Capítulo 1

Emergent Software Systems: Theory and Practice

Roberto Rodrigues Filho (UFