Emergent Software Systems

Roberto Rodrigues Filho

BSc. Computer Science, Federal University of Goiás, Brazil (2010)
MSc. Computer Science, Federal University of Goiás, Brazil (2013)

A Thesis presented for the degree of
Doctor of Philosophy

School of Computing and Communications
Lancaster University, UK

January, 2018

With love to Carmen and Roberto

Emergent Software Systems

Roberto Rodrigues Filho

Submitted for the degree of Doctor of Philosophy
January, 2018

Abstract

Contemporary software systems often have millions of lines of code that interact over complex infrastructures. The development of such systems is very challenging due to the increasing complexity of services and the high level of dynamism of current operating environments. In order to support the development and management of such systems, autonomic computing concepts have gained significant importance.

The majority of autonomic computing approaches show significant levels of expert dependency in designing adaptive solutions. These approaches usually rely on human-made models and policies to support and guide software adaptation at runtime. These approaches mainly suffer from: i) a significant upfront effort demanded to create such solutions, which adds to the complexity of creating autonomous systems, and ii) unreliability given the high levels of uncertainty in current operating environments, leading the system to degraded performance and error states when subjected to unpredicted operating conditions and unexpected software interactions.

Motivated by the problems and limitations of state-of-the-art autonomic computing solutions, this thesis introduces the concept of Emergent Software Systems. These systems are autonomously composed at runtime from discovered components, and are autonomously optimised based on the operating conditions, being able to build their own understanding of their environment and constituent parts. This thesis defines Emergent Software Systems, presenting the challenges of implementing such approach, and presents a fully functioning emergent systems framework that demonstrates this concept in real-world, fully functioning datacentre-based software.

Declaration

I declare that the work undertaken in this thesis is my own and has not been submitted in any other form for the award of a research degree.

This work has been conducted within the context of the "Deep Online Cognition in Modular Software" project. The entire emergent framework modules (both the local and distributed framework) were entirely written by me. Also, the datacentre software (load balancer and web cache) were also entirely developed by me. The web server, in particular, was extended by me from the open source web server example made available with the Dana programming language. Finally, the clients, used in the evaluation of this thesis, were extended by me from a simple HTTP client application. The HTTP client application, the web server example and the Dana language were developed by Dr. Barry Porter, who directly supervised this work.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Barry Porter for his invaluable advice, and great support in the realisation of this work. Barry is a great mentor and has become a dear friend. I would also like to thank my supervisor Prof. Gordon Blair for giving me the opportunity to join his research group, and for his support and crucial feedback on all important milestones in the PhD process.

I would like to thank all my colleagues in the School of Computing and Communications who made this journey less lonely and more enjoyable. In special I would like to thank Dr. Vatsala Nundloll, Dr. Faiza Samreen, Dr. Sirak Kaewjamnong, Dr. Yehia Elkhatib, Dr. Izhar Ullah, Dr. Leandro Marcolino, Dr. Rodrigo Siqueira, Dr. Vicent Sanz, Willy Wolff, James Hadley, Ben Taylor, Alex Wild, Wyatt Lindquist, the Metalab and DSG group for the fruitful discussions and the enriching feedback.

I am also deeply grateful to all my friends and family in Brazil who supported me throughout this journey. In special I would like to thank my dear wife Débora Caetano for supporting and motivating me in my decision to pursuit a PhD degree. I would like to thank my parents and brother for their unconditional love and support, Taciano Moraes for the fun online chats we had in the difficult moments, and Cheryl Towey who helped me throughout a good portion of my PhD studies with our weekly philosophical chats and exercise training sessions.

Finally, I would like to thank *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES) Brazil for the grant Proc. BEX 13292/13-7. This thesis would not have been possible without their financial support.

Thank you all,

Roberto Rodrigues Filho

Publications

- Defining emergent software using continuous self-assembly, perception and learning. **Rodrigues Filho, R.** & Porter, B. *ACM Transactions on Autonomous and Adaptive Systems. 12, 3, 25 p. 16 (TAAS) (2017)*

- Experiments with a machine-centric approach to realise distributed emergent software systems. **Rodrigues Filho, R.** & Porter, B. *Proceedings of ACM 15th Int. Workshop on Adaptive and Reflective Middleware (ARM) (2016)*

- Losing control: the case for emergent software systems using autonomous assembly, perception and learning. Porter, B. & **Rodrigues Filho, R.** *IEEE 10th Int. Conf. on Self-Adaptive and Self-Organizing Systems (SASO) (2016).*

- REX: a development platform and online learning approach for Runtime emergent software systems. Porter, B., Grieves, M., **Rodrigues Filho, R.** & Leslie, D. S. *Proceedings of the 12th Symposium on Operating Systems Design and Implementation. USENIX Association, p. 333-348 (OSDI) (2016)*

- A runtime framework for machine-augmented software design using unsupervised self-learning. **Rodrigues Filho, R.** & Porter, B. (Poster) *IEEE International Conference on Autonomic Computing p.231-232 (ICAC) (2016)*

- Demonstrating a runtime machine-centric emergent software architecture framework. **Rodrigues Filho, R.** & Porter, B. (Demo) *IEEE International Conference on Autonomic Computing p.239-240 2 (ICAC) (2016)*

- Environmental IoT: programming cyber-physical clouds with high-level system specifications. **Rodrigues Filho, R.**, Porter, B. & Blair, G. (Workshop) *IEEE/ACM 7th Int. Conf. Utility and Cloud Computing (UCC) (2014)*

"Intelligence is the ability to adapt to change."

Stephen Hawking

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Overview

Contemporary software systems have reached a level of complexity that is beyond human capacity to fully understand [53]. Besides the innate complexity of software development [13], which continues to be a problem, systems are becoming larger, with millions of lines of code, and are deployed on highly dynamic and complex infrastructures, running on distributed heterogeneous platforms across the globe. In addition, these systems require high levels of scalability, reliability and security to deliver services to millions of users worldwide. Building and managing systems to maintain such properties using classical engineering approaches is becoming inefficient [19,50,53]. For instance, fluctuations on incoming workload yield unpredictable resource utilisation demands and complex interactions among software modules.

In classical software engineering approaches, which do not aim at creating self-management software solutions, the design decisions are made in the design phase and often become invalid at runtime due to unpredictable fluctuations in the operating environment. Furthermore, management decisions are made entirely by experts, who demand time to profile the system and analyse historical workloads to create models to predict workload changes before manually optimising the system. This scenario makes classical engineering methods unsuitable as the level of uncertainty increases in operating environments of contemporary software systems [19,32,50,53].

In this context, a desired property of contemporary systems is the ability to

autonomously change their behaviour to accommodate changes in the environment. This property is known as self-adaptation [76]. The most relevant idealisation of autonomous software management and its related challenges were defined by IBM in 2001 [43, 53]. The authors defined the term Autonomic Computing and motivated subsequent research to tackle the challenges in building systems capable of self-management. However, equipping software systems with self-adaptive properties has consequences. The development of autonomic systems is very complex and they often present undesired emergent properties and behaviours [53, 74]. The complexity of creating self-adaptive solutions involves the development of self-adaptive mechanisms and strategies to guide software adaptation and evolution at runtime.

Current approaches handle these complexity aspects by applying adaptation mechanisms to only specific parts of the system and using predefined models and policies to guide online software adaptation. Although, this strategy to handle complexity in designing self-adaptive systems is a first step, it is not sufficient to cope with the increasing levels of complexity in contemporary systems. The use of policies and models to guide software adaptation rely on predictions that might not happen, leading systems to error and malfunctioning states. Furthermore, the dependence of policies might result in undesired emergent properties due to unforeseen events in the operating environment and unpredicted effects of software interactions. Finally, these predefined models and policies require upfront human effort in creating such systems, which adds to the complexity and cost in creating autonomous solutions.

The existence of undesired behaviours that emerge from unpredicted interactions among self-adaptive software modules prompted the creation of the organic computing research field [74, 77]. The goal of this community is to study mechanisms to control undesired emergent behaviours whilst exploiting the benefits of self-adaptive software systems. However, this thesis argues against such controlling mechanisms because they restrict the potential of self-adaptive software, limiting the software's ability to autonomously deal with unforeseen situations. Instead, this thesis argues for the creation of self-adaptive systems with no predefined adaptation rules and the application of self-adaptive mechanisms not only to specific parts of the system but rather to the composition process of the entire system.

Therefore, this thesis defines a new engineering approach where software systems are composed out of small units of behaviours, which enable the system to actively experiment with a variety of component variations in order to assemble optimal software compositions to handle different operating environments. This approach embraces the emergent properties and behaviour in systems as a way to manage software complexity, and handles undesired behaviour through a continuous runtime learning process, where the software itself learns according to high level goals, and observations of the system health status whether its composition is acceptable.

As a consequence, this approach handles the problem of system complexity by handling uncertainties in the operating environment and interactions among the software modules, as well as addressing the complexity aspects of creating self-adaptive systems by reducing the influence of humans in the design and management of software systems. This approach is named Emergent Software Systems.

The 'emergent systems' term is not precisely defined in the literature and is usually used as a characteristic of systems with emergence properties. In this thesis, emergence means: i) to autonomously compose software from small units of behaviour, where the resulting combination of software units is greater than the sum of its parts, ii) to autonomously compose software as a result of operating conditions, and iii) the possibility of the approach to autonomously find unexpected optimal compositions. Furthermore, our approach is intended as a method for creating Autonomic Computing systems and to increase their capability of self-management. This thesis discusses the proposed approach in the context of datacentre-based software.

1.2 Critique of Existing Approaches

Autonomic Computing approaches were the focus of several research in the literature tackling a variety of challenges and issues to provide self-adaptive mechanisms to software systems [18, 47, 60, 86]. An analysis of the most relevant research shows that they share one or more of the following characteristics: i) significant human-dependency in their design, ii) assurance mechanisms for adaptation, and finally, iii) adaptation mechanisms are applied to very specific parts of the system.

The majority of self-adaptive solutions show significant levels of expert dependency in designing adaptive solution. These approaches usually rely on models and policies to support and guide software adaptation at runtime [48, 55]. These design decisions are made at design time which results in two main problems.

- These systems suffer from high levels of uncertainty in contemporary operating environments [32];
- These approaches require significant upfront effort to develop [32].

The dynamism in contemporary operating environments leads systems to an error and malfunctioning state when they are subjected to events not predicted in the predefined adaptation policies. Also, as systems become larger, the task to define models to guide software adaptation becomes increasingly complex to a point where is unwieldy to manually specify all adaptation rules, making such approaches not scalable, due to increasing upfront effort required to develop such approaches [32].

Assurance mechanisms in self-adaptive systems sound very appealing at first, because they promise to maintain the system within a safe operating state. However, there are two main concerns with those mechanisms. The first is that the provision of assurance to self-adapting systems is very complex. In many systems a combination of valid behaviours might result in undesired behaviour, and it is difficult to anticipate such combinations before they happen [18, 32]. The second problem is the presence of controlling mechanisms to support assurance in self-adaptive systems. This is because they constrain the systems' ability to freely explore and find more suitable compositions to better handle their operating environment, limiting the benefits of incorporating self-adaptive mechanisms to the system.

Finally, the adaptation mechanisms are usually applied to very specific parts of the system, for example, to support adaptive mechanisms to control process allocation on web servers [60], or to enable a self-managing load balancer for memcached servers [86]. Although this is a safe research strategy because it isolates specific aspects of software adaptation for better exploration, it requires experts to predict which parts of the system might need to adapt. This thesis argues that contemporary software requires the application of self-adaptive techniques in the development process and management of the entire system, independent of its application domain.

1.3 Research Goals

Classical software engineering methods, which do not consider autonomous software management, are becoming inadequate as the levels of system complexity increases [53]. In this context, this thesis explores the application of self-composition concepts in the software development process, where software systems are the result of their operating environments, constantly building and optimising their own structure. More specifically this thesis aims to validate the hypothesis stated below:

> **The use of fine-grained software components in tandem with reinforcement learning as a method to develop self-adaptive software systems is key to address operating environment dynamism whilst minimising design complexity.**

Component-based technologies [21, 70] have been demonstrated as a suitable solution to minimize software complexity by maximising software re-usability and maintainability [52]. These technologies are also the bedrock that seamlessly support runtime architectural software adaptation and evolution [21]. Complementarily, the application of reinforcement learning approach [79] allows systems to learn at runtime how their constituent parts act in diverse environments. In tandem, these two technologies have the potential to empower systems with the ability to understand their architecture and choose compositions depending on the operating conditions.

This work presents an engineering approach to software development and management that transcends the boundaries of current self-adaptive and autonomic computing approaches. Furthermore, this thesis demonstrates the feasibility of creating software capable of actively building its own understanding of its architecture whilst self-optimising according to its goals and operating environment. As these systems are autonomously composed as a product of their operating environment, this concept is named Emergent Software Systems.

The validation of the hypothesis consists of i) verifying that, for different operating environments, different software compositions have different performance optimality, and ii) showing that the system is able to accurately locate optimal available

composition for the different operating environment with no predefined models or domain-specific information. This thesis shows the validity of Emergent Software Systems in both local and distributed instances. The validation steps unfold into the following research questions:

[**RQ 1**] Do different software compositions have different performance when subjected to different operating environments?

[**RQ 2**] Is it possible to autonomously locate optimal software composition (within a set of available options) with no predefined nor domain-specific information?

[**RQ 3**] How can an autonomous system coordinate multiple instances of emergent software to converge to optimal available global compositions in distributed settings?

1.4 Methodology

The methodology used to conduct this research is a prototyping, experimental-driven and iterative approach. This approach was chosen for two main reasons: i) developing prototypes and designing experiments often forces one to consider important technical limitations, which otherwise would not be considered, that may directly affect the validity of the proposed concepts, and ii) an iterative methodology suits the explorative nature of defining the novel concepts presented in this thesis.

The methodology consists of executing the steps illustrated in Fig.1.1. The first step is the definition of a concept or idea for investigation (**1. Concept Definition / Refinement**), the proposed concept is then implemented (**2. Prototype creation**) and evaluated (**3. Prototype Evaluation**). The results of the evaluation are discussed with supervisors and research group and, if considered relevant, are written in the form of scientific paper and submitted to a high quality conference of the area (**4. Reporting/Discussing Findings**). The feedback from the discussion phase is then analysed (**5. Feedback Analysis**) and the original idea or concept is refined, triggering subsequent iteration for further investigation.

This thesis explores the Emergent Software System concept on two scenarios: i) local software and ii) distributed software. The first scenario explores the fundamen-

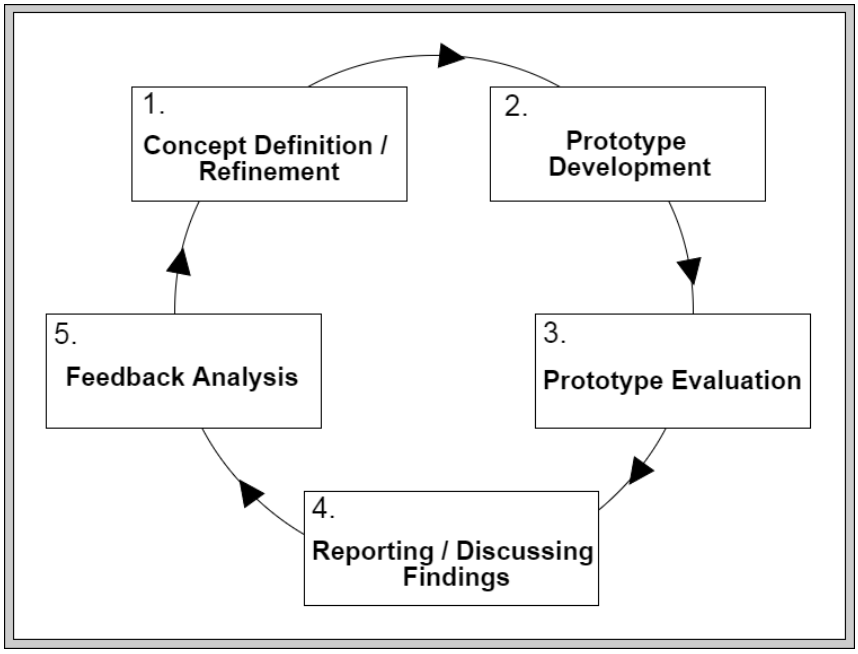

Figure 1.1: Phases of the iterative process.

tal concepts that form the basis of Emergent Software Systems focusing only on the autonomous composition of a single software instance. The second scenario explores Emergent Systems consisting of multiple interacting software instances, exploring learning coordination and emergent behaviour in distributed emergent systems.

The first step of this research was to define datacentre-based software as the case study. Specifically, web servers were chosen as the primary target to apply Emergent Software System concepts, due to their timely importance in contemporary systems, which are in their majority web-based systems and because of the notable difficulty in manually optimise and configure them [87].

During the exploration of the first scenario, a single instance of an emergent web server was created, which has the ability to autonomously self-assemble into a fully functional web server, and to self-optimise according to a variety of operating conditions, addressing the research questions RQ1 and RQ2 (see Sec. 1.3). The second scenario explores the autonomous assembling of entire distributed web platforms, where the roles of the participating machines emerge as a result of the natural self-composing property of emergent solutions, and hence the explored aspects are mainly related to research question RQ3, but also addresses RQ1 and RQ2 in a distributed scenario.

1.5 Contributions

The main contributions are i) the proposal of the Emergent Software System concept overcoming significant human dependency in creating self-adaptive systems, and ii) the validation of the concept, showing that different software compositions have different performances under different operating conditions, and that emergent systems are able to autonomously converge towards optimal performance. The proposal of the Emergent Software System concept is the result of extensive practical experiments using real-world datacentre-based software, making it a concept deeply grounded in reality. In addition, this thesis demonstrates the potential of emergent systems approach by minimising human efforts in creating autonomous systems, through a complete automated creation and evolution of systems' adaptation logic. More specifically this thesis:

- Presents a list of key challenges in realising Emergent Systems, whilst also presenting solutions to the most important identified challenges. This thesis presents solutions to: i) handling large search spaces resulted from the combinatorial nature of the autonomous software composition process, ii) online classification of dynamic operating environments, and iii) coordination of software composition and learning in distributed systems environment.

- Introduces a domain-independent framework to orchestrate Emergent Software Systems. The framework captures the essence of Emergent Software System in its main modules that are used to autonomously assemble software systems, monitor the operating environment and the systems health status, and learn about the systems operating environment and the systems constituent parts, with no predefined models or adaptation rules. The framework implementation is also made available for replication of the main results reported in this thesis, and can be downloaded at [33], [34], [35] and [36].

- Argues for a paradigm shift in the autonomous software creation process, by changing the focus from autonomous software adaptation to autonomous software composition. Software adaptation becomes a consequence of composition,

rather than the main focus for creating autonomous solutions. This paradigm shift is at the core of Emergent Systems, and reduces the up-front effort and complexity involved in designing autonomous solutions, by pushing design decisions to be made at runtime by the system itself.

- Applies the concept of Emergent Systems to create a complete emergent web platform. The emergent web platform does not require the classical process of optimising web platforms, which consists of: expert analysis of historical workload, profiling the web platform, the creation of models to predict workload changes, and a manual tuning of parameters of web servers, load balancers and web caches. Instead, the emergent web platform self-composes and self-optimises based on real-time observed workload.

1.6 Thesis Structure

The rest of this thesis is structured in the following manner:

- **Chapter 2** discusses the background for this thesis and presents a comprehensive analysis of recent and most relevant work in the literature.
- **Chapter 3** defines the concept of Emergent Software Systems, describing the problem space and challenges of implementing the concept.
- **Chapter 4** introduces the Emergent Software System framework, detailing the implementation of each of its modules and presenting the solution for the most important challenges described in the problem space (listed in Chapter 3).
- **Chapter 5** evaluates the framework presented in Chapter 4 applied to a web platform case study, demonstrating the practicality of Emergent Software Systems in both local and distributed software instances.
- **Chapter 6** concludes the thesis, highlighting the main contributions, and presenting future challenges to consolidate the Emergent System paradigm.

CHAPTER 2

Background and Related Work

This chapter introduces the main concepts discussed in this thesis, whilst also describing the most relevant work as application examples of such concepts. Furthermore, this chapter presents the shortcomings of current approaches contrasting with the concept of emergent software systems, as well as how these approaches complement and assist emergent software system to be consolidated as a complete method to develop autonomous solutions.

2.1 Autonomic Computing

This section describes the Autonomic Computing concept and vision, and its importance for understanding the novelty and contribution of Emergent Software Systems. This section starts by presenting the IBM vision for Autonomic Computing to tackle the increasing complexity in managing software systems. The concept of Organic Computing is also introduced in this section as a complementary research area that focuses on autonomous system solutions. Organic Computing solutions also aims at controlling the interaction of multiple autonomic systems to guide emerging solutions whilst restraining the occurrence of undesired behaviours. Finally, the concept of self-adaptive systems is also introduced as an umbrella term for autonomous systems, and as a fundamental property to enable different aspects of autonomous systems (e.g. self-optimising, self-healing, self-protecting, self-configuring).

2.1.1 IBM Vision

Motivated by the increasingly complexity in contemporary systems, IBM introduced the concept of Autonomic Computing [53] in a manifesto in 2001. Contemporary systems are characterised by their large size and their interconnectivity with other large systems. This scenario requires skilled software experts to install, configure, tune and maintain such systems. In [53], Kephart and Chess argue that these systems were reaching the limit of human capacity to adequately address managing tasks and to timely react to unexpected events from the systems. Thus, inspired by the autonomic nervous system in human beings, the Autonomic Computing vision aims to enable systems to self-manage based on goals defined by administrators.

The realisation of the concept of Autonomic Computing requires systems to implement autonomous behaviour towards a specific system aspect (e.g. security, performance, fault-tolerance, etc.). These systems aim to maintain certain properties with minimum human interference and are known as a self-* system. The self-* properties are dimensions of autonomous behaviour that is incorporated in a system to address a certain system aspect. Autonomous systems may implement the following self-* properties: self-protecting, self-optimising, self-healing and self-configuring capabilities. Self-protecting systems (e.g. [85]) are capable of identifying possible security threats and operating risks for the system well functioning. Furthermore, these systems are able to properly handle and prevent malicious users to exploit possible security threats by, for example, autonomously changing its internal structure. Self-optimising systems (e.g. [46]) are systems that are able to change their structure to autonomously improve some aspect of their performance, for example by reducing the systems response time. Self-configuring systems (e.g. [51]) are able to autonomously set up when introduced to a distributed system or change their configuration to accommodate the insertion of new systems. Self-healing systems (e.g. [2]) is capable to maintain a level of reliability by discovering faults in the system's behaviour and deciding on a course of action to maintain system execution.

The area of autonomic computing brought a variety of scientific and engineering challenges to be addressed. Many of these challenges were introduced and discussed

in 'The Vision of Autonomic Computing' paper (see [53]). The main challenges involve the life-cycle of autonomous systems which comprehends systems design, test, management, monitoring and upgrading systems. The management of interactions among autonomous systems which includes defining services provided and required by autonomous systems, services discovery and negotiation of providing services to multiple autonomous entities. Another challenge is the definition and representation of global system's goals which represent the interface between humans and autonomous systems. Also, learning and optimisation techniques exploring and advancing the state-of-the-art in machine learning techniques and statistical and probabilistic models to support system adaptation. These challenges are still in debate in the related scientific communities with promising solutions and future directions.

2.1.2 Organic Computing

The Organic Computing research initiative started in Germany with the focus to address and explore the self-organisation in technical software systems, inspired by neuroscience and molecular biology concepts and software engineering [77, 82]. Originated around the same time as the Autonomic Computing vision, the Organic Computing initiative also focused on the problem of having multiple instances of interacting autonomous systems, which may result in conflicts and undesired emergent behaviour affecting the resulting system [77].

In detail, autonomous systems with multiple goals might spontaneously interact with other autonomous systems in order to achieve and maintain global systems goals. The interaction among autonomous system instances is not a futuristic far-fetched idea. A concrete example given in [77] draws attention to the contemporary cars and the multiple interacting devices that are required to support the basics cars functions. In this example, devices requires data from other devices in order to provide its functionality, and the orchestration of such devices results in the car and all its available functions. Problems with the interactions among such systems, for example, by delaying the deliver of information to devices may lead to miscalculations and compromise service execution. Not only problems with orchestration and synchronisation may occur in such scenarios, a key problem is the emergent undesired

behaviours resulted from unpredicted interactions among systems devices.

Given this scenario of interacting autonomous systems, several studies were conducted to validate multiple controlling mechanisms that could be used to prevent undesired emergent behaviour in a variety of scenarios [62, 74] as well as to address further aspects (self-* properties) of autonomous systems [16, 40]. These studies are described in Sec 2.2.5, and later contrasting with the concept of Emergent Systems.

2.1.3 Self-adaptive Systems

The term 'self-adaptive systems' are usually used as an umbrella term, by different research communities, to refer to the systems ability to change its structure to accommodate changes. Whereas in distributed environments the term 'self-organising systems' is frequently used to refer to systems capable of reorganising its distributed architecture to cope with changes and achieve global system goals. In order to build systems that are capable of self-organising and self-adapting, the system requires the definition of an **adaptation logic** to guide software adaptation, the implementation of **adaptation mechanisms** to propagate changes to the actual system structure, and **coordination** among the system nodes to ensure a coherent change and convergence towards global system goals.

The **adaptation logic** is responsible to capture the knowledge that allows the system to detect events of interest and decide its next course of actions, which include the decision to maintain the system in its current state or to adapt to another configuration. The adaptation logic can be represented in different forms: the policy-based approaches, which represent adaptation logic using expertly-crafted rules defined in the design phase; mode-driven approaches, which apply models representing system properties, QoS traits, system architectures, system goals and equations that supports runtime reasoning and adaptation; and bio-inspired algorithms are used to encode adaptation logic imitating behaviours found in nature, e.g. ant-colony.

The **adaptation mechanisms** are responsible to propagate changes in the actual software. These mechanisms are classified either as parametric or architectural approaches. Parametric approaches consider the system as a black-box, having only dials buttons to influence the system behaviour. This approach is limited as com-

pared to architectural, because it can only change the system within a predefined range of parametric values. The architectural adaptation approach allows the system to change its structure by replacing components or changing its architecture pattern to maintain system properties (e.g. to use a load balancer architecture to maintain performance or to replicate services to cope with system failures). The architectural adaptation allows profound changes in the system, making it more flexible to cope with changes in the operating environment.

In a distributed scenario where multiple autonomous entities, with individual goals and capable of making their own adaptation decisions, are interacting with each other, it is important to create mechanisms to ensure that the system achieves its global goals whilst maintaining non-functional requirements. The **coordination of software adaptation** is key to ensure a coherent adaptation of distributed systems and convergence towards a common global system goal. Many approaches for coordinating adaptation were investigated in the multi-agent research community where the autonomous entities (agents) sought cooperation and consensus in order to make decisions to converge the system towards the global desired behaviour. Some approaches use voting schemes, action predictions, and other consensus schemes.

Common activities of a self-adaptive and self-organising system involve: monitoring the system in execution, analysing the monitored data, deciding on and executing the course of action the system needs to take to maintain its desired properties and performing adaptation to the systems behaviour. The feedback loop conceptual framework was well-adopted to implement self-adaptive system for capturing these essential activities to equip systems with autonomous adaptation ability. The most famous feedback loop conceptual framework in self-adaptive systems and autonomic computing field is the MAPE-K loop (as in [7]), which stands for Monitoring, Analysing, Planning, Executing activities which are guided by the Knowledge defined by experts at design phase and the results of the previous actions of the system. In order to give autonomous capabilities for systems, Autonomic Computing approaches enclose a MAPE-K framework to a system to support monitoring, analysis, planning and executing of system adaptation.

2.2 Self-adaptive Approaches and Mechanisms

This section surveys relevant work that illustrates different aspects of developing self-adaptive solutions. The works described in this section represent a comprehensive survey of techniques to support reasoning and planning for self-adaptation and change propagation throughout the actual software structure and configuration. Moreover, these works are evidence of the shortcomings and limitation of current approaches, as well as illustrations of the design process of current autonomic computing solutions. Some approaches are complementary and may assist further development and consolidation of the emergent software paradigm.

The first two subsections are divided to showcase approaches that use i) static policies and ii) model approaches as different techniques to represent systems adaptation logic, which is responsible to support reasoning and decision making in the adaptation process of the system's behaviour and structure. These two aspects of software adaptation is described in Sec. 2.2.1 and Sec. 2.2.2 respectively, along with a description of the adaptation mechanisms that the presented works apply to ensure changes in the actual software structure. Furthermore, Sec. 2.2.3 introduces the concept of reinforcement learning and discusses how this learning paradigm has been used in self-adaptive systems. Sec. 2.2.4 describes relevant examples of multi-agent systems, illustrating how agents are used to autonomously organise systems, and to autonomously coordinate distributed adaptation to achieve global systems goals. Sec. 2.2.5 presents relevant work implementing assurance and controlling mechanisms to avoid undesired emergent behaviour. Finally, Sec 2.2.6 describes biologically inspired approaches that are used to solve a variety of optimising problems, including problems in the context of self-adaptive systems.

2.2.1 Policy-driven Approaches

The policy-driven approaches implement the simplest and most straight-forward method to represent the systems adaptation logic. Due to its simplicity these were the most used methods in designing early self-adaptive systems. The approach consists of manually writing static rules in the design phase to describe software state

or operating conditions and make the system aware of what software configuration it should change to in case those conditions are detected. Some approaches implement very straightforward and static rules not leaving room for autonomous reasoning and decision making in the process, as described in [48], and other approaches, such as in [55], the software engineers describe adaptation rules in terms of logic expressions, which allow a small level of reasoning in the adaptation process.

The most relevant example of manually crafted rules to guide software adaptation is described in [48]. The work applies component-based technologies to introduce the concept of open overlays into middleware architectures. The open overlays concept supports the configuration and reconfiguration of network overlays allowing a flexible composition of virtual network resources and services at runtime. The paper describe a series of fine-grained software components that are used to compose overlay-related behaviour in wireless sensor network nodes at runtime, guided by predefined declarative XML-based expressions. These expressions are used to define rules that are processed at runtime to determine the most appropriate overlay behaviour for the system. For instance, a rule can be created for the case where 'multicast' is requested by the application (which represents the system goal), stating that in those cases when the underlying network infrastructure provides no support to IP multicast (which represents the software state or environment condition), then the Tree Building Control Protocol (TBCP) overlay components should be selected (which represents the desired software configuration). This work is a very relevant example of a rule-based approach as a way to express the adaptation logic and guide software adaptation at runtime, and it applies component-based technology to compose and adapt software as the system executes.

Another example of policy-driven approaches is the work described in [55]. This is an important example that illustrates the use of logic expressions to support software adaptation. The paper describes the application of temporal requirements in the system's adaptation rules. The work is conducted in the context of a Cybercar concept which is described as a public transport system with 'automated driving capabilities'. The vehicle is equipped with different positioning systems devices such as GPS and WiFi. These devices can be used to support location services in

different manners, for instance by using GPS, or by using WiFi as a location-aware device, or using a combination of both GPS+WiFi. The different compositions have different effects in the non-functional properties of the system, e.g. energy consumption and accuracy. The approach requires the definition of the systems configuration and reconfiguration in the design phase. These static policies define the events and temporal requirements responsible to trigger software adaptation, as well as the configuration to which the system will adapt. The mechanism used to propagate changes to the actual software is also realised through a component-based runtime. The authors apply the Fractal [14] component-based model, a well-known framework that supports fine-grained and runtime component adaptation.

These policy-driven approaches are very effective when all system state are well known a prior and the deployment environment are fairly static. For that reason these approaches were used in early examples of self-adaptive solutions. Due to the high levels of dynamisms in the operating environment of contemporary applications, the static policy-driven approach is not adequate to support the required levels of flexibility in the software adaptation process. In case where the system suffers from fluctuations in the workload or unexpected failures in the underlying structure, these systems are not be able to react accordingly and accommodate the changes in a timely manner, leading the system to malfunctioning or degraded performance.

2.2.2 Model-based Approaches

Model-driven approaches are engineering techniques widely used to tackle software complexity, specifically in the development and adaptation of large software systems. These approaches use representations of systems in an abstract and simplified form, supporting autonomous reasoning of the system state and facilitating runtime optimisation of large systems. In particular, there are three main situations in which models are used to tackle software complexity and are briefly introduced.

Firstly, models are used to capture high level goals, for instance in platform independent models, that will drive system code generation, facilitating and accelerating the software development process. Secondly, models are used as an abstract representation of the system (the entire system or only parts of it). This

abstract/simplified representation allows runtime reasoning of the system state, supporting decision making that enables system adaptation. Finally, models are used to guide software adaptation at runtime, through code generation techniques or command interpretation to change the running system architecture.

The model-driven approaches represent the majority of work that support self-adaptation. In the literature, there are a variety of models capturing different aspects of software and domain information used to support autonomous software adaptation. These models, as opposed to expertly-crafted policies, support more elaborate autonomous reasoning of the software state and its configurations, enabling a wider range of more complex adaptation possibilities.

This section categorise different model-based approaches considering the content that is represented by the models. This categorisation method yielded three categories that capture well the variety of model-based approaches to support self-adaptive solutions. The first category is *'domain-specific and system-representative models'*, which considers models that either capture domain-specific information or the representation of the system itself and are used to support online reasoning of the system state and adaptation. The second category is *'predefined analytical models'* and refers to models that capture in formal manner the adaptation process, for example by using mathematical equations. Finally, the third category is the *'autonomously generated analytical models'*, representing the leading approaches to support software adaptation, consists of models that capture the adaptation process in a formal manner, but require less up-front engineering effort in specifying models. These categories were designed not only to show the variety of ways to use models to support adaptation, but also to showcase the progression of how model-based approaches tackles the challenges of supporting system adaptation.

Domain-specific and System-representative Models

The domain-specific and system-representative models require the definition of models in the design phase that capture the system state, domain-specific information, the system goals or QoS traits and models that correlate all these informations.

In the paper [17], the authors describe a model-driven approach to create self-

adaptive systems. This is built on the core idea that there is no one-to-one mapping between requirements and solutions, and that alternative solutions to satisfy the system requirements impact systems properties in different ways. The approach requires two manually crafted models. One representing the system goals, and the other representing the architecture design decisions captured in the form of a decision tree. The goal model captures all systems requirements, both functional (goals) and non-functional (softgoals) and their correlation. The goals in the model are connected to tasks which are in turn connected to architecture design choices capable of carrying out the tasks. One requirement is often connected to multiple alternative goals, meaning that at least one of the goals must be satisfied in order to satisfy the requirements. This alternative relation between goals gives the system the choice of how to satisfy its requirements by selecting the goals it will satisfy. Considering that each goal is attached to a different task, by selecting a set of new goals the system is selecting a different architecture configuration responsible to achieve the new set of goals. The system considers its monitored information to decide what goals to satisfy and consequently what architectural configuration to choose in order to maintain system properties and the quality of service provided. The result is the generation of a incrementally transformed models of the current running architecture, which can be used to adapt the actual system. This work is a relevant example of using models to reason about the systems operational profile and make software adaptation decisions. This paper focuses solely on its goal-oriented requirement model and architecture design decision model to support adaptation, not advocating in favour of any mechanism to conduct adaptation in the actual system. The paper mentions that once the adapted architecture model for the system is generated, architecture-based management middleware can be used to propagate changes to the actual system. Component-based models and service-oriented adaptation techniques are suggested as adaptation mechanisms.

