure, the algorithm converges faster for higher exploration degrees compared to lower ones. However, we noticed during the experiments that whenever we increase the exploration degree, the application performs badly. Actually, with higher values such as 0.5 and 0.8, the algorithm selects very frequently random an potentially bad actions which reflects badly on the application performance by frequently violating the SLA terms. However, with a lower value such as 0.1, the application occasionally violates the SLA whereas the learning session lasts longer. When $\epsilon = 0.3$, the algorithm converges faster compared to $\epsilon = 0.1$ and violates less the SLA terms compared to 0.5 and 0.8. The value 0.3 seems to have a good tradeoff between speed and application performance. We set the exploration degree to $\epsilon = 0.3$ for the rest of experiments.

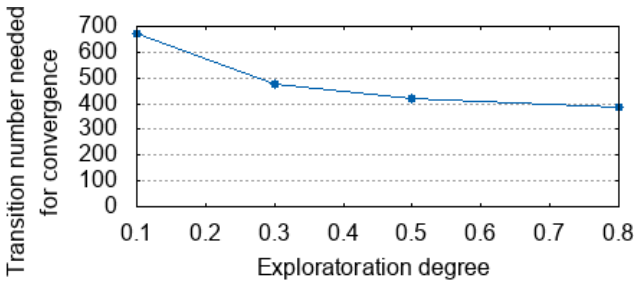


Figure 6.5: Convergence speed of multi-step vs. exploration degree ϵ

The third set of experiments was intended to select the suitable values for the eligibility factor λ and the parameter ϵ_z. We run the multi-step learning algorithm with different values for λ from 0.2 to 0.8. We also varied ϵ_z from 10^{-1} to 10^{-4}. These two parameters determine how far the multi-step learning can look back in its previously visited couples (s, a) to determine the eligible ones for Q value function update. Therefore, we conducted the experiments shown in Figure 6.6. We computed the number of transitions needed to the algorithm to converge while varying the parameters λ and ϵ_z.

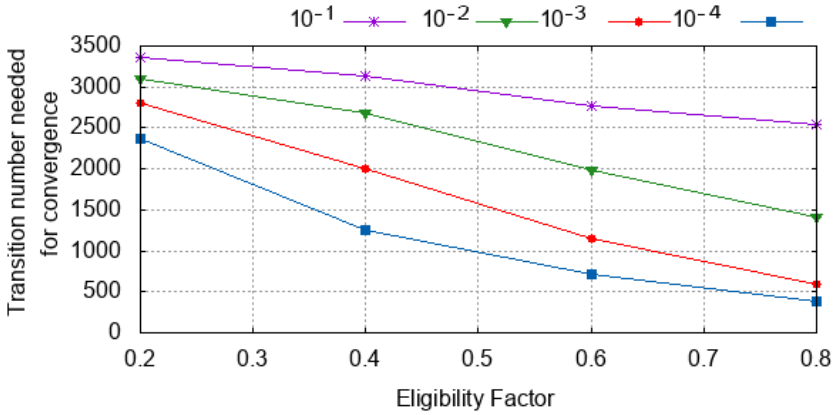


Figure 6.6: Convergence speed of multi-step vs. λ and ϵ_z

For $\lambda = 0.2$, when $\epsilon_z = 10^{-1}$, the multi-step algorithm acts exactly like the one-step learning. Actually, these values do not allow for a backward view of any previously visited couple to update their Q value functions. Thus, the number of transitions required to the algorithm to converge remains high. When ϵ_z ranges from 10^{-2} to 10^{-4}, the algorithm has a narrow view on its previously visited couples. It still could update their respective Q value functions and then slightly reduce the time needed for its convergence as shown in this figure. For $\lambda = 0.8$, when $\epsilon_z = 10^{-1}$ the transition number needed to convergence is slightly enhanced compared to $\epsilon_z = 10^{-4}$ that shows a wider margin of transition number improvement. In fact, when $\lambda = 0.8$ and $\epsilon_z = 10^{-4}$, we allow the algorithm a sufficient view on its previously visited couples to update their Q value functions. These parameter values reflects positively on reducing the transitions needed to the algorithm to converge.

During the experiments, we noticed that when ϵ_z goes below 10^{-4} to tend toward 0, the transitions number needed for convergence does not necessarily enhance. However, we observed slowness of few milliseconds at each transition in terms of CPU time. Actually, when the multi-step algorithm had to look back to a larger number of visited couples, its takes longer time to perform its Q value updates upon each performed transition. Therefore, the most relevant parameters in our case study are: $\lambda = 0.8$ and $\epsilon_z = 10^{-4}$.

Now that we empirically defined the RL parameters that directly impact the learning efficiency, we can use them to evaluate the application performance after its transformation.

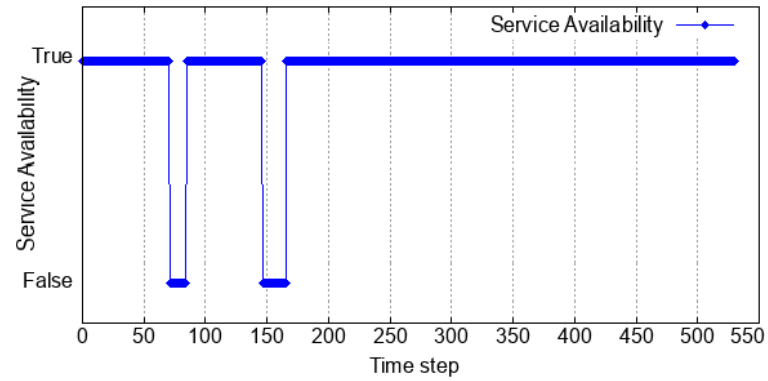
6.4.3 Application Performance

In the following experiments, we examine the application performance by evaluating its self-adaptivity to its context dynamics, performance optimization during learning phase and its SLA guarantee during and after learning phase. Recall that the context dynamics are determined by the number of simultaneous service calls that we generated as workload. As previously introduced, three intensities of workload are randomly mixed during and after learning: light, medium and heavy. We also randomly triggered breakdowns with service unavailability of the Catalog component to observe the application behavior before and after its transformation. We compared the application behavior before and after its transformation to assess the impact of using the autonomic container on its performance under the described context dynamics.

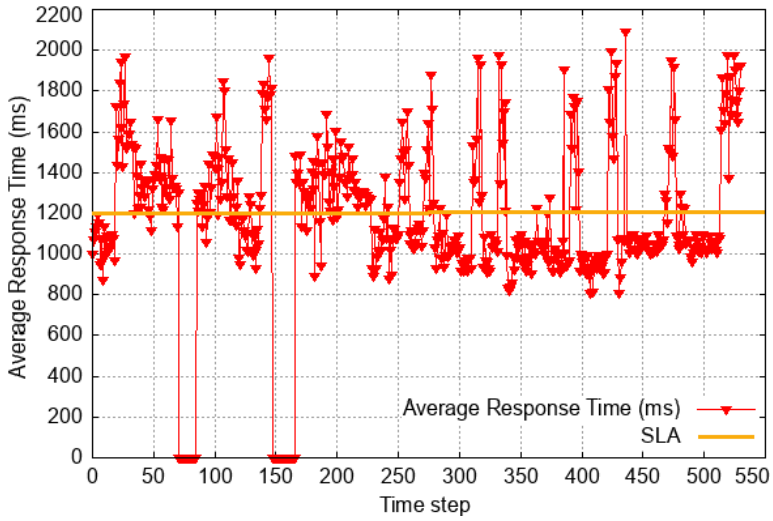
We also needed to evaluate the performance optimization during the learning phase of the multi-step algorithm. To do so, we implemented the one-step version of our algorithm and compared its performance with the multi-step version that we use. Our objective is to compare the SLA guarantee of both versions during the learning phase. In the following, we introduce the set of experiments that we performed to evaluate the applications performance.