Another relevant example is described in [61]. Malek et al. introduce a framework for Self-architecting Service-oriented Systems (SASSY). This approach rely on a series of manually-crafted models to compose, adapt, analyse and evaluate service-oriented systems. Examples of these models capture 'QoS architectural patterns'

and software adaptation patterns to compose and evolve service-oriented architectures. The 'QoS architectural patterns' model, for instance, correlates QoS metrics to architectural decision. For example to increase fault-tolerance the system should replicate services, or the system should opt for a load balancer architecture to increase throughput. The adaptation pattern, on the other hand, is an analytical model that specifies how an architectural pattern influences QoS metrics of interest, serving as a bedrock to system online adaptation. This work is an important and relevant example of the upfront effort required to create models to abstract systems to support online adaptation. This work is classified in model-driven approach, where static analytical models are defined in the design phase to support and guide adaptation at runtime. The approach uses architectural patterns for service composition, and selection of service providers to compose and adapt system at runtime.

A final example, discussed in this section, to demonstrate the use of domain-specific models is presented in [67]. This paper describes an end-to-end approach that allows end-users to change systems configuration using high level/domain-specific concepts at runtime. This approach does not support any self-adaptive mechanisms but abstracts the system to facilitate a user-led adaptation process by using domain-specific models. The approach applies the concept of i-DSVM (Interpreted Domain-specific Virtual Machine) capable of executing models, instead of translating models into high-level languages (a common approach in models@runtime community). In order to support model execution, in a cheap and reusable manner, the authors introduce the idea of a MoE (Model of Execution) and DSK (Domain-specific Knowledge) as two separate elements. The idea considers the creation of a generic MoE detached from domain-specific concepts, so that the MoE can be reused in multiple domains when provided access to multiple DSK. The authors also introduce the concept of Intent Models (IM), a tree-based model composed of multiple procedures that represent different alternatives to execute high level commands. The IM gives flexibility to this approach by allowing the middleware to find at runtime the most appropriate way to carry out high level commands, considering the system state, user preferences and a repertoire of procedures. Although this approach provides flexibility to the system in executing high

level commands, it is still very human-centric. The technical concepts that make up IMs, for instance, are predefined and manually crafted, in other words, humans define the range of possible execution paths that the system can choose from at runtime, there is no online learning and consequently knowledge are not created and new/unpredictable paths of executions are not encountered. Furthermore, this approach does not have any embedded autonomous decision making process, relying solely on domain-specific models to guide software adaptation at runtime.

Predefined Analytical Models

A different type of model used to support software adaptation is the analytical models. An analytical model can represent the system, the system's goals or the system's domain concepts, however, this section details two examples of models that mathematically captures the knowledge used to conduct software adaptation.

The first example is the paper [60]. This paper describes a self-adaptive system using Control-Theory to guarantee service delays in web servers. The work focuses on connection delays in HTTP 1.1 web servers, which implements persistent connections. These connections are a response to the connection delays occurred in HTTP 1.0, where every client request were closed after the servers response, forcing the client to (re)open a new connection for each new request. The persistent connections maintain connections open after the server sends its response, so that the client can reuse the same connection to make further requests, reducing delays caused by the ritual of establishing connections. Commonly, servers allocate an executing thread per client connection from a pool of executing threads. In this persistent connection concept, for every connection established an executing thread is dedicated to handling the requests from that connection, keeping the thread occupied even when requests are not being processed. A common strategy to guarantee delays in this scenario is to allocate threads to classes of requests, so that the server can reserve a bigger number of threads to classes of requests that require a small delay tolerance. This work defines controllers to better allocate threads to classes, by using an adaptive proportional share policy for class-thread allocation determined by mathematical models for delay guarantees. These models are defined in

the design phase and are composed of a set of difference equations that capture the systems adaptation logic, supporting decisions based on monitored information frequently collected from the system, and acting accordingly to fluctuations of the workload and predefined desired delays. The adaptation mechanism for the system is responsible to change the allocation policy of incoming requests, not having to change structural details of the server (e.g. architectural adaptation) but rather performing parametric changes to the connection scheduler (component responsible to allocate incoming connections to executing threads). This work is classified as an example of applying control-theory in creating a static predefined adaptation logic, and to use parametric mechanisms for system adaptation.

A more recent example is described in [86]. The paper describes NetKV, a self-managing load balancer for memcached clusters for web application performance optimisation. NetKV is a solution for the problem of imbalance requests to memcached clusters which occurs as a result of poor management of popular content. According to the authors, current approaches apply client-side proxies to redirect requests to cached content to the memcached cluster. This proxy approach hinders the development of a proper content distribution policy for the cluster, contributing to overload servers that store popular content. As a response to that problem, Zhang et al. present a centralised load balancer that explores Network Function Virtualisation to quickly forward requests to the cluster, whilst identifying hot content using stream-analytic techniques and applying a static analytical model to determine content replication in the clusters. This approach eliminates the need for client-side proxies by having a centralised load balancer, and optimises the web infrastructure by popular content replication, distributing a more balanced load across the servers. The adaptation mechanism implemented by this approach changes the system behaviour by (re)placing and replicating content across servers, not requiring changes in the software implementation or parameters. The models that support the systems decisions and adaptation are fixed and defined in the design phase, and it is designed to adapt the system according to variation of popular content. This approach is classified as static analytical model, using predefined models to guide systems adaptation. Furthermore, this approach does not implement any adaptation

mechanism for the system implementation (e.g. replacing architectural components or changing systems parameters). This approach enables structural system changes by replicating and placing stored content across different memcached servers.

These predefined analytical models require a deep and complex analysis of the systems conditions and external environment to define the models. This condition makes this approach unwieldy to implement, demanding a considerable effort in deducing models to realise the system.

Autonomously Generated Analytical Models

As a response to the complexity and upfront analytical effort demanded from the predefined analytical models, which make these approaches highly complex to realise, recent approaches explore the autonomous development of these models, reducing the complexity of realising such approach and facilitating the development of complex self-adaptive systems.

The work in [39] proposes the use of Control-theory and its formal properties to support software adaptation. As previously illustrated by [60], control-theory based approaches involve the definition of complex equations to capture the nuances of the software adaptation space, which makes such approaches difficult to engineer. Furthermore, the dynamic nature of operating environment, e.g. with workload fluctuations, might invalidate previous-defined models. In response to that situation, this paper is another example of an alternative to manually define the adaptation logic of self-adaptive systems, and to minimise the up-front effort to develop the adaptation logic. Software adaptation decisions are made based on quantitative measurements of the specified non-functional requirements, and the actual software adaptation is done by adjusting the systems parameters. In order to generate the controller the system requires the input of tunable parameters of the system and specified non-functional requirements the system is expected to satisfy. There is no prior need of information that maps system parameters and the requirements. A simple example that describes the synthesising process is the web service case, where the parameters are the number of servers to provide the service and the response time is the controlled system aspect. The controller synthesising process

have two phases, the first phase profiles the system and generate a linear model of the system, it then uses that model to synthesise the system controller, creating a set of differential equations to control the self-adaptive software. Online model updates are also considered in this approach, which allows the system to re-synthesis controllers adjusted to new operating conditions. This approach is classified (for the purpose of this thesis) as 'dynamic', regarding the dynamic synthesis of the adaptation logic, and as 'parametric', considering the adaptation mechanisms.

Another very relevant work in this space is described in [32]. The authors also argue against the use of manually defined analytical models for the system adaptation logic to guide online software adaptation. The properties of these analytical models are defined in the design-phase as an assumption of how the operating environment will act. As a consequence of the dynamism of current operating environment, unforeseen events might occur, leading the system to performance degradation or malfunctioning. In response to this scenario, the authors advocate in favour of a learning process as an alternative to a fixed analytical model with predefined properties for the self-adaptive system's adaptation logic. This learning process consists of generating mathematical functions that correlate the presence of software features in the system with the satisfaction levels of the systems goals. These mathematical functions are autonomously created as a result of offline training by running the system and exposing it to a set of operating conditions the system is expected to find in production. After this offline learning process, the system will have generated an analytical model that will be used to make software adaptation at runtime. Furthermore, as the system is exposed to new operating conditions, it is capable of fine-tuning the functions coefficients to incorporate the new operating condition and maintain a consistent and reliable adaptation logic. The features are captured in the design phase, and are responsible for providing an abstract representation of the system that supports software adaptation by defining the parts of the system that are adaptable. This work is classified as one of the most relevant examples in the literature presenting a learning approach to autonomously define and evolve the systems adaptation logic, as opposed to having a predefined manually crafted analytical model. This work also applies a feature model to define specific adaptable

parts of the system, and to establish correlations between features and goals.

The 'analytical model' approach captures the complexity and nuances required to guide software adaptation at runtime in dynamic and uncertain environments. However, creating analytical models requires substantial effort. The presented 'Autonomously Generated Analytical Models' is a way to mitigate this problem. This approach enables automated learning or deduction of analytical models with minimum human involvement. This autonomous model generation facilitates the realisation of complex self-adaptive software, whilst supporting the evolution of such models as new operating conditions emerge.

2.2.3 Reinforcement Learning in Self-adaptive Systems

Reinforcement Learning is a learning paradigm in which an agent learns to correlate a set of actions to reward values [75, 79]. An agent, in this context, can be defined as a software that is capable of executing actions and collecting rewards as a consequence of the performed actions. This learning paradigm allows software systems to learn, at runtime, whilst the system interacts with the environment, which actions and in which order will yield maximum reward values. This characteristic of this learning paradigm applied to self-adaptive systems enables the system to learn the adaptation logic as it executes, driving software adaptation at runtime, and evolving its adaptation logic as the system encounters new, unforeseen situations [4, 54].

Several papers on self-adaptive systems that apply reinforcement learning algorithms in the adaptation process, such as [4], or any machine learning algorithm for that matter [30, 57, 88], focuses on parametric adaptation. Parametric adaptation consists of changes in software parameters as if the system were a black box with dials. In this context, the system is only capable of adapting within a predefined set of values/actions, limiting the capacity of reinforcement learning algorithms to learn beyond the actions that was established by experts in the design phase.

This limitation of parametric tuning approaches also affects some architectural-based approaches such a [32, 54, 61], due to the fact that, in these approaches, the system can only change its architecture based on a predefined range of architectural configurations, and that expanding this fixed range of architectural options requires

the system and the learning algorithm to be re-evaluated and re-defined.

In [54], Kim and Park demonstrate the application of Q-learning, a well-known and widely used reinforcement learning algorithm, to architectural change in a robot simulation case study. This is one of the earlier and most representative examples of the direct application of reinforcement learning to build and evolve adaptation logic in self-adaptive systems. The paper describes how to create learning tables with actions and systems states, detailing a method that can be followed to apply Q-learning to self-adaptive systems. The paper ends with an evaluation of the application of reinforcement learning to robot simulations, showing that a robot that applies such approach can successfully learn and evolve its adaptation logic.

The main limitation of applying reinforcement learning to self-adaptive systems in the literature is the lack of abstraction of actions and state for the machine learning algorithm. As papers such as [4,54] illustrate, reinforcement learning algorithms are applied to self-adaptive systems to optimise software in specific domains, with a predefined set of actions observable in specific case studies. This shows high levels of human dependency to define, for each application domain, the set of actions and states on which the learning algorithm will be developed upon, limiting the potential and generalisation of reinforcement learning approaches in self-adaptive systems.

This thesis argues that in order to fully enable learning and evolution of the adaptation logic of software systems, it is required to i) apply reinforcement learning to the composition process of systems, and ii) 'abstract'/remodel the system to be easily handled by machine learning algorithms. The proposed Emergent Systems concept demonstrates how these two points can be realised by unifying component-based model and reinforcement learning algorithms.

2.2.4 Multi-agent Approaches

Multi-agent systems are a popular approach to build self-organising systems. This section presents some key work in the literature to illustrate the application of such systems. Two relevant papers describing an architecture addressing the problem of supporting coordination of adaptation processes in a self-organising system, and one paper illustrating the use of agents as programme instructions that has the ability

to combine themselves to autonomously compose algorithms.

In order to support a coherent adaptation process to achieve a global desired behaviour in distributed self-organising systems, a key problem is the creation of coordinating mechanisms of the adaptation process. An easy solution for this problem is the creation of a centralised decision making entity that has a global view of the system and thus can provide a better support for a global system adaptation. However, the centralised coordination approach provides a single-point of failure and hinders scalability, not being able to support a system with large numbers of agents. On the other extreme side of the spectrum lies the complete decentralised decision making process, where each agent adopts an adaptation plan according to its local conditions and demands. Although this approach is extremely scalable, it often makes the system converge towards local optimal configuration which might not satisfy the global system requirements. Moreover, in a competitive scenario, i.e. where agents might compete for resources or service provision, this type of solution may lead to agent's resource starvation and disruption of services. A reliable self-organising processes require an intermediate solution that supports agent coordination in the adaptation process.

In [72], the authors present discusses an approach for structural adaptation of the coordination process of self-organising systems. The system is designed using the concept of agents with local goals that interact among themselves to achieve a global goal. In this context, the system requires orchestration and consensus of predefined adaptation plans among the agents in order to support the distributed adaptation process. As a result of the coordination process the system is capable of performing structural adaptation to maintain systems properties and global desired behaviour. A decentralised coordination process is often necessary in distributed systems, as opposed to a centralised decision making process, to avoid single point of failures and enable scalability. The paper presents a voting scheme as a way to reach an agreement among agents, deciding which adaptation plan to adopt once an adaptation condition is identified. The paper presents an entire architecture to support the dissemination of information among agents to support coordination of the adaptation process. Other solutions, described in [68] involve predicting the

actions of an agent's neighbourhood (a network of agents defined by proximity) by understanding how they react to certain events, and then use that knowledge to approximate and converge their interactions.

A key property of multi-agent systems is their ability to implement emergent behaviour. An important examples of this is in the paper [44], where the authors present an emergent programming platform supported by multi-agents to enable emergent software design. The paper proposes a multi-purpose programming language where the instruction set is composed of autonomous agents. The instruction-agent compose fully functioning computer programmes by partnering agents. The agents link to other agents from which data are received, and to which data are sent, forming an execution flow as data is received from one instruction-agent and passed down to another instruction-agent. This approach enables an autonomous agent-based software design and composition by having the agents try different instructions combinations to create the desired behaviour.

2.2.5 Assurance and Controlling Mechanisms

This section describes examples of work that apply controlling or assurance mechanisms to guide software adaptation and avoid undesired emergent behaviour in the process. A research community that tackles such challenges is Organic Computing, and thus this section presents a paper that illustrate the concept. Furthermore, this section also presents mechanisms used to provide assurance in autonomous systems.

A descriptive example of the Organic Computing concepts is described the paper [74]. The authors propose an architecture to support organic computing. The main goal of organic computing is to enable the creation and interaction of autonomous systems whilst minimising and restraining the occurrence of undesired emergent behaviours. This challenge motivated the authors to propose an observer/controller theoretical architecture to support this vision. The architecture is composed of two main elements; the observer and the controller. The observer is responsible for monitoring the underlying system, collecting information of the system status and feeding this information to the controller. The observer is also responsible for predicting and identifying system status. The controller has access to user-provided

system goals and uses the system status collected from the observer module to influence the underlying system, in order to maintain it within the system goals. As an adaptation system, the observer/controller architecture expects that the system is implemented as a multi-agent system, and changes are done by changing networks of agents that can influence its 'neighbourhood' and have global system impact. This approach illustrates the goals of the organic computing research community, and presents a theoretical architecture to support the concept. The architecture implements a feedback loop to avoid undesired emergent behaviours whilst guiding systems changes, based on user-provided goals and system representative models.

In the paper [62], the authors present a variety of evolutionary computing (i.e. genetic algorithms and genetic programming) techniques to mitigate uncertainty in designing self-adaptive solutions and increase assurance in the adaptation process. The paper suggests the application of evolutionary computing in different stages of the software life-cycle to tackle uncertainty. For example, in the development time, evolutionary computing techniques can be used to assist program development by evolving algorithms to be more robust when facing diverse operating conditions. This process is done by having potential algorithms as individuals, and the desired high level behaviour as the selective filter used to guide the evolutionary process. This process can be used to generate software variations capable of addressing different operating conditions. The advantage of this method is in its ability to find solution autonomously for complex problems and optimise them for specific operating conditions. A disadvantage of this approach is that it requires software engineers to know details of the operating environment in the design phase.

McKinley el al. [62] also present the exploration of potential operating conditions that system might encounter after deployment. The described approach generates operating conditions used to simulate system variations. The results of the simulations are then recorded, showing how the system performed in different conditions. The recorded data is then captured as an utility function for requirement monitoring after the system deployment. This solution is only effective in cases where the generated conditions match a high number of real operating conditions.

At runtime, McKinley el al. illustrate the use of evolutionary computing to assess

runtime monitoring problems. A system with a high frequency monitoring process provides a more precise picture of the system state, which in turn require more resources to support the monitoring process. The paper mentions an genetic algorithm approach that balances resource consumption and online monitoring accuracy to support better decision making in the adaptation process. Finally, the authors describe the use of evolutionary computing to address the selection of adaptation paths when the system presents multiple intermediary configurations with different requirements impact, enabling optimisation considering conflicting requirements.

These approaches demand the involvement of domain experts to define the system's acceptable conditions, limiting the potential of autonomous system to freely explore and find unexpected optimal configurations for the operating conditions. Also, in certain scenarios, such as in self-driving vehicles, it is extremely hard to predict and create the proper assurance mechanisms to avoid undesired behaviour.

2.2.6 Biologically Inspired Approaches

Biologically inspired approaches are often used in optimisation problems. These approaches encode behaviours found in nature in algorithms to solve optimisation problems. A very popular example of this approach is the ant-colony where agents imitate ant behaviour in finding food by leaving a trail of pheromone that other ants can sense [27]. A great variety of papers explored the application of such methods in a variety of problems such as the travelling sales [28], vehicle routing [42], graph colouring [20] and project scheduling [63]. Another popular example of this approach is evolutionary computing [31]. This concept applies the basic idea of biological evolution to solve problems. This approach defines a population of individuals, then randomly insert genetic variations into the population and submit them through a selective process eliminating some individuals and leaving the fittest according to a selection function. The remaining individuals are used to build the next generation of individuals that go through the process again.

Bio-inspired approaches were used to directly provide systems with self-adaptive capabilities. For example, the paper [26] describes an algorithm that encodes the behaviour of Neuroendocrine Immune System to support autonomous composition

and adaptation of web services. Another examples is described in [73] the used of evolutionary computing to predict in the design phase the operating environment that the system will be facing and how the diverse available software configurations would best suit different environment maintaining desired QoS levels.

2.3 Overall Analysis

The proposed Emergent Systems concept unifies component-based models and reinforcement learning approach to push design decisions to the software architecture at runtime. This concept also aims to eliminate human involvement in creating rules and models to guide software adaptation at runtime. In this sense, this concept provides a framework to support systems to create its own understanding of its environment and constituent parts. This concept is described in Chapter 3 with a detailed description of the problem space. This section revisits the presented approaches and contrast them to the concept of Emergent Systems. Finally, this section is concluded with a table, summarising the presented approaches, illustrating the gap in the literature which the concept of Emergent System aims to address.

Policy-driven Approaches

The Emergent Software System paradigm differs from the expertly-crafted policy-driven approaches by having the system learn its own adaptation rules. As part of the learning process, the system learns how the available different software compositions are used in the detected operating conditions as the system executes. The Emergent Software System approach, as alternative, reduces the engineering effort and discards expert knowledge to create adaptive systems. The resulting system is also able to cope with high levels of dynamism in the operating environment.

Autonomously Generated Analytical Models

In the Emergent Software System paradigm, the concept of learning and developing analytical models to support adaptation lies in the core of the emergent system approach. The main difference, however, is that the learning process happens at

runtime, rather than offline training the system. The online learning approach in tandem with a component-based model is fundamental to enable software to build its own understanding of its environment and structure, being able to evolve and cope with high levels of uncertainty and dynamism in the operating environment, representing a step beyond current approaches.

Reinforcement Learning in Self-adaptive Systems

Reinforcement learning is at the core of the proposed concept of Emergent Software Systems. Its role in the proposed concept is to support runtime learning and evolution of the adaptation logic in autonomous systems, with minimum human intervention. However, the state-of-the-art application of reinforcement learning algorithms in self-adaptive systems suffers from the lack of an autonomous method to abstract state and actions necessary to implement these algorithms. This is mainly because software systems have an infinite set of states and actions, and the extraction of the ones that have the most impact on system's goals requires domain specific knowledge. The key difference of the work in the literature and the proposed approach is to change the paradigm from having human-selected actions to provide self-adaptation, to defining only one action: deciding if a set of components should be in or out of the software architectural composition. By changing the paradigm from self-adaptation to self-composition, this thesis aims to generalise and abstract autonomous software composition using the reinforcement learning paradigm.

Multi-agent Systems

Emergent Software System concept shares commonalities with multi-agent system. Both concepts argue that systems should be built from small units of software to form systems. In this regard, multi-agent system engineering techniques and frameworks can be used to realise Emergent Systems, and more specifically, learning techniques and coordination to reach consensus among agents that are investigated in the multi-agent community can be used in the learning module of Emergent Systems. On the other hand, emergent software system essentially differs from multi-agent system when considering the classical agent definition. In emergent software sys-

tem, components is not seen as agent because they do not hold a particular goal. The components only perform very specific tasks. The Emergent Software System concept implements a meta-structure through which software is composed and optimised as it runs, maintaining the target software functioning despite its ability to freely explore among composition variants.

Assurance and Controlling Mechanisms

Assurance and controlling mechanisms are often considered a desired property in autonomous systems. These techniques require the provision of information predicted by engineer expert in the design phase, or controlling mechanisms to guide system to maintain a correct behaviour. This technique often makes the system vulnerable when unpredicted events emerge, leaving the system in unknown states. In the Emergent Software System paradigm, however, the system is a product of its operating environment, being composed as a consequence of the perceived operating conditions. This characteristic gives emergent systems the ability to learn how to respond to changes, and how to deal with unexpected situations.

Bio-inspired approaches

The Emergent System concept is different from the bio-inspired approaches for not incorporating the adaptation logic in the algorithm. Instead, emergent systems learns as it executes, as a consequence of autonomously experimenting with different software compositions which composition works best for the detected executing environment. Biological inspired approaches, on the other hand, have been successfully applied to a variety of optimisation problems. Considering the Emergent Software System paradigm, the bio-inspired approaches is seen as complementary. Particularly in assisting autonomous system optimisation as a method to find optimal architectural compositions in specific application domains, for example, by using genetic algorithms to autonomously generate software components.

Other Related Work

The Autonomic Computing area is vast and multidisciplinary, spreading through diverse disciplines such as Artificial Intelligence, Software Engineering, Distributed Systems, and so on. Because of this multidisciplinary characteristic, many research communities use their own concepts and techniques to propose systems with self-* properties. Many of these works use different taxonomy to express and describe their contributions and methods. Although, previous sections already present the main articles carefully selected to clearly show the gap that this thesis addresses, for completeness, this section describes other articles that complement this narrative.

On the mechanisms of self-adaptation and propagation of changes in the software structure, some work such as [8, 9, 49] use (micro)services to support software composition and self-adaptation. Works such as [15, 29, 58, 66], as opposed to use inflexible orchestration, create choreographies that are enacted at runtime to compose and integrate distributed systems. Furthermore, articles such as [10, 11, 23, 81] describes models, languages used by experts to express goal-oriented self-integration systems in an autonomous setting. Although, these articles illustrate a different perspective (goal-oriented self-integration and composition), they still fall within the previously defined sections, where autonomous systems rely on rules, and models mainly defined at design phase to implement their self-* properties.

Another relevant class of work that tackles engineering and management of complex system is described in [12, 24, 41]. These articles apply the concept of holons to create a goal-oriented model for engineering autonomous systems. These works are highly complimentary to the work presented in this thesis, since they provide ways to express high-level goals to guide system integration and composition that can be used in the context of Emergent Software System concept proposed.

Finally, projects such as PetaBricks [5, 6, 25, 69], ZettaBricks [3] and more recent examples such as [84] use machine learning techniques to generate models and user-defined algorithmic choices at design phase and exploit these models and choices at runtime to optimise systems. The system optimisation examples brought by these papers are in the domain of compilers that optimise programs to better explore cores in a multi-core architecture [64], to reduce temperature levels in datacentres [78] or

address energy efficiency [84]. These are very relevant and important work that take advantage of machine learning techniques to be able to keep up with the speed of changes in CPU architectures, GPU architectures and datacentre infrastructures. However, the application of machine learning, or languages that define adaptation options in the design phase have different challenges than applying reinforcement learning techniques at runtime, specially to learn and evolve adaptation logic of self-adaptive systems. The problem space of online learning applied to self-adaptive systems, which is the main focus of this thesis, is fully presented in Chapter 3.

State of the Art and Research Gap

The realisation of self-adaptive solutions require the development of an adaptation logic, adaptation mechanisms and often a coordination strategy (in distributed contexts). Analysing the presented work in this domain with regards to adaptation logic creation and evolution, a research gap is identified in the literature, which is illustrated and summarised in Table 2.1. The table shows the lack of work addressing the entirely autonomous creation and evolution of adaptation logic, which would enable the development of solutions capable of addressing the high levels of uncertainty commonly presented in developing adaptive solutions (as described in [19]). Furthermore, the creation of systems able to autonomously develop their adaptation logic would reduce the engineering efforts of developing autonomous solutions.

Considering the adaptation logic, the state-of-the-art approaches implement different techniques including policy-driven (e.g. Grace, et al [48]), system representative models (e.g. SASSY [61]) and analytical models (e.g. Netkv [86]). The majority of the approaches require the definition of adaptation information in the design time. Some techniques allow the adaptation logic to evolve as new events occur in the operating environment, enabling a limited evolution of its adaptation logic to accommodate new events from operating environment (e.g. FUSION [32]). This approach, however, can not evolve if new software is added, requiring to rerun its offline training approach to generate its new adaptation logic. Finally, Georgé et al. [44], have a complete autonomous multi-agent approach, that is constantly trying to optimise towards a goal, but never builds nor evolves an adaptation logic. This

Work \ Aspects	Adaptation Logic Definition					Adaptation Logic Evolution			
	None	Manual	Hybrid	Automated (offline)	Automated (online)	None	Manual	Partially Automated	Completely Automated
Grace, et al [48]		✓					✓		
SASSY [61]		✓					✓		
Netkv [86]			✓			✓			
FUSION [32]				✓				✓	
Preisler, et al. [72]		✓					✓		
Georgé, et al. [44]	✓					✓			
Ramirez, et al. [73]				✓		✓			
Emergent Systems					✓				✓

Table 2.1: The table shows a gap in the literature, with a lack of approaches that enable both autonomous creation and evolution of adaptation logic with no human involvement.

approach is not able to 'remember' optimal solutions for previously seen situations, having to search for solutions even when previously seen conditions occur.

The Emergent Software System paradigm differentiates itself from all these approaches by incorporating an online reinforcement learning approach to compose software systems. This strategy is a step beyond current approaches and reduces the effort of creating the system, enabling the system to build its own understanding of its components and operating conditions (i.e. its adaptation logic). Furthermore, the main focus of Emergent Software solutions is in self-composition instead of self-adaptation. As a consequence of this change of focus, the Emergent Systems paradigm is expected to create software solutions capable of handling the increasingly complexity of software systems by postponing decisions that are commonly made in the systems design phase, pushing software composition and design decisions to be made at runtime whilst the system is exposed to operating conditions.

2.4 Summary

This chapter presented the main approaches to realising autonomous systems solutions. In detail, this chapter described the main concepts in the literature that support autonomous system adaptation. The most relevant concept is the vision of

Autonomic Computing and how self-managing systems can tackle the complexity aspects in developing, configuring, optimising and healing systems in a timely and effective manner. These systems target highly dynamic and volatile operating environments, and aim at limiting human involvement in high level system tasks, leaving the system in charge of low level, repetitive and cumbersome managing tasks.

Complementary concepts to support the Autonomic Computing vision were also described. **Organic Computing** originated from the necessity of interacting multi autonomous systems. This approach focuses mainly on adding constraints and assurance mechanisms to control software adaptation and avoid undesired emergent behaviour. Another complementary research area that supports the development of autonomous systems are **Model-driven approaches**. These are very popular techniques to support reasoning of software properties and adaptation decisions at runtime. Finally, the concept of **Multi-agent systems** was also discussed as a platform to enable self-organising systems through the definition of autonomous agents with local goals capable of autonomously compose distributed systems.

Another important concept introduced in this chapter was **self-adaptive and self-organising systems**. These are systems capable of changing its behaviour when facing changes in the operating environment or requirements. This term is used as umbrella term for autonomous systems and is a feature to realise autonomous solutions. In order to support any of the self-* properties of Autonomic Computing (e.g. self-optimising, self-healing) the system has be to able to self-adapt. The self-adaptive system must implement the adaptation logic of the system, responsible to guide software adaptation at runtime, the adaptation mechanisms responsible to propagate changes in the actual software system, and in case of distributed systems, coordination mechanisms responsible to guarantee a consensus among autonomous elements of the system on the new global system configuration.

The chapter concludes with a brief analysis of the differences and similarities of Emergent Software System and state-of-the-art approaches to design autonomous systems. The Emergent Software System paradigm differs from some principles described in this chapter, reinforces other principles by incorporating them into the approach, and use some areas as complement or promising directions to address the

challenges in developing emergent solutions (discussed in details in Chapter 3).

In particular, Emergent Software Systems reinforce the following ideas:

- Autonomous learning and deducting analytical models to guide software adaptation. But distinguishes itself from current approaches by adopting online learning technique (no offline training), allowing the creation of self-adaptive systems with no prior domain-specific knowledge, or human interference.
- Application of component-based technologies as a deployment mechanism, and model to support software (re)composition at runtime, enabling the application of reinforcement learning approaches.

Emergent Software Systems differ from the following ideas:

- Manually defining static policies and models (system representation or analytical models) to guide software adaptation;
- Manually defining a system architectural models or features to support propagation of changes in the actual system;
- Implementing parametric software adaptation mechanisms;
- Implementing assurance and controlling mechanisms that constraints emergent behaviour;

Emergent Software Systems differ from those ideas because they restrain system exploration, hindering the system's ability to autonomously find unexpected optimal compositions. Furthermore, these ideas also require expert knowledge in the design phase, increasing the complexity of self-adaptive system design and implementation.

CHAPTER 3

Emergent Software Systems: Concept and Challenges

This chapter introduces Emergent Software Systems, which is the core concept of this thesis, presenting the definition of this concept and its problem space. Emergent Software Systems are presented as a complete methodology to support the entire life-cycle of self-composing systems. In detail, this chapter discusses the main principles that support the creation of systems capable of self-composition and self-optimisation with minimum human interference, as well as presenting the challenges involved in this process. Although this chapter presents the full range of challenges involved in the implementation of emergent systems in both local and distributed settings, only the key challenges are explored in this thesis. In particular, this thesis addresses the fundamental challenges to demonstrate the feasibility of emergent software solutions as a means to validate this thesis' hypothesis. The Emergent Systems concept and problem space definition described in this chapter was published in [38], [71] and [37]. Sec. 3.1 defines the concept of Emergent Software System, presenting an overview of the concept, and its definition. Sec. 3.2 introduces the challenges based on our practical experience in building Emergent Software solutions. Finally, Sec. 3.3 presents the thesis scope, highlighting the challenges that are addressed in the remaining chapters.

3.1 Emergent Software Systems Concept

Emergent Software Systems are systems built from small and reusable units of software behaviour, which are capable of self-composition and self-optimisation as a result of the characteristics of their operating environment. The concept consists of unifying component-based models and a reinforcement learning approach to enable systems to make design decision at runtime, when new conditions emerge. Component-based technologies allow reinforcement learning algorithms to be used to experiment with software composition at runtime, and learn by experimenting with different software compositions the one that has the best performance levels for the executing operating environment. This section provides an abstract definition of Emergent Systems. A concrete implementation is presented in Chapter 4.

Problem Definition

The realisation of Emergent Software Systems requires the existence of a system goal G and a finite set of small software units SU. The goal G defines the main purpose of the system (comprehending both functional and non-functional requirements) and SU is composed of a variety of software units u. For some $u \in SU$, there exist implementation variations of u, meaning that unit variants provide the same functionality in different forms. For example, if the functionality is to compress a stream of bytes, there should exist software units implementing variations of compression algorithms (e.g. gzip, zlib, etc.). The functionalities are defined in a form of interfaces which define the available functions the software unit offer or require, as well as the kind of data that can be passed in or out of those functions. The software units are connected to each other through their interfaces, connecting the units that require a specific interface to the units that provide the same interface. This connecting units process is what results in software architectural composition that form a single instance of software in a single node.

The presence of variations of software units enable a variety of architectural compositions. Each assembly influences how the nodes interact with each other resulting in the global system architectural composition. In other words, there are no

predefined specifications of the system topology, protocols or data semantics, these are determined by local architectural composition. Furthermore, the existence of unit variants allow the resulting system to be assembled to achieve G in different ways, enabling the system to maintain required QoS in diverse operating conditions. Furthermore, some of the software units are required to emit a stream of metrics and events. Metrics are numeric values that represent the health status (e.g. performance) of the system and are also used as indicators of satisfaction levels of G. The events, on the other hand, are used to classify the operating environment, representing systems inputs and deployment characteristics such as CPU and memory percentage usage. The classification of the environment is important to permit a fair comparison among multiple existing compositions metrics, ensuring that the perception of the system health of a given architecture is fairly compared to another composition, giving that both were exposed to the same operating condition.

The definition of Emergent Software System consists of autonomously composing systems from small reusable software components, according to metrics and events (numbers and labels that represent system's health status and its operating condition) and user-provided goals. This process ensures that software is autonomously composed as a response to external stimuli and desired goals.

Problem Statement

The problem emergent systems have to solve is to find the composition of software units to best satisfy G at runtime. The system identifies its G satisfaction levels by analysing the collected metrics, choosing the most suitable composition by analysing metrics within the same operating condition, which is characterised by the collected events. The problem is aggravated by the need to involve multiple nodes to accomplish G. In a distributed scenario, the system must find the best global composition to satisfy G coordinating local composition across the available nodes.

3.2 Emergent Software Systems Challenges

This section describes the challenges of implementing the presented vision of Emergent Software Systems. These challenges are primarily based on attempts to realise emergent software solutions. These systems are able to self-compose and optimise its architecture at runtime guided by adversities found in the operating environment and according to user-provided goals. This section expands on the challenges of implementing these systems, considering the issues involved in the online learning process, the innate challenges to realise these systems, and the problems accentuated when considering distributed system scenarios.

3.2.1 Online Learning Challenges

The definition of the concept of Emergent Software Systems allows a range of possibilities to implement the details of the learning process to support system self-composition and optimisation at runtime. A key property of the learning process of emergent systems is its ability to experiment with the live system and learn about the operating environment and available components as the system executes. This section focuses on the challenges of implementing online learning strategy considering our experience to support software composition at runtime.

The reinforcement learning approach, explored in this thesis, consists of two parts, the exploration and exploitation phases. In the exploration phase, the system selects an architectural composition for use, then it waits for a period of time (named observation window) as the chosen composition is executing. At the end of the observation window the system collects the generated metrics and events. These collected metrics and events support the system to understand how well a particular composition performed under a specific operating condition. This process is repeated until all compositions are tried at runtime. In the end of the experimentation cycle, the system compares the performance of the tried compositions and selects the best performing architecture for the perceived operating environment. Once the system experiments and locates the appropriate software assembly for a specific operating environment, it stores that information in its knowledge base, so that

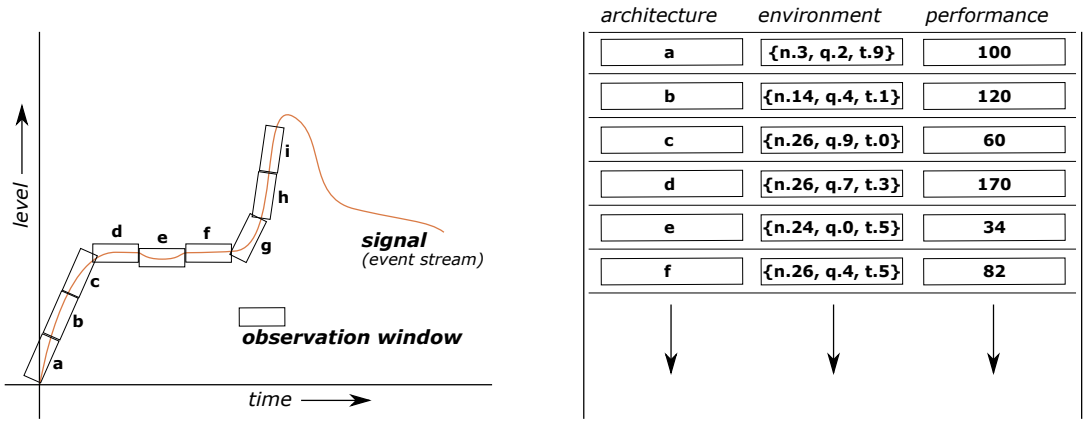

Figure 3.1: The emergent software online learning problem. The way the environment changes over time is illustrated on the left. On the right, the data collected at the end of each observation window is illustrated.

if the system encounters this operating condition again, it automatically changes its architectural composition to the one it previously learned, without realising the exploration process again. In the exploitation phase, the system maintains the located optimal composition running, whilst observing the operating environment. In case new conditions emerge, the system returns to the exploration phase.

Fig. 3.1 illustrates the learning problem in its abstract form. On the left side of the figure is the environment representation (event stream) to which the system is subjected. These events represent the system perception of the environment which is generally outside the control of the system. The graph is built with a collection of events periodically generated by components integrating the running architectural composition. On the right side of Fig. 3.1, the data collected from the components both events and metrics are displayed. In this example, the system changes its architecture in each successive observation window. This behaviour is common during the exploration phase of the online reinforcement learning approach. From this example a number of challenges are discussed:

- **Comparison difficulty:** Comparing different architectural compositions is an essential part of the online learning approach. This comparison process is difficult because of the dynamism of the operating environment that might change its conditions at any point of the exploration process. For instance, changes in operating environment on subsequent observation windows makes

the compositions experimented before the change incomparable with the architectural compositions experimented after the change. This is illustrated in Fig 3.1 on the right side in architecture (a), (b) and (c) as compared to (d), (e) and (f) that executes in the operating conditions. Scenarios that present a constantly changing operating environment present a great challenge to online learning and an interesting avenue for future work.

- **Mid-window changes:** Changes in the operating environment also can happen *during* an observation window. This is illustrated in architecture (g), where the graph representing the environment changes to an upward slope. This is particular difficult to detect depending on the size of the observation window and the type of collected events. Poorly defined observation window size or the absence of certain events may hide this transition from the system. This hinders the system's ability to properly compare compositions, impacting the quality of the learning process outcome.

- **Self-referentiality:** The operating environment is perceived through the components that compose a certain architecture. This might create a distorted perception of the environment. For example, in the architecture (e) there is an apparent dip in the environment graph. If the environment is represented by the number of requests made to the system, at that point it looks like the system received less requests as compared to architectures (d) and (f). This can happen if architecture (e) is slower than architecture (d) and (f), handling less requests in the same period of time. The perceived environment condition is often influenced by the executing architecture. This concept is further discussed in Sec. 3.2.2 when describing 'Dynamic Fitness Landscape'.

- **Observing the past:** The system perception, in terms of its performance or in terms of its operating environment, is always observing the past. Because the metrics and events are only collected after the observation window period (i.e. the system has already processed them). The online learning approach considers a reactive strategy making decisions assuming that the system will continue to behave as it recently did. A prediction strategy, on the other hand,

might consider the general trend of the operating condition, making decisions assuming the direction the environment is taking, rather than the information it has just collected. This thesis explores a reactive approach, but considers the possibility of exploring both approaches in parallel as future work.

- **Hidden trends:** The selection of the aspects to characterise the environment is important in order to have a clear picture of the environment and how it maps onto the perceived performance of the system. An example of this is when the system captures only the volume of data the system is handling but not the type of data. Thus a high increase in the volume of a specific type of data might not impact on the system performance as much as a low volume increase of another type of data. Ignoring the data type in this situation makes the system unable to establish a precise correlation between the environment and its performance, making the learning process more challenging.

- **Multi-dimensionality:** The graph in Fig. 3.1 represents only one dimension of the operating environment. In reality, a proper characterisation of an operating environment considers multi dimensions, requiring the system to report multiple event types on different aspects of the environment. This characteristic makes all previous described challenges multi-dimensional in nature, increasing the difficulty in implementing the learning process.

In summary, the reinforcement learning approach has two main tasks. The first task consists of the proper characterisation of the operating environment so that the experimented architectural compositions can be compared in equivalent environments. This enables the system to establish correlations between architecture features (e.g. the presence of a certain component) and operating conditions characteristic (e.g. the volume level of a specific input type). This task is key to ensure that the system is able to learn what architectural composition is most suitable for a specific operating environment. The second task, on the other hand, is to ensure that the learning approach balance the trade-off between exploration and exploitation. The exploration is important to find optimal composition in case of changes in the operating environment or when new components are added. Exploration is also

important to increase the system knowledge about the operating environments and components and possibly find better architecture assemblies. At the same time, it is desirable that the system exploits the optimal architectures as much as possible. This balance is essential considering this reinforcement learning happens in the 'live system', having real consequence when the system operates sub-optimally during exploration. This is a reactive learning approach, where the system learns when new operating conditions emergent. In parallel the system could use a predictive approach, having an offline algorithm running separately analysing all information gathered by the reinforcement learning process.

3.2.2 Local Emergent Systems Challenges

The challenges described in this section are an intrinsic part of the presented definition of emergent software solutions. These are challenges and properties of emergent system, and they are: divergent optimality, everything is relative, abstracting the environment, data quality and perception errors, application to critical systems, search space complexity, self-referential fitness landscapes, propagating errors as degraded health and developer interaction.

Divergent optimality

The divergent optimality concept comes from the idea that component variants of a given functionality will enable the system to perform optimal in diverse environments. Depending on the components provided, the system might find one optimal composition commonly applicable to divergent environments, or it might find different optimal assemblies for diverse operating conditions. In either case, emergent systems autonomously find, through the composition process, the optimal architectural composition **available** for unexpected environment conditions. It is expected that the components selected to emit metrics and events have their variants also emitting the same data, to facilitate the comparison process.

Everything is Relative

Emergent systems actively build their own understanding of their constituents components and operating environment with no expertly provided knowledge. Thus these systems have to build their own baseline to determine what is 'good' performance through the reinforcement learning process. Additionally, the concept of a good architecture may change when a better architecture is found during exploration. This scenario, makes it imperative for the system to explore and experiment with the available components as much as possible to gain a more comprehensive knowledge of the system. This learning process has to be balanced with the exploitation of the discovered optimal architectures to exploit as much as possible the system findings and maintain a good overall performance. Furthermore, what is known to be a good performing architecture in one operating environment, might not be true under a different operating condition. All the information and knowledge produced from the exploration process needs to be stored and accessible to the system so that it maintains information of the moving performance baseline for each case, avoiding the constant exploration whenever changes occur.

Abstracting the environment

Abstracting the operating environment is an essential feature of emergent systems. This process ensures the representation of the operating condition the system is subjected to. As previously described in 'Everything is Relative' property, the identification of the operating environment is fundamental for the system to create a performance baseline to compare its compositions under equivalent conditions.

The process of abstracting the environment involves the identification of the important aspects that will be monitored from the environment. These aspects should be selected to provide a correlation between performance and the composition components. Furthermore, the value ranges of each aspect are unknown for the system, which aggravates the identification of diverse conditions. The autonomous extraction of such aspects is a difficult problem for machine learning [45]. In addition, the ranges that differentiate environments should also avoid overlaps, preventing the system to oscillate between nearby conditions, triggering unnecessary adaptation.

Data quality and perception errors

The poor quality of the metrics and events collected from the system's components may give the system a false perception of its performance and operating conditions, misleading the system to accumulate false knowledge which may trigger unnecessary adaptation or the selection of suboptimal architectures. In addition, as a related challenge, the absence of an important perception data hides trends in the operating environment from the system. For example, the system may find its performance degrading over time with no apparent change in the operating condition. This scenario occurs when selected events do not match all important aspects of the environment that impact on the system's performance, or when event averages are miscalculated hiding fluctuations in the environment. This apparent mismatch between how the system perceives its environment and the actual environment has direct impact on the emergent system's ability to locate optimal architectural composition. Ideally, emergent systems should detect such blind spots so that the appropriate adjustments can be made to give the system a more precise perception of its environment.

Application to critical systems

The reinforcement learning process may lead the system to temporarily inconsistent states as the result of exploring faulty architectural compositions at runtime. This is the consequence of the learning process advocated in the emergent system context. Considering this fundamental characteristic of emergent systems, it is not advisable to **directly** apply this concept to critical applications, i.e. applications where errors may lead to catastrophic or life-threatening situations. Examples of this type of application are: self-driving vehicles [83] or applications that assist and perform surgical procedures [80]. However, note that these applications also suffer from the dynamism of their operating environment, whilst having minimum tolerance for errors. Emergent software systems might still be suitable for applications in this context, specially because they allow software to handle the unexpected. To mitigate the risk of harmful consequences, however, the system should be extensively trained under carefully designed situations to accumulate enough knowledge and repertoire

to make more accurate decisions when facing the unexpected, still allowing a degree of freedom to react to unknown conditions. The author considers this as a promising avenue for future research.

Dynamic fitness landscapes

Fitness landscapes, considering the defined concept of emergent systems, are graphs representing the fitness values of the available architectures executing on an operating environment. These graphs have an oscillatory shape where the highest peak represents the architecture with the highest fitness value, meaning that it is the optimal architectural composition for the operating environment. As components are removed or replaced from the architecture with the highest fitness value, other valid architectural compositions are formed with lower fitness value. In the context of the emergent software system, as described in the challenge 'everything is relative', for different operating conditions (external stimuli) the fitness landscape of the system changes, meaning that the fitness values of the available compositions change, including the change of the most suitable composition to another composition when the environment changes. This scenario characterises emergent software systems as having **dynamic fitness landscapes**. In [16] Cakar et al defines two types of dynamic fitness landscapes, the one influenced by external stimuli and the one influenced by the system's constituent components. For the changes in the fitness landscape caused by the system internal elements themselves, Cakar et al refers to this changes as **self-referential fitness landscapes**, in this thesis this concept is simply referred to as **self-referentiality** (discussed in Sec. 3.2.1).

Emergent software system's fitness landscapes suffer from both external stimuli and from the interaction of the components that form the system's architecture. The fitness value of a specific architecture is determined by the metrics and events generated by the system's components. Thus the perceived conditions are directly influenced by the executing composition itself. This scenario makes it difficult for the system to differentiate real changes from a distorted perception of the environment. The dynamic fitness landscapes of emergent systems in tandem with the distortion perception problem represent a key challenge to realise emergent systems.

Propagating errors as degraded health

The provided Emergent Software vision considers components in the repository to be correct. For that, this thesis assumes that they are subjected to unit tests. However, unit tests are not sufficient to eliminate or prevent errors in the system, specially considering the interaction of a great number of components. This scenario requires emergent systems to be able to detect and handle errors resulted from the autonomous online composition process. Thus the emergent system's errors are propagated as degraded health. In other words, when a component fails to execute a given function, the components that depend on it also fail to provide their functions. This effect is naturally propagated through a chain of components as a consequence of the error, i.e. the faulty component stops executing returning an error value that propagates throughout the chain of dependent components. This error propagation is then captured by a metric-emitting component associating the performance of the executing architecture to the worst possible metric value. The emergent system can then learn that for that particular operating condition the executing architecture have an unacceptable performance, avoiding it in the future. Furthermore, the system might tag the particular composition to be debugged by human developers.

Developer interaction

The ability to experiment with architectural compositions at runtime and produce information about the operating environment and the components that compose the system, makes emergent software system a valuable member of its own development team, placing the burden to analyse and determine the appropriate software composition into the hands of the machine, reducing the complexity of modern software development. Emergent Software System can be integrated to the software development cycle as a platform to assemble software autonomously and test different architectural compositions at runtime. The knowledge generated by these systems can be used by developers to implement new component variants to be tested at runtime and assist the system to better exploit operating conditions. The use of these systems in the software development process reduces substantially the com-

plexity involved in developing complex systems, considering that the burden to find appropriate software architectures is placed in the system itself.

Aggregation functions

The collection of information generated by the system as a result of monitoring its health status generates a large quantity of data to be stored and handled by the live system. As a consequence, the use of aggregation functions are often necessary to timely handle the monitoring data in a way to not interfere with the system's performance. Different aggregation functions can be used, and depending on the application domain, some functions may mask or distort the system data perception. Furthermore, the perception of the system status may vary from the observer. The use of an inadequate aggregation function may give the system distorted perception of its performance. For example, depending on the aggregation function, few clients might perceive the system as 'very slow', but the system may perceive its performance as 'very fast'. This can be due to some clients (perceiving the system as 'very fast') being prioritised over other clients (perceiving the system as 'very slow'). This can create variability of results across similar experiments, depending on the aggregation function selected. Thus, in some application domains, it is important to consider the appropriate aggregation function other than simple arithmetic average such as geometric means, harmonic means, confidence intervals, and so on.

3.2.3 Distributed Emergent Systems Challenges

This section expands on the intrinsic challenges presented in the previous section, focusing on the emergent systems' challenges and properties that stand out considering a distributed scenario. These challenges are: the combinatorial explosion in the search space, the locus and personality control of meta-structures, information sharing, interference effects and behavioural mismatch.

Combinatorial explosion in search space

The presented Emergent Software vision is fundamentally based on a combinatorial learning process. These systems are built out of the process of combining smaller

components into architectures and testing these architectures at runtime. As components are added to the repository the number of possible architectural assemblies increase exponentially. This scenario becomes considerably worse when considering distributed systems, where the global system architecture is the result of the combination of a set of micro-architectures. Considering that the addition of one component to the repository exponentially increases the number of possible micro-architectures, in the distributed scenario this property has a cascading effect when considering the quantity of participating nodes in the system. The global system architecture is the overall combination of all participating node's micro-architectures. To illustrate how quick the search space grows in a distributed scenario, consider the following example: Two identical web servers with 50 compositions each running on two distinct nodes, and one load balancer with 10 valid compositions running on another node. The load balancer is responsible to forward requests to both web servers. This system has 25,000 valid compositions (50 times 50 times 10). If we add another web server node with 50 composition, the number of global compositions increases to 1,250,000 (50 times 50 times 50 times 10). This is a inherent property of emergent systems and an essential problem to be addressed in order to show the feasibility and to guarantee the consolidation of the emergent system paradigm.

The combinatorial explosion problem is an open issue, and the author does not claim to provide the ultimate solution for this problem. However, this thesis shows how to explore some properties and characteristics of some application domains to mitigate the combinatorial explosion innate of emergent systems. For more details on the proposed solutions, please refer to Chapter 4.

Locus and personality of control

The meta-structures of emergent systems is responsible to i) compose system using software units, ii) collect perception data generated by the components at runtime and iii) learn about the collected information. These meta-structures can operate in different locations with different 'personality'. The meta-structures can operate controlling arbitrary groups of the nodes in the system. They can control the operating of individual nodes in a complete decentralised fashion, or clusters of two,

three or four nodes, etc., up to all nodes of the system, maintaining a centralised system's control. Additionally to that, meta-structures can adopt different personality by, for example, acting in an entirely selfish manner according to the groups local interests, ignoring the rest of the system, or by acting in an altruistic way making local decisions to benefit the global system's interests.

Interference effects

Interacting nodes in a distributed emergent system might interfere with each other's perception of the world, influencing both their metrics and events. Metrics are influenced from interacting nodes when there is a functionality dependency between nodes. For instance, when a load balancer running on a node receives a request, it forwards the request to a web server running on another node to process it. In this scenario, the load balancer's response time depends on the time the web server takes to handle the request. If the web server is in a sub-optimal composition, it directly affects the load balancer's response time, even if the load balancer is in an optimal composition. The same happens to events. For instance, if events are registering information about the requests (e.g. type of files requested – image, text, video, etc.), the perception of the operating environment of the web server is determined by the files the load balancer is forwarding to the web server. In case the load balancer changes its scheduling policy, the web server perceive a change in the operating condition. These interference effects, in a large distributed environment, can compromise the quality of the learning process, making the system less efficient.

Information sharing

Information sharing is essential to reach consensus in a distributed scenario and to optimise the exploration phase in the learning process. Sharing information among groups of nodes controlled by meta-structures gives the group an external perceptive of the system and how the group is effecting the system's goal. Furthermore, information sharing can help optimise the learning process by diving architectural exploration among nodes with similar resources and share the outcome among them. This mitigates the combinatorial explosion problem by diving the search space among

equivalent nodes, and can help manage interference effects by enabling exploration to specific groups whilst their adjacent nodes remains in a fixed architecture. After the learning process occurs the knowledge is shared among the neighbourhood nodes, minimising interference effects. Information sharing occurs in the level of meta-structures, and the definition of the information to be shared and the related challenges involved in the sharing process is an issue to be further explored.

Behavioural mismatches

In a distributed system scenario, the micro-architectures in different nodes might implement behaviours that do not match to other interacting nodes. For examples, one micro-architecture might encode different schemes of encryption algorithms, making the nodes incompatible. A trivial solution for this problem is to manually define rules or impose constraints regarding aspects of the system behaviour. As discussed before, this solution is against the philosophy of emergent systems due to its restraining nature, limiting the reinforcement learning process. Emergent systems should have the freedom to explore and learn autonomously whilst simultaneously maintaining the system in a globally valid composition.

3.3 Thesis Scope

In this chapter a series of challenges and properties have been presented in both local and distributed settings of Emergent Software Systems. Although these challenges are important to realise the concept of these systems, some of them are open issues and demand further investigation. This thesis focuses on the main challenges that allow the creation of the basis on which emergent solutions can be built on, showing the feasibility and the benefits of pushing emergent systems to the real world. In particular the scope of this thesis is on the challenges of self-assembling mechanisms, component-monitoring mechanisms and the reinforcement learning algorithm. These three main concerns, discussed in detail in Chapter 4, form the main challenges that support the implementation of emergent systems. Considering the specific challenges presented in this chapter, this thesis addresses the following:

- **Online Learning Challenges (Sec. 3.2.1):** Comparison difficulty, self-referentiality and multi-dimensionality;
- **Local Emergent Systems Challenges (Sec. 3.2.2):** Divergent optimality, everything is relative, abstracting the environment and dynamic fitness landscapes;
- **Distributed Emergent Systems Challenges (Sec. 3.2.3):** Combinatorial explosion, locus and personality of control, interference effects and behavioural mismatches.

This thesis presents a baseline learning algorithm that addresses the comparison difficulty, everything is relative, abstracting the environment, dynamic fitness landscapes and multi-dimensionality challenges and properties of emergent systems. As an extension of the baseline algorithm, this thesis also presents a feature-based learning algorithm that focuses on the combinatorial explosion problem. This thesis also presents an environment classification algorithm that directly tackles self-referentiality, multi-dimensionality and dynamic fitness landscapes challenges. Finally, this thesis explores different coordination algorithms (complete centralised, complete decentralised and a hybrid approach) for locus and personality of control in distributed settings. More specifically, this thesis proposes a hierarchical coordination learning approach (a hybrid approach – i.e. partially decentralised) to handle combinatorial explosion, interference effects and behavioural mismatches challenges. Finally, the divergent optimality property, which is essential to realise emergent systems, is very evident in the selected case study, making it ideal to explore the concept of emergent software systems. On the other hand, the following challenges are out of this thesis' scope:

- **Online Learning Challenges (Sec. 3.2.1):** Mid-window changes, observing the past and hidden trends;
- **Local Emergent Systems Challenges (Sec. 3.2.2):** Data quality and perception errors, application to critical systems, propagating errors as degraded health and developer interaction;

- **Distributed Emergent Systems Challenges (Sec. 3.2.3):** Information sharing.

These challenges are not explored because i) they are not crucial to this thesis' research goals, and ii) they require substantial investigation. Particularly, with regards to the 'observing the past' challenge, this thesis focuses on a reactive learning approach, rather than a mixed approach combining offline workload prediction and online observations of the executing environment, which is an interesting path for future work. The 'information sharing' challenge, for the evaluated scenario, is seen as an improvement over the learning approach, rather than an essential part of the solution. Nonetheless, the author recognises the importance of information sharing in distributed learning solutions, and its key role in supporting global optimal convergence, which is an interesting challenge to explore in the future. The 'propagating errors as degraded health' and 'data quality and perception errors' challenges are partially addressed, but further investigation is needed, specially in terms of scenarios deliberately designed with high levels of error occurrence. The 'mid-window changes' and 'hidden trends' are identified limitations of the proposed environment classification algorithm and demand further investigation. Finally, the 'application to critical systems' and 'developer interactions' challenges are important to further explore the potential of Emergent Systems, but are not part of the research goals.

3.4 Summary

This chapter has defined Emergent Software Systems. These systems are autonomously composed from a set of component units according to a goal and influenced by the operating environment condition. The composition of such systems occurs at runtime considering the events affecting the system in production. The composition process is executed through an reinforcement learning approach, where the system architectural composition is assembled and experimented with at runtime, giving the system information to determine what the most appropriate architectural design is for the presented environment. This information is stored and used when previously seen conditions are encountered in the future. Reinforcement learning approaches

have to balance the exploration process that allows the system to accumulate knowledge and the exploitation of the most suitable architectural composition. Balancing between exploration and exploitation is key to maintain a good system performance, considering that inadequate compositions has real impact and consequences.

This chapter also presented the key properties of such systems and the challenges to realise the vision of emergent systems. An important challenge is the implementation of the reinforcement learning process. This process presents challenging issues such as: the difficulty to compare multiple architectural compositions on dynamic operating environments with unexpected changes, and the process to abstract and characterise operating environments. Another key challenge to realise emergent systems is the combinatorial explosion problem, which is severe in large distributed systems. This is a key challenge to address considering the combinatorial nature of the proposed approach. Other challenges were also described such as: hidden trends and multi-dimensionality characteristic of the operating environment, the interference effects in the learning process of adjacent nodes, the mechanisms to share information among nodes to minimise impact on the system performance and the application of emergent system concept to critical applications. Although these challenges were presented in this chapter, it is important to remark that this thesis only focuses on the challenges that demonstrate the feasibility of emergent solutions. This involves: the combinatorial explosion problem, the interference effect and the abstraction of the operating environment. The challenges that are not investigated in depth in this thesis are presented as promising research directions to further consolidate the Emergent Software Systems paradigm.

CHAPTER 4

Implementation

This chapter introduces the framework to enable the Emergent Software Systems concept in both local and distributed environments. The local framework consists of three modules: **Perception**, **Assembly** and **Learning**, which names the framework PAL. These modules are the base of the concept of Emergent Systems and support online software composition, monitoring and learning. The distributed framework is an extension of the PAL framework, and support the interaction of emergent systems in distributed settings. The distributed framework modules are: **Factory**, **Registry** and **Validator** modules. This chapter describes the implementation of each module, and details their interaction to realise the concept of Emergent Systems. The chapter is divided into the following sections: Sec. 4.1 presents an overview of the local modules and their interaction to support the Emergent System concept. Sec. 4.2 presents details of the Assembly module and all the concepts and principles behind autonomous software composition. Sec. 4.3 introduces the Perception module and the concepts that enable the system to collect information about its health status and its operating conditions. Sec. 4.4 details the reinforcement learning approach. Finally, the distributed framework modules are introduced in Sec. 4.5. These modules enable the expansion of the PAL framework to support Emergent Systems in distributed environments, realising the concept of Distributed Emergent Systems.

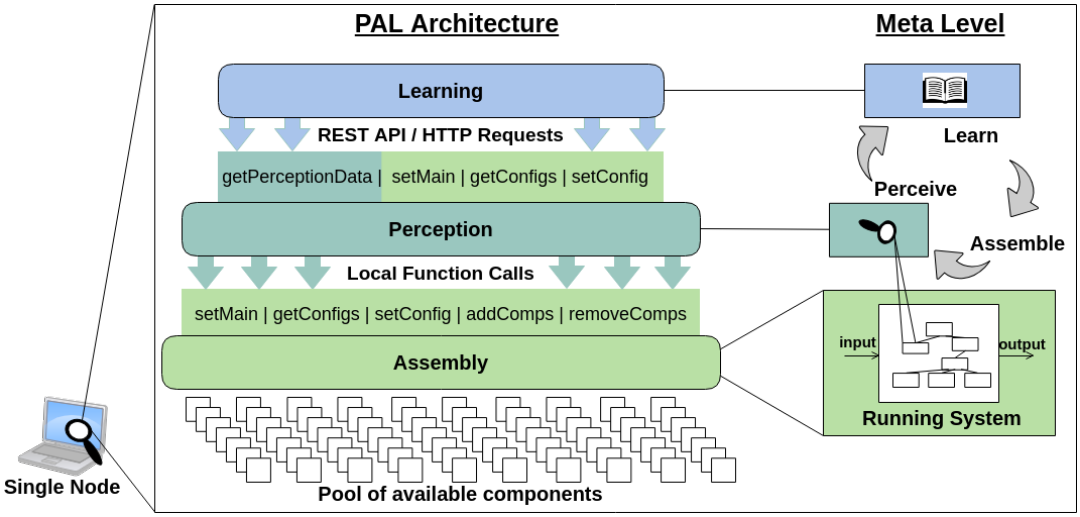

Figure 4.1: PAL framework architecture.

4.1 Local Emergent Systems

The local framework supports the concept of Emergent Software System based on three modules: **Perception**, **Assembly** and **Learning**, illustrated in Fig 4.1. As described in Chapter 3, Emergent Systems have the ability to autonomously compose software architectures from small and reusable software components according to the observed operating environment. The Assembly searches for components in the repository, creates an in-memory representation of all available architectural compositions the system can be assembled into, and supports composition changes at runtime. The Perception module generates and adds proxy components to the system's architectures to monitor the system health status and the operating environment. Finally, the Learning module leads the entire autonomous software design-by-composition process through the reinforcement learning approach implemented.

The interaction among the three modules is depicted in Fig. 4.1. The framework is organised in a multi-tier architecture in which the upper layers use the functions of the lower-level layers to abstract fine-grained function-calls and provide higher level functionalities. The multi-tier modular architecture ensures that each of the participating modules implement a well-defined set of functionalities to the upper-level. Although the main modules were designed to be generic and applicable to any application domain, this modular structure allows each module to be replaced by a

variant, when applicable, to explore specificities of particular domains. This characteristic makes the framework flexible, facilitating, for example, the experimentation and comparison of a variety of learning algorithms. The following sections focus on the implementation details and the functionalities provided by the PAL framework.

4.2 Assembly Module

This section provides details on the Assembly module, covering the component-based model used to enable the concept of Emergent Software System and the provided functionalities for the upper level layers. The definition of Emergent Software Systems is based on the composition of fully functioning software architectures from small components. In order to assemble components into functioning architectures, the Assembly module searches for components in the repository, gathers information about them (and their variants) and assemble them into a fully functioning system with composition options. The Assembly module uses component-based runtime provided functions to execute component replacement and composition at execution time, whilst abstracting the processes that support software architectures composition and (re)composition used by the Learning module to realise the reinforcement learning process. The following sections describe the role of a component-based model in the autonomous composition process, enabling architectural adaptation by component replacement, and the API functions provided by the Assembly module.

4.2.1 Component-based Model

Component-based models are essential to support the concept of Emergent Software. Mainly because they provide the necessary information about components to realise autonomous software composition. The information provided supports autonomous component connections to form functioning software architectures, avoiding random component compositions and offline testing to find compositions that work. Furthermore, since component models allow developers to express features through interfaces that are explicitly expressed when coding components, the component-based model eliminates the need to use extra models to label the system code and

define features. This extra models label code to define what parts of the system are adaptive and how to adapt them (e.g. what features could be replaced, and how to replace them). For example, feature models (such as in [32]) are often used to represent how pieces of code interact with each other, and how they could be replaced to support adaptation. Using such models, the developers have to code the software and then describe a model capturing the relationship between different parts of the code, increasing development effort. The component model, on the other hand, is a way to abstract that need, integrating fundamental information for component adaptation in components. This characteristic reduces development efforts during the system design phase, facilitating the provision of software adaptation throughout the entire system, rather than to specific parts that was previously modelled.

This work uses the Dana[1] component model described in [70]. Dana is a multi-purpose programming language that inherently provides a component-based model and runtime support to component adaptation in fine-grained complex modular structures. Component-based models require the definition of interfaces and components. Interfaces define functions signatures (i.e. function names, return types and the list parameters with their expected data types). Each interface expects *at least one* component implementing all functions defined by the interface. Components can *provide* implementation for multiple interfaces and *require* other interfaces to support their own implementation. Also, multiple components might implement the same interface using different approaches, those are often referred to as **component variants**, and by replacing those variants in an executing software is how the adaptation process occurs. This 'provide-require' policy is a central part in a component-based model, allowing the language runtime to connect different components (following the provided-required policy) to create fully functioning software. Software evolution or adaptation is realised by replacing components in the resulting software architecture at runtime without interrupting the system services. Besides changing the implementation detail of a system's functionality, online component replacement might also change the system's architecture by adding a set of new

[1]Please refer to http://www.projectdana.com for more practical information on the language, its component-based model and its online adaptation mechanism.

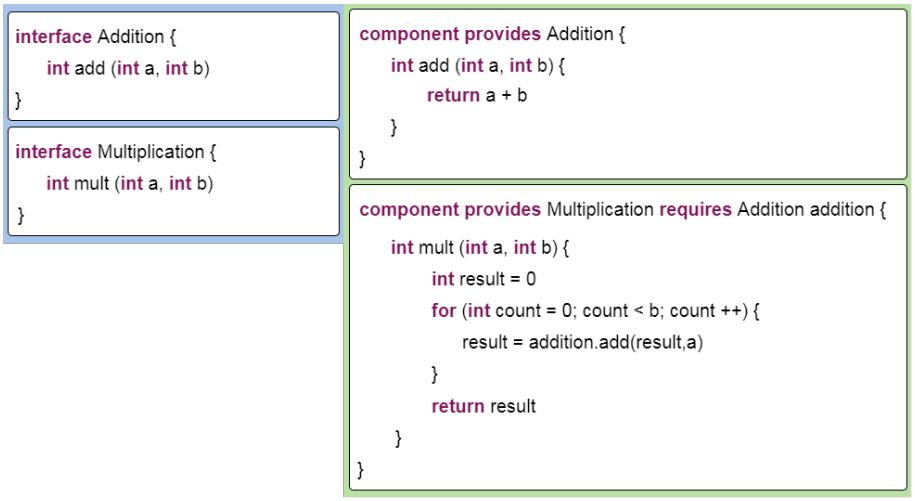

Figure 4.2: Example of interfaces and components. Note that Multiplication requires an external interface (Addition) to complete its implementation. The code is written using the Dana programming language syntax.

components required to support the new component's implementation.

An example of the Dana language syntax and its component-based model is illustrated in Fig. 4.2. The image shows two interfaces (on the left side of the picture): *Addition* and *Multiplication*. Each interface presents the definition of functions with the function's name, return type, and a list of parameters with their types. On the right side of the image, two components providing implementation for aforementioned interfaces are illustrated. The *Multiplication* component is a simple example that illustrates a component implementation depending on another interface. In this case, the *Multiplication* component relies on the interface *Addition* to support its implementation. Therefore, whenever the *Multiplication* component is used, any component providing *Addition* is required to be connected to it. Considering those interfaces and components used to create a **Calculator** program: if there are multiple components providing *Addition* interface, the *Multiplication* component, in order to work, could be connected to any component providing *Addition*. Furthermore, we could also assume the existence of a *Multiplication* component variant that, for example, does not require the *Addition* interface, thus implementing the *Multiplication* interface in a different form. Therefore, in this example, there exists a variety of architectural options to realise the **Calculator** program, each option implementing the operation *Multiplication* in a different way, by either connecting

Multiplication with a variety of *Addition* components or replacing *Multiplication* with a variant that does not require the *Addition* interface.

The Dana runtime provides basic functions to abstract the online architectural composition and adaptation process. The abstraction provided by the language supports, through a single function call, the realisation of software adaptation or composition, involving: `loading` new components to memory, `pausing` old component execution, `transferring` component states (when applicable – i.e. *replacing* stateful components) and `connecting` the new component to the requiring component. Dana transparently executes the aforementioned tasks without further actions of the developer. The Assembly module then interacts and further extends these low level functions to determine and control the composition and adaptation process at runtime. A list of functions provided by the Assembly module is detailed below.