6.4.3.1 Before the Transformation

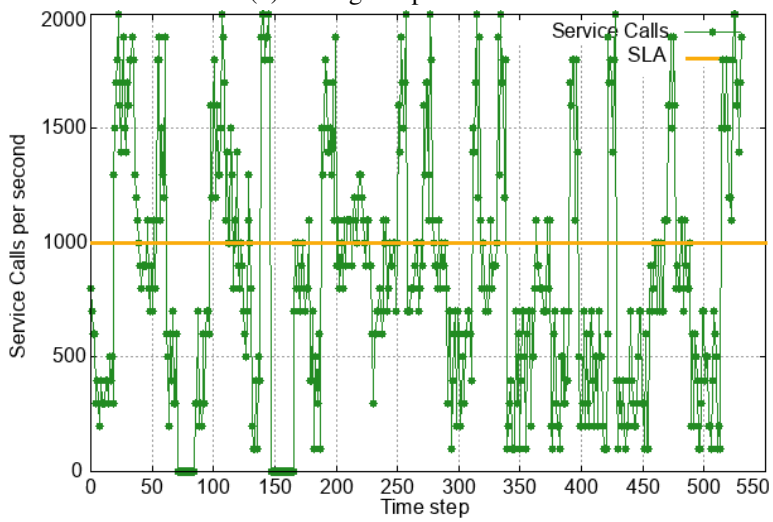
We deployed the application without transformation and run it under the described context dynamics. Figure 6.7 depicts the values v_{avai} of the service availability metric, v_{resp} of the average response time metric and v_{calls} of the simultaneous service calls metric at execution time and at each time step before the transformation. The three metrics compose the observed state of the Catalog component.



(a) Service availability



(b) Average response time



(c) Service calls

Figure 6.7: Performance metrics before transformation of the application

As shown in this figure, the metrics values frequently violate the SLA terms. We noticed that whenever the service calls exceed the Catalog component required capacity (Figure 6.7c), its average response time significantly slows down (Figure 6.7b). Meanwhile, the breakdowns caused continuous service interruptions (Figure 6.7a). We needed to relaunch the application manually in order to continue the service delivery.

6.4.3.2 After the Transformation with Online One-Step Learning

In the following experiments, we integrated the ECA rules to the framework to initialize the Q value function. Then, we transformed the application as previously introduced in Figure 5.11 of the case study. We deployed the transformed application and run it with random starting state of the Catalog component (due to the random workload).

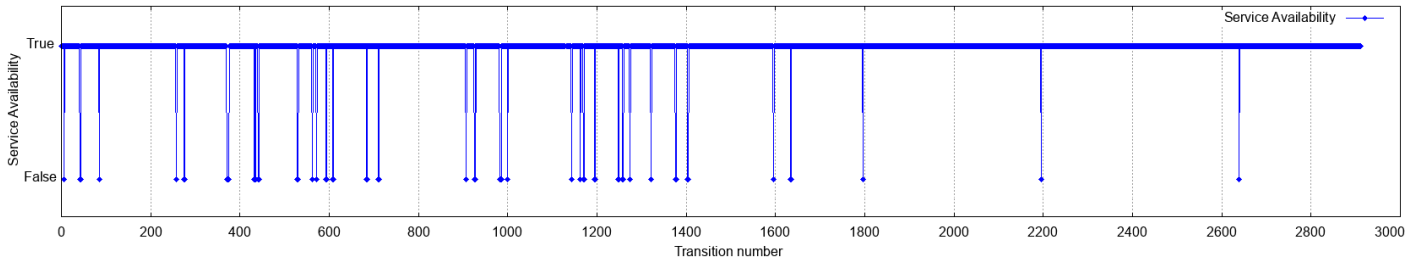
These set of experiments aim to show the effectiveness of the RL approach in terms of self-adaptivity of the transformed application. Therefore, we run the one-step learning algorithm and evaluated the self-adaptivity criterion under the same context dynamics. Note that we set the following RL parameters determined earlier to $\gamma = 0.9, \alpha_0 = 0.2, \epsilon = 0.3$. These parameters remain valid and provide us with best results for our application for both one-step and multi-step learning.

Figure 6.8 depicts the metrics values monitored at execution time of the service availability v_{avai} (Figure 6.8a), the average response time v_{resp} (Figure 6.8b) and the simultaneous service calls v_{calls} (Figure 6.8c). It also depicts the reward function in Figure 6.8d) at execution time which illustrates positive and negative values that reflect the improvement or degradation of the Catalog component's state.

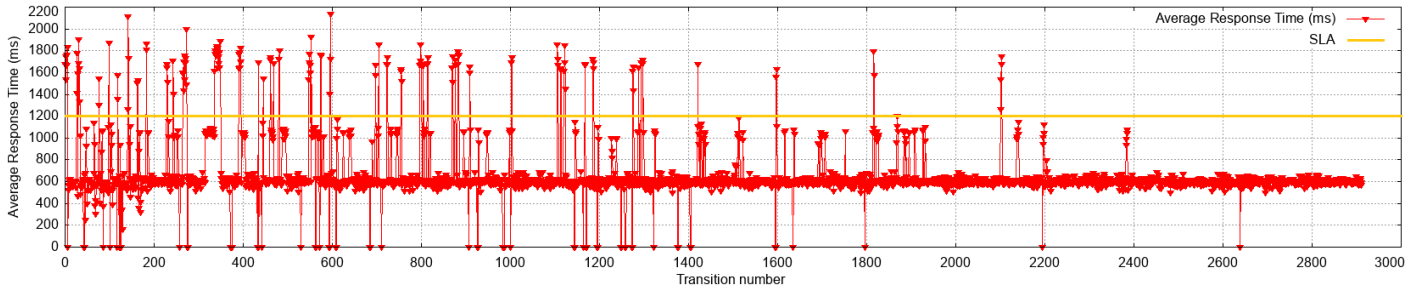
During the first learning phases, we noticed that the three metrics values violated the SLA terms frequently before starting to space out these violation intervals. Violation spacing occurs around the 1800^{th} transition for service availability (Figure 6.8a), the 1400^{th} transition for average response time (Figure 6.8b) and the 2000^{th} transition for service calls (Figure 6.8c). Actually, over the transitions, the algorithm gradually acquired enough experience on its context to learn how to avoid bad action selection. The algorithm started then to self-adapt to workload dynamics and service breakdowns in order to apply the best possible action.

The learning session lasted for 2420 transitions before the algorithm comes to convergence after which the Catalog's state has stabilized as shown in the three metrics figures. In addition, after convergence, the collected rewards are mainly positive which demonstrated an optimal action selection after convergence which is reflected on the quality of the catalog state.

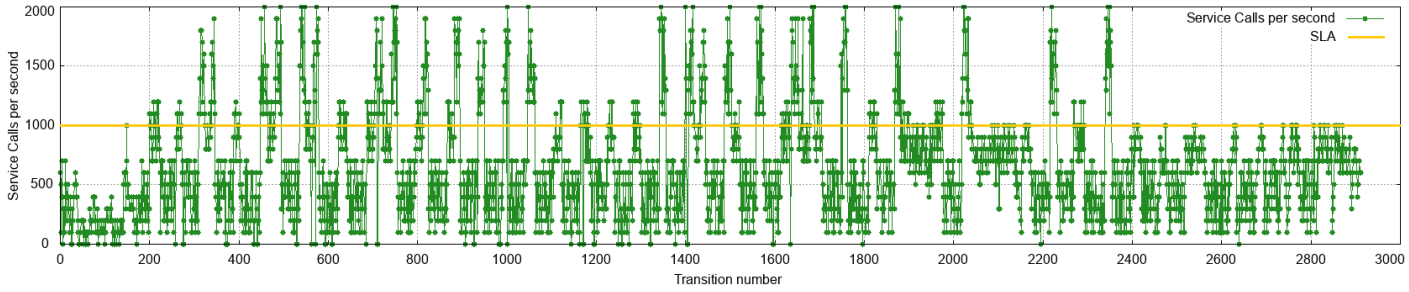
However, an efficient meeting of quality requirements during the learning phase was hard to achieve. The one-step algorithm spent a long time in SLA violation which is not tolerated for real world applications. Actually, the algorithm had to go through each couple (s, a) of the search space several times in order to learn the optimal policy. This kind of algorithm converges very slowly, especially when the state and action spaces are large.



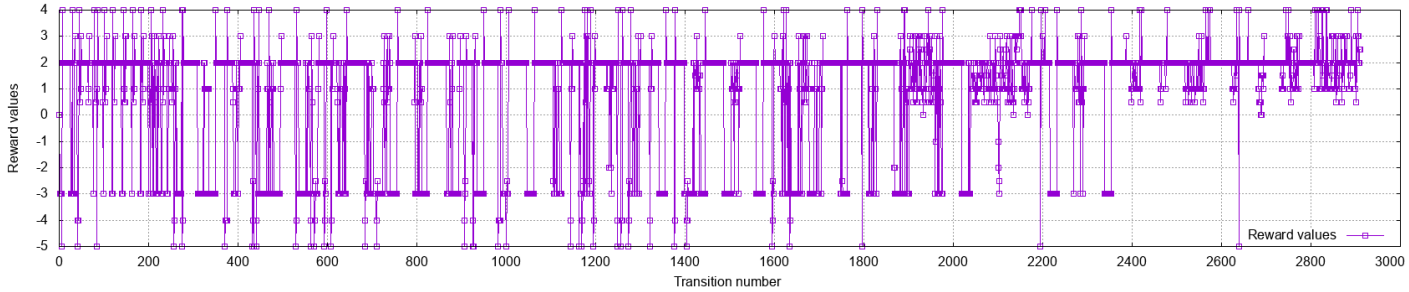
(a) Service availability during transitions



(b) Average response time during transitions



(c) Service calls during transitions



(d) Reward function during transitions

Figure 6.8: Performance metrics after transformation of the application during online one-step learning phase

6.4.3.3 After the Transformation with Online Multi-Step Learning

Similarly, we integrated the ECA rules to the framework to initialize the Q value function then transformed it with the multi-step learning algorithm. Afterwards, we deployed the transformed application and executed it with random starting state of the Catalog component. With the following experiments, we intended to evaluate the SLA guarantee and learning optimization criteria of the transformed application. Thereby, we run the multi-step learning algorithm and assessed these criteria under the same context dynamics. We previously determined the RL parameters that guarantee best results for our application and set them to $\gamma = 0.9, \alpha_0 = 0.2, \epsilon = 0.3, \lambda = 0.8, \epsilon_z = e - 4$.

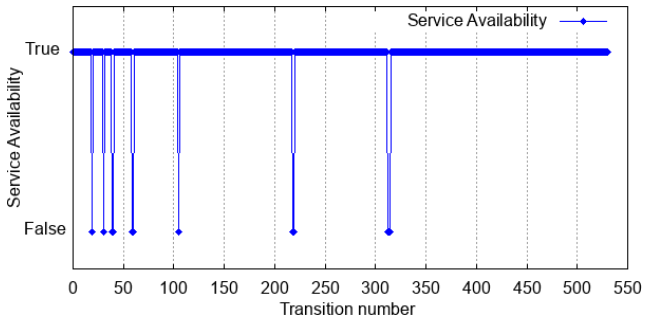
Figure 6.9 shows the metrics values observed at execution time of the service availability (Figure 6.9a), the average response time (Figure 6.9b) and the simultaneous service calls (Figure 6.9c). It also shows the reward values (Figure 6.9d) at execution time which reflects the improvement and degradation of the Catalog state.

At the beginning of learning, as shown in this figure, the metrics values violate the SLA terms less frequently compared to one-step learning. Furthermore, the multi-step algorithm started reducing its time spent in SLA violation around the 100^{th} transition for service availability (Figure 6.9a), the 150^{th} transition for average response time (Figure 6.9b) and the 200^{th} transition for service calls (Figure 6.9c). These results show that not only this algorithm started to self-adapt to the workload dynamics and the unexpected service breakdowns, it also showed that over the iterations, it quickly acquired enough experience on its context dynamics to circumvent bad action selection.

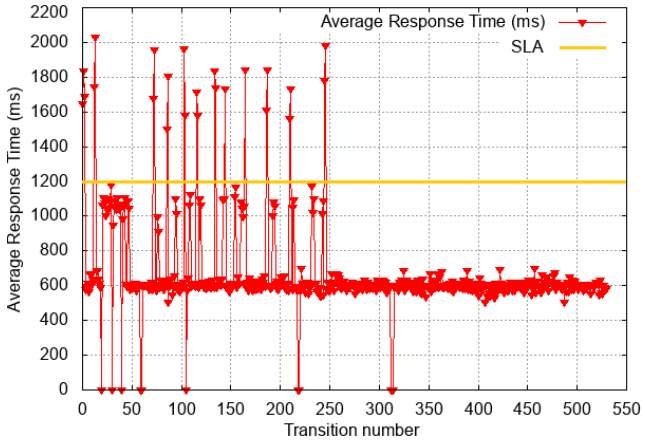
The multi-step algorithm reached convergence after 320 iterations. The convergence is proven by the Catalog's state that has stabilized starting from this transition and the accumulation of positive rewards due to the optimal decision actions it makes. This algorithm could self-adapt to its environment dynamics and also optimized its learning performance to finally optimize the application performance. Consequently, it significantly reduced the time spent in SLA violation even during the learning phase. This algorithm is best suited for real world applications with large state and action spaces.

It is worthy to note that to make its decisions during the learning phase, the Analysis component required the times: $average = 0.054ms$, $min = 0.009ms$, $max = 4.444ms$ and $standardDeviation = 0.115ms$. These values are valid for both one-step and multi-step learning.

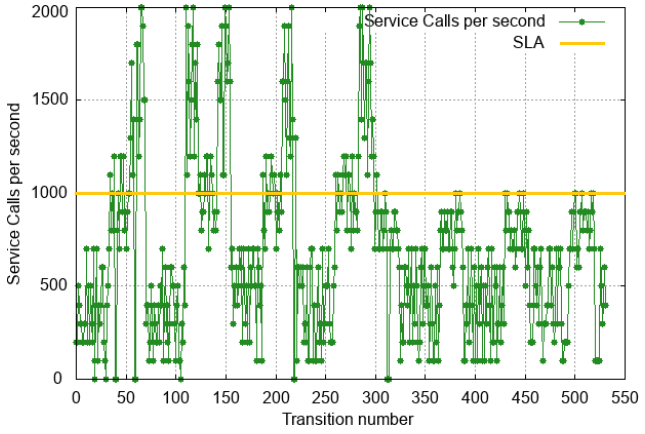
We conclude that our multi-step learning approach performs better than the one-step approach. It is best suited for real-world applications due to the reduced time spent in SLA violation during learning phase. It allows a fast self-adaptation to context dynamics for applications with large state and action spaces.