4.2.2 Assembly Module API

The Assembly module provides the following functionalities:

- `bool setMain(char compName[])`: This function is responsible to start the assembling process of the target system's architecture. The Assembly module is capable of assembling an entire system's architecture and its variations from a main component. This procedure is detailed later in this section.

- `String[] getConfigs()`: This function returns a list of the available architectural descriptions. These descriptions are string representations of architectures, containing a list of components and how these components are connected in the architecture. Furthermore, these descriptions are used by the Learning module to reason and learn about architectural compositions.

- `bool setConfig(char configDesc[])`: This function is used to change the executing architectural composition to another at runtime. This function receives as parameter the architectural description of the new composition.

- `char[] getConfig()`: This function returns the executing software composition at the moment that this function is invoked.

- `bool removeComp(char compName[])`: This function removes a specific component from the list of components being used by the Assembly module to form the available software compositions. As a result, there is a decrease in the number of available software compositions. This function expects the name of the component to be removed.

- `bool addComp(char compName[])`: This function adds a component to the list of available components. As a result, there is an increase in the number of possible architectural compositions. This function receives the name of the component, searches for it in the repository and loads it into the Assembly module to create more architectural compositions.

The `setMain()` function sets the procedure to compose the software architecture and variations of it from a root component. The root component implements the main function, i.e. the point where the program starts its execution. From that component, the Assembly module extracts the name of required interfaces, and for each of the listed required interfaces the Assembly module searches in the repository for components that provide those interfaces. In case the Assembly module locates multiple components providing the same interface, it gathers these component variants (multiple components providing the same interface) and attaches them to the interface creating an adaptation point, i.e. a point where there are multiple components to choose from. These adaptation points are represented in Fig. 4.3 in the interfaces A, B and C. After loading all components that provide the interfaces required by the main component, the Assembly module repeats the same steps for each of the loaded components, and continues with the process of loading components and their variants for the required interfaces until the entire architectural tree is composed with the necessary components to realise at least one working software composition, i.e. having at least one component providing every required interface.

A working composition consists of the selection of one component in each required interface on the tree (an example is illustrated in Fig. 4.3). The components marked in blue, in the image, are part of the executing composition. After the process of loading components and their variants, the Assembly module selects and runs a

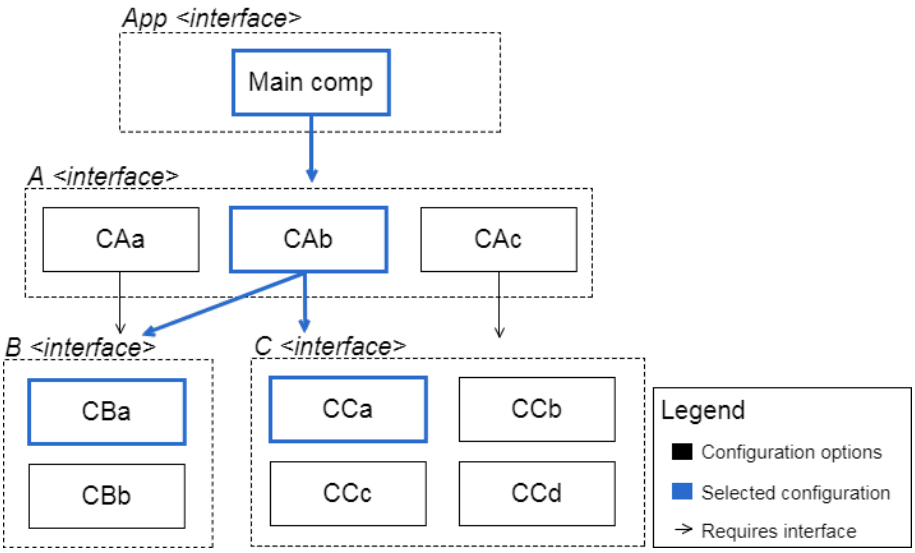

Figure 4.3: This is an illustration of a generic architecture represented by the Assembly module. This is the result of executing the Assembly module function *setMain*.

random architectural composition. The illustrated example in Fig. 4.3 presents a total of 14 available architectural composition. The Assembly module provides functions to add new components (which were not in the repository when `setMain` was first executed), and a function to remove components as candidates to provide a specific interface, not allowing the removal of components that have no variants. These two functions influence the number of available architectural composition. Furthermore, the Assembly module provides functions to return a list of architectural descriptions of the available architectural compositions, a function to return the description of the executing composition, and a function to change from one option to another. These functions work by acting on the information of the available components, illustrated as a tree in Fig. 4.3, stored in the Assembly module.

The Assembly module provides architectural descriptions to enable external modules to reason about the running software structure and to interact with the Assembly module to change the running composition. This is a text-based description that provides information about components integrating a specific composition and their relationship, i.e. how the components are connected. The architectural descriptions are useful in the PAL framework for two main reason: i) enable external modules to interact with variations of assembly module and ii) enable external modules to reason and infer information about the architectures. The use of ar-

chitectural descriptions facilitate the interactions between the framework modules (mainly Perception and Learning) and the Assembly module. Since the description contains all details to reassemble any architectural composition, they enable, for example, the Learning module to reason about the available architectural composition to understand the impact of specific components in the architecture. Furthermore, the descriptions support the creation of different implementation versions of the Assembly module, enabling them to internally represent compositions in different forms, whilst maintaining a consistent format to refer to valid software compositions. For more information on architectural descriptions, refer to Appendix A.

4.3 Perception Module

The Perception module uses the Assembly module functions to enable the system to monitor its health status and collect information about the operating environment. Furthermore, the Perception module also provides, through a RESTful API, the Assembly functions to the modules interacting with Perception, facilitating access to functions that support the learning process, such as: functions to change to the software composition, to get a list of possible software compositions, to add new components or remove components, and to get the executing composition description. This section is divided into two subsections: the first describes the perception data (events and metrics), which are used to represent the system's health status and to classify operating environments. The other subsection describes a special type of component (proxy) that is autonomously generated and inserted into the software architecture to collect metrics and events from executing compositions.

4.3.1 Perception Data (Events and Metrics)

The perception data (Event and Metric) are data types through which the system represents its health status and operating conditions, both essential information for the system to learn about its internal composition and execution environment. These data are used by the learning algorithm to classify environments and determine the level of satisfaction of the system's goal, serving as the base on which the

```
data Metric {
    char name[]
    dec value
    bool preferHighValue
}

data Event {
    char type[]
    char name[]
    dec value
}
```

Figure 4.4: Metric and Event data types in the Dana programming language.

system learns and makes design decision choices. The metrics are used to represent aspects of system health status (e.g. performance, security level, etc.), whilst events are created to store environment features values (e.g. input patterns, hardware characteristics). In tandem, both events and metrics are used by the Learning module to realise its reinforcement learning approach. As an important characteristic, both data types (see Fig. 4.4) were designed to capture information regardless of the application domain, enabling the system to learn about any type of system by setting the appropriate event types and metrics to ensure a satisfying learning outcome.

The metric data type is composed of the `name`, `value` and `preferHighValue` attributes. The `name` attribute represent the aspect of the system related to its performance and health status that is being monitored. This field could be set as *'response time'* or *'number of active threads'* or any other *string* that represents aspects of the system that can be quantitatively measured and monitored. The `value` attribute stores the correspondent value associated to the metric. For example, for *'response time'* the system may store time in milliseconds that the system takes to process incoming requests. Additionally, metrics are not only used to represent system status, but also to express system's goals. The `preferHighValue` when set to **true** tells the system to find the architectural composition that maximises the `value` attribute. Contrarily, when `preferHighValue` is set to **false** the system searches for the composition with the lowest `value` attribute. The `name` and `preferHighValue`

is often manually define by the domain expert, and the system obtains the `value` as it executes. Note that it is possible to create multi-goal Emergent Systems using the concept of metrics. However, the focus of this thesis is to prove the feasibility of the Emergent System approach, and not to explore autonomous optimisation of multi-goal systems. Thus, to narrow the focus of the thesis, I focus on single goal (e.g. performance) optimisation, defining a single metric for the entire system.

Events are used to represent different features of the operating environment. The event data type represent features with the following attributes: `type`, `name` and `value`. The `type` attribute stores the name of the feature, for example, in cases where it is important to characterise request patterns, a possible event type could be defined as *'request type'*. The attribute `name` defines the name of the environment feature that was perceived. In our request pattern example, the `name` attribute could be set to *'text'* when the system received requests to retrieve text files, or *'image'* to represent different labels of the same *'request type'*. Finally, the attribute `value` quantifies the observed attribute. For example, storing the size (e.g. in bytes) of the requested files. This allows the system to understand the operating environment it is running on, in this example, by understanding how differences in the file size of certain request types affect the system operation. The `type` values is often manually defined, but the `value` and `name` are collected by the system.

After the collection of events and metrics from the executing system, other important attributes can be inferred, for example: the number of times a certain attributed was perceived (e.g. number of requests for text files). This could be an important information that assists the system to determine whether an event is recurrent or not, which might be an important aspect to characterise the operating environment. Another important information is the time the metrics and events were collected. Time-stamped data enables the system to establish a time-line with information on how the system and the operating environment behaves in time period. The collection of events and metrics is done by proxies, a special type of component autonomously generated to collect (and time-stamp) information about the system. The Perception Proxies are discussed in the next session (Sec. 4.3.2).

4.3.2 Proxy Components

Proxy components are essential elements in the Perception module. These special types of components are responsible to extract events and metrics from the system and operating environment. The proxy component complies with the same policy of other components in a component-based model, and thus are indistinguishable from other components to the Assembly module in terms of code structure, syntax and how to connect to other components. The only difference is that proxy components are annotated to allow the Assembly module to locate them to extract the collected metrics and events and to avoid connecting two or more proxy components together.

A special characteristic of all proxies is that they need to provide and require, at the same time, the interface the proxy is supposed to monitor. By requiring and providing the same interface, the proxy component can be inserted between two components; one that requires the interface (e.g. component A) and the another that provides the interface (e.g. component B). Then the proxy can intercept function calls from A to component B, and extract information from the interaction between A and B. Fig. 4.5 illustrates a proxy component inserted in the system.

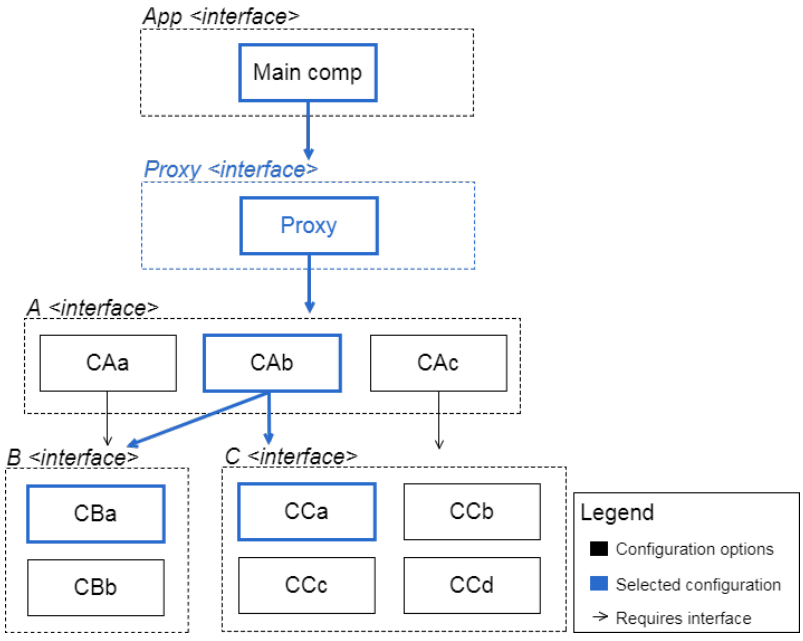

Figure 4.5: Proxy component inserted in the system's architecture.

The Perception module was designed to support a proxy-based monitoring solution, mainly because proxies are non-intrusive, i.e. they do not require code to be inserted into the component to extract information for the system. Thus it is possible to separate the code responsible to monitor a specific component from the component itself, not requiring component developers (human or a machine – for machine-generated components) to code functions to monitor the component, avoiding it to be incompatible with the applied learning algorithm. Furthermore, this separation of concerns (component code and monitoring code) also makes the component development process more flexible, eliminating the need for a model to dictate how components should be coded in order to be monitored.

This proxy-based approach implemented by the Perception module also enables the autonomous generation of proxy components at runtime, according to the necessity and goals of the Learning module. The proxy generation procedure is further detailed in the subsequent subsection. Furthermore, considering that a Proxy component is generated to monitor target interfaces rather than components, and that the same interface might be required by multiple components throughout the system, a Proxy Expression Language was designed to allow the Learning module to express exactly what interface to place the proxies in the system structure. The Proxy Expression Language is also described in detail in a subsequent section.

Proxy Generation Process

The proxy generation process gives the system flexibility by autonomously generating the appropriate proxy components for the goals of the Learning module, regardless of the system, the application domain or components to be monitored. The proxy generation process is triggered by the Learning module, once it obtains the system goals and the list of available architectural compositions, the Learning module triggers the proxy component generation process. This generation process requires as input the target interface, and components responsible to generate the metrics and events to be collected from the target interface components.

The Perception module provides a range of components that implement the collection of specific metrics and events and allow the creation of new metric/event-

collectors type of components. For example, to collect the response time in milliseconds, i.e. the time (in milliseconds) that takes a certain function to execute. The component `ResponseTime` offers the code necessary to collect such metric. This component provides a *timer* that is triggered right before the function is invoked, and stopped right after the function finishes processing, returning the calculated time the function took to execute. The same applies for collecting event information. A given example for event is, in the context of a system that handles HTTP requests, the collection of mime-types from incoming requests. The component `MimeType` is also implemented and provided. This component provides a function that receives the request in a raw-text format and returns the extracted mime-type of the requested resource. Both described examples of components that generate (i.e. calculate or collect) metrics and events are provided by the Perception module. As new components of this kind are implemented, the Perception module makes them available to be used in the generation of proxy components.

The metrics and events being generated in the proxy are timestamped once they are created and stored in an object named `Container`. This object is responsible to average the metrics and events of the same type, update the a 'count' variable to express the number of a certain event or metric and make them available to the Perception module to format them into a JSON string as **perception data** to send to the Learning module upon request. Note that averaging events and metrics values is an important process to reduce the amount of data stored in the proxy component. Considering that each input processed by the systems has the potential to create metrics and events, storing each individual event and metric generated (not averaging them) consumes unnecessarily large memory quantities. An example of a generated proxy component is in Appendix C.

Proxy Expression Language

Proxy components focus on monitoring interfaces, rather than particular components. The reason is that a proxy designed to monitor an interface monitors a standardised feature and thus is able to monitor all component variants that provide such feature, independent of individual component implementation. Furthermore, in case

the system adapts its architectural composition from a component that provides the monitored component to a variant, the Perception module can replace the component without replacing the proxy component, reducing the amount of changes performed by the Assembly module. The main problem of monitoring interfaces rather than components is that an interface can be required by multiple component present in the software architecture. As a result of this property, when placing a proxy to monitor an interface, the Perception module places multiple proxy components to monitor *every* component that requires the interface, adding multiple proxy components throughout the architecture, impacting the performance of the system. In order to provide fine-grain control over the process to place proxy components in the architectural composition, whilst avoiding the Learning module to indicate what component (instead of interface) to be monitored every time an architectural change occurs, the concept of the Proxy Expression Language was designed.

The Proxy Expression Language (PEL) is a tool to precisely express where to place proxy components in the architectural composition. This language is used by the Learning module to ensure that the generated proxy components are kept in place, monitoring the intended interface even when the software compositions are constantly changing. The Perception module is responsible to interpret the expression and make a list of architectural compositions without proxy components associating these compositions with their equivalent with proxy components added to the right place (interface or component) according to the expression created by the Learning module. Therefore, whenever the Learning module requests the software composition to be changed, the Perception module compares the new architecture description with its list and decides: if the new composition matches one of the compositions in the list, the Perception module changes the software composition to the corresponding new composition with the proxy component in the right place as according to the expression. This strategy enables the Learning module to work with architectural descriptions with no proxy component added, the Perception module is responsible to apply the PEL expression provided by the Learning module once (before it starts the experimentation process) and to transparently add proxy components to the appropriate place in the software architectural composition.

The language provides levels of control to the Learning module when placing proxy components in the system, allowing it to place proxies to only one specific component, or to an interface, or to any level between the two, for example, by placing a proxy component to monitor an interface but only if that interface is required by a specific component (as oppose to every component that requires that interface). The Learning module, using the expression language, is able to create expressions as generic or as specific as it requires, having the Perception module to use that expression against the architectural description to determine the place to add the proxy components in the software architecture. Adding multiple proxy components to different parts of the software architecture gives a more precise perspective of the system status and its operating environment, but adds overhead to process and collect events and metrics that impacts the system performance. An important challenge in the process of adding proxies to the software architecture is to identify the amount of proxy components and their optimal position to reduce the number of proxies whilst maximising the quality of the collected data. Tests with the case study (see Chapter 5) shows that for the performance (response time) proxies the higher in the software architecture these components are placed the better they capture the overall system performance. Considering that in Emergent System software architecture a component calls functions in lower level components (required components), by placing proxies in the highest level possible the performance proxy is able to calculate the time a stack of function calls takes to execute. This is not a general rule, and may vary according to the components part of the architecture. An exception to this scenario, is when, at some point in the function call stack, a function is called to be executed in a different *thread*. Although further research is required to solve this challenge, this thesis demonstrates that this simple strategy to place proxy components in the highest part of the architecture tree works well for the explored application domain and used components.

The Proxy Expression Language is a flexible and precise way to express proxy components placement in architectural compositions. For the interested reader, further details of the language, as well as examples, are described in Appendix B.

4.4 Learning Module

The Learning module is responsible to guide the processes to realise Emergent Systems. This module, accessing lower level modules (Perception and Assembly), triggers the deployment and composition process, requires Proxy generation by selecting Metrics and Events generators according to the system goals, and controls the entire active learning process, triggering the exploration and exploitation phases, classifying the operating environment and identifying the most suitable architectural composition. This section describes details of the learning process (exploration and exploitation phases), showing two learning strategies: i) the *Brute-Force* approach, and ii) the *Feature-based* approach. Furthermore, this section also describes the environment classification algorithm used in this thesis.

The learning process executes with no prior knowledge about the target system, nor any information about the operating conditions to which the system will be exposed. The main task of the Learning module is to understand the correlations between assembled collection of components (i.e. the system's behaviour), and the system's perception of its performance, in each identified operating conditions. The Learning module executes its main task by requesting the Assembly module to change the system's architectural composition to experiment with the identified operating environment changes and by observing its own performance and the operating conditions through Perception. Sec. 4.4.1 details the learning process.

4.4.1 Learning Algorithms

The generic approach to realise the reinforcement learning involves three main tasks. First, it must be able to characterise and classify features in the software's operating environment (derived from the stream of events being emitted) so that the performance of different compositions can be compared in equivalent environments, and so that the learning module can 'remember' which compositions work best in each environment (i.e. to save re-learning each time a recurring environment is encountered). Second, after finding the most suitable architectural composition to the perceived environment, the system exploits the optimal composition whilst observing the en-

vironment and the system performance to detect any changes and trigger learning again, in case it detect unforeseen changes. Finally, the third task, as in any online learning system, the learning module must balance the trade-off between exploring options for which there is insufficient information and exploiting options known to be good [79]. This third task is important because our emergent software framework operates on live software, and sub-optimal performance has real consequences.

As discussed in Chapter 3, performing online reinforcement learning is highly challenging. The software is not in control of its operating environment and so cannot know in advance when it may be able to reliably compare any two software compositions against the same operating condition. In addition, there are complex interactions between the process of exploration itself and the environment, where selecting a 'good' composition may impact on the system performance and change the perceived environment. The learning approach proposed is inspired by reinforcement learning [79], but tailored for our particular problem space. The two learning approaches presented in this section continually discovers optimal assemblies by exploring the search space whilst simultaneously classifying observed operating condition's features into labelled environments. The *Brute-force* approach explores all available compositions before opting for the best performing option. This approach is presented as a base line to compare with other learning strategies, because it guarantees (when operating conditions are maintained during exploration) to locate the global optimal composition. The *Feature-based* approach, on the other hand, explores the search space analysing features instead of individual compositions, and thus it reduces the search space by focusing on representative compositions of available features. This approach relies on domain specific assumptions and explores them to be sufficiently scalable to support learning in distributed systems.

Brute-force strategy

The *Brute-force* algorithm applies a standard 'exploration activity' to both characterise the current environment and also identify the best composition for that environment, shown on lines 3-7. The system triggers exploration whenever it encounters high uncertainty in its decision making process – where this uncertainty

comes either from (i) having no information at all (i.e. system startup); (ii) the current environment characteristics deviating outside of expected ranges from existing experience, or (iii) current system performance deviating beyond its expected range.

Algorithm 1 Experimental Learning Algorithm (brute-force strategy)

```
1: while running do
2:     //perform exploration activity - brute-force approach
3:     for each c in assembly.getConfigs() do
4:         assembly.selectConfig(c)
5:         wait for w_t
6:         store perception.getPerception() for c
7:     end for
8:     //select the new composition to use
9:     store environment ep as max : min of event types
10:    assembly.selectConfig(best known for ep)
11:    //wait for conditions to change
12:    newExploration = false
13:    while newExploration == false do
14:        wait until (different environment ep detected) or
15:                (performance degrades) for >= w_t * 3
16:        if different ep and ep is previously known then
17:            assembly.selectConfig(best known for ep)
18:        else
19:            newExploration = true
20:        end if
21:    end while
22: end while
```

The exploration activity tries every possible composition for a fixed-length 'observation window' w_t, such that the total time spent exploring is:

$$w_t * \texttt{length(getConfigs())}$$

The observation window w_t is defined according to the application domain. A reasonable time frame value (w_t) for the case study investigated in this thesis is 10 seconds for high quality learning results. The `getConfigs()` function is provided by the Assembly module and returns all available software compositions. After trying every composition, the learning module then characterises the data collected over the entire exploration process to determine the best course of action.

Specifically, after an exploration activity, the learning module selects the best-performing composition for use and enters its exploitation state. The selected action continues to be monitored every w_t and analysed for its suitability. A change may occur if either (i) perceived events during w_t show that this is a different event pattern, or (ii) perceived metrics during w_t show degraded performance. To avoid frequent oscillation, in either case the algorithm waits for $w_t * 3$ of consistently observed behaviour before changing its current course. In case (i), if the detected event pattern is one that has been previously seen, the matching best composition is simply selected. In all other cases a new exploration activity is triggered. This process of exploration / exploitation repeats continually, where the amount of exploration reduces as fewer new environments are seen. Note that this learning algorithm, which is based on an exhaustive exploration phase, is not designed to scale up to large systems with thousands of compositions but rather serves as a proof of concept and useful baseline against which to compare more sophisticated algorithms.

Feature-based Strategy

The *Feature-based* learning strategy is an alternative solution for the experimentation learning process. Instead of experimenting with all available architectural compositions, the system experiments with representative architectures for a specific feature. A feature is a functionality defined by an interface, and therefore, in this context, the words feature and interface are used interchangeably. A variety of components can be created to provide a single feature (interface) by implementing the same functionality differently, requiring, for example, other features (sub-features) in their implementation. The algorithm exploits that property, by choosing the component variant for a specific feature F considering the sub-features required by the component variants of F. Furthermore, the algorithm does not try all variants of all features (interfaces) in the system's architecture. Instead, it only tries component variants for the most suitable features, testing only one component variant for features that were not suitable for the operating environment. Therefore, this approach reduces the search space and supports a faster learning process, providing a more scalable solution. However, this approach relies on the assumption that the

worst component variant for the best feature is better than the other components of the other features, which might not be true for every application domain.

The *Feature-based* strategy maintains the essence of the reinforcement learning algorithm, i.e. the entire process described in the Brute-force strategy still applies (including most of Algorithm 1). The only part of the algorithm that the feature-based strategy changes is the code between lines 3-7. This part of the algorithm experiments every available composition c provided by the Assembly module. This feature-based approach, however, navigates through the software architecture from top-down deciding on the component variants it encounters for each feature (interface) that has multiple component variants. Once it determines the best component variant for a feature, it transverses down the tree considering only the chosen component variant, eliminating the other branches defined by the remaining (not selected) variants. This process continues until the algorithm decides on every component variant of every feature it considers relevant, resulting in the optimal discovered architecture. The rest of the algorithm, including the environment classification and the details of the exploitation phase, are the same as in Algorithm 1.

The *Feature-based* approach is detailed in Algorithm 2. The algorithm considers a tree-like structure that represents all architectural compositions for the software, as illustrated in Fig. 4.3 in the Assembly module section (Sec. 4.2). Considering a tree-like structure representing the available architectural compositions, the algorithm selects the first interface (from the root down) with component variants. Note that Emergent Systems only have composition options `if and only if` there exist component variants for at least one required interface. The system has no option for required interfaces with only one component implementing their functionalities. Therefore, the algorithm navigates the tree from top-down defining the best component variant for only the required interfaces that has ≥ 2 components implementing the interface. The algorithm tests all interfaces with component variants that were required by a selected component in a higher level, ignoring the interfaces with component variants that are required by components that were not selected at any level of the tree. The selection of component variants in each required interface is done by executing architectural compositions containing the component variants,

Algorithm 2 Feature-based Algorithm (this replaces lines 3-7 in Algorithm 1)

```
1: tree = assembly.getConfigs() // configs in form of a tree
2: interface = first interface with multiple comp variants from the root of tree
3: config = list of components with no variants from the root to interface
4: while interface != null do
5:     for each component in interface do
6:         testConfig = config
7:         testConfig += component
8:         assembly.selectConfig(testConfig)
9:         wait for w_t
10:        store perception.getPerception() for testConfig
11:    end for
12:    config += component that had the best performance in the loop (line 5)
13:    interface = first interface with multiple variants in tree from component
14:    if there are more than one interface in the same level then
15:        interface = a combination of components of interfaces with variants
16:    end if
17: end while//config stores the components of the best performing architecture
```

and selecting the variant which was in the best performing architecture (lines 5-11 in Algorithm 2). Once a component variant is selected, it becomes part of the 'optimal' composition, then the algorithm moves on to the next interface with component variants following the branch defined by the selected component. This process is repeated until there are no more interfaces with component variants, making the composition in $config$ the 'optimal' architectural composition.

Fig. 4.6 illustrates the Feature-based strategy in action, considering the generic architectural compositions defined in Fig. 4.3. The image illustrates the case where the selected component variant requires two interfaces with multiple component variants. In those situations, the algorithm combines all components in those interfaces for experimentation. This situations, where the algorithm combines components of interfaces, results in the worst case scenario for the feature-based search. As a quick comparison between the brute-force and feature-based, considering the generic architectural compositions in Fig. 4.6, the *Brute-force* approach takes $w_t * 14$ time (2.3 min for $w_t = 10$ secs) to explore all possible compositions. For the *Feature-based* strategy, considering the worst case scenario, the algorithm explores compositions

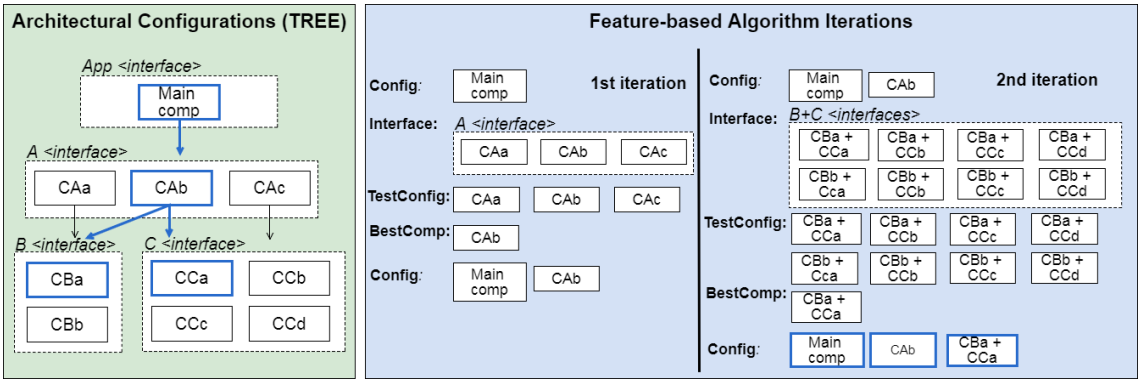

Figure 4.6: Feature-based strategy in action, considering the generic architectural compositions in in Fig. 4.3.

in $w_t * 11$ time (1.8 min for $w_t = 10$ secs). Considering the best case scenario, the *Feature-based* approach takes $w_t * 5$ time (50 secs for $w_t = 10$ secs), in case the best feature is B (component CAa) rather than $B + C$ (component CAb). The average case, in this example, is when the best feature is C (component CAc), resulting in an execution of $w_t * 7$ (1.16 min for $w_t = 10$ secs). In this particular example, the best case scenario for the feature-based approach has a significant advantage over the brute-force approach. The worst case scenario, however, is not as significant having a difference of only 30 secs (and not guaranteeing a global optimal), but considering that for every exploration the brute-force approach always executes in 2.3 secs (considering $w_t = 10$ secs) the feature-based search is still very advantageous. A more expressive scenario based on real systems is described and explored in Chapter 5, showing the advantages of the *Feature-based* search approach.

4.4.2 Environment Classification

Environment classification is an important part of the learning process. The Learning module classifies operating environments for two main reasons: i) to compare architectural compositions exposed to equivalent external stimuli to guarantee fair comparisons, and ii) to 'remember' the best software composition for previously seen conditions, to avoid unnecessary re-learning. The process to classify environments, as described in Chapter 3, is very challenging in the Emergent System concept. That is mainly because the perceived conditions may be distorted by the executing

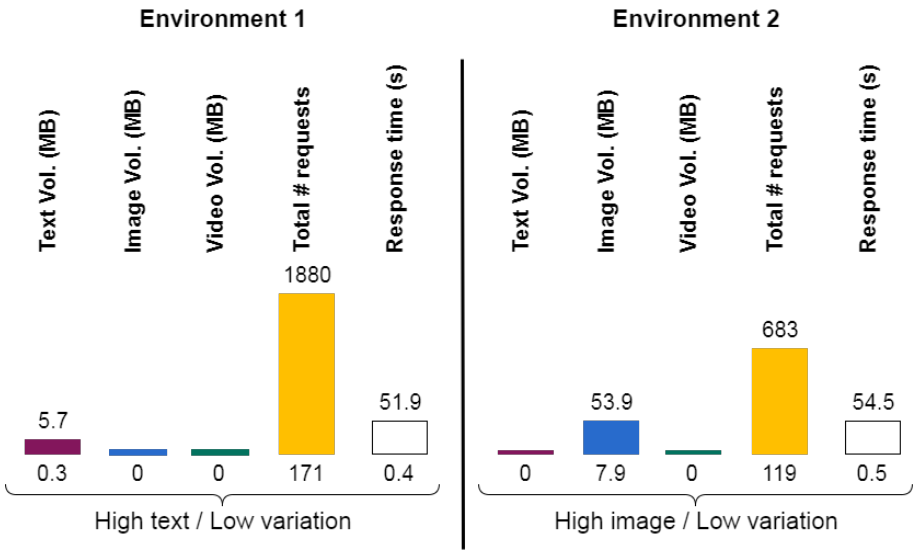

Figure 4.7: Example of classified environments using ranges of collected *Event* and *Metric* values. First environment consists of requests of large size text files with low variation. Second environment consists of large size image files with low variation.

architectural composition. These distortions create the illusion of changes in the environment, for example, the workload has increased, when in reality the running architecture is able to process more requests in less time. This section describes the environment classification algorithm used in Algorithm 1.

After collecting information about the architectural compositions explored (lines 3-7 in Algorithm 1), the Learning module receives a list of *Events* and *Metrics* attached to each composition executed, and based on that information the system classifies the operating environment. This approach minimises the effects of environment distortion by classifying environments with ranges. The algorithm creates ranges for each Event and Metric collected with a minimum value defined by the lowest event or metric value perceived and a maximum value defined by the highest value registered by the compositions as illustrated in Fig. 4.7.

The image (Fig. 4.7) illustrates the classification of two operating environments obtained from the tests with the case study (see Chapter 5). Considering a Web Server program and the operating environment as the patterns of request handled by the server. The first environment consists of large size text files requests with low variations, i.e. the majority of text file requested were repeated files. The second environment classified, on the right side of the image, consists of large size image

file requests with low variation (i.e. large volumes of requests to repeated files)..

During the exploitation phase, the system has already selected the best option to execute under the perceived operating environment, and only observes metrics and events collected from the executing architectural composition. If the collected values are within the ranges established by the environment class, and if there is not an extra event type and value (e.g. from a pattern of text-only requests, a few video requests start appearing) then the system understands that the environment has not changed. On the other hand, if the collected values are outside the range of any event type or metric (e.g. the system has a significant performance decrease) then after 3 iterations (the threshold to trigger exploration or architectural changes) the system compares the values it is perceiving with previously defined environments, in case these values are within all the ranges established in classified environments, the system changes to the composition attached to the new environment, otherwise the system triggers exploration. This process is defined in lines 12-21 of Algorithm 1.

This range-definition classification approach has some limitations. For a more accurate classification of environments, for instance, this approach works best when there are no changes in the environment during exploration, as will be demonstrated in the evaluation chapter (see Chapter 5). Also, it is difficult to 'remember' environments when there are overlaps among their defined ranges, preventing the system to accurately determine the operating environment based only on data collected from a single architecture (which is often the case to trigger exploration or composition changes during exploitation phase). Furthermore, this approach, as any current machine learning approach, suffers when important features of the environment is not defined as Event, leaving, for example, two or more distinct environments to be classified as the same. As previously mentioned in Chapter 3, the implementation of the reinforcement learning approach is very challenging, having a range of open issues to be addressed. Therefore, further research is required to better explore and overcome these issues. One of the main contributions of this thesis is to define the problem space to realise Emergent Systems, and to show the feasibility of this approach by presenting solutions (some preliminary) to enable the realisation Emergent Systems.

4.5 Distributed Emergent Systems

The distributed emergent system concept is an extension of the design-by-composition process in local machines across a distributed infrastructure. In a distributed environment, each software instance, running on each node in the distributed infrastructure, is composed at runtime through the PAL framework. The PAL framework implements the necessary elements to learn the best available composition of the software in local settings, but requires additional modules to allow the autonomous composition of distributed systems. To maintain the process of composition consistent across multiple instances, it is necessary to transform the process of **topology definition**, **service discovery**, and **service negotiation** among interacting software to locally selecting units of behaviour. Furthermore, the concept of distributed emergent systems involves not only finding optimal compositions of a program, but also the definition of the program and its diverse compositions in each system's node, which is part of the service negotiation problem. Transforming distributed design decisions to component replacement in local nodes is the base of the concept of distributed emergent systems, and is itself a challenge to realise the concept.