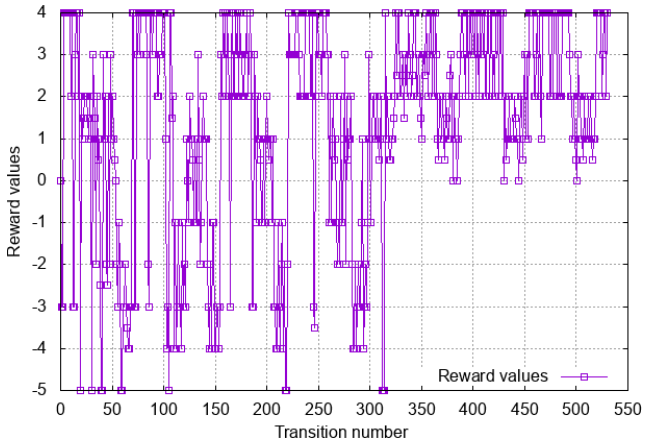
(a) Service availability during transitions



(b) Average response time during transitions



(c) Service calls during transitions



(d) Reward function during transitions

Figure 6.9: Performance metrics after transformation of the application during online multi-step learning phase

6.4.4 Synthesis

We deduce the effectiveness of the transformed application regarding self-adaptivity to context dynamics, hardly achievable without transformation. Indeed, in both one-step and multi-step scenarios, RL algorithms showed their effectiveness regarding their self-adaptivity to the changing context. However, an efficient meeting of quality requirements is hard to achieve for one-step learning. Spending a long time in SLA violation is not tolerated for real-world applications. When the state and action spaces are large, the one-step algorithm needs more time to visit all couples (s, a) several times before it actually learns a good quality policy.

The experimental results show clearly an efficient learning phase for the multi-step algorithm. Not only the algorithm can self-adapt to its context dynamics but it is also able to learn faster to optimize its performance and so as the application performance. Consequently, it reduces considerably the time spent in SLA violation even during the learning phase. We deduce that the multi-step learning approach outperforms the one-step approaches and quickly learns an optimal policy. It allows a fast self-adaptation to context dynamics for applications especially with large state and action spaces.

6.5 Conclusion

In this chapter, we proposed series of evaluations to validate our contributions. We demonstrated their feasibility and efficiency by performing different experiments on the framework and its multi-step learning algorithm that conducts its adaptation logic. The results showed that the autonomic container is lightweight by showing a negligible overhead before and after the transformation of the application. The results also showed a self-adaptive behavior of the transformed application to its context dynamics. They also showed a reduced time spent in SLA violation during the learning phase along with an optimized performance of the multi-step algorithm compared to the one-step algorithm. The evaluations showed promising results and proved the efficiency of our research work that we believe open several perspectives. In the near future, we target collaboration interactions between autonomic containers for a global and consistent management of Autonomic Computing Systems. We are currently working on collaboration algorithms for a consistent and collaborative learning of decision policies at execution time.

Chapter 7

Conclusion and Perspectives

Contents

7.1 Fulfillment of Contributions Objectives

Application developers have never had as much pressure and labor as they have nowadays. They are in constant race to provide both innovative and performing services. As consequence, their built applications have scaled in complexity resulting in increased manual management interventions to cope with service quality requirements. Autonomic Computing came as a relief to give room to autonomy rather than human intervention. However, the developers were caught again by the increased development labor of their adaptation aspects due to the inherent dynamic evolution of their applications owing to their dynamic contexts. To reduce the development labor of building nowadays Autonomic Computing Systems (ACS), two major aspects should be addressed. The first aspect is to enable ACS to learn from their past experiences and dynamically build their adaptation logic. The second aspect is to facilitate the construction of ACS to allow the developers to focus the efforts more on their business logic. In this thesis manuscript, we addressed these aspects by making the following proposals:

Regarding the first aspect, we proposed a learning-based approach to enhance the adaptation logic of ACS. In most cased, ACS are based on a static adaptation logic upon which an autonomic MAPE-K loop acts

to manage its associated Managed component. We proposed to enhance this logic by equipping the Analysis component of the MAPE-K loop with Reinforcement Learning (RL) capabilities. The application components can then dynamically and autonomously self-adapt their behavior to their context dynamics. To do so, we modeled the decision making process of the Analysis component as a Markov Decision Problem. We solved this problem by introducing Markov Decision Process's concepts that are related to our approach. We also introduced a reward function algorithm that we used to guide the Analysis component behavior. This function rewards good action selection leading to good performance and also punishes bad action selection leading to bad performance. We also proposed a decision learning algorithm that derived the Analysis behavior in dynamically learning its optimal decision policy. The RL algorithm that we used is based on a multi-step learning approach. It is a combination of Temporal Difference methods and Monte Carlo methods. This algorithm helped the Analysis component to learn from its past experiences and computed dynamically the optimal decision policy at execution time. When experimenting our approach, we found that the usage of a learning algorithm shows a self-adaptive behavior to context dynamics. The Experiments also showed that the multi-step algorithm converged faster compared to existing one-step learning approaches. As demonstrated, the multi-step algorithm outperformed the one-step algorithm by reducing the time the system might spend in SLA (Service Level Agreement) violations during learning phase.

Concerning the second aspect, we proposed an approach aiming to support the application developers in building the autonomic behavior of their applications. The objective is to reduce their development labor and help them focus more on their business requirements rather than the adaptation aspects. To do so, we proposed a generic framework to build self-adaptive applications by transforming their existing components to equip them with the needed autonomic behavior. The framework provides reusable and extensible functionalities that support the developers in transforming their existing applications. Its abstract design enabled extending and overriding the autonomic behavior to meet the specific management requirements of the functional components and their deployment contexts. We strengthened this framework with a design based on Service Component Architecture standard which allows its integration with different applications. The framework is designed as an autonomic container that holds the non-functional components of Proxy and MAPE-K, provided in separated components. The application is transformed by encapsulating its components in autonomic containers to render them self-adaptive. Depending on the management needs, this transformation is enabled on different granularity levels from a atomic to a composite component. Consequently, the transformed components can dynamically learn optimal policies at execution time based on their multi-step learning algorithms. We assessed the framework usage with a case study of a product composite application. We customized the framework to render it adequate to our management needs. Accordingly, we described the performed transformation after introducing the set of elements that we provided, extended and implemented. The experiments related to this transformation demonstrated that the container is lightweight by showing a negligible overhead during and after the application transformation.

7.2 Future Research Directions

Our research work opens several research perspectives that can be accomplished in short, middle and long terms.

7.2.1 Short Terms Perspectives

First, we plan to enhance our decision learning algorithm. More specifically its function $selectAction(s)$ introduced in Algorithm 2 (line 8). We need to integrate aspects related to the actions cost into the action selection process of this function. Actually, actions may have different costs, whether financial (i.e., in monetary value) or computational (e.g., CPU usage). For instance, a migration action is more expensive compared to a binding action. The migration action would consume more financial resources than the binding action since it involves the allocation of Cloud resources. For its execution, the migration action would also consume more computational resources than the binding action. The function $selectAction(s)$ could then select less expensive actions especially when they can both lead the system to approximately similar performances. For instance, in case a scale-out action and a migration action can provide similar results when the Managed component is overloaded, thus, the function should select a scale-out action, if it is less expensive. We think that this enhancement of the function $selectAction(s)$ will help optimizing the utilization of system resources.

We are aware that the case study that we experimented remains small compared to real-world, large and complex applications. We think that we should work farther on the evaluation aspects of the framework. We need to assess in details its genericity and usage for other applications examples and other deployment environments. We plan on evaluating the framework on large applications with larger number of components deployed on very dynamic and distributed environments such as the IoT (Internet of Things) and the Cloud.

7.2.2 Middle and long terms perspectives

As middle and long terms future work, we plan on improving the autonomic container behavior. To do so, we should make the autonomic container considering not only local decision making but also collaborative decision making with other containers for a global and consistent management of an entire distributed application.

7.2.2.1 Motivation and Research Problems

When managing a distributed complex application, we encapsulate each of its components/composites into an autonomic container for a distributed autonomic management. Indeed, using a single autonomic container for a centralized management represents a potential bottleneck. Instead of enhancing the system's performance, a centralized management may inversely cause performance degradation. Several autonomic containers are then necessary to properly manage an entire complex application.

Therefore, several autonomic containers may coexist in the system. Each, taking into consideration solely the management concerns of its encapsulated component and its deployment environment which both constitute its management scope. Thus, an autonomic container acts according to its own adaptation logic (during and after learning) that behaves to accomplish its specific management needs. Thereby, the containers act independently from each others when selecting actions during their decision making processes. This independence may affect each others performance by affecting their system quality (e.g., degraded response time). We define the system quality as its ability to comply with its Service Level Objectives (SLOs).

Actually, the application components maintain functional interactions that are promoted by the autonomic containers after their transformation and binded to compose the transformed application. By focusing solely on its management concerns, a container may end-up selecting a decision action that, certainly enhances the

performance of its system (i.e., Managed component), but can potentially degrade the systems performances of other containers and affect their functioning since they may be impacted by this decision. As they act independently, the containers are unaware of each others decisions. They may end-up selecting different or mismatched decisions which may cause either the failure of the management decisions and then compromise the application performance. They may also select contradictory decisions that may lead to corrupted results or instability of the system state and in worst cases causing failure of the application execution. The containers may also oscillate on their contradictory decisions causing decision instability.

These issues motivate the need to design autonomic containers that collaborate for a consistent management of an entire distributed application. They should collaborate to verify that their selected decision actions will not compromise each others systems performances. We need to enhance the autonomic containers behavior with collaboration algorithms to enable them to interact during their decision making processes. The purpose is to make them consider a global decision making in addition to their already existing local decision making.

7.2.2.2 Related Work

There have been research work undertaking studies on how to make several Autonomic Managers (AMs) collaborate to achieve a global management objective. Some of them have addressed either collaboration or coordination mechanisms in different environments, such as the Cloud, agent-based and distributed environments.

An early work where Li and Parashar presented Rudder [83], a peer to peer agent framework designed to perform element compositions (i.e., service compositions) for autonomic applications in decentralized and distributed environments. To perform the compositions, the framework offered an autonomic approach based on discovering, composing and controlling the elements along with a coordination approach for the agents. Two types of agents were introduced: the component agent (CA) and the composition agent (CSA). The agent CA uses profiles to identify elements and behavioral rules to control their behavior. A profile assembles functional and non-functional attributes of an element and is described using an XML-based ontology. The agent CSA discovers and composes dynamically the elements as applications. The authors addressed conflicting decisions that should be avoided by the agents by negotiating to determine a satisfying situation for both local and global scopes. To do so, the framework defines three protocols to enable coordination: (a) discovery protocol: allows the agents to register, unregister and discover the elements profiles using a specific ontology domain; (b) control protocol: enables the agents to control elements states by using control messages that are sent to an element for execution; (c) interaction protocol: aims to help a group of agents to negotiate, cooperate and interact. The negotiations between the CSAs are handled by the Contract-Net Protocol (CNP) [84] to enable them to discover, negotiate and select dynamically the most suitable CAs to compose the application elements. Nevertheless, the agents act according to predefined rules.

Mola and Bauer presented in [85, 86] their work that addressed SLA violations in the Cloud by improving the response time of a web server. Authors organized the AMs in a hierarchical manner where upper level AMs have more authority over lower level AMs. An AM of a lower level keeps allocating resources (i.e., Memory, CPU, Maximum number of clients) to its web server to enhance its response time. If no more allocations are possible then the AM requests another AM of upper level for further allocations. AMs of different Cloud layers

collaborate by exchanging predefined messages by means of an interface called ManagedObjects provided by each AM. This interface encapsulates properties, metrics and associated actions of the managed objects. A message broker is used between AMs of different levels to accept and forward the messages. However, a simple set of fixed and predefined ECA (Event Condition Action) policy was used to trigger predefined collaboration messages as events whenever a SLA violation occurs so that AMs of lower level ask AMs of upper level to perform the allocation. We think that the usage of a single broker between two layers of AMs may be source of bottleneck when overloaded. Furthermore, AMs of the same level do not collaborate and the AMs that do collaborate based on ECA rules. Furthermore, AMs of different levels collaborate based on ECA rules and AMs of the same level do not collaborate.

De Oliveira et al. [87, 88] proposed a framework for the coordination of multiple AMs in the Cloud. They introduced two types of AMs: Application AM (AAM) and Infrastructure AM (IAM). AAM is used to manage an application in Software-as-a-Service (SaaS) layer while IAM is responsible for optimal placement of VMs (Virtual Machines) on PMs (Physical Machines) in Infrastructure-as-a-Service (IaaS) layer. An AAM is associated to each application, whereas a single IAM operates for all IaaS resources. Each AAM maintains a local and private knowledge about its application while the IAM holds a global and shared knowledge. The global knowledge is split into critical and non-critical sections. The critical sections should not be accessed simultaneously by different AMs. Therefore, the authors proposed mechanisms of synchronization coordination based on predefined events and actions patterns (e.g., predefined ECA scenarios). AAMs collaboratively bring changes to the shared knowledge by means of a synchronization protocol to handle concurrency and render the global knowledge available to both SaaS and IaaS resources. The synchronization by assigning a token for each critical section of the knowledge and predefining a set of messages to pass the token between the AMs. The coordination mechanism is provided by some predefined coordination scenarios based on triggered events and their corresponding actions between the IAM and the AAMs. However, the AAMs propose solely a coarse-grained management for their applications. Furthermore, the management of IaaS level is centralized with the single IAM that is used which is undoubtedly source of bottleneck. Moreover, the coordination policy is based solely on predefined ECA coordination scenarios.

A learning-based approach was proposed to provide self-management properties with collaboration mechanisms. Mbaye and Krief [89] introduced collaborating knowledge planes to provide network components with self-management and collaborative abilities in distributed environments. They proposed to equip network components with self-organization and self-adaptation capabilities. The self-organization capability uses a sharing algorithm that enables neighboring network nodes to communicate by exchanging their knowledge and strategies they locally acquired. The sharing algorithm helps avoiding non-conflicting strategies and have a unified view of the system. The self-adaptation capability uses a learning algorithm based on Inductive Logic Programming (ILP) [90] to adapt the state of a network resource in order to compute the strategies. The learning algorithm allows acquiring new strategies when the system encounters unknown situations. The newly acquired strategies are examined to decide whether they can be kept or not. However, their learning approach required the availability of data samples or past experiences to build strategies upon them. Furthermore, we think that the sharing algorithm may not be applicable for different components with different management requirements.

7.2.2.3 Draft of Proposal: Collaboration Approach for a Global Management and Global System Quality

To propose an approach that enables the autonomic containers to collaborate during their learning phase, we first need to equip them with local and global definitions of their system quality. Thus, they will determine when they should collaborate for a global management of their system and when they do not need to collaborate.

We define the system quality as the system ability to meet its Service Level Objectives (SLOs) held in the Service Level Agreement (SLA). We determine the system quality by comparing the monitored values of the system state to the SLOs. As previously introduced in Chapter 4, the system state is composed of the monitored values of the metrics that are targeted in the SLA. To evaluate the system quality, we define a compliance degree function that compares the system state to the SLOs. The function verifies whether the monitored values of the metrics complies with (or violates) the SLOs to provide an indication on the improvement (or degradation) degree of the system quality. Based on this function, we previously classified the system quality to five classes according to their desirability order. These classes are ordered from the most to the least desirable based on the SLA (i.e., VERY-GOOD, GOOD, NEUTRAL, BAD, VERY-BAD).