The process to autonomously compose distributed systems suffers from two additional challenges: ii) scalability due to the fast growth in the search space, and iii) system failures during exploration. Both of these problems are addressed in single emergent systems by the PAL framework and are extended to address the needs of the distributed scenario. The scalability issue is a main concern in emergent solutions and it is addressed in local instances by specifically exploring software features rather than individual compositions. System failures during exploration is addressed by the used of component-based models, that serve both as deployment mechanism and a model to avoid invalid architectural composition (through the require-provide interface policy defined at design phase). The distributed emergent system framework modules implement two main concepts that aim to mitigate the aforementioned issues: the concept of **External Reference** and the concept of **Hierarchical Coordination**, which are realised by the **Factory**, **Registry** and **Validator** modules and will be described in the following sections.

```
data ServerInfo {
    char[] serviceName
    char[] IPAddr
    int port
}
interface ExternalReference {
     ServerInfo[] getServersInfo()
}
```

Figure 4.8: The External Reference interface code.

4.5.1 External References Concept

The **External Reference** concept extends the component-based model to allow exploration of communication patterns and the system topology through component replacement in the local software instances. The External Reference concept solves two main problems: i) the inability of component developers, using a component-based model, to write components that require to interact with external services, and ii) enabling the Learning module to experiment, through the component-replacement experimentation learning process, different system's topologies, for example, by connecting the component to the service running in node A and B, or only to node C. This section details how the development of the External Reference interface in tandem with the Registry and Factory modules realise the External Reference concept.

The `ExternalReference` interface (Fig. 4.8) is used by components developers to express a component's requirement to interact with external systems. The interface provides a function that returns a list of *IP addresses* and *port* of executing emergent systems. This information is dynamically generated after the system's deployment, and are used by components to locate and interact with their required external services. The details of the `ExternalReference` interface is depicted in Fig. 4.8, illustrating the single function (`ServerInfo[] getServersInfo()`) and the `ServerInfo` data type. In a component-base model, a component that requires an interface expects to be connected to a component that provides such interface at runtime. Based on that assumption, the External Reference concept uses the Registry and Factory modules to generate components that implement

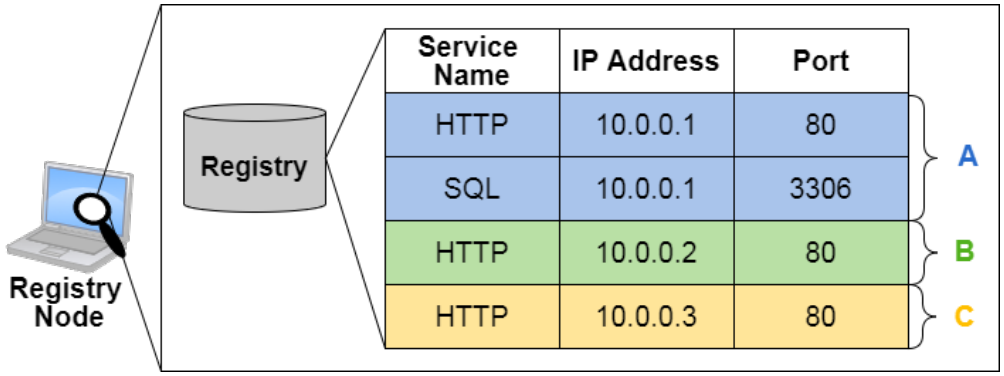

Figure 4.9: The image shows the Registry information table with data about potential services provided by nodes A, B and C.

the `ExternalReference` interface. The generated components are connected to the component who requires the `ExternalReference` interface, providing it with information about the nodes where the services are running. The component generation process and the generated components are described later in this section.

The Registry module is a regular naming service that contains the services' name and the server information running the services (*IP address* and *port number*). Considering that in an Emergent Software System the services are autonomously assembled at runtime, and that the services provided are changed when the nodes composition changes, the Registry stores information of the nodes and the potential services each node might provide. For example, as illustrated in Fig. 4.9, the server *A* (*IP address*: 10.0.0.1) has the potential to either provide a HTTP service on *port* 80, or provide a SQL service on *port* 3306. The Registry data illustrated in Fig. 4.9 informs that the node *A* has the potential to be a HTTP server (i.e. web server, web cache or a load balance) **or** a SQL server (i.e. a relational database), depending on its composition. Based on the information stored in the Registry, the Factory generates components that provide the `ExternalReference` interface.

The Factory module provides the function '`generateExternalRefVariations()`' responsible to generate components that provide the `ExternalReference` interface. This function expects a JSON string (`char servicesInformation[]`) as parameter with the data to generate the components (e.g. `"IPAddr"`, `port`, etc.). An example of data required to generate the `ExternalReference` components is presented:

```
[{"name": "HTTP", "IPAddr": "10.0.0.1", "port": "80"},
```

```
{"name": "SQL", "IPAddr": "10.0.0.1", "port": "3306"},
{"name": "HTTP", "IPAddr": "10.0.0.2", "port": "80"}]}
```

Based on the example of data to generate `ExternalReference` components, the Factory module generates component variants to each of the participating nodes. The example described on the list above illustrates the C node's (in Fig. 4.9) request to the Factory module. The C node requires the Factory module to generate `ExternalReference` component variations passing as parameters the *IPAddr* and *port number* of all the other nodes in the system (excluding itself) in the Registry. Based on the information provided, the Factory module generates 7 component variations (Fig. 4.10). The Factory considers the services information (*IPAddr* and *port number*) passed as parameters as a set. The Factory then generates the **power set**[2], excluding the empty set, of all the elements it receives. Each `ExternalReference` component generated by the Factory is a set of the resulting power set. In this example, the Factory generates 7 `ExternalReference` components, making combinations of the items `1 by 1`, `2 by 2` and `3 by 3`. The resulting components produced by the Factory to C node, considering the presented input example, is shown in Fig. 4.10.

As a result of generating `ExternalReference` components, the External Reference concept enables the experimentation of the system's topology by exploring `ExternalReference` components in the local architecture. For example, node C, depending on the `ExternalReference` component connected to its architecture, enables the node to communicate to only node A on port `80` (when selecting A_{http} component in Fig.4.10); or to both A on ports `80` and `3306`, and node B on port `80`, when the component A_{HTTP} A_{SQL} B_{HTTP} is connected to the architecture. This allows the PAL framework, through the experimentation process, to discover the optimal communication pattern and topology to best satisfy the system goal.

The External Reference concept enables the exploration of communication patterns and system topology by the Learning module, but does not prevent invalid global compositions resulted from Behavioural Mismatches, e.g. a node sending HTTP requests to another node running a relational database (see Sec.3.2.3 in Chap-

[2]A power set is a mathematical concept and is defined as: a set composed of all subsets of a given set. E.g.: Given set S {0,5}, the power set of S is {{}, {0}, {5}, {0,5}}

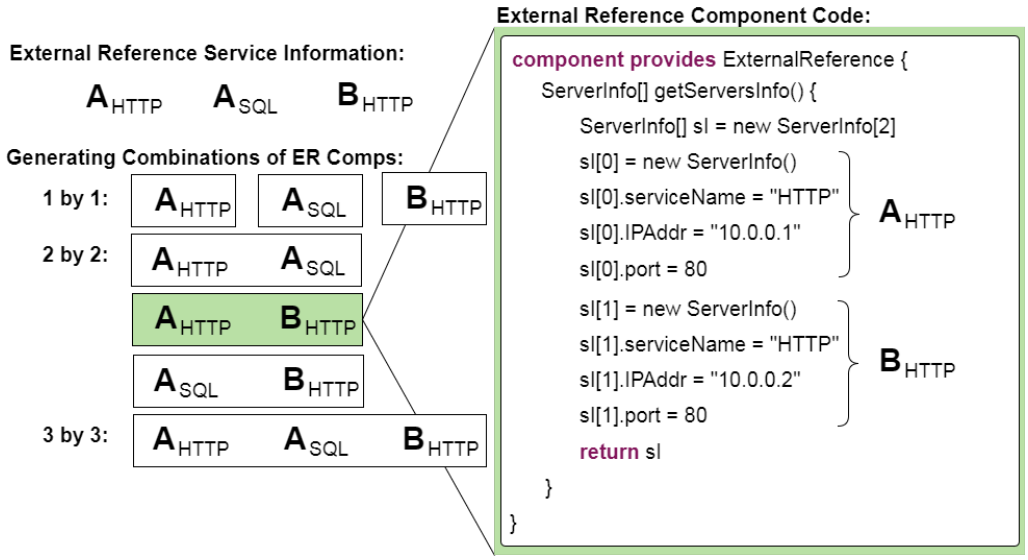

Figure 4.10: Example of resulting External Reference components. All possible components generated from services information are illustrated on the left part of the image. An External Reference component code is represented on the right.

ter 3). The next section (Sec. 4.5.2) presents the **Validator** module responsible to identify invalid global compositions and prevent system failures during exploration.

4.5.2 Hierarchical Coordination Strategy

The Hierarchical Coordination is a learning coordination strategy that avoids a centralised element of coordination. Instead, this approach creates multiple points of coordination exploiting hierarchical structure of systems, mitigating the exponential growth in the search space in distributed environments. Additionally, this coordination approach applies the External Reference concept to filter valid compositions and support a reliable learning process avoiding system failures. This section describes the Hierarchical Coordination learning strategy, showing how the Validator module is used to support this learning strategy.

The solution to the Behavioural Mismatch problem enables the system to explore architectural compositions in different nodes avoiding system failures. The Validator module solves that problem whilst supporting the Hierarchical Coordination learning strategy by relying on the following assumptions:

1. The target system is a hierarchical system which has a single *Entry Point*, i.e. a machine that receives the systems input;

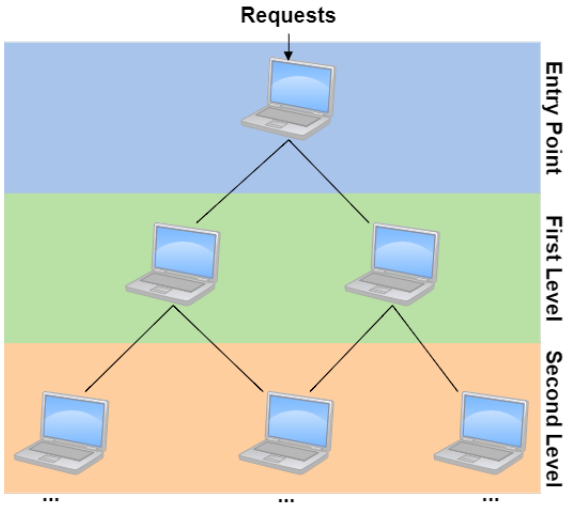

Figure 4.11: Hierarchical topology used in the Hierarchical Coordination approach.

2. The target system is built to satisfy a single common system goal, in a cooperative environment;

3. Connecting two machines A and B, where A requires a service in B and B is in a composition that provides such service, will always work;

4. The Registry module information will not change as the system executes (i.e. external services are not added or removed).

The Validator module, based on the information on the Registry and further information annotated on the components by developers, generates component groups to avoid invalid global composition. The generated groups are used by the Learning module to avoid behavioural mismatches in the system during composition exploration in distributed environments. The groups contain compositions of all nodes that do not result in behavioural mismatches, enabling a free exploration of compositions. The group formation considers a hierarchical topology, where high level nodes composition determines available compositions on lower level nodes. The hierarchical communication pattern is illustrated in Fig. 4.11. The hierarchical topology is a logical topology based on the communication at the application level determined by the generated `ExternalReference` components.

The group formation algorithm starts from the Entry Point node (or root node), and based on the `ExternalReference` component selected it determines the set

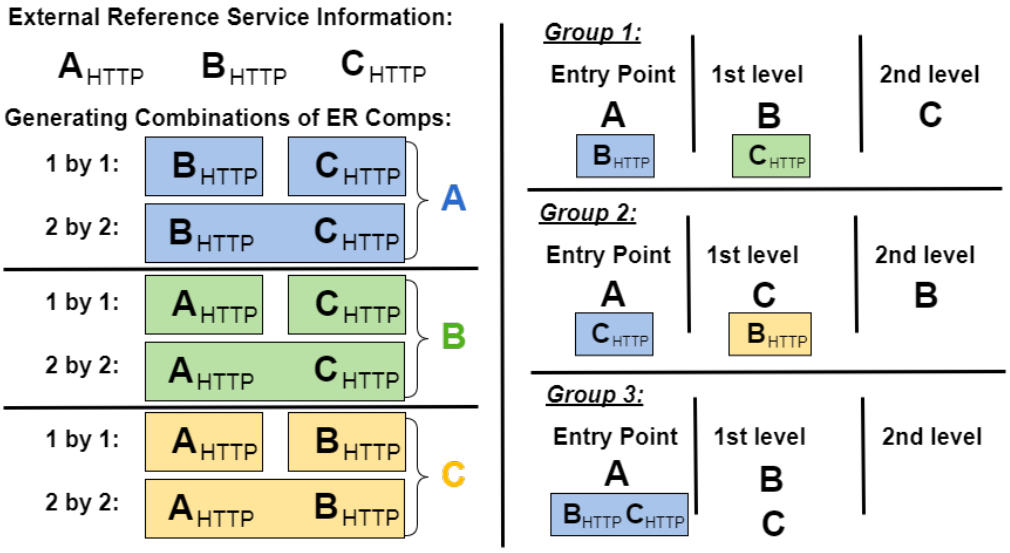

Figure 4.12: Example of exploration groups created by the Validator module. The available External Reference components are represented on the left. The exploration groups resulted from the components is illustrated on the right.

of nodes that are placed in the first level of the hierarchy, then it removes all `ExternalReference` components in the first level that allows the first-level machines to connect to either the Entry Point level or to connect among the machines in the first level, allowing first-level machines to only connect to machines that have not yet been placed in any level. The algorithm then analyses the `ExternalReference` components in the first level and determines the second level of the hierarchy applying the same rules to remove `ExternalReference` components. The aforementioned process continues until the algorithm determines the machines and components that are part of the last level of the hierarchical topology, forming a group. The algorithm restarts selecting different `ExternalReference` components in the root node and repeating the aforementioned process until it defines all exploration groups.

Fig. 4.12 illustrates the resulting group using an example set of `ExternalReference` components. On the left side of the image, there are the available `ExternalReference` components generated by the Factory module. The Validator module, with access to those components and the information that the node *A* is fixed as the Entry Point, starts building the tables (on the right side) determining the exploration groups. This example considers communication among the machines using a unique application level protocol: the HTTP protocol. This situation, where all machines use and offer services on the same communication protocol, might still generate global

invalid compositions. These undesirable compositions, in this example, occurs due to circular topologies. For instance, when server A forwards requests to B which, in turn, forwards to C, which finally forwards the request back to A. The same situation occurs in real web platforms, where all nodes could be assembled into a Load Balancer composition, forwarding requests indefinitely.

The algorithm creates different exploration groups (tables on the right side of the image) based on the `ExternalReference` component chosen by the *Entry Point* (the machine A). The algorithm creates one group containing each of the available generated `ExternalReference` components. In the image example, there are 3 `ExternalReference` components forming 3 groups. The first group is defined by the selection of the component that allows the node A to directly communicate with node B, and because of that choice, server B is then placed in the first level of the hierarchy. The algorithm then analyses the `ExternalReference` components that B is able to choose. Following the algorithm rule that eliminates components with information about itself and information about higher level nodes (to avoid cycles), the only available option is to allow B (in case it chooses a distributed composition) to only communicate with node C. The algorithm then applies the same rule to select the `ExternalReference` components to be made available to node C. According the the algorithm rule, node C is only allowed to be in a local composition, i.e. a composition that does not contain a component that requires external service. This process repeats for the other two `ExternalReference` components available to node A. The other options are to select the component that allows a direct communication with node C, placing C in the second level, or to simultaneously communicate with B and C, placing both in the second level, and forcing them to only experiment with local compositions. Due to the assumption 4 (the Registry information does not change), the group definition process is executed only once.

After the Validator module divide the components into exploration groups, the Hierarchical Coordination learning strategy is executed. This hierarchical learning strategy is partially decentralised, meaning that multiple Learning modules executes in parallel in multiple nodes when exploring a set of compositions in a specific group. This parallel learning execution occurs in machines that are on the same hierarchical

level. The selection of the group being explored, however, is determined by a central Learning module, the module executing in the *Entry Point* machine. The Hierarchical Coordination strategy applies the Feature-based Learning Algorithm described in Sec. 4.4. The PAL framework running on the *Entry Point* machine executes as it normally would in a single instance, having only access to the `ExternalReference` component that is part of the first exploration group (the other nodes are also in the first group). In case where the *Entry Point* machine decides for a distributed composition, instead of a local one, the Hierarchical Coordination is started in the Learning module that triggers learning in the nodes that are part of the second level which finds the best local composition in parallel. Once the second level machines fishes learning, they trigger learning in the third level and so one. Once all levels finish the learning process, the leading Learning module starts the same exploration procedure but exploring the second group and so on. The decision for the best global composition, as well as the classification of the operating environment, is done by the leading Learning module. This module stores the best performing compositions of each exploration group and compares them to choose the overall best.

4.6 Summary

This chapter presented the implementation details of the modules that supports the Emergent Software System concept. The modules that compose the local framework are: the **Perception**, **Assembly** and **Learning** modules. The Assembly module applies a component-based model to guide an autonomous software composition at runtime, allowing software composition changes whilst avoiding system failures due to the process of replacing a component to an incompatible or faulty one. This module is at the bottom of the multi-tier architecture and provides an API that abstracts the component composition and (re)-organisation process. The Perception module uses the function provided by the Assembly module to provide its main function responsible to collect metrics and events about the system health status and the operating environment. The Perception module also provides a RESTful API and access to the Assembly module indirectly through the Perception module itself. The

RESTFul API enables the Learning module to control the system composition and the collection of data about the system remotely (running on a external machine) to avoid impact on system performance in cases where, for example, the Learning algorithm is CPU-intensive. The Learning module leads the entire composition, perception data collection and reinforcement learning process, discovering the system architectural assemblies that are optimal to the classified operating conditions.

This chapter also presented the distributed framework modules and their roles in supporting autonomous software composition in distributed environments. In order to enable the application of the PAL framework to autonomously design distributed software systems, this thesis introduces two concepts: the External References and the Hierarchical Coordination strategy. These concepts are supported by the distributed framework modules: the **Factory**, **Registry** and **Validator** modules. In tandem with the Factory, Registry modules and the `ExternalReference` interface, the system is able to extend the local component-based model to allow component developers to require external services through the External Reference concept. This concept also enables the system to experiment with different system topology and communication patterns. The Validator module, based on the information in the Registry module, creates distinct groups of architectural compositions in the nodes of the system that are interoperable, avoiding system failures during the reinforcement learning process. Although the Hierarchical Coordination concept relies on a set of assumptions, the approach shows the feasibility of applying the Emergent System concept to design distributed systems. The realisation of fully Distributed Emergent System has open issues and interesting challenges that are part of future work discussed in Chapter 6.

CHAPTER 5

Case Study and Evaluation

This chapter presents the results of evaluating Emergent Systems. The evaluation is conducted in a web server case study due to the key role these servers play in supporting contemporary applications, underpinning major systems such as: banking applications, social media, search engines and so on. Furthermore, web servers are known to be difficult to optimally configure, particularly when subjected to different client workloads over time [87]. In addition, current optimisation approaches heavily rely on manual workload analysis and parametric configuration. The evaluation was conducted in two phases: i) local and ii) distributed emergent web platforms. Sec. 5.1 describes the local emergent web server scenario and presents the results of applying the PAL framework to a single software instance, demonstrating that: i) different compositions for the web server have different performance depending on detected request patterns (RQ1)[1], and ii) the emergent web server is able to detect and converge to optimal solutions in reasonable time with no domain specific knowledge or offline training (RQ2). Sec. 5.2 introduces the distributed emergent web platform (composed of web servers, web caches and load balancers) and presents the results of extending the PAL framework, evaluating different learning coordination strategies (RQ3). Finally, Sec. 5.3 evaluates additional aspects of online learning, specifically the environment classification algorithm and the effects of different observation window sizes in the learning process. The evaluation of these learning aspects unfolds from RQ2, to test the limits of system convergence in extreme conditions.

[1]RQ1, RQ2 and RQ3 are research questions defined in Chapter 1.

5.1 Local Emergent Web Server

This section introduces the local case study scenario and evaluates the emergent software framework, using the web server as an emergent system example. The evaluation was conducted with a real implementation of the emergent web server, and the emergent software framework, running on rackmount servers within Lancaster University's infrastructure. These servers have an Intel Xeon E3-1280 v2 Quad Core 3.60 GHz CPU, 16 GB of RAM, and run Ubuntu 14.04. Similar specification machines were used as clients to generate workloads; these client machines were on a different subnet to the servers (in a different building). The experiments were conducted with a mixture of custom-built workload patterns designed to explore emergent system's characteristics, and a real-world trace from NASA [1].

The presented evaluation aims to show that the concept of Emergent Systems is feasible, and supports autonomous system convergence to optimal software compositions with no predefined rules, policies or models to guide software adaptation in single software instances. First, this section shows that for different workload patterns a different architectural composition has better performance. This motivates the need for adaptive solutions, whilst presenting the ground truth to validate and show the convergence accuracy of the proposed learning approach. Second, this section shows, based on the ground truth, that the system autonomously learns and converges towards the optimal performance. Finally, this section shows the convergence rate and accuracy of both the brute-force and feature-based algorithms, addressing research questions regarding search space growth. The results described in this section was published in [71] and [38]. The code used in the evaluation, along with instructions on how to repeat all experiments in this section is available at [36].

5.1.1 Case Study: Emergent Web Server

The local scenario consists of evaluating the autonomous composition and optimisation of an emergent web server. The **Emergent Web Server** is a single web server program autonomously assembled from small reusable components in a repository. The web server architecture, illustrated in Fig. 5.1 has the following main

interface that support the realisation of a web server program: `RequestHandler`, `HTTPHandler`, `Cache` and `Compression`. Each of these interfaces realise a part of the web server, and they have component variants (i.e. components that implement the same functionality in a slightly different manner). The `RequestHandler` defines the concurrency model of handling incoming TCP requests, having two components providing two different models. The component `RequestHandlerTPC` creates a thread per connection received. In contrast, the component `RequestHandlerPT` creates a pool of threads and assigns each incoming connection to a thread. In case all threads are busy, the system places the connections to a queue in the thread. The threads are kept in a circular list, and the connection is always assigned to the next thread in the list (i.e. round robin).

The interface `HTTPHandler` implements the HTTP protocol. This standard implementation is provided by the `HTTPHandler` component, which fetches requested files from disk and send them directly back to the client. Slightly different implementations are provided by `HTTPHandlerCH`, `HTTPHandlerCMP` and `HTTPHandlerCHCMP` variants which require other interfaces for their implementation. The component `HTTPHandlerCH` implements the HTTP protocol, particularly the GET method, by firstly checking whether the requested content is stored in an in-memory cache; if the file is cached, the component returns the stored requested file, otherwise the component fetches the content from the disk and directly returns the file to the client. After returning the requested file, the component caches the file's content. The `HTTPHandlerCMP` component, on the other hand, fetches the requested file on the disk and, before sending it back the client, compresses the file content before returning it to the client. Finally, the component `HTTPHandlerCHCMP` works as a combination of `HTTPHandlerCH` and `HTTPHandlerCMP` components. It verifies whether the requested file is in cache; in case it is not, the component loads the file from the disk, compresses the file, and sends the compressed file to the client. After the compressed version of the file is sent, the component stores it in cache.

The `Compression` and `Cache` interface support different HTTP implementations, used specifically by the `HTTPHandlerCH`, `HTTPHandlerCMP` and `HTTPHandlerCHCMP` components. The components that implement `Cache` and `Compression` interfaces

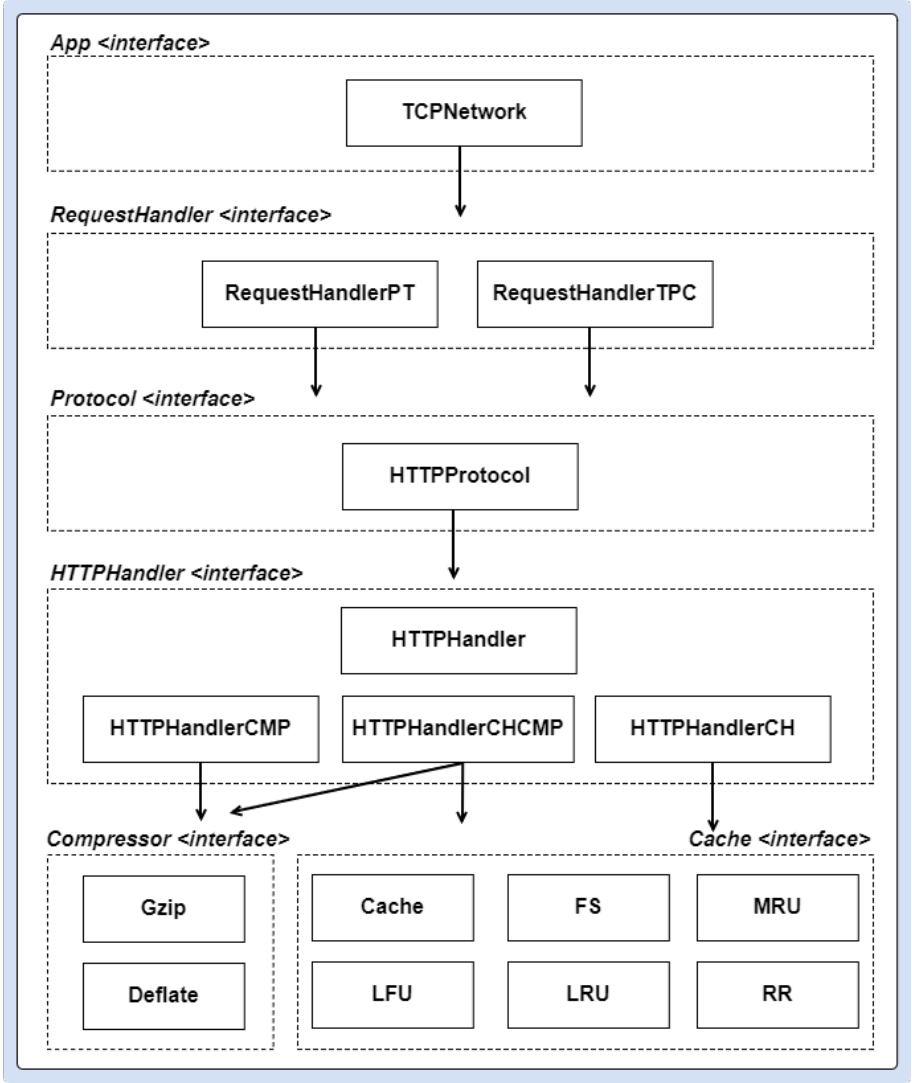

Figure 5.1: Architectural representation of the web server compositions.

are generic and reusable. The cache component, for example, stores a generic string of bytes, and each of its variants implements a different item replacement strategy. The `FS` component implements a simple First-In-First-Out (FIFO) strategy, where the first item cached is the first item replaced once the cache is full. The `LFU` component implements the Least Frequently Used strategy, replacing the item that has the lowest number of access. The `MRU` component implements the Most Recently Used strategy, replacing the last accessed item. The `LRU` component implements the Least Recently Used replacement strategy, where the item that was first accessed (i.e. the accessed 'timestamp' is the earliest) is replaced. Both `MRU` and `LRU` requires a time stamp or (age bits) to determine when the items were accessed in order to select the

most recently accessed or the least recently accessed. The `RR` component implements the Random Replacement strategy, where the items selected to be replaced are chosen randomly. The `Cache` component does not implement any replacement strategy. Instead it considers an infinite amount of cache space and keeps caching new items until it runs out of memory. Finally, the `Compression` interface has two component variants: the `GZ` and `Deflate` components. According to the HTTP 1.1 specification RFC 2616 (`https://tools.ietf.org/html/rfc2616}`): the `GZ` component implements the **gzip** compression algorithm (RFC1952), and the `Deflate` component uses the **zlib** format and the deflate compression mechanism (RFC 1950).[2]

A functional web server software instance is realised by selecting a single component in each one of the interfaces defined in the architecture in Fig. 5.1. For example, by selecting a single component available in the `RequestHandler` interface, another component in the `HTTPHandler` and depending on the component selected in that interface, it is required to select another component in `Cache` and/or `Compression` interface. The number of compositions possible is calculated by adding 2 (number of compositions when selecting `HTTPHandlerCMP`) plus 1 (number of composition when selecting `HTTPHandler` component) plus 12 (number of compositions when selecting `HTTPHandlerCHCMP` – which is the result of multiplying 2, the number of compression components, times 6, the number of cache variants) plus 6 (number of compositions when selecting `HTTPHandlerCH` component). The result of the addition is 21, considering that any of those 21 could be composed using either `RequestHandlerPT` component or `RequestHandlerTPC` variant, we multiply 21 times 2, resulting in a total of **42 unique compositions** for the web server.

5.1.2 Evaluating Compositions Under Different Workloads

This section demonstrates that different optimal compositions exist for different workloads. This result is used to show the emergent web server autonomously learning such optimal compositions with no predefined information. Four workload patterns were chosen to illustrate the different performance of compositions when

[2]For more information on gzip and zlib compression algorithms, please refer to https://tools.ietf.org/html/rfc1952 and https://tools.ietf.org/html/rfc1950 respectively.

implementing different behaviour. Three of the four patterns are synthetically made to explore when caching, compression and the combination of these two features or the absence of them impacts the system's performance. These patterns consists of sequential requests to i) repeated small (∼3KB) text-based html file, ii) repeated small (∼64KB) image file, and iii) a diverse set of small text-based html files. The fourth workload is a real-world workload pattern from NASA [1]. The NASA workload consists of a log of requests issued to the NASA web site in 1995 and can be downloaded at [1]. For the purpose of this evaluation, the requested files were recreated from the trace, which contained the size of each of the requested resources and their types. The fabricated files were generated having the same type and the same size, but with fake/random content. For example, if a 5KB PDF file named "foobar.pdf" is requested at any point in the NASA trace, a 5KB "foobar.pdf" file with random content is created and placed in the 'htdocs' of the emergent web server. This workload was chosen due to its accessibility (facilitating results replication), and because the workload has requests to diverse files with a variety of mime-types (text, image, video), thus being a relevant representation of real-world workloads.

The results from custom-defined request patterns are shown in Fig. 5.2 and Fig. 5.3 (a); whilst results from the NASA trace are shown in Fig. 5.3 (b). These graphs show four specific compositions' performance, from the 42 available.

Four groups of architectural compositions were chosen to represent the results in Fig. 5.2 and Fig. 5.3. These compositions are representative of the four main functionalities of the available web server architecture: `Cache`, `Compression`, `Both` (i.e. when selecting `HTTPHandlerCHCMP` component in Fig. 5.1) and `None` (when selecting `HTTPHandler` component in Fig. 5.1). For each of the lines in the graphs (Fig. 5.2 and Fig. 5.3) the best performing composition of each group was chosen to represent the group. For example, the 'Compression' line represents the best architectural composition that has a component implementing a compression algorithm.

Fig. 5.2 shows the average response time of the web server for request patterns in which the same file is repeatedly requested (low variation). When this is a text file, Fig. 5.2 (a) shows that compositions with in-memory caching and without compression have better average response times than compositions with both caching and

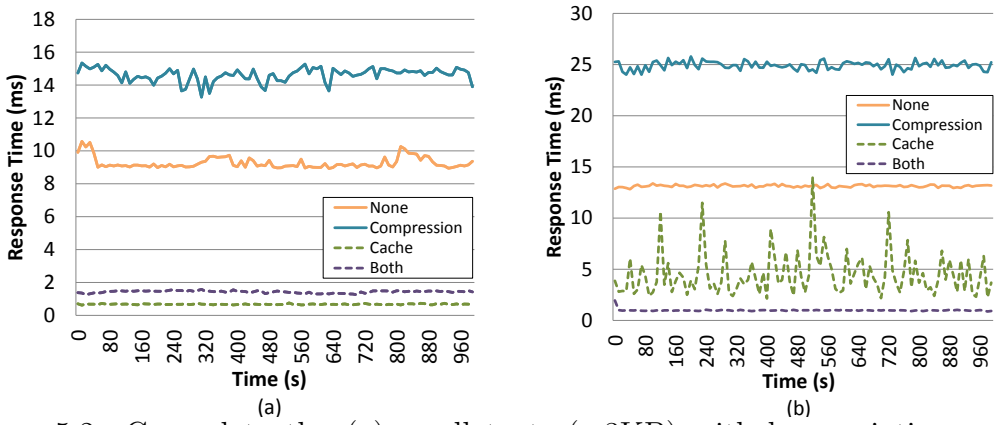

Figure 5.2: Ground truth: (a) small texts (∼3KB) with low variation workload pattern; (b) small images (∼64KB) with low variation workload pattern.

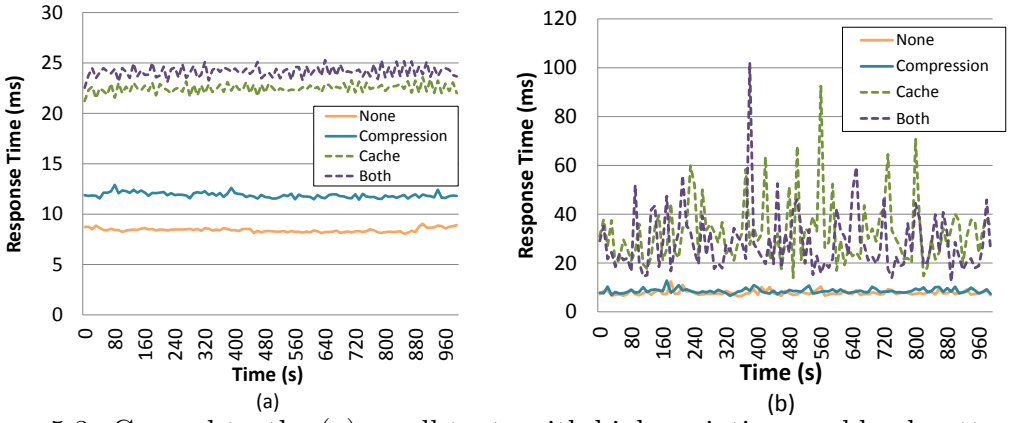

Figure 5.3: Ground truth: (a) small texts with high variation workload pattern; (b) NASA workload trace from [1].

compression. However, for image files, Fig. 5.2 (b) shows that the opposite of this is true, i.e. the compositions that contain both cache and compression components have better performance when exposed low variation small image workload.