An autonomic container is responsible for the management of its local system represented by its Managed component. It is equipped with a local compliance degree function that allows it to evaluate the system quality from its local point of view by comparing the monitored state of its Managed component against the SLOs. The local compliance degree function is a function that is used by the reward function to compute reward values in Algorithm 4.3.3. In a system where several containers coexist to efficiently manage the overall system, we need to evaluate the system quality for a set of containers from a global point of view. We define the global system quality as the ability of a group of autonomic containers to comply with their desired SLOs. We evaluate the global system quality by considering the local system quality at each container of the group. We then plan to define a global compliance degree function that uses the local compliance degree functions for each container to evaluate the global system quality. Depending on the needs, the global compliance degree function aggregates the local compliance degree functions either by: computing their average, conditional average or their weighted average in case some containers in the system are more important or prioritized than others. Finally, the global compliance degree function will assign the global system to a quality class.

Each container of the system is equipped with a local multi-step learning algorithm that estimates its own Q value function locally. We use the estimation of the containers' local Q value functions during learning phase to define the estimation of a global Q value function for a group of containers. We determine the estimation of the global Q value function by computing a function that aggregates the local Q value functions. This global function may be estimated by aggregating the local Q-value functions either by computing their average, conditional average or their weighted average in case some containers in the system are more important or prioritized than others. This global function can be very useful to consult by the containers for future action selections. They can compute and consult it before executing actions that may cause degraded global performance.

During the learning phase, a container makes decision actions and learns about them in order to compute locally its system quality. Actually, a decision action may be executed solely by a single container or collaboratively by several coexisting containers in the system. Note that these containers promote functional

interactions and for a consistent management, some actions are collaboratively executed. For instance, suppose that in a pervasive context a container detects a week network signal affecting its service delivery time. The container may generate a compressor adapter as a local action to compress the service request before sending it. The containers receiving the compressed response cannot consume it since they cannot decompress it. The container that initially executed the local compressor adapter action should have executed a collaborative action instead. In this collaborative action, we should have a local Management service call for the generation of compressor adapter action and remote Management service calls to ask for the execution of decompressor adapter actions from remote containers consumers. Thus, the Management services of a container should be remotely accessible by other containers to collaboratively execute collaborative actions.

An action may have either a local or a global impact on the system. An action has a local impact when it locally affects the Managed component of the container executing it. Otherwise, it has a global impact when it globally affects several Managed components and its execution involves several containers. We define the impact scope as the set of containers impacted by a decision action. The perspective of the global scope is what we use to delimit the system components for which we compute the global system quality at each collaborative decision action of a given container. Actually, we can determine the impact scope either by studying the historical data of the executed actions or by supposing that the impact scope is provided as an input. For simplicity reasons, we assume that the impact scope is provided as input by the framework user when developing its decision actions.

In our approach, several containers coexist to efficiently manage the entire system. To ensure a consistent decision making, each container should not only make decisions regarding its local system quality, it should also take into consideration the global system quality for the containers that are impacted by its decisions. Therefore, during learning and prior to executing a selected collaborative action, a container should verify that its execution will enhance the global system quality and not just its local system quality. To do so, the container should check the impact scope of its selected action to interact with the impacted containers (if any). Before deciding to execute the collaborative action, the container can request their local Q-value functions for the remote actions (of the collaborative action) they may execute in the future. The purpose is to compute the estimation of the global Q value function of all impacted containers. Depending on this global function, the containers decide to execute the collaborative action. Otherwise, the container that chose initially this collaborative action should select another action with a better estimation of the global Q value function.

In case the containers decided to execute this collaborative action, the containers need to learn collaboratively about its actual effectiveness. The objective is to learn whether this action has effectively enhanced the global system quality as estimated or not. Thereby, the container (that selected the collaborative action) interacts with the impacted containers to request their current local system qualities in order to compute the current global system quality. The previous global system quality should have been computed previously so, the container compares it with the current one. This comparison allows it to compute a collaborative reward in order to learn about the effectiveness of its collaborative action. The collaborative reward is used to update the local Q-value functions of all the impacted containers. The objective is to take into consideration what they learned from this action for future collaborative decisions.

Appendices

Appendix A

List of publications

1. **Nabila Belhaj**, Djamel Belaïd, Hamid Mukhtar, Framework for Building Self-Adaptive Component Applications based on Reinforcement Learning. In: IEEE World Congress on Services, IEEE International Conference on Services Computing (SCC), San Francisco, California, USA, July 2-7, pp. 17-24, (2018).

2. **Nabila Belhaj**, Djamel Belaïd, Hamid Mukhtar, Self-Adaptive Decision Making for the Management of Component-based Applications. In: On The Move, 25th International Conference on Cooperative Information Systems (CoopIS), Rhodes, Greece, October 23-27, pp. 570–588, (2017).

3. **Nabila Belhaj**, Imen Ben Lahmar, Mohamed Mohamed, Djamel Belaïd, Collaborative Autonomic Management of Distributed Component-based Applications. In: On The Move, 23rd International Conference on Cooperative Information Systems (CoopIS), Rhodes, Greece, October 26-30, pp. 3–18, (2015).

4. **Nabila Belhaj**, Imen Ben Lahmar, Mohamed Mohamed, Djamel Belaïd, Collaborative Autonomic Container for the Management of Component-based Applications. In: IEEE 24th International Conference on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), Larnaca, Cyprus, June 15-17, pp. 41-43, (2015).

Bibliography

[1] OASIS, "Service Component Architecture Assembly Model Specification Version 1.1," OASIS Standard, Tech. Rep., March 2009, available online. [Online]. Available: http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd03.pdf xv, 13

[2] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology," Tech. Rep., 2001. xv, 14, 15

[3] O. Sigaud and O. Buffet, *Markov Decision Processes in Artificial Intelligence*. Wiley-IEEE Press, 2010. xv, 15, 16, 50, 90

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction. Second edition, in progress.* A Bradford Book, The MIT Press, Cambridge, Massachusetts, London, England, 2012. xv, 18, 19

[5] J. Soldani, T. Binz, U. Breitenbücher, F. Leymann, and A. Brogi, "ToscaMart: A method for adapting and reusing cloud applications," *Journal of Systems and Software*, vol. 113, pp. 395–406, 2016. [Online]. Available: https://doi.org/10.1016/j.jss.2015.12.025 xv, 23

[6] IBM, "An Architectural Blueprint for Autonomic Computing," Tech. Rep., 2005. xv, 14, 24

[7] J. Adamczyk, R. Chojnacki, M. Jarzab, and K. Zielinski, *Rule Engine Based Lightweight Framework for Adaptive and Autonomic Computing*. Springer-Verlag, 2008. xv, 5, 24, 25, 26, 37, 38, 40

[8] C. Ruz, F. Baude, and B. Sauvan, "Component-based Generic Approach for Reconfigurable Management of Component-based SOA Applications," in *3rd International Workshop on Monitoring, Adaptation and Beyond*, 2010. xv, 5, 24, 25, 26, 37, 38, 40

[9] M. B. Alaya and T. Monteil, "FRAMESELF: A Generic Context-Aware Autonomic Framework for Self-Management of Distributed Systems," in *WETICE*. IEEE Computer Society, 2012, pp. 60–65. xv, 5, 26, 27, 37, 38, 40

[10] S. Frey, A. Diaconescu, D. Menga, and I. M. Demeure, "Towards a Generic Architecture and Methodology for Multi-goal, Highly-distributed and Dynamic Autonomic Systems," in *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*. USENIX Association, 2013, pp. 201–212. xv, 5, 27, 28, 38, 40

[11] F. Paraiso, P. Merle, and L. Seinturier, "soCloud: a service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds," *Computing*, vol. 98, no. 5, pp. 539–565, 2016. xv, 5, 28, 29, 37, 38, 40

[12] L. Beishui, Y. Yuan, and G. Ji, "Conceptual Model and Realization Methods of Autonomic Computing," *Journal of Software*, vol. 19, pp. 779–802, April 2008. xv, 30

[13] S. M. K. Gueye, N. D. Palma, and E. Rutten, "Component-Based Autonomic Managers for Coordination Control," in *COORDINATION*, 2013. xv, 5, 31, 32, 38, 40

[14] J. Cano, G. Delaval, and E. Rutten, "Coordination of ECA rules by verification and control," in *COORDINATION*, 2014. xv, 5, 32, 38, 40

[15] Gerald Tesauro, "Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies," *IEEE Internet Computing*, 2007. xv, 6, 35, 38, 40

[16] M. Amoui, M. Salehie, S. Mirarab, and L. Tahvildari, "Adaptive Action Selection in Autonomic Software Using Reinforcement Learning," in *Fourth International Conference on Autonomic and Autonomous Systems , ICAS*, March 2008, pp. 175–181. xv, 6, 36, 38, 40

[17] P. Nawrocki and B. Sniezynski, "Adaptive Service Management in Mobile Cloud Computing by Means of Supervised and Reinforcement Learning," *Journal of Network and Systems Management*, vol. 26, no. 1, pp. 1–22, Jan 2018. [Online]. Available: https://doi.org/10.1007/s10922-017-9405-4 xv, 6, 36, 37, 38, 40

[18] M. P. Papazoglou, "Service-oriented computing: concepts, characteristics and directions," in *Proceedings of the Fourth International Conference on Web Information Systems Engineering*, Dec. 2003, pp. 3–12. 1, 12

[19] IBM, *An Introduction to Creating Service Component Architecture Applications in Rational Application Developer Version 8.0*, 2010. 1, 12, 60, 61

[20] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," 2003. 2, 14, 44, 45

[21] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," in *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, ser. ICAC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 65–73. [Online]. Available: https://doi.org/10.1109/ICAC.2006.1662383 3, 4, 6, 35, 38, 40, 44

[22] M. R. Nami and K. Bertels, "A survey of autonomic computing systems," in *The 3rd International Conference on Autonomic and Autonomous Systems.*, 2007. 3, 44

[23] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola,

J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, B. Schmerl, D. B. Smith, J. a. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer-Verlag, 2013. 3, 44

[24] L. P. Kaelbling, M. L. Littman, and A. P. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, 1996. 3, 15, 44

[25] M. L. Puterman, *Markov Decision Processes: discrete stochastic dynamic programming*, 1994. 4, 15, 44, 46, 52

[26] S. M. LaValle, "Sequential Decision Theory," in *Planning Algorithms*. Cambridge University Press, 2006, ch. 10. 4, 44

[27] R. S. Sutton, "Learning to predict by the methods of temporal differences," in *MACHINE LEARNING*. Kluwer Academic Publishers, 1988, pp. 9–44. 4, 7, 18, 34, 54

[28] Oracle, "Java Management Extensions Specification, version 1.4," Sun Microsystems, Tech. Rep., November 2006. 5, 22, 37, 38, 40

[29] A. Brogi, A. Canciani, J. Soldani, and P. Wang, "Modelling the Behaviour of Management Operations in Cloud-based Applications," in *PNSE @ Petri Nets*, ser. CEUR Workshop Proceedings, vol. 1372. CEUR-WS.org, 2015, pp. 191–205. 5, 26, 37, 38, 40

[30] R. Agrawal, M. Behrens, B. R. Byreddy, M. Carlson, and al., "Cloud Application Management for Platforms Version 1.2," Tech. Rep., 2017. 5, 23, 37, 38, 40

[31] M. B. Alaya and T. Monteil, "FRAMESELF: an ontology-based framework for the self-management of machine-to-machine systems," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 6, pp. 1412–1426, 2015. 5, 26, 37, 38, 40

[32] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, and A. F. Yassin, "A Practical Guide to the IBM Autonomic Computing Toolkit RedBook," Tech. Rep., April 2004, available online. 5, 24, 38, 40

[33] OASIS, "Topology and Orchestration Specification for Cloud Applications Version 1.0," Tech. Rep., 2013. 5, 22, 37, 38, 40

[34] S. Frey, A. Diaconescu, and I. Demeure, "Architectural integration patterns for autonomic management systems," in *9th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems (EASe)*, 2012. 5, 27, 38, 40

[35] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, "No power Struggles: Coordinated Multi-level Power Management for the Data Center," in *ASPLOS'13*, 2008. 5, 30, 38, 40

[36] S. Lightstone, M. Surendra, Y. Diao, S. S. Parekh, J. L. Hellerstein, K. Rose, A. J. Storm, and C. Garcia-Arellano, "Control Theory: a Foundational Technique for Self Managing Databases," in *ICDE Workshops*. IEEE Computer Society, 2007, pp. 395–403. 5, 30, 38, 40

[37] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, "A Control Theory Foundation for Self-managing Computing Systems," in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, April 2005, pp. 2213–2222. 5, 30, 38, 40

[38] R. Abid, G. Salaün, and N. D. Palma, "Asynchronous synthesis techniques for coordinating autonomic managers in the cloud," *Sci. Comput. Program.*, vol. 146, pp. 87–103, 2017. [Online]. Available: https://doi.org/10.1016/j.scico.2017.05.005 5, 33, 38, 40

[39] S. M. K. Gueye, N. D. Palma, and E. Rutten, "Coordinating energy-aware administration loops using discrete control," in *The Eighth International Conference on Autonomic and Autonomous Systems, ICAS 2012*, 2012, pp. 99–106. 5, 31, 38, 40

[40] H. Wang, X. Zhou, X. Zhou, W. Liu, W. Li, and A. Bouguettaya, "Adaptive Service Composition Based on Reinforcement Learning," in *ICSOC*, 2010. 6, 34, 38, 40

[41] H. Wang, X. Wang, X. Hu, X. Zhang, and M. Gu, "A multi-agent reinforcement learning approach to dynamic service composition," *Inf. Sci.*, vol. 363, pp. 96–119, 2016. 6, 34, 38, 40

[42] G. Tesauro, "Online Resource Allocation Using Decompositional Reinforcement Learning," in *Proceedings, AAAI'05*, 2005. 6, 34, 38, 40

[43] J. Rao, X. Bu, K. Wang, and C.-Z. Xu, "Self-adaptive Provisioning of Virtualized Resources in Cloud Computing," in *SIGMETRICS'11*, 2011. 6, 35, 38, 40

[44] O. Sigaud, O. Buffet, and A. e. a. Dutech, *Processus décisionnels de Markov en intelligence artificielle*, 2008. 7, 16, 17, 18, 34, 46, 51, 52, 54, 89

[45] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web Services Architecture," *W3C Working Group Note*, vol. 11, pp. 2005–1, 2004. 12

[46] S. Burbeck, "The Tao of e-business services: The evolution of Web applications into service-oriented components with Web services," IBM Software Group, 2000. [Online]. Available: http://www.citeulike.org/user/tvolland/article/1688947 12

[47] IBM, *IBM WebSphere Application V7 Feature Pack for Service Component Architecture*. IBM, December 2008. [Online]. Available: http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.wasfpsca/wasfpsca/1.0/Overview/WASv7SCA_Overview_sca_Introduction/player.html 13