Fig. 5.3 (a) and (b) show the average response time of the web server for request patterns in which many different files are requested (high variation). In detail, Fig. 5.3 (a) shows results from a custom request pattern in which each request is for a different small (∼3KB) text file; whilst Fig. 5.3 (b) shows results from replaying the NASA trace (which also has a high degree of file requests variations). In both of these graphs (in Fig. 5.3) the best compositions are the composition that do not implement cache, i.e. either 'None' or 'Compression' compositions, which are different from the low variations workload (Fig. 5.2). The compositions that are best for low variation are the worst two compositions for high variation workload.

The results demonstrate that different compositions of our web server will per-

form differently when subjected to different request patterns at runtime. In particular, request patterns with high variation do not benefit from compositions that use caching, whereas request patterns with low variation do. Additionally, the performance of compositions that include compression is impacted by the compression ratio of the files being requested in that pattern. While these results may be intuitive to experts; the next section evaluates the feasibility of autonomously learning this information from no prior experience – the basis of emergent software systems whose design is a product of their environment. Furthermore, this environment includes the physical hardware platform on which the system is running, a factor which is automatically accounted for by emergent software systems in continually forming the most ideal composition of the software over time.

5.1.3 Experimental Learning Analysis

This section evaluates the emergent systems capability of learning optimal compositions for the previously presented workloads. The learning process, as previously described (see Chapter 4), continuously and autonomously locates optimal component compositions for the target system, by analysing the currently available perception data (perception data – events and metrics) and exploring how the various available compositions of behaviour affect the perception of metrics across different environments. Note that the learning process applies only unsupervised online learning, with no human input and with no application-specific aids. This section evaluates two learning strategies. First, this section evaluates the brute-force (baseline) learning strategy to locate optimal compositions. This strategy samples every available composition every time the learning process is triggered, and because of that the strategy guarantees to find the optimal composition available. Finally, this section presents the results of comparing the feature-based approach against the brute-force, showing both learning accuracy and convergence rate.

Baseline Learning Strategy

The evaluation of the brute-force learning strategy is depicted in two graphs (Fig. 5.4 and Fig. 5.5). The graphs illustrate the brute-force algorithm converging to the

optimal architectural composition when exposed to different workload patterns. In particular, Fig. 5.4 shows a request pattern consisting of sequential requests for small ($\sim$ 3KB) text files for 700 seconds, followed by sequential requests for small ($\sim$ 1MB) image files for 1200 seconds, and finally returning to small text files. This experiment was chosen as it contains two distinct kinds of request patterns for which different web server compositions are known to be optimal, as shown in Sec. 5.1.2.

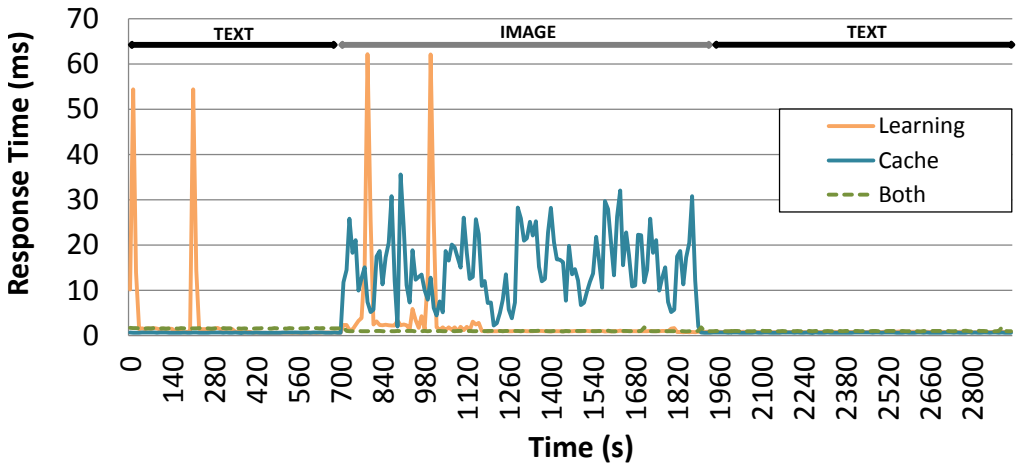

Figure 5.4: Performance comparison between fixed web server architectures and our emergent platform, using two different request patterns over time.

The graph (in Fig. 5.4) shows the performance of the brute-force learning approach, exploring available compositions, compared to the performance of two different static web server compositions that are known to be optimal for the different parts of this request pattern. At the beginning of the experiment, the learning system starts with no information and so must go through the entire learning process to discover the architecture most suited for the currently observed conditions.

In detail, when a new pattern is detected, the learning module performs an exploration activity to find the best composition for that pattern. This takes 420 seconds (each composition runs for $w_t = 10$ seconds) and is clearly visible on the graph as two large spikes; each spike shows experimentation with a particularly poorly-performing composition for this pattern. When learning is complete, the system converges to the optimal composition. This can be seen at two times, one at time 250, and the other at time 1200. At time 1900 another request pattern transition occurs, but (in this case) to a pattern that the learning system has already

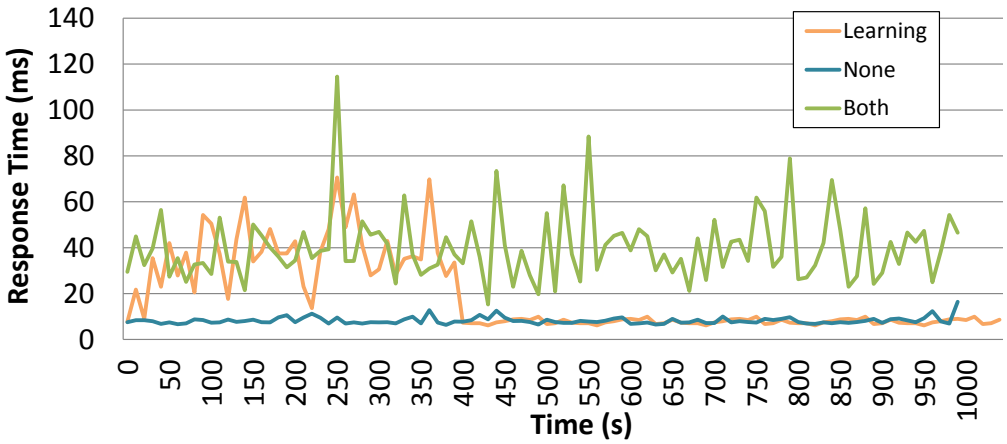

Figure 5.5: Performance comparison between two fixed web server architectures and the emergent system framework when exposed to the NASA trace [1].

seen. In this scenario, the system does not trigger a further learning phase. Instead, the system simply picks the best composition from prior experience. Comparing this against the two static compositions, the emergent system framework maintains optimal performance for the longest period of time, whilst both static compositions are optimal when exposed to one of the two tested workloads.

Fig. 5.5 shows an experiment with our learning system using the NASA trace, which is characterised by having small files ($<$ 20MB) with a high degree of variation, meaning that the same file is rarely requested consecutively. This trace was chosen as a representation of a real-world scenario. Starting from no information at the beginning of the experiment, the learning process maintains the same time of 420 seconds to learn the most suitable composition – again needing to experiment with each available composition for 10 seconds. The graph (Fig. 5.5) shows the result of comparing the brute-force strategy to the performance of two fixed architectures that had the best and the worst performance for this pattern, showing that after 420 seconds the learning system converges to an architecture with an equivalent level of performance of the best performing architecture. Note that, when compared to the results in Fig. 5.4, both the learning and static architectures in this case have a relatively erratic level of performance caused by a relatively high degree of variation in this request pattern. Furthermore, note that after 420 seconds the emergent system outperforms the worst composition in an average of 32 ms per request.

These results demonstrate that, starting with no information at all, the system can learn and converge to an optimal composition in real-time. As more data is collected by the learning module, more experience is gained and less learning occurs – the approach always maintains the ability to detect and react to new conditions.

Feature-based Search Learning Strategy

This section presents the results of comparing the feature-based and the brute-force learning approach. The comparison is made considering both convergence rate, i.e. how fast the feature-based learning converges to the optimal composition, and the learning accuracy, i.e. the capacity to converge to the best performing composition.

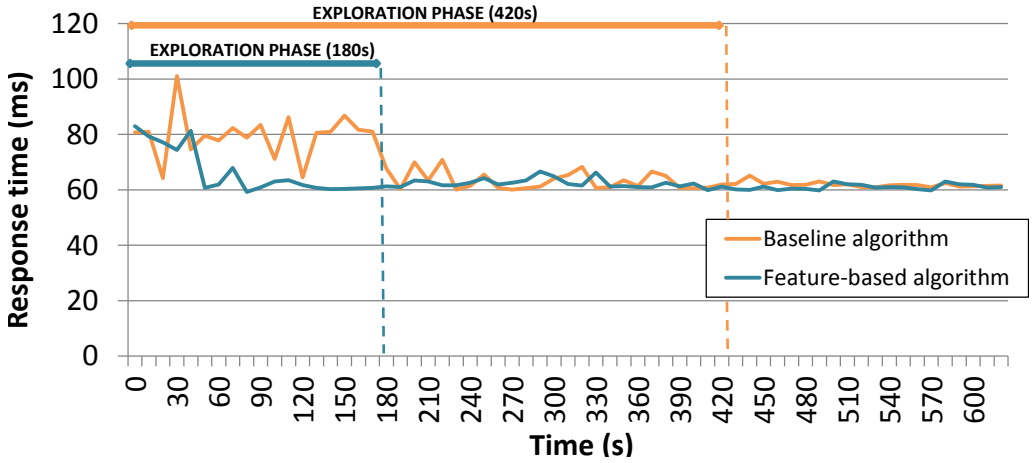

Figure 5.6: Performance comparison between the brute-force algorithm and the feature-based algorithm when exposed to the small text with low variation workload.

Fig. 5.6 shows both feature-based and the brute-force learning strategies subjected to the small text workload (see Sec. 5.1.2, Fig. 5.2 (a)). The brute-force strategy always takes 420 seconds to explore the available compositions. In this particular case, however, the last set of compositions experimented by the system coincidently matched the group of architectures that had optimal performance. Although the system was still exploring, the system performance had already converged. For the feature-based approach, the graph demonstrates that even for the worst case scenario (180 seconds for the exploration phase), the featured-based approach converges faster than the brute-force and also accurately. The 180 seconds for the worst-case scenario is due to the exploration of the two variants for the `RequestHandler` inter-

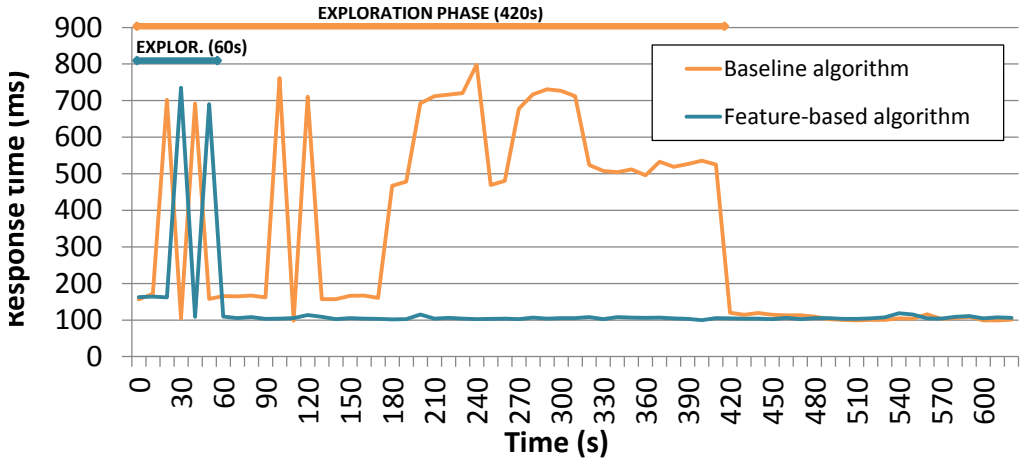

Figure 5.7: Performance comparison between the brute-force algorithm and the feature-based algorithm when exposed to the small text with high variation workload.

face (see Fig. 5.1), plus the exploration of the four variants for the `HTTPHanlder` and, finally, the exploration of the twelve variants for the `HTTPHandlerCHCMP` component which combines two compression variants and six cache variants, making a total of 18 compositions to explore. Given that the observation window is 10 seconds, the worst-case scenario takes 180 seconds. As opposed to the brute-force approach that coincidently experimented with the best performing group of architectures, the feature-based approach is programmed so this phenomenon happens on purpose, since the system locates the best performing feature and then searches for the best composition within the group of architectures implementing that feature.

Fig. 5.7 shows the comparison between the two learning strategies subjected to small text workload with high variation (see Sec. 5.1.2, Fig. 5.3 (a)). The graph shows the feature-based accurately converging in its best case scenario with an exploration phase that only takes 60 seconds to complete. The exploration phase in the brute-force approach, however, always takes 420 seconds, with the observation window set to 10 seconds. Both graphs show the effectiveness and accuracy of the feature-based approach applied to the web server domain. In the best case scenario, the feature-based approach accurately converges 360 seconds before the brute-force approach. In the worst case scenario, the feature-based approach accurately converges 240 seconds before the brute-force strategy.

The results clearly demonstrates that the feature-based approach performs well,

as long as its assumptions is upheld. The feature-based approach has a shorter exploration phase time and often converges before the exploration phase finishes, due to its focus on finding the best feature, and then finding the best composition that implements such feature. The disadvantage of the feature-based strategy is that it relies on the assumption that the worst composition of the best performing feature has to be better than the best of the other features. This assumption works for the chosen case study domain, but might not be extendible to other domains. Finally, both strategies work well for the local scenario for the experimented workloads. The feature-based approach has faster convergence rate, and it is key to demonstrate the potential of emergent systems in distributed settings, evaluated in the next section.

5.2 Distributed Emergent Web Platform

This section evaluates the feasibility of the Emergent System concept in distributed settings. In particular, this section evaluates a centralised, decentralised and a hybrid learning coordination strategy considering the growth of the search space in distributed systems and the detection and management of invalid compositions. Similar to Sec. 5.1, all evaluation described in this section was conducted with a real implementation of the emergent web platform, and the distributed emergent software framework, running on rackmount servers within Lancaster University's infrastructure. These servers have an Intel Xeon E3-1280 v2 Quad Core 3.60 GHz CPU, 16 GB of RAM, and run Ubuntu 14.04. Similar specification machines were used as clients to generate workloads; these client machines were on a different subnet to the servers (in a different building). The following sections provide detail of the evaluation, describing the emergent web platform case study, and further details of the distributed emergent web platform evaluation procedure.

5.2.1 Case study: Emergent Web Platform

The evaluation of the distributed framework is in the context of a web platform. The investigated platform is composed of web servers (Sec. 5.17), load balancers and web caches. This section introduces the load balancer and the web cache architectures

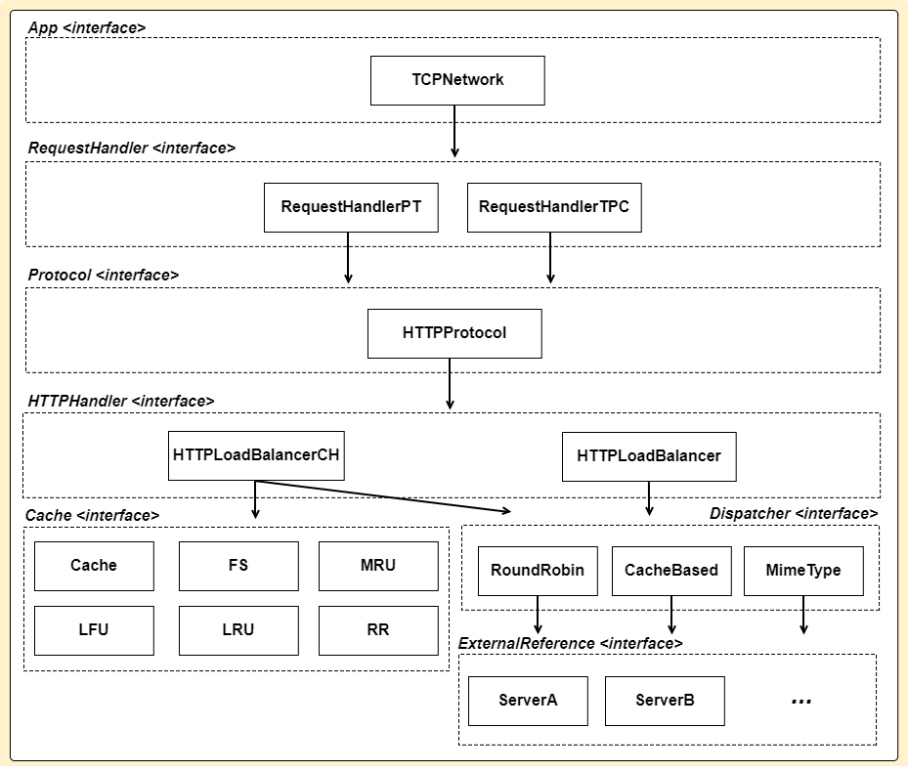

Figure 5.8: Architectural representation of the load balancer and web cache compositions.

and the two contexts on which they are used to evaluate the emergent framework.

Similarly to the emergent web server, the **Emergent Load Balancer** and **Emergent Web Cache** (Fig. 5.8) are single software instances that are autonomously assembled from small components in the repository. The emergent load balancer is responsible to receive HTTP requests and forward them to be concurrently processed in web servers running on different machines, and once the load balancer receives the response back from the servers, it returns the response back to the appropriate client. The emergent web cache is a special load balancer composition, and as such the web cache also forwards requests to other servers but caches their responses locally, so that subsequent requests to a cached content do not have to be forwarded. The load balancer and web cache share some of the components used to realise the emergent web server previously described. The components in the interfaces `App`, `RequestHandler` and `Protocol` are the same components used in the emergent web server. Consequently, the components selected to provide those interfaces are also the same. The difference is in the components providing the `HTTPHandler` interface.

In the emergent load balancer architecture, the component that provides the `HTTPHandler` interface is the `HTTPLoadBalancer` component. This component requires the interface `Dispatcher` responsible to decide how to forward the requests to the available servers in the systems. The `Dispatcher` interface has three component variations that use the concept of External Reference (see Chapter 4) to interact with external emergent services (i.e. services running on a different node in the system). The `RoundRobin` component forwards requests equally among the available servers, having the servers in a circular list, and forwarding the incoming request always to the next server on the list. The `CacheBased` component variant remembers the server to which a request for a given file was last sent (if any) and forwards further requests to that same file to the same server. In cases where the requests are made to a unseen file, the component applies the round robin strategy. Finally, the `MimeType` component variant considers the MIME type of each HTTP request and builds a mapping of MIME types to servers, such that, for example, all image requests are forwarded to server A, all text requests to server B, and so on.

The emergent web cache, as a special composition for a load balancer, shares almost all interfaces with the load balancer architecture including `App`, `RequestHandler`, `Protocol` and `Dispatcher` interface and their component variants. The difference is in the `HTTPHandler` interface, particularly the `HTTPLoadBalancerCH` component which combines both `Cache` and `Dispatcher` interfaces. The `HTTPLoadBalancerCH` uses the `Dispatcher` variants to be able to forward requests, and uses the `Cache` variants to check if requested content is locally cached. In cases when the content is not cached, the `HTTPLoadBalancerCH` uses one of the `Dispatcher` variants to forward the request to another server. Once the requested file's content returns to the web cache, it gets locally stored by a `Cache` component variant.

The distributed evaluation scenario is divided into two sub-scenarios to evaluate different aspects of the Learning module. The first sub-scenario evaluates the Learning module locus and personality of control using three machines. The first machine, the network entry point, runs both the load balancer and web cache architectures as illustrated in Fig. 5.8. The remaining machines run the emergent web server architecture depicted in Fig. 5.1. The second sub-scenario aims to evaluate

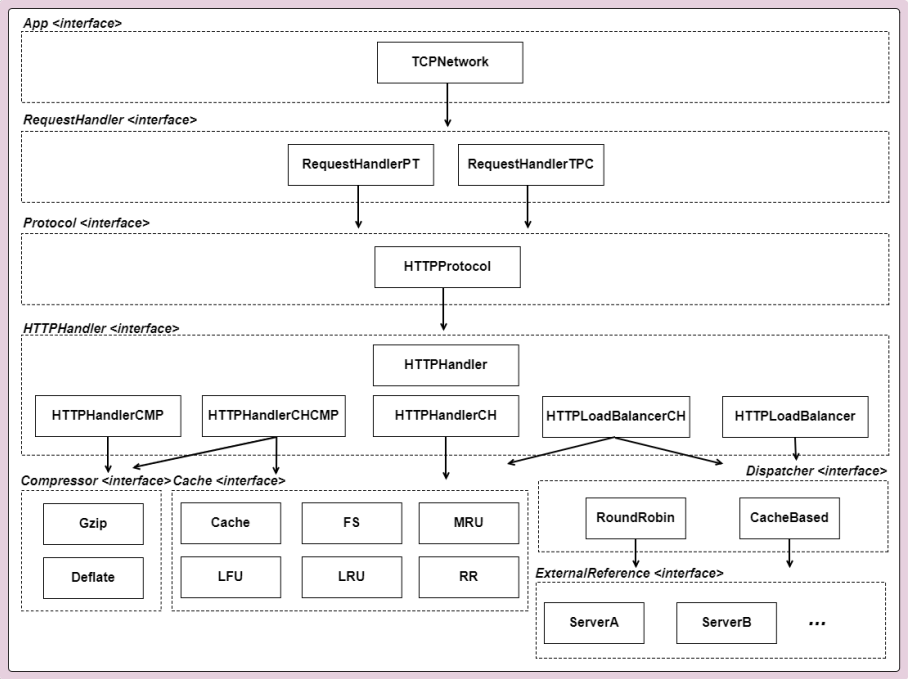

Figure 5.9: Unified architectural of the emergent web platform.

the hierarchical learning coordination in a larger search space scenario with four machines, one being the entry point. For this scenario, all components that form the web platform are available for the machines (as in Fig. 5.9). Furthermore, a change in the system's composition may also change the software running on a machine from a program to another (e.g. from a web server to a load balancer).

5.2.2 Locus and Personality Control Analysis

This section presents the results of evaluating the locus and personality control of the Learning module, using our load balancer / web server system. This gives key insights into some of the major challenges in building distributed emergent systems. The main results demonstrated in this section are: (i) different optimal compositions of our example system exist under different environments; (ii) these optimal compositions can be autonomously discovered by our framework; and (iii) coordinated and decentralised approaches in the learning dimension provide significantly different overall behaviours in emergent software systems. The experiments in this first scenario were performed with two web servers running on two identical rackmount servers in a typical datacentre environment, the load balancer on a third machine,

and a client (generating workloads) residing on a fourth machine in a different subnet. The results shown in this section was published in [37]. The source code used in this evaluation, along with instructions to replicate the results is available at [33].

The locus and personality control evaluation was conducted considering three machines. The entry point machine (referred to as LB) had a subset of the compositions for the load balancer and web cache in Fig. 5.8, whilst the other machines (referred to as WS1 and WS2) had a subset of the web server compositions in Fig. 5.1. The total composition for the load balancer / web cache is 4, in order to restrict the search space. The 4 compositions is the result of making available only one variant for the `Cache`, `Dispatcher` and `ExternalReference` interfaces. For the `Cache` interface, the `RR` component is the only one available. For the `Dispatcher`, the component available is `RoundRobin`. Finally, for the `ExternalReference` the only component available is the component with information of the two remaining servers (WS1 and WS2). The remaining machines run 4 compositions for the web server architecture. The 4 compositions results from making available only one variant for `RequestHandler`, `Cache` and `Compression` interfaces. For the `RequestHandler` interface, the only component available is the `RequestHandlerPT`. For the `Compression` interface, the component available is the `Gzip`. Finally, for the `Cache` interface, the component available is `RR`. The distributed composition is determined by the combination of the composition in each one of the participating nodes. Considering that the 3 participating nodes have 4 compositions each, the total number of global compositions, combining all machines, is 64.

Evaluating Compositions Under Different Workloads

The results shown in Fig. 5.10 and Fig. 5.11 indicate that: i) different global distributed system compositions behave very differently in the same operating environment conditions; ii) for two different environments, there are notably different global system compositions that perform optimally; and iii) there are very clear *groups* of compositions with similar performance levels in both environments.

In detail, Fig. 5.10 and Fig. 5.11 show the average response time to client requests, as reported at the entry point (LB) machine, for every possible global dis-

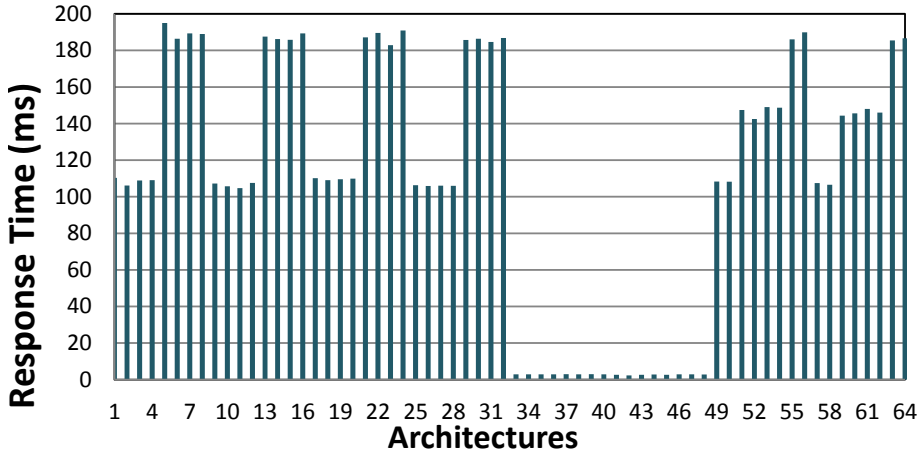

Figure 5.10: Average response time of every available distributed composition for the first distributed scenario when using Workload 1.

tributed system composition (i.e. every possible composition of components for every local node, combined) for two tested environments, characterised by different workloads (Workload 1 and Workload 2). The Workload 1 consists of small text file (html) requests with low variation (similar to Fig. 5.2 (a)). The Workload 2, on the other hand, consists of small text file (html) requests with high variation, meaning that a different text file is always requested (similar to Fig. 5.3 (a)).

For Workload 1, shown in Fig. 5.10, there is a difference of 193 ms in request handling latency between the best and worst performing distributed compositions. Similarly, for Workload 2 shown in Fig. 5.11, the difference in request handling latency between the best and worst architecture is 126 ms. This clearly shows that different compositions have significant impact on overall performance (result (i)).

Comparing these two graphs, it is also noticeable that the best performing global architecture in both scenarios are different (result (ii)). For Workload 1, the best architecture is the architecture number 33, which has the following composition: cache in the LB machine, only compression in WS1, and only compression in WS2 (LB-C, WS1-GZ, WS2-GZ)[3], whilst for Workload 2 the best architectural composition is the number 49 that has the global composition: LB-RR, WS1-GZ, WS2-GZ.

[3]A global composition is described as: <Machine-composition> format. LB stands for load balancer, WS1 stands for web server 1, WS2 stands for web server 2. The compositions are indicated by their initials: C for Cache, GZ for Compression, CGZ for Cache and Compression and RR for Round Robin. Therefore, LB-C, WS1-GZ, WS2-GZ, means load balancer with cache, web server 1 with Compression and web server 2 with Compression.

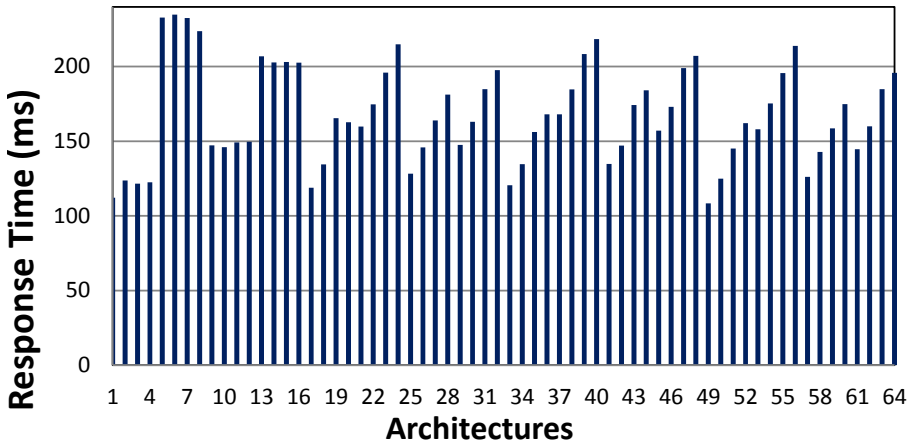

Figure 5.11: Average response time of every available distributed composition when using Workload 2.

Considering that Workload 1 consists of one client repeatedly requesting only one text-only html file, global architecture number 33 performs best because, in this composition, the web servers always compress the requested files, and once the file is returned to the load balancer, it is stored in a small content cache at the load balancer (i.e. web cache composition). Thereafter, all subsequent requests for the same file can be retrieved instantly by the entry point machine itself, from its local cache, avoiding forwarding requests, and thus significantly reducing response time.

On the other hand, for Workload 2, which consists of one client repeatedly requesting a different text-only html file for every request, architectures with caching will not perform well due to the frequency of cache misses. As a result, the best performing architecture is one which does not use cache at either the entry point or the web servers. The architecture number 49 defines a round-robin scheduling algorithm for the load balance, and web servers that return compressed files (from disk) as responses. As for the load balancer scheduling algorithm (LB-RR), each incoming request is evenly distributed among to the web servers.

The last notable result here is the equivalent performance of large groups of architectural compositions (result (iii)). This is easily identified in both workloads, though is visually more obvious in Fig. 5.10. One clear reason for this is that both web servers are running on machines with the exact same hardware features and capacity. Thus, a global distributed composition that sets WS1 to composition

X and WS2 to composition Y is essentially the same (performance-wise) as having WS1 set to Y and WS2 to X. Furthermore, whenever requests are limited to a subset of the system's nodes, all compositions of the unreachable nodes do not affect the system's performance, making those compositions indifferent for the system. This situation is observed in two cases: the first case happens when the load balancer, due to its scheduling algorithm, forwards 100% of the incoming requests to only one web server. This is observed in architectures number 1 to 32 in Workload 1, and from architectures 1 to 16 in Workload 2. The second case happens when the load balancer forwards the incoming requests only once to one web server, and all subsequent requests are handled locally due to the entry point web cache composition. This happens only in Workload 1 and is observable for architectures number 33 to 48.

Learning Coordination Analysis

The above results provide a ground truth, informing us which distributed system compositions are the best options for the two workloads. The different locus and personality strategies is examined using the emergent framework. In both cases the brute-force learning algorithm is used, examining a single and centralised instance of the Learning module controlling the entire system, and also in a complete decentralised fashion, where every node runs an instance of the Learning module.

The results are shown in Fig. 5.12 and Fig. 5.13. Both graphs show the learning process and its convergence to the optimal composition. Additionally, both graphs show the performance of three different compositions of the system, for comparison: the learning line is the version running the emergent framework to control the system (the orange lines in both Fig. 5.12 and Fig. 5.13), whereas the blue and green lines are fixed architectural compositions used as reference points (i.e. their compositions do not change over time). The composition represented by the blue line is LB-C, WS1-GZ and WS2-GZ, and is the best performing distributed composition for Workload 1, as shown by the earlier results. The green line then represents the architectural composition LB-RR, WS1-GZ and WS2-GZ, which is the best performing composition for Workload 2. At the midpoint of both experiments the workload is changed from Workload 1 to Workload 2, therefore demonstrating the way in which the framework learns and reacts to changes in the environment.

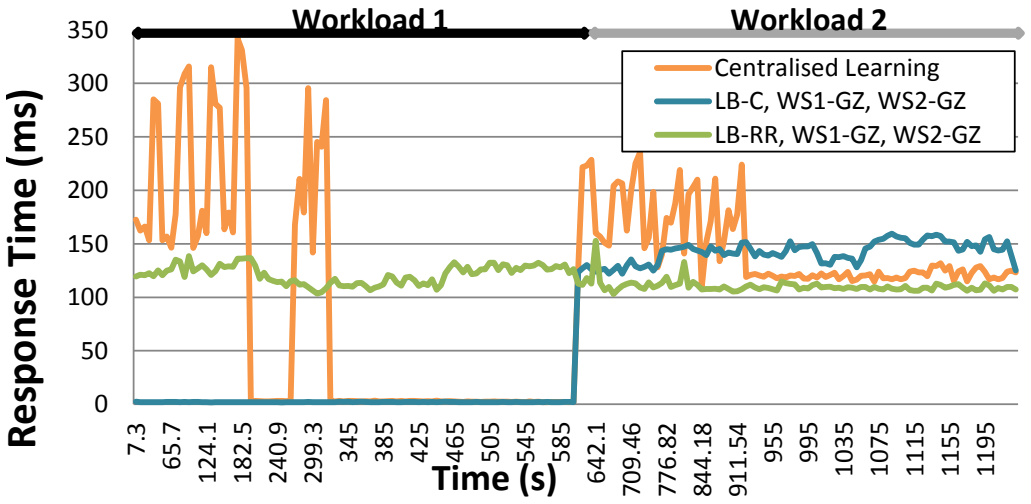

Figure 5.12: Performance of our coordinated learning approach for two different workloads, compared with static baseline compositions. The spikes at the beginning of the coordinated learning curve for both workloads represent the exploration phase.

These graphs show that: i) the centralised learning approach, in both workloads, autonomously identifies the optimal global architectural composition, with no prior information; ii) the decentralised learning approach, for Workload 1, converges faster than the coordinated learning approach, but it never converges for Workload 2.

In detail, the centralised (altruistic) learning approach, as shown in Fig. 5.12, takes 320 seconds (∼5 minutes) to find the optimal composition for each environment it encounters. This is because the brute-force learning algorithm works by exposing each of the 64 available compositions and observing them for 5 seconds, after which it selects the composition that had the lowest average response time. The advantage of this approach is that it will always find the optimal solution, due to its exhaustive search. On the other hand, this approach does not scale to larger numbers of components and/or nodes due to the combinatorial explosion problem.

The decentralised (selfish) learning approach, by comparison, converges 15 times faster than the centralised approach when exposed to Workload 1, as shown in Fig. 5.13. The rapid convergence is due to the decentralised learning process, which at the entry point machine must only iterate through 4 different compositions, quickly identifying the local content cache in the entry point machine as the best option, this composition suffers little performance impact from the web servers (as was seen in Fig. 5.10). However, this approach never converges to the optimal

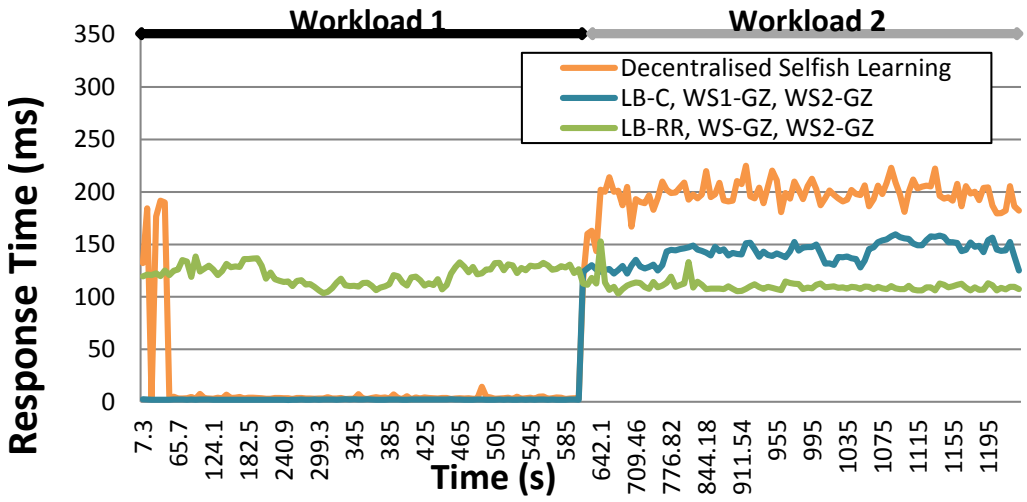

Figure 5.13: Performance of our decentralised learning approach for two different workloads, compared with static baseline compositions. The spikes at the beginning of the selfish learning curve for Workload 1 represent the exploration phase.

solution when exposed to Workload 2. In this more complex workload, in which the 'local content caching' solution at the entry point machine is not the best option. This lack of convergence is due to a complex set of interference conditions: first, because each local emergent system is exploring independently, it is unlikely that all nodes will simultaneously happen to be in the globally optimal set of compositions at the same moment; second, the locally optimal solution for the web server nodes is not necessarily the globally optimal one; and third, the act of exploring different compositions on different nodes causes constant changes in observed metrics and observed events at both the load balancer and the web servers. In the more complex Workload 2, these conditions mean that all nodes are in a constant state of learning.

5.2.3 Hierarchical Coordination Analysis

This section presents the results of evaluating the hierarchical coordination learning algorithm. This coordination strategy enables the system to learn not only the composition for each of the participate machines in the system, but also to determine their role (i.e. the executing program, e.g. a web server, a web cache or a load balancer) in the system. As the system becomes able to autonomously learn more system's aspects, the learning problems (see Chapter 3) become more prominent and challenging. This section focuses on evaluating the learning approach regarding

the i) search space growth, ii) interference effects during learning, and iii) invalid global compositions. The section also describes the scenario on which one of the feature-based approach assumption is not held, exemplifying the undesirable effects.

The experiment deploys the architectural compositions in Fig. 5.9 in every participating node, and uses four machines to realise the system, one of these machines being the entry point. Note that the `MimeType` component in that architecture is not included in the `Dispatcher` interface, because it breaks the feature-based assumption that the worst composition for the best feature has to be best than any other composition of other features (see Chapter 4). In total, the number of unique compositions in Fig. 5.9 is 70. Therefore, considering 4 machines, the total number of global compositions is $70^4 = 24,010,000$, which includes invalid compositions and only one component available for the `ExternalReference` interface. The hierarchical coordination algorithm separates the valid compositions in groups. One of these valid groups is the case where the entry point machine communicates directly with the remaining three machines, placing them in the first level of the system's hierarchy. This is the case used to evaluate the system learning, having 70 available compositions in the entry point and 42 compositions in the remaining three machines resulting a total of $70 * (42^3) = 5,186,160$ global compositions.

Evaluating Compositions Under Different Workloads

In the experiment described in Sec. 5.2.2, the role (i.e. the executing program) of each of the machines in the system was predefined. The entry point machine was fixed as load balancer / web cache and the remaining two machines were fixed as web servers. In this experiment, however, the machine's roles in the system is also autonomously defined. For that, the entry point machine has to learn in what situation is advantageous to handle all incoming requests locally, and when to forward the requests to be handled by other participating nodes.

The graphs show two synthetic workloads that demonstrate (in Fig. 5.14 (a)) the web server role is more advantageous than load balancer, and (in Fig. 5.14 (b)) the opposite, i.e. when forwarding requests is better than handling them locally. The workloads used in those graphs consist of requests to dynamic content, i.e. content

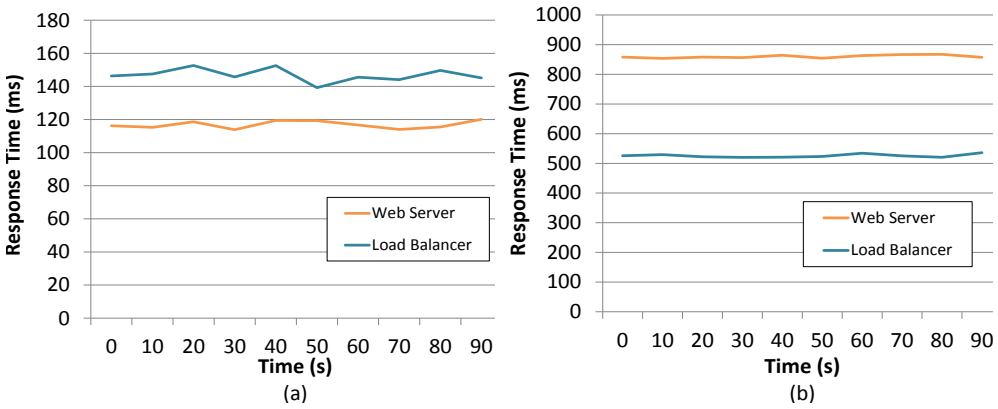

Figure 5.14: Performance of a web server and load balancer composition exposed to the Dynamic Workload A (a) and Dynamic Workload B (b).

that is processed and generated upon each request. Dynamic content was preferred over static content used in previous evaluation, because they clearly differentiate the need for Web Server or Load Balancer. The Dynamic Workload A (Fig. 5.14 (a)) is composed of sequential cycles of 5 simultaneous requests to dynamic content. Once the 5 simultaneous requests are handled, the client casts 5 more requests, and so on. The dynamic content in the Dynamic Workload A takes ∼87 ms to be processed. Similarly, the Dynamic Workload B (Fig. 5.14 (b)) is composed of cycles of 10 simultaneous requests to content that takes ∼397 ms to be processed.

The results in the graph show that as the average time to handle dynamic content increases, forwarding requests to be processed in parallel by other machines becomes more advantageous. As the processing time decreases, there is no benefit for the system to forward requests. That is why the dynamic content was chosen to realise the experiment. The static content workload maintain very low processing time (i.e. only disk access time), making it less noticeable when to choose one composition over the other. As the processing time increases, it is more advantageous to spend time forwarding the requests, whilst saving time in splitting the processing time among three servers, than processing all requests in one machine. This result is well-known in distributed systems field, but is not evident for the Learning module, which executes with no prior information about the workload or the target system.

Learning Coordination Analysis

This section shows the hierarchical coordination approach and the feature-based learning algorithm locating optimal distributed compositions. For the Dynamic Workload A, the learning system falls in the local case scenario explored in Sec. 5.1, where the results demonstrate that using either the brute-force and featured-based approach the system is able to learn the optimal compositions locally. Therefore, this section focuses on the learning system when exposed to the Dynamic Workload B.

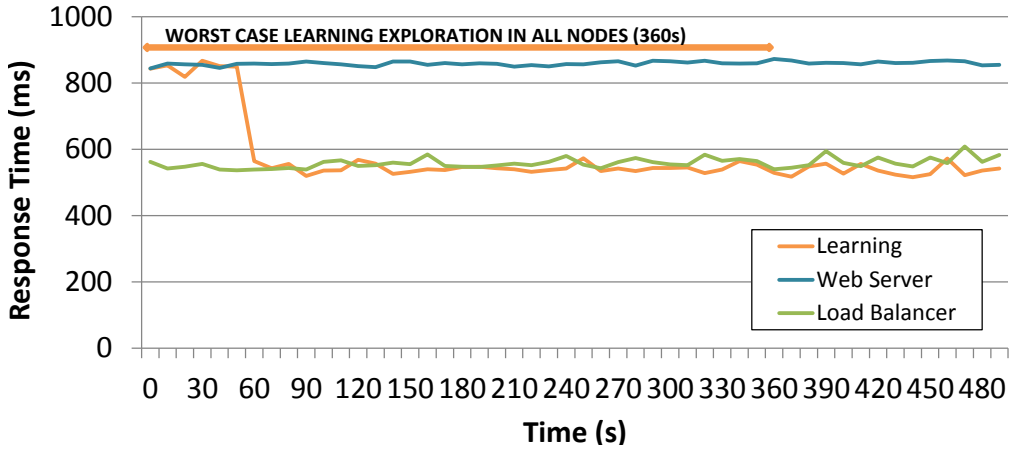

Figure 5.15: Performance convergence (60s) of the hierarchical coordination when exposed to the Dynamic Workload B. The number of valid global compositions is 5,186,160. The brute-force learning would theoretically converge in 1.6 years.

As shown in Fig. 5.14 (b), when the system is exposed to Dynamic Workload B the best performing composition is a distributed composition, which has the entry point node to forward requests to the remaining servers. Fig. 5.15 shows the learning system undergoing through the process of locating the optimal composition in the distributed environment. The 'Web Server' and 'Load Balancer' line represents the performance of the best and worst fixed composition (no learning, nor adaptation happening) for the workload. The 'Learning' line (the orange line in the graph), on the other hand, is the performance of the entry point node running the learning system throughout the entire experiment. The system starts exploring using the feature-based algorithm in the entry point machine. Considering the architectural compositions in the entry point (represented in Fig. 5.9) the feature-based system first tests two compositions for the `RequestHandler` interface, and then tests six

compositions for the `HTTPHandler` interface, converging to a `LoadBalancer` component at that level. From that part of the exploration phase onwards, the system has already converged to the optimal performance, given that the 42 web server compositions running on the remaining servers have similar response time, and thus they do not affect the system's performance even during their exploration phase.

The system (from the 80 sec time forward) continues its exploration phase comparing the two different component variants for the `Dispatcher` interface, and deciding for the best between the two available ones. The Dynamic Workload B requests dynamic content with high variation (each request to a different content) making the `CBS` and `RoundRobin` to forward requests evenly among the remaining servers, which results in them having similar performance. After locating the optimal composition in the entry point node, the system then triggers learning in the remaining servers concurrently. Since the remaining servers are in the first level of the architecture (according to the exploration group of the experiment), the learning system executing in the remaining nodes works just as in the local scenario, locating the best performing composition among the 42 web server composition. Due to the experimented workload, the web server compositions have very similar performance and again they do not affect the global system performance.

This experiment show that the system converges in 60 seconds, but the exploration phase, considering the learning executing in the remaining 3 web servers, may last (in the worst case scenario) for 360 seconds. That is 180 seconds running the exploration phase in the entry point node, and 180 seconds concurrently running the exploration phase in the remaining servers, resulting in a total of 360 seconds. Therefore, the hierarchical coordination is able to explore larger search spaces in less time by locating the optimal feature before zooming in on the search to locate the optimal composition. Furthermore, by diving the exploration phase in groups, the system avoids invalid global and interference effects in learning. The latter is avoided by executing the learning process in each level of the hierarchy in different times. In this case particularly, the learning starts in the entry point machine before triggering learning in the web servers which are in subsequent level.

Overall, the hierarchy coordination in tandem with the feature-based algorithm

implement concepts that allow the management of the search space, interference effects and invalid composition challenges. However, the presented implementation is not applicable to every domain. For example, the case including the `MimeType` component for the `Dispatcher` interface. In that scenario, for the Dynamic Workload B, the system will only forward requests to only one server, since all requests are considered 'php' mime type, thus concentrating the processing in only one machine, making this composition worst than any web server composition. Therefore, this composition breaks the assumption of the feature-based search that the worst composition for the best feature must be better than the best composition of the other features. In this special case, the feature-based algorithm could coincidently compare the `MimeType` component as the representative composition for the load balancer / web cache against a web server composition, and mistakenly conclude that being a load balancer is worse than being a web server in that case.

5.3 Additional Learning Aspects

This section evaluates other fundamental aspects of the online learning process. In detail, this section evaluates the environment classification algorithm and the observation window size. The environment classification is key to enable the system to 'remember' previously seen operating conditions, and also to enable a fair comparison between compositions operating under equivalent conditions. The environment classification algorithm is executed in extreme scenarios to evaluate the situations on which the algorithm is not suitable. The observation window size, on the other hand, influences directly the quality of learning, affecting directly the perception of the operating conditions through the executing compositions. The observation window size evaluation consists of testing different window sizes to find the one that supports the best learning process.

5.3.1 Environment Classification

The environment classification algorithm is described in Chapter. 4. The presented algorithm uses ranges of values for each of the features to classify the operating con-

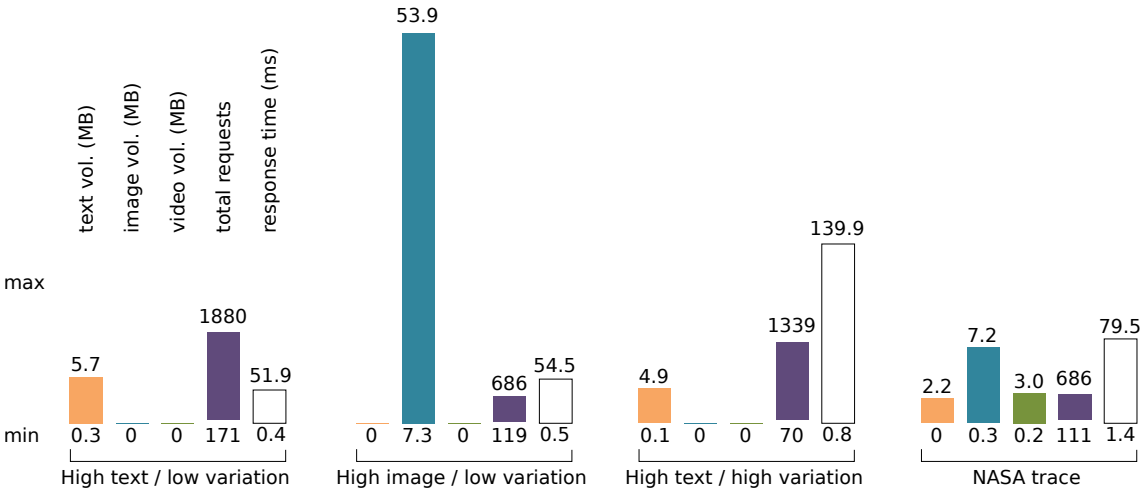

Figure 5.16: Classification results: environment ranges detected for each workload.

ditions during the exploration phase, to mitigate the self-referential fitness problem, particularly the distortions resulted from extracting information of the environment through different architectural compositions (for more information see Chapter 3). The evaluation of the classification algorithm is divided into two parts. Firstly, this section examines the classified ranges that were identified for the workloads described in Sec. 5.1.2. Secondly, this section evaluates the limitations of the range-based classification algorithm showing two graphs that explores i) classification of environments that have **overlapping ranges**, and ii) **mid-exploration workload changes**, which are the factors that most affect the proposed solution.

The classification ranges of the workloads of the local scenario are shown in Fig. 5.16. The graph demonstrates that the rages-based classification algorithm is able to detect most of the differences between the tested workloads, but not all. Specifically, the low-variation text, low-variation image, and NASA workloads all have different classified environment ranges, allowing their corresponding optimal architectures to be easily distinguished. By contrast, the low-variation text and high-variation text workloads have very similar classified ranges, lacking a clear distinguishing feature; this is because the events that are generated by the web server do not capture the idea of variation (the ratio of adjacent files in a request sequence that are different) within a single media type. Developing ways to infer this kind of attribute from the primary event stream, or otherwise report that additional event detail would be useful to distinguish ranges, are directions for future work.

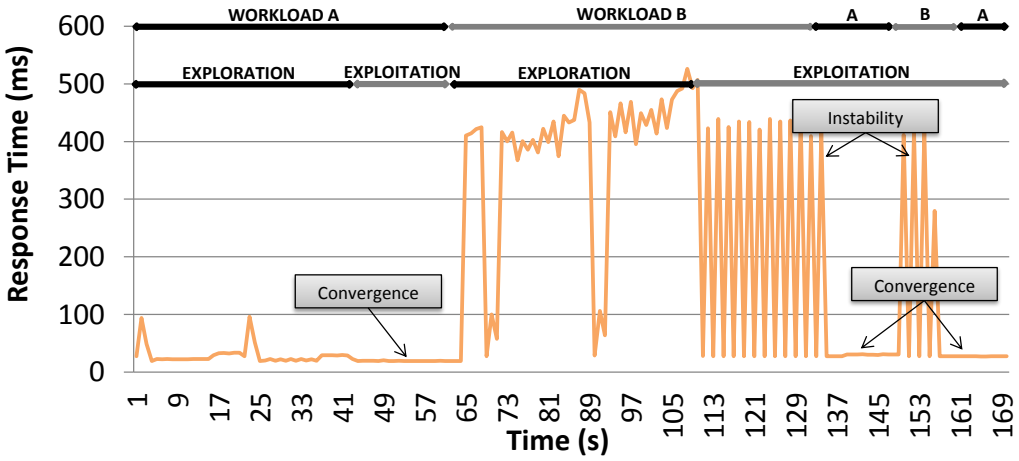

Figure 5.17: Classification experiment using request patterns that are difficult to distinguish.

The next graphs show the learning process under difficult environment classification conditions to further highlight the challenges involved. Fig. 5.17 shows the possible effects of being unable to distinguish between two environment ranges, as is the case with high- and low-variation text scenarios. At the beginning of this experiment, the system is subjected to a low-variation text workload (Workload A) until the system converges to an optimal composition (a composition using cache). Then, after converging and switching to exploitation mode, the workload is changed to high-variation text (Workload B). At this point (time 67) the Learning module observes a degradation in performance, indicated by the large spike in response time, and waits for three consecutive observations in this performance change. The Learning module then triggers a new exploration activity, which finishes learning at time 111 and converges to an optimal composition for this workload (a composition that does not use cache, in this case). At this point (time 113) an undesirable effect occurs: the system oscillates between two compositions, one of which performs very poorly and the other which performs very well. This is because, having chosen the non-cache composition, this brings response time back down to a low level, within the limits of the response time used in the classified range of Workload A. This triggers the system to assume that the environment is now Workload A, and so the system selects the best known composition for that environment, which is a composition that uses cache. However, this composition performs very badly under

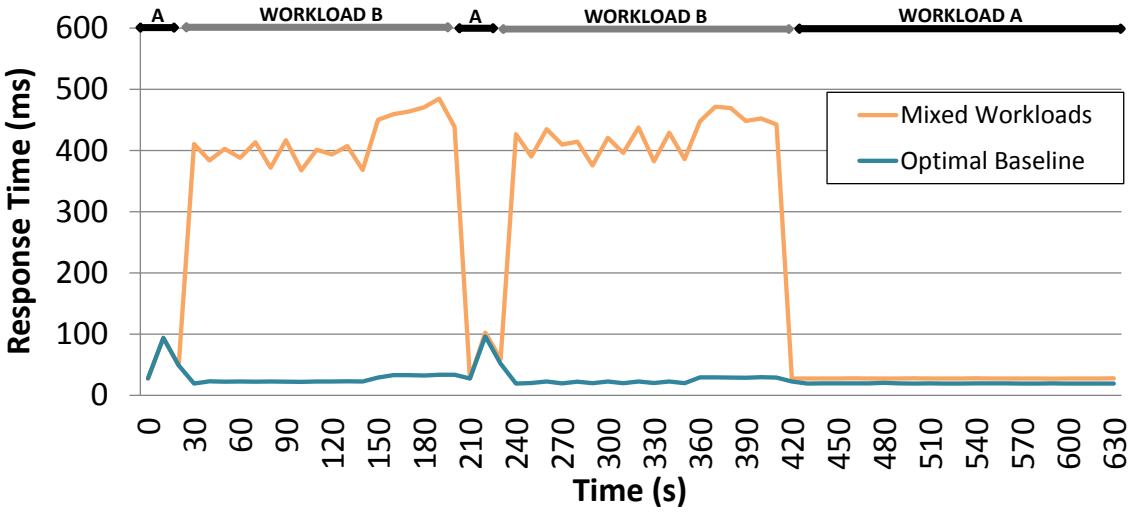

Figure 5.18: Classification experiment using mid-exploration workload changes.

the high variation workload, which in turn pushes response time back into the range constructed for Workload B. This causes the system to assume that the environment is now Workload A, and so selects the composition that does not use cache. This cycle then repeats until the workload changes at time 137.

The inability to appropriately distinguish environment ranges from one another, combined with the use of response times to help trigger current environment range checks, in this case causes a continuous cycle of poor decisions that need to be constantly corrected. This demonstrates the critical nature of environment classification in emergent software, and the challenge in doing this when the environment's effects are partially coupled with the performance characteristics of the system itself.

Fig. 5.18 shows the possible effects of the workload changing during an exploration activity. Here the system starts in Workload A, then during the exploration activity the environment changes to Workload B, then back to A and finally B again. Once the exploration activity finishes (420 seconds), the environment transitions back to Workload A. When exploration finishes, the graphs shows that the system converges to a composition that is not quite as good as the known optimal composition for Workload A, having been unable to accurately compare all architectures under the same conditions during exploration. This highlights the need to better distinguish environment ranges during exploration activities. However, note that the end result of this experiment does still leave the system relatively close to

the optimal solution, despite our simple approach to classification, suggesting that this is not as significant an issue as that highlighted in Fig. 5.17.

5.3.2 Window Size Effects in Online Learning

The common observation window size for most of the results previously shown in this chapter is 10 seconds. This section investigates alternative window sizes and their affects on learning behaviour. The evaluation was conducted by running the entire NASA trace, with each request in the trace delivered in sequence, to the local emergent web server, using five different observation window sizes, from 1 second up to 20 seconds. The effects of this are shown in Fig. 5.19 and Fig. 5.20, with further extracted details shown in Table 5.1. In each experiment the web server is run for the same portion of the NASA trace. The evaluation shows the total number of adaptations that occur, and the number of exploration activities that are triggered.

The analysis of the window size illustrated in Fig. 5.19 and Fig. 5.20 are relatively complex, with a set of interacting features. Overall the graphs show that the experiment using a 10 second observation window size completes the trace fastest, indicating that it had the best overall response times. There are two different reasons for this. First, the use of smaller observation windows causes *over-reaction* to the variations in the workload, and an unnecessarily large number of adaptations. These adaptations, whilst cheap, do momentarily impact the response time of the web server, resulting for example in the 5 second experiment having far higher peaks than the 10 second experiment shown in Fig. 5.20. The data for the 1 and 2 second window sizes, in Fig. 5.19, show even more volatility as the system attempts to over-fit to the environment. Second, the use of larger observation windows causes

	1sec	2sec	5sec	10sec	20sec
# of classes	7	6	3	2	2
# of adaptations	480 (186)	376 (124)	141 (15)	104 (20)	84 (0)
Experiment duration	1 hour	1 hour	50 min	39 min	48 min

Table 5.1: Adaptation details of observation window experiments, showing total number of adaptations performed, and adaptations performed under exploitation shown in brackets.

under-reaction to the variations in the workload. This is exemplified by the 20 second observation window experiment, which has two exploration phases, but never performs any adaptations outside of those phases during exploitation, indicating that it has classified very broad environment ranges, missing important details.

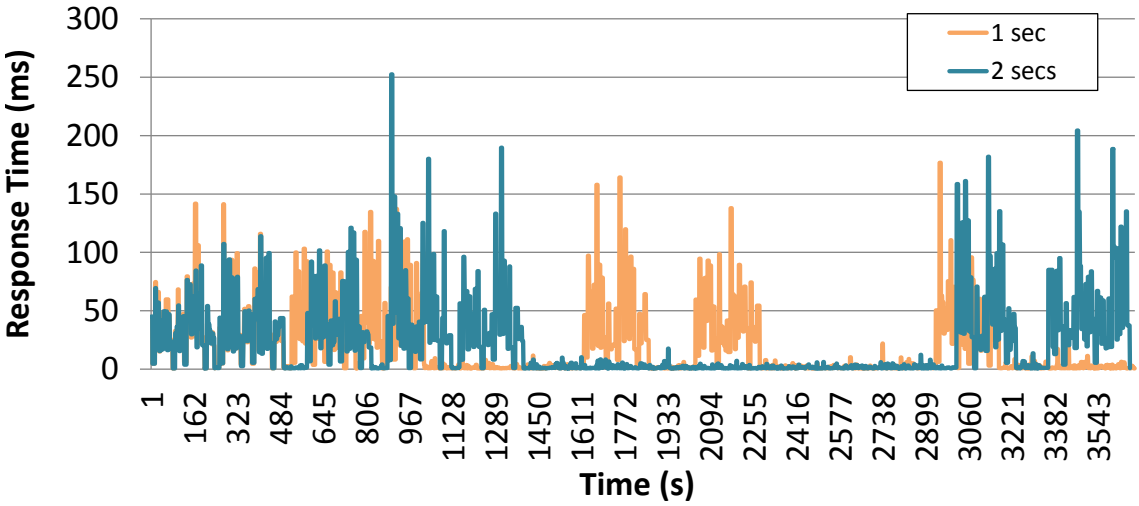

Figure 5.19: Observation window experiments of 1 and 2 seconds.

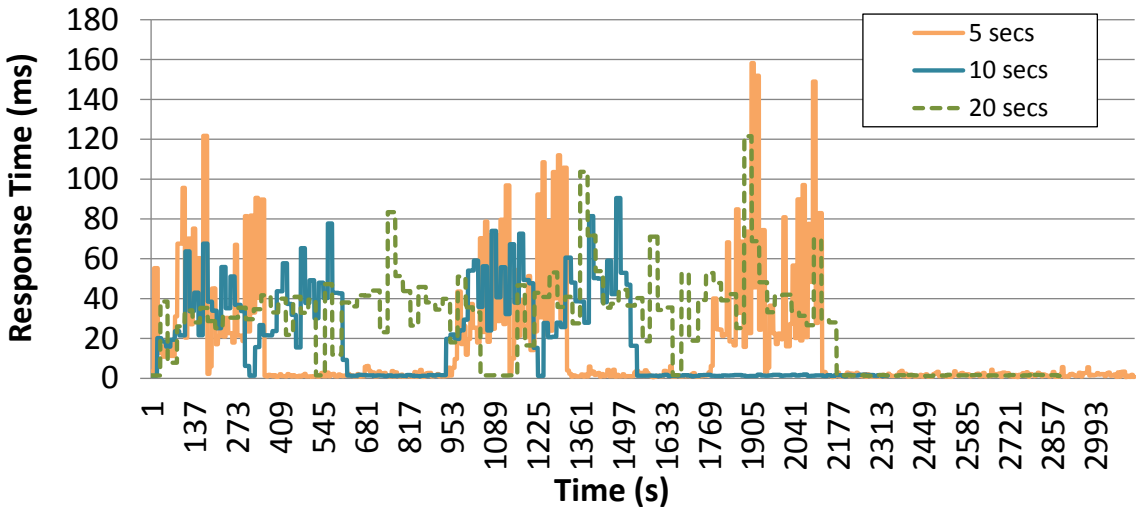

Figure 5.20: Observation window experiments of 5, 10 and 20 seconds.

This demonstrates that the composition of the optimal observation window size is itself a challenge, and may have a different ideal setting across different types of application, different workload characteristics, and different hardware platforms. Beyond the static observation windows used here, it may also be useful in the future to explore dynamic observation window sizes.

5.4 Summary

This chapter presented two main scenarios that were used to evaluate the emergent system framework described in Chapter 4. The first scenario consists of evaluating the emergent framework to autonomously assemble and optimise a single instance of a web server. The second scenario consists of exploring the emergent framework in distributed settings. Both scenarios aim to evaluate learning strategies in terms of convergence rate (i.e. how fast it converges) and accuracy (i.e. if the system locates the optimal composition). Furthermore, this chapter also evaluates additional aspects of the learning process, such as the environment classification algorithm and the size of the observation window used during the learning process.

In detail, for the local scenario, this chapter evaluates both the brute-force and feature-based approach applied to a single instance of the web server. The results show that the emergent system concept enables the continuous composition and optimisation of the web server according to the workloads. Furthermore, the results show that the feature-based approach converges, in the best-case scenario, 7 times faster than the brute-force, and more than 2 times faster in the worst-case scenario.

In the distributed scenario, the chapter evaluates different coordination strategies for the learning process. The results show that the centralised altruistic approach converges to the optimal solution, but this approach is inefficient as the number of composition grow exponentially in distributed settings. The decentralised selfish strategy converges faster and accurately in some environments, but suffers from learning interference effects in other scenarios. Finally, for the coordination analysis, this chapter show that a hybrid coordination strategy (the hierarchical coordination) converges faster and accurately considering a larger search space, whilst avoiding interference effects and invalid global compositions. This approach, however, is only applicable in domains where the feature-based assumptions are maintained.

Finally, the chapter evaluates the environment classification algorithm, and shows the results of experimenting with different sizes for the observation window. The results show that the environment classification suffers from classifications that present overlapping ranges, mid-exploration workload changes and when some important

environment feature is not defined as an event (perception data, see Chapter 4). The overlapping ranges classification confuses the system during exploitation phase, causing instability in the system performance. The mid-exploration changes 'deceives' the system to converge to composition that is not as good as the known optimal. The observation window size depends on the application domain and on the system's workload, so that for different scenarios, different sizes might be better. The results for the observation window size in the explored scenario and workload points to the 10 second window as the most appropriate size for a more accurate learning. However, further investigation on this aspect of the system is necessary, including an investigation on dynamic size of the observation window.

Although the different aspects of the emergent framework evaluated in this chapter is not applicable to every application domain, nor covers every learning challenge in Chapter 3, the presented solution implements important concepts that were demonstrated to be key to realise the paradigm of emergent systems. Furthermore, the shortcomings of the presented solution establish the next research steps that are detailed in the next chapter as future work.

CHAPTER 6

Conclusion

Autonomic computing has emerged as an important area to address the increasing complexity in developing, configuring, and managing software systems. Due to the uncertainty of highly dynamic operating environments, the development of such solutions becomes very challenging. As a response to the complexity of creating autonomous solutions, this thesis has introduced Emergent Software Systems, with a real-world implementation of the concept and an extensive empirical evaluation.

The main goal of Emergent Systems is to autonomously assemble and optimise software systems in both local and distributed environments with minimum human involvement. As a consequence of the presented concept, systems become able to accommodate changes in their operating environment with no offline training nor predefined domain information, reducing the engineering effort required to create autonomous solutions. Furthermore, the ability to freely explore the available design possibilities, with no restrictions, enable the system to handle unexpected changes.

This chapter revisits previous chapters, highlighting this thesis' main contributions and the novelty of the presented approach. Moreover, this chapter revisits the research questions defined in Chapter 1, addressing the questions with references to the results presented in Chapter 5, and concepts discussed in both Chapter 3 and Chapter 4. Finally, this chapter describes several avenues for future work, and concludes the thesis with the final remarks.

6.1 Thesis Summary

Chapter 1 introduced the thesis, contextualising the work and its relevance to the Autonomous Computing research area. This chapter also described this thesis' main objectives and contributions, concluding with a presentation of the thesis' structure.

Chapter 2 presented the background concept used in this thesis, whilst also presenting relevant related work. This chapter presented a literature analysis, comparing the concept of Emergent Systems with key work in autonomous systems, and detailing the advantages of Emergent Systems and the gap this concept addresses.

Chapter 3 defined the concept of Emergent Software Systems in terms of autonomously composing software systems from small units of behaviour and online learning techniques. This chapter also introduced the main challenges in realising Emergent Systems in both local and distributed scenarios.

Chapter 4 described the framework implementation that supports the concept of Emergent Systems. This chapter introduced the Perception, Assembly and Learning (PAL) modules as the core framework to realise Emergent Systems in local settings. Furthermore, this chapter presented an extension to the PAL framework to support the concept of Emergent Systems in distributed scenarios.

Chapter 5 evaluated the emergent framework applied to datacentre software. The results presented in this chapter demonstrate the feasibility and potential of Emergent Systems. Furthermore, this chapter also analysed the limitations of the presented solutions, presenting further challenges to be addressed in future work.

6.2 Revisiting the Hypothesis

Hypothesis: The use of fine-grained software components in tandem with reinforcement learning as a method to develop self-adaptive software systems is key to address operating environment dynamism whilst minimising design complexity.

This thesis presented the concept of Emergent Software System as derived from the hypothesis. The concept applies component-based modelling and an online learning process to assemble and optimise systems with no predefined knowledge of both components implementation and execution environment. The component-based model is used by the emergent framework to assemble valid software architectures and enable runtime adaptation, eliminating the need for extra models to define parts of the software that should be adaptable as well as the process to adapt those parts (e.g. feature-based models, adaptation policies, etc.). The utilisation of a component-based model, with the presence of component variants (different implementations for the same functionality), supports the reinforcement learning approach. This learning strategy allows the system to experiment with different architectural compositions as the system runs in its executing environment and handles incoming requests. This strategy enables the system to build its own understanding of its constituent parts when exposed to varying operating conditions, eliminating the need for experts to create rules / models to support systems adaptation.

The utilisation of component-based models and online learning approach minimises the complexity of creating autonomic systems, because they eliminate the need to create adaptation rules and models, supporting the online learning approach that pushes design decisions to be made at runtime by the system itself. The evaluation (Chapter 5) focused on showing that the component-based model in tandem with the reinforcement learning approach works, i.e. the system is able to learn under different operating conditions optimal software compositions with no predefined information. The results show the system autonomously composing real-world data centre software, and learning optimal compositions in both local and distributed settings, driven by real-world, relevant workloads. Hence, the author argues that this thesis demonstrates that the Emergent Software Systems concept enables a complete autonomous creation and evolution of adaptation logic, minimising the upfront effort of creating autonomous systems.

6.3 Revisiting the Research Questions

[**RQ 1**] Do different software compositions have different performance when subjected to different operating environments?