[48] M. C. Huebscher and J. A. McCann, "A Survey of Autonomic Computing - Degrees, Models, and Applications," 2008. 14

[49] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2003. 15

[50] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction*. A Bradford Book, The MIT Press, 2017. 17, 18, 19, 51, 52

[51] P. Gerard, "Apprentissage par Renforcement: Apprentissage Numérique," 2008. 17

[52] OASIS, "Web Services Business Process Execution Language Version 2.0," Tech. Rep., April 2007, available online. 23, 48

[53] OMG, "Business Process Model and Notation (BPMN) Version 2.0," Tech. Rep., 2011. 23

[54] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – A Runtime for TOSCA-based Cloud Applications," in *11th International Conference on Service-Oriented Computing*, ser. LNCS. Springer, 2013. 23

[55] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneijder, and L. A. Stein, "OWL Web Ontology Language Reference," World Wide Web Consortium (W3C), Recommendation, February10 2004, see http://www.w3.org/TR/owl-ref/. 26

[56] "Etsi. ETSI ETSI TS 102 690 V1.1.1: Machine-to-Machine communications (M2M); Functional architecture (2011-10)," Tech. Rep. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/102600_102699/102690/01.01.01_60/ts_102690v010101p.pdf 26

[57] B. Michelson, "Event-Driven Architecture Overview," *Patricia Seybold Group, Feb*, 2006. 27

[58] S. Kanaparti, "A Distributed-SOA Model for Unified Communication Services." in *ICWS*. IEEE Computer Society, 2008, pp. 529–536. 27

[59] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, "GraphML Progress Report: Structural Layer Proposal," in *Proceedings of the 9th International Symposium Graph Drawing (GD '01) LNCS 2265*. Springer-Verlag, 2002, pp. 501–512. [Online]. Available: http://www.inf.uni-konstanz.de/algo/publications/behhm-gprsl-01.ps.gz 27

[60] H. Chen and C. Cao, "Research and Application of Distributed OSGi for Cloud Computing," in *International Conference on Computational Intelligence and Software Engineering (CiSE)*, December 2010, pp. 1–5. 27

[61] A. McEwen and H. Cassimally, *Designing the Internet of Things*, 1st ed. Wiley Publishing, 2013. 27

[62] W. L. Brogan, *Modern Control Theory (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991. 29

[63] G. F. Franklin, D. J. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. 29

[64] S. Martello, D. Pisinger, and D. Vigo, "The Three-Dimensional Bin Packing Problem," *Oper. Res.*, vol. 48, no. 2, pp. 256–267, Mar. 2000. [Online]. Available: http://dx.doi.org/10.1287/opre.48.2.256.12386 31

[65] A. Filieri, M. Maggio, K. Angelopoulos, N. D'ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel, "Control Strategies for Self-Adaptive Software Systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 11, no. 4, pp. 24:1–24:31, Feb. 2017. [Online]. Available: http://doi.acm.org/10.1145/3024188 31, 38, 40

[66] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, "Self-managing systems: A control theory foundation," in *In ECBS Workshop*, April 2005, pp. 441–448. [Online]. Available: https://ieeexplore.ieee.org/document/1409947/ 31

[67] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Berlin, Heidelberg: Springer-Verlag, 2006. 31

[68] INRIA-Institute, *Heptagon/BZR Manual*, October 2013. [Online]. Available: http://bzr.inria.fr/pub/bzr-manual.pdf 31

[69] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic, "Synthesis of Discrete-Event Controllers based on the Signal Environment," *Discrete Event Dynamic System: Theory and Applications*, vol. 10, no. 4, pp. 325–346, October 2000. 33

[70] INRIA-Institute, *Reference Manual of the LNT to LOTOS Translator, (Formerly: Reference Manual of the LOTOS NT to LOTOS Translator)*, January 2018, available online. [Online]. Available: ftp://ftp.inrialpes.fr/pub/vasy/publications/cadp/Champelovier-Clerc-Garavel-et-al-10.pdf 33

[71] H. Garavel, F. Lang, and R. Mateescu, "Compositional verification of asynchronous concurrent systems using CADP," *Acta Inf.*, vol. 52, no. 4-5, pp. 337–392, 2015. [Online]. Available: https://doi.org/10.1007/s00236-015-0226-1 33

[72] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998. 35

[73] N. Fernando, S. Loke, and W. Rahayu, "Mobile cloud computing: A Survey," *Future Gener. Comput. Syst.*, pp. 84–106, 2013. 36

[74] S. B. Kotsiantis, "Supervised Machine Learning: A Review of Classification Techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2007, pp. 3–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=1566770.1566773 36

[75] P. Bianco, G. Lewis, and P. Merson, "Service Level Agreements in Service-Oriented Architecture Environments," Tech. Rep., 2008. 44

[76] P. Lalanda, J. A. McCann, and A. Diaconescu, *Autonomic Computing - Principles, Design and Implementation*, ser. Undergraduate Topics in Computer Science. Springer London, 2013. 45

[77] R. Bellman, *Dynamic Programming*, 1st ed. Princeton, NJ, USA: Princeton University Press, 1957. 53

[78] R. Bellman and S. Dreyfus, *Functional Approximation and Dynamic Programming*. Math. Tables and other Aids Comp., 1959, vol. 13. 53

[79] C. J. C. H. Watkins and P. Dayan, "Q-learning," in *Machine Learning*, 1992, pp. 279–292. 57

[80] Philip K. McKinley, Seyed Masoud Sadjadi, Betty H.C. Cheng, and Eric P. Kasten, "Composing Adaptive Software," 2004. 61

[81] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani, "Reconfigurable SCA Applications with the FraSCAti Platform," in *Proceedings of IEEE International Conference on Services Computing.*, ser. SCC'09, Bangalore, India, 2009, pp. 268–275. 77

[82] S. Pat, "Oracle SCA - The Power of the Composite," An Oracle White Paper, Tech. Rep., August 2009. 84

[83] Z. Li and M. Parashar, "A Decentralized Agent Framework for Dynamic Composition and Coordination for Autonomic Applications," in *DEXA Workshops*, 2005. 102

[84] R. G. Smith, "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver," *IEEE Trans. Comput.*, vol. 29, no. 12, pp. 1104–1113, Dec. 1980. [Online]. Available: http://dx.doi.org/10.1109/TC.1980.1675516 102

[85] O. Mola and M. Bauer, "Collaborative Policy-based Autonomic Management: In a Hierarchical Model," in *CNSM*, October 2011, pp. 1–5. 102

[86] ——, "Towards Cloud Management by Autonomic Manager Collaboration," in *International Journal of Communications, Network and System Sciences*, 2011, pp. 790–802. 102

[87] J. F. A. De Oliveira, R. Sharrock, and T. Ledoux, "Synchronization of Multiple Autonomic Control Loops: Application to Cloud Computing," in *Coordination Models and Languages - 14th International Conference, COORDINATION 2012, Stockholm, Sweden, June 14-15, 2012. Proceedings*. Springer, 2012, pp. 29–43. 103

[88] J. F. A. De Oliveira and R. Sharrock, "A Framework for the Coordination of Multiple Autonomic Managers in Cloud Environments," in *7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO*, September 2013, pp. 179–188. 103

[89] M. Mbaye and F. Krief, "A Collaborative Knowledge Plane for Autonomic Networks," in *Autonomic Communication*. Springer US, 2010, pp. 69–92. 103

[90] S. Muggleton and L. D. Raedt, "Inductive Logic Programming: Theory and Methods," *JOURNAL OF LOGIC PROGRAMMING*, pp. 629–679, 1994. 103