This thesis has extensively demonstrated that for different environments, different compositions present different performance. Particularly, Chapter 5 shows that i) different features of the environment directly affects different software composition for incorporating different functionalities, and ii) in a large search space of software compositions, many distinct compositions have very similar performance for implementing similar functionalities. For the result (i) this thesis shows that:

- Low variation workloads (i.e. small set of different requested content) benefit web server compositions with cache components. Contrarily, high variation workloads (i.e. a large set of different requested content) benefit web server compositions that do not have cache components. This is shown in Sec. 5.1.2 for a single web server and in Sec. 5.2.2 in a distributed setting.

- Workloads with small files (∼3KB) benefit compositions that have cache but do not have compression components. Contrarily, workloads with bigger files (∼64KB) benefit compositions that have cache and compression components. This is shown in Sec. 5.1.2.

- Workloads with dynamic content requested in sequential cycles of 10 simultaneous requests benefit load balancer compositions. Contrarily, workloads with static or dynamic content requested in sequential cycles of 5 simultaneous requests benefit web server compositions. This is shown in Sec. 5.2.3.

This thesis shows that in a large search space of software compositions, many distinct compositions have similar performance (ii). This result is shown in 5.2.3, and it was used as motivation to propose the feature-based learning algorithm (Chapter 4) to handle significant growth in the search space, particularly in distributed settings.

This phenomenon of having different software compositions presenting different performance in different environment is named 'Divergent Optimality', and it was

presented as a property of Emergent Software Systems in Chapter 3. This property is essential to the concept of Emergent Systems because i) it motivates the creation of adaptive software, and ii) shows the ground truth that were used to demonstrate that Emergent Systems are able to accurately converge towards optimal composition.

[**RQ 2**] Is it possible to autonomously locate optimal software composition (within a set of available options) with no predefined nor domain-specific information?

This thesis shows that emergent systems accurately converge towards optimal software compositions with no predefined information in both local (Sec. 5.1) and distributed settings (Sec. 5.2) when exposed to a variety of workloads in different contexts, with few exceptions. Particularly, this thesis shows:

- Fig. 5.4 shows the system accurately converging when exposed to distinct synthetic workloads using the brute-force algorithm. This graph also shows the system quickly changes its composition when detected a previously seen workload and takes 420 seconds to learn a newly identified environment.
- Fig. 5.5 shows the system accurately converging when exposed to NASA trace.
- Fig. 5.6 and Fig. 5.7 show the system accurately converging using the featured-based learning approach, which converges 360 seconds faster than the brute-force approach in the best-case scenario, and 240 seconds faster than the brute-force approach in the worst-case scenario.
- Fig. 5.12, Fig. 5.13 (Workload 1) and Fig. 5.15 show the system accurately converging in distributed settings, with the exception of the decentralised approach when exposed to high variation of requested files (Workload 2 in Fig. 5.13).

The results shown in this thesis demonstrate that emergent systems are capable of autonomously locate optimal compositions of software in both local and distributed settings, specially using the brute-force (and centralised, in distributed settings) approach. One of the main problems, however, is the growth of the search space that directly affects convergence rate. For that problem, this thesis presented

a featured-based algorithm that are shown to perform significantly better than the brute-force (baseline) approach. Nonetheless, the featured-based approach relies on the assumption that the worst software composition of the best located feature has to be better than any composition of the remaining software features. Further investigation to propose an approach with no such limitation is required. The exception cases where emergent systems could not accurately converge are:

- Fig. 5.13 (Workload 2) shows the distributed coordination approach exposed to high variation of requested files. Due to interference effects from simultaneously executing learning modules, the system is unable to converge.
- Fig. 5.17 shows instability during exploration phase when the system are exposed to environments that present overlaps in the classification ranges.
- Fig. 5.18 shows the system converging to a suboptimal composition due to mid-exploration workload change.
- Fig. 5.19, Fig. 5.20 and Table 5.1 suggests that different observation window sizes can affect the convergence accuracy of system, showing observation window of 1 and 2 seconds significantly affecting the system's overall performance.

For the distributed environment case, where the system suffered from interference effect, this thesis presented the hybrid approach (hierarchical coordination, Sec. 5.2.3, Fig.5.15) as alternative. The hybrid approach relies on a set of assumptions to work, thus further investigation is required to enable accurate convergence in scenarios where those assumptions are not held. Furthermore, the limitations of the environment classification algorithm and the effects of the observation window size in the convergence accuracy require further investigation.

[RQ 3] How can an autonomous system coordinate multiple instances of emergent software to converge to optimal available global compositions in distributed settings?

The coordination strategy that had the best results in terms of convergence rate and accuracy was the hybrid approach, which is a coordination approach that is

between a completely distributed and completely centralised solution. This thesis explores three coordination approaches: a centralised, a completely distributed and a hybrid approach. The evaluation (Chapter 5) shows that the centralised learning always converges, but it is inefficient because it explores every single composition in every participating node, always converging in the number of available global compositions times the observation window size (i.e. `getConfigs()`$*w_t$) as described in Chapter 4 and evaluated in Sec. 5.2.2 from Chapter 5. The decentralise learning, however, converges faster (15 times faster considering the characteristics of the experiment demonstrated in Sec. 5.2.2 from Chapter 5), because the search space is divided among the system's nodes. The disadvantage of this approach is the learning interference form running multiple learning modules concurrently, which affects the system's convergence. Finally, the hierarchical approach (the hybrid approach) divides the search space by running multiple learning modules, and avoids learning interference by triggering learning level by level in the hierarchy. This approach was shown to be the most suitable in terms of convergence rate and learning accuracy, whilst minimising learning interference effects, being used to not only find optimal compositions but also to determine the node's role in the system, as demonstrated in Sec.5.2.3, particularly in Fig.5.15 from Chapter 5.

6.4 Contributions

The main contributions of the thesis are i) the proposal of the Emergent Software System concept overcoming significant human dependency in creating self-adaptive systems, and ii) the validation of the concept, showing that different software compositions have different performances under different operating conditions, and that emergent systems are able to autonomously converge towards optimal performance. The proposal of the Emergent Software System concept is the result of extensive practical experiments using real-world datacentre-based software, making it a concept deeply grounded in reality. In addition, this thesis demonstrates the potential of emergent systems approach by minimising human efforts in creating autonomous systems, through a complete automated creation and evolution of systems' adaptation logic. More specifically this thesis:

- Presented key challenges in the Emergent Software System problem space (Chapter 3), and addressed the most important challenges to realise emergent systems, which resulted in the following contributions:

    1. An online learning algorithm (Brute-force algorithm in Chapter 4) that uses runtime adaptation to exhaustively explore alternative system compositions in different operating conditions to move towards optimal solutions, with high convergence accuracy (see Chapter 5);
    2. An extension of the brute-force algorithm (Featured-based algorithm in Chapter 4) that addresses the rapidly growth in the search space of available software compositions. As a consequence, the feature-based approach is key to realise the Emergent Systems in distributed settings;
    3. An environment classification algorithm (described in Chapter 4) based on perception data ranges, handling distortion effects resulted from the collection of perception data through different software compositions;
    4. The External Reference approach and Hierarchical Coordination algorithm (described in Chapter 4) that support the extension of the local design-by-composition process to realise distributed emergent systems.

- Introduced a framework to orchestrate Emergent Software Systems. The construction of the framework yielded the following contributions:

    1. The PAL framework architecture (Chapter 4) itself which encapsulates the three main tasks to realise Emergent Systems. The tasks of Perception (for system monitoring), Assembly (for autonomous system composition) and Learning (for learning optimal available software compositions);
    2. The availability of the framework implementation for results replication. The framework implements the algorithms and concepts described in this thesis, and are available for download at [33], [34], [35] and [36].

- Argued for a paradigm shift in the autonomous software creation process, by changing the focus from autonomous software adaptation to **autonomous software composition**. The focus on autonomous software composition is

one of this thesis' key contributions. Software adaptation becomes a consequence of composition, rather than the main focus for creating autonomous solutions. This paradigm shift is at the core of Emergent Systems, and reduces the up-front effort and complexity involved in designing autonomous solutions, by pushing design decisions to be made at runtime by the system itself.

- Applied the concept of Emergent Systems to create a complete emergent web platform. The contribution is the creation of a web platform that does not require the classical process of optimising web platforms, which consists of: expert analysis of historical workload, manually profiling the web platform, creation of models to predict workload changes, and manual parametric tuning of the web platform. Instead, the emergent web platform continually self-composes and self-optimises based on real-time observed workload.

6.5 Future Work

The defined Emergent System concept to create autonomous systems contributes to the Autonomic Computing vision in minimising human involvement, mainly in the fine-grained software development, composition and optimisation tasks. To further advance this vision and consolidate the presented Emergent System paradigm, this section presents a list of avenues for future work.

Distributed Detection of Invalid Distributed compositions: The hierarchical coordination algorithm presented in this thesis is a centralised and offline solution to detect and avoid invalid compositions. Although this approach was shown to address the problem, further investigation is needed. A decentralised detection of invalid distributed compositions, for instance, would make the detection and avoidance of invalid compositions more scalable. Furthermore, the learning approach could benefit from information sharing, and consider problems derived from system's errors. A promising area to explore is the multi-agent system's concept of Distributed Constrained Optimisation (DCOP) [59]. These problems consist of finding optimal actions when constraints and variables are distributed among agents. Further investigations on DCOPs might provide new insights to create large-scale distributed emergent systems.

High-level Goals for Emergent Systems: Although the concept of Emergent System considers high-level goals to guide software optimisation, this thesis does not address any aspects of systems goals. Future research in this area is important, specially to define the format to express system goals and how these goals should be used to generate perception data (events and metrics) – which is currently manually defined. Relevant research with Domain-specific (Modelling) Languages (DSL and DSML) have been conducted to address the definition of high level systems goals [22, 56, 67]. The principles discussed in these papers could be complementary and applicable to further develop the emergent system concept.

Other Application Domains and Platforms: The application of the Emergent System concept to other application domains is important to further validate and expand the concept. A timely and important scenario is the creation of highly heterogeneous and volatile distributed infrastructures (e.g. systems running on both IoT and cloud platforms). The application of emergent systems to underpin such platforms increases their potential, leveraging the creation of future applications. Furthermore, emergent systems could be explored in critical systems for allowing software to handle the unexpected, e.g. in self-driving vehicles [83] or applications that assist and perform surgical procedures [80]. Finally, Emergent Systems could be integrated to the software development cycle as a platform to assemble software autonomously and test different architectural compositions at runtime. The knowledge generated by these systems can then be used by developers to further improve the system, making software an active member of its own development team.

Range-based Environment Classification Limitations: The environment classification algorithm presented handles important challenges in realising Emergent Systems such as perception data distortions, environment abstraction and multi-dimensionality. However, the presented approach was shown to not handle mid-exploration changes, and environments with overlapping ranges. Future research is required to address these limitation. A promising area to explore is Partially Observable Markov Decision Process (POMDPs) [65] to handle overlapping ranges, where the system is uncertain about which environment it is executing.

Mixed (Online / Offline) Learning Approach: The explored online learning approached consider a reactive strategy, making decisions assuming that the system will continue to behave as it recently did. A prediction strategy, on the other hand, might consider the general trend of the operating condition, making decisions assuming the direction the environment is taking, rather than the information it has just collected. Also, a pure offline learning approach that uses spare computing resources, might benefit application domains that are latency-sensitive. Exploring both approaches in parallel is an interesting perspective for future research.

Different Metrics and Multi-goal Emergents Systems: The definition of metrics in the Chapter 4 as a set of numeric values related to the systems' health opens possibilities to represent a plethora of system's characteristics. In this thesis, performance (in terms of response time) was explored, but other metrics as resource usage or system's unavailability can create different dimensions for the Learning module to exploit. Also, as contemporary systems are often multi-goal, a future research avenue would be to explore multi-goal (multi-metrics) emergent systems.

6.6 Final Remarks

The author argues that the presented concept of Emergent Software Systems is key to realise the Autonomic Computing vision in minimising human involvement in fine-grained system's management tasks. This thesis presented the definition of such concept and described the challenges of creating such systems. Furthermore, this thesis presented the first emergent framework applied to realised emergent data centre software. The results presented in this thesis demonstrate the feasibility and potential of the concept in both local and distributed systems, whilst also pointing areas for further investigation. Finally, this thesis has presented the very first fully functioning emergent web platform capable of self-assembling and self-optimising with no prior domain-specific information, reducing the up-front effort of optimising web platforms. The author invites the autonomic research community to further investigate the potential of emergent systems and to consolidate its concept.

Bibliography

[1] NASA web site trace: http://ita.ee.lbl.gov/html/contrib/nasa-http.html, 1995.

[2] D. Al-Jumeily, A. Hussain, and P. Fergus. Using adaptive neural networks to provide self-healing autonomic software. *International Journal of Space-Based and Situated Computing*, 5(3):129–140, 2015.

[3] S. Amarasinghe. Zettabricks: A language compiler and runtime system for anyscale computing. Technical report, Massachusetts Inst. of Technology (MIT), Cambridge, MA (United States), 2015.

[4] M. Amoui, M. Salehie, S. Mirarab, and L. Tahvildari. Adaptive action selection in autonomic software using reinforcement learning. In *Autonomic and Autonomous Systems, 2008. ICAS 2008. Fourth International Conference on*, pages 175–181. IEEE, 2008.

[5] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe. Siblingrivalry: online autotuning through local competitions. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 91–100. ACM, 2012.

[6] J. J. A. Ansel. *Autotuning programs with algorithmic choice*. PhD thesis, Massachusetts Institute of Technology, 2014.

[7] P. Arcaini, E. Riccobene, and P. Scandurra. Modeling and analyzing mape-k feedback loops for self-adaptation. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 13–23. IEEE Press, 2015.

[8] K. Baylov and A. Dimov. An overview of self-adaptive techniques for microservice architectures. *Serdica Journal of Computing*, 11(2):115–136, 2017.

[9] K. Baylov and A. Dimov. Reference architecture for self-adaptive microservice systems. In *International Symposium on Intelligent and Distributed Computing*, pages 297–303. Springer, 2017.

[10] K. L. Bellman and C. Landauer. Integration science: more than putting pieces together. In *Aerospace Conference Proc., 2000 IEEE*, volume 4, pages 397–409.

[11] K. L. Bellman and C. A. Landauer. Early work on the brain patch, a reflective service for system of systems integration. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pages 249–254. IEEE, 2015.

[12] G. Blair, Y.-D. Bromberg, G. Coulson, Y. Elkhatib, L. Réveillère, H. B. Ribeiro, E. Rivière, and F. Taïani. Holons: towards a systematic approach to composing systems of systems. In *Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware*, page 5. ACM, 2015.

[13] F. Brooks and H. Kugler. *No silver bullet*. April, 1987.

[14] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in java. In *Component-Based Software Engineering*, volume 3054, pages 7–22. Springer Berlin Heidelberg, 2004.

[15] B. Butzin, F. Golatowski, and D. Timmermann. Microservices approach for the internet of things. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, pages 1–6. IEEE, 2016.

[16] E. Cakar, S. Tomforde, and C. Müller-Schloer. A role-based imitation algorithm for the optimisation in dynamic fitness landscapes. In *Swarm Intelligence (SIS), 2011 IEEE Symposium on*, pages 1–8. IEEE, 2011.

[17] B. Chen, X. Peng, Y. Yu, B. Nuseibeh, and W. Zhao. Self-adaptation through incremental generative model transformations at runtime. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 676–687, New York, NY, USA, 2014. ACM.

[18] B. H. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, et al. Using models at runtime to address assurance for self-adaptive systems. In *Models@ run. time*, pages 101–136. Springer, 2014.

[19] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[20] D. Costa and A. Hertz. Ants can colour graphs. *Journal of the operational research society*, 48(3):295–305, 1997.

[21] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. on Comp. Systems*, 26(1):1:1–1:42, Mar. 2008.

[22] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. *A Language-Based Approach to Autonomic Computing*, pages 25–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[23] B. Debbabi, A. Diaconescu, and P. Lalanda. Controlling self-organising software applications with archetypes. In *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on*, pages 69–78. IEEE.

[24] A. Diaconescu, S. Frey, C. Müller-Schloer, J. Pitt, and S. Tomforde. Goal-oriented holonics for complex system (self-) integration: Concepts and case studies. In *Self-Adaptive and Self-Organizing Systems (SASO), 2016 IEEE 10th International Conference on*, pages 100–109. IEEE, 2016.

[25] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. OReilly, and S. Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices*, volume 50, pages 379–390. ACM, 2015.

[26] Y. Ding, H. Sun, and K. Hao. A bio-inspired emergent system for intelligent web service composition and management. *Knowledge-Based Systems*, 20(5):457–465, 2007.

[27] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[28] M. Dorigo, V. Maniezzo, A. Colorni, and V. Maniezzo. Positive feedback as a search strategy. 1991.

[29] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[30] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[31] A. E. Eiben, J. E. Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[32] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: A framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 7–16, New York, NY, USA, 2010. ACM.

[33] R. R. Filho. Source code from arm paper with instructions: http://research.projectdana.com/arm2016rodrigues.

[34] R. R. Filho. Source code from saso paper with instructions: http://research.projectdana.com/saso2016rodrigues.

[35] R. R. Filho. Source code for the icac demo paper with instructions: http://rex.projectdana.com/, 2017.

[36] R. R. Filho. Source code from acm taas paper with instructions: http://research.projectdana.com/taas2017rodrigues, 2017.

[37] R. R. Filho and B. Porter. Experiments with a machine-centric approach to realise distributed emergent software systems. In *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*, ARM 2016, pages 1:1–1:6, New York, NY, USA, 2016. ACM.

[38] R. R. Filho and B. Porter. Defining emergent software using continuous self-assembly, perception, and learning. *ACM Transactions Autonomic Adaptive Systems*, 12(3):16:1–16:25, Sept. 2017.

[39] A. Filieri, H. Hoffmann, and M. Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, pages 299–310. ACM, 2014.

[40] D. Fisch, M. Janicke, B. Sick, and C. Muller-Schloer. Quantitative emergence–a refined approach based on divergence measures. In *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*, pages 94–103. IEEE, 2010.

[41] S. Frey, A. Diaconescu, D. Menga, and I. Demeure. A generic holonic control architecture for heterogeneous multiscale and multiobjective smart microgrids. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(2):9, 2015.

[42] L. Gambardella and G. Taillard. A multiple ant colony system for vehicle routing problems with time windows.[in:] corne d., dorigo mp: New ideas in optimization, 1999.

[43] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, Jan. 2003.

[44] J.-P. Georgé and M. P. Gleizes. Experiments in emergent programming using self-organizing multi-agent systems. In *CEEMAS*, pages 450–459. Springer, 2005.

[45] K. Glocer, D. Eads, and J. Theiler. Online feature selection for pixel classification. In *Proceedings of the 22nd international conference on Machine learning*, pages 249–256. ACM, 2005.

[46] S. Götz, T. Kühn, C. Piechnick, G. Püschel, and U. Aßmann. A models@run.time approach for multi-objective self-optimizing software. In *Adaptive and Intelligent Systems*, pages 100–109. Springer, 2014.

[47] P. Grace, G. Coulson, G. Blair, B. Porter, and D. Hughes. Dynamic reconfiguration in sensor middleware. In *Proceedings of the international workshop on Middleware for sensor networks*, pages 1–6. ACM, 2006.

[48] P. Grace, D. Hughes, B. Porter, G. S. Blair, G. Coulson, and F. Taiani. Experiences with open overlays: A middleware approach to network heterogeneity. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 123–136, New York, NY, USA, 2008. ACM.

[49] S. Hassan and R. Bahsoon. Microservices and their design trade-offs: A self-adaptive roadmap. In *Services Computing (SCC), 2016 IEEE International Conference on*, pages 813–818. IEEE, 2016.

[50] M. C. Huebscher and J. A. McCann. A survey of autonomic computing – degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, Aug. 2008.

[51] K. Jeong and R. Figueiredo. Self-configuring software-defined overlay bypass for seamless inter-and intra-cloud virtual networking. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 153–164. ACM, 2016.

[52] A. Kaur and K. S. Mann. Component based software engineering. *International Journal of Computer Applications*, 2(1):105–108, 2010.

[53] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[54] D. Kim and S. Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*, pages 76–85. IEEE, 2009.

[55] O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In *Formal Aspects of Component Software*, pages 234–253. Springer, 2014.

[56] F. Křikava, P. Collet, and R. B. France. Actress: Domain-specific modeling of self-adaptive software architectures. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 391–398, New York, NY, USA, 2014. ACM.

[57] G. Liao, K. Datta, and T. L. Willke. Gunther: Search-based auto-tuning of mapreduce. In *Euro-Par 2013 Parallel Processing*, pages 406–419. Springer, 2013.

[58] J. C. Lima, R. C. Rocha, and F. M. Costa. An approach for qos-aware selection of shared services for multiple service choreographies. In *Service-Oriented System Engineering (SOSE), 2016 IEEE Symposium on*, pages 221–230. IEEE, 2016.

[59] J.-S. Liu and K. P. Sycara. Exploiting problem structure for distributed constraint optimization.

[60] C. Lu, Y. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, 2006.

[61] S. Malek, N. Esfahani, D. Menasce, J. Sousa, and H. Gomaa. Self-architecting software systems (sassy) from qos-annotated activity models. In *Principles of Engineering Service Oriented Systems, 2009. PESOS 2009. ICSE Workshop on*, pages 62–69, May 2009.

[62] P. K. McKinley, B. H. Cheng, A. J. Ramirez, and A. C. Jensen. Applying evolutionary computation to mitigate uncertainty in dynamically-adaptive, high-assurance middleware. *Journal of Internet Services and Applications*, 3(1):51–58, 2012.

[63] D. Merkle, M. Middendorf, and H. Schmeck. Ant colony optimization for resource-constrained project scheduling. *IEEE transactions on evolutionary computation*, 6(4):333–346, 2002.

[64] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, et al. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. In *International Workshop on Applied Parallel Computing*, pages 328–342. Springer, 2012.

[65] G. E. Monahan. State of the art – a survey of partially observable markov decision processes: theory, models, and algorithms. *Management Science*, 1982.

[66] F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[67] K. A. Morris, M. Allison, F. M. Costa, J. Wei, and P. J. Clarke. An adaptive middleware design to support the dynamic interpretation of domain-specific models. *Information and Software Technology*, 62:21–41, 2015.

[68] S. Niemann, C. Mueller-Schloer, and M. Pacher. Solving distributed dynamic optimization problems in self-optimizing systems by approximating the interaction between agents. In *Architecture of Computing Systems (ARCS), Proceedings of 2013 26th International Conference on*, pages 1–12. VDE, 2013.

[69] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 431–444. ACM, 2013.

[70] B. Porter. Runtime modularity in complex structures: A component model for fine grained runtime adaptation. In *Component-Based Software Engineering*, pages 26–32. ACM, June 2014.

[71] B. Porter and R. R. Filho. Losing control: The case for emergent software using autonomous perception, assembly and learning. In *Proc. of the 10th IEEE International Conf. on Self-Adaptive and Self-Organizing Systems*, 2016.

[72] T. Preisler, T. Dethlefs, and W. Renz. Structural adaptations of decentralized coordination processes in self-organizing systems. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*.

[73] A. J. Ramirez, A. C. Jensen, B. H. Cheng, and D. B. Knoester. Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 568–571. IEEE Computer Society, 2011.

[74] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck. Towards a generic observer/controller architecture for organic computing. *GI Jahrestagung (1)*, 93:112–119, 2006.

[75] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[76] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. on Autonomous and Adaptive Systems (TAAS)*, 2009.

[77] H. Schmeck. Organic computing-a new vision for distributed embedded systems. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 201–203. IEEE, 2005.

[78] H. Shoukourian, T. Wilde, D. Labrenz, and A. Bode. Using machine learning for data center cooling infrastructure efficiency prediction. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 954–963. IEEE, 2017.

[79] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. A Bradford book. Bradford Book, 1998.

[80] R. H. Taylor, A. Menciassi, G. Fichtinger, P. Fiorini, and P. Dario. Medical robotics and computer-integrated surgery. In *Springer handbook of robotics*, pages 1657–1684. Springer, 2016.

[81] S. Tomforde, S. Rudolph, K. Bellman, and R. Würtz. An organic computing perspective on self-improving system interweaving at runtime. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*, pages 276–284.

[82] S. Tomforde, B. Sick, and C. Müller-Schloer. Organic computing in the spotlight. *arXiv preprint arXiv:1701.08125*, 2017.

[83] M. M. Waldrop et al. No drivers required. *Nature*, 518(7537):20–20, 2015.

[84] Z. Wang, Z. Tian, J. Xu, R. K. Maeda, H. Li, P. Yang, Z. Wang, L. H. Duong, Z. Wang, and X. Chen. Modular reinforcement learning for self-adaptive energy efficiency optimization in multicore system. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 684–689. IEEE, 2017.

[85] E. Yuan, S. Malek, B. Schmerl, D. Garlan, and J. Gennari. Architecture-based self-protecting software systems. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, pages 33–42, 2013.

[86] W. Zhang, T. Wood, and J. Hwang. Netkv: Scalable, self-managing, load balancing as a network function. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*, pages 5–14. IEEE, 2016.

[87] W. Zheng, R. Bianchini, and T. D. Nguyen. Massconf: Automatic configuration tuning by leveraging user community information. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 283–288, New York, NY, USA, 2011. ACM.

[88] W. Zheng, R. Bianchini, and T. D. Nguyen. Massconf: automatic configuration tuning by leveraging user community information. In *ACM SIGSOFT Software Engineering Notes*, volume 36, pages 283–288. ACM, 2011.

APPENDIX A

Architectural Descriptions

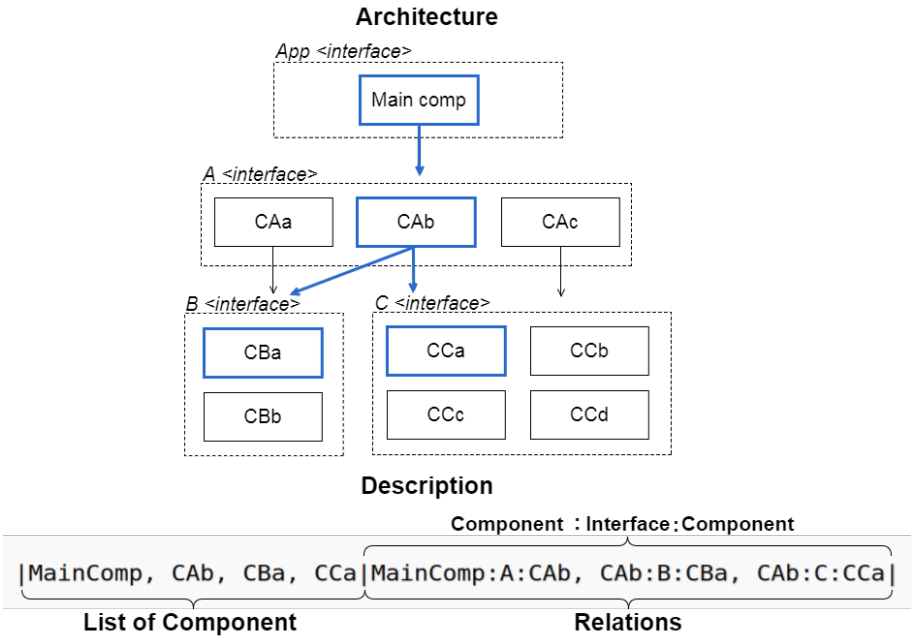

Figure A.1: Example of architectural description.

Here I introduce the architectural description notation, used by the PAL framework to uniquely describe architectural compositions. This description is used by the Learning module to interact with the Perception+Assembly modules when collecting a list of available compositions or requesting the Assembly module to change the software composition to another. An example of an architectural description is illustrated in Fig A.1. The image has a visual representation of an architectural composition on top of the image, and at the bottom there is a representation of the description of the same composition. The architectural description has two parts:

the first part is a list of the components that are part of the architecture, the second part is the relationship among the components. In this example, the relation determines that a component A is connecting to component B through an interface I, i.e. $A{:}I{:}B$. These are unique descriptions, meaning that there cannot exist the same description for two different architectures, since the description itself contains the necessary information to replicate the actual architecture.

APPENDIX B

Proxy Expression Language

The Proxy Expression Language (PEL) is a tool to precisely express where to place proxy components in the architectural composition. This language is used by the Learning module to ensure that the generated proxy components are kept in place, monitoring the intended interface even when the software compositions are constantly changing. The Perception module is responsible to interpret the expression and make a list of architectural compositions without proxy components associating these compositions with their equivalent with proxy components added to the right place (interface or component) according to the expression created by the Learning module. Therefore, whenever the Learning module requests the software composition to be changed, the Perception module compares the new architectural description with its list and decides: if the new composition matches one of the compositions in the list, the Perception module changes the software composition to the corresponding new composition with the proxy component in the right place as according to the expression. This strategy enables the Learning module to work with architectural descriptions with no proxy component added, the Perception module is responsible to apply the PEL expression provided by the Learning module once (before it starts the experimentation process) and to transparently add proxy components to the appropriate place in the software architectural composition.

The expression language explores the concept and format of the architectural description to determine what interface the proxy component must monitor. An example of PEL is illustrated in Fig. B.1. The image shows two architectures (1

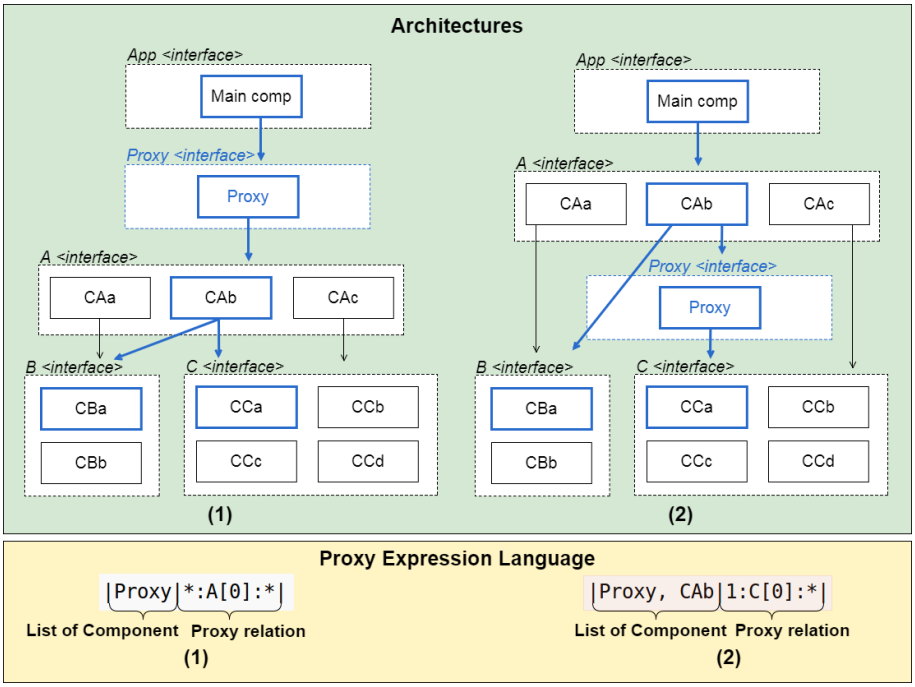

Figure B.1: Example of PEL expressions.

and 2) with the proxy component placed to monitor two different interfaces. The expressions are similar to composition descriptions having two parts. The first part of the expression is a list of components, the second part is the expression that informs the Perception module where to place the proxy component. The expression itself is also similar to the 'relation' in a composition description, having a the following format: $component{:}interface{:}component$. The difference in the expression is the addition of the element [0] after the interface name. This element indicates that the component in the first position in the list of components (first part of the expression) must be placed to monitor that specific interface and between the specific components named on the right and on the left of the expression. The expression allows the use of the special character $*$ to indicate *any* component, and numbers to indicate a specific component on the list of components specified in the first part of the expression. The first expression places the proxy on the interface A between components that require and provide A. Contrarily, in architecture 2, the expression places the proxy on interface C, specifically in the composition with component CAb requiring interface C connecting to any component providing C. Furthermore, the expression language also provides logic operators that enable the

composition of expressions to place different proxy components in different places in the architecture. The operators are: **or** (represented by the symbol ";") and **and** (represented by the symbol "&"). Examples of these kind of expressions are:

i) `|ProxyA, ProxyC,CAb|(*:A[0]:*) & (2:C[1]:*)|`

and

ii) `|ProxyB, ProxyC, CAa, CAb|(2:B[0]:*) ; (3:C[1]:*)|`

The first expression (i) places the proxy component $ProxyA$ on interface A, **and**, at the same time, whenever the component CAb is present in the architecture, the system also places the $ProxyC$ on interface C. Note that when the architecture composition has CAc instead of CAb, $ProxyC$ is not used. The second expression (ii), places $ProxyB$ on interface B in cases when component CAa is in the architecture , **or** in cases when component CAb is in the architecture, the system places $ProxyC$ on interface C.

APPENDIX C

Proxy Components

```
// Generated: HTTPProxy
component provides http.handler.GET.HTTPGET, monitoring.Monitoring requires http.handler.HTTPGET httpGET,
     monitoring.Container, metrics.ResponseTime, events.MimeType {

    /* standard code: never change */
    static Container monitor
    implementation Monitoring {
        Event[] Monitoring:getEvents() {
            if (monitor == null) { monitor = new Container() }
            return monitor.getEvents()
        }

        Metric[] Monitoring:getMetrics() {
            if (monitor == null) { monitor = new Container() }
            return monitor.getMetrics()
        }

        void Monitoring:turnMonitorOn() {
            if (monitor == null) { monitor = new Container() }
            monitor.turnMonitorOn()
        }

        void Monitoring:turnMonitorOff() {
            if (monitor == null) { monitor = new Container() }
            monitor.turnMonitorOff()
        }
    }

    implementation HTTPGET {
        void HTTPGET:handleRequest(HTTPMessage httpHeader) {
            /* standard code: never change */
            if (monitor == null) {
                monitor = new Container()
                monitor.turnMonitorOn()
            }
            /* using metrics and events components to collect metric and events */
            ResponseTime metric = new ResponseTime()
            MimeType event = new MimeType()
            metric.start()
            /* error as degraded performance */
            if (httpGET.handleRequest(httpHeader)) {
                metric.finish()
                monitor.addMetric(metric.getName(),metric.getResult(),metric.preferHigh())
            } else {
                monitor.addMetric(metric.getName(),INT_MAX,metric.preferHigh())
            }
            monitor.addEvent(event.getType(),event.getName(httpHeader),event.getValue(httpHeader))
        }
    }
}
```

Figure C.1: Example of a generated proxy component responsible to collect information from the *HTTPGET* interface, generating *ResponseTime* as metric and *MimeType* as event. This component was used to monitor the Web Platform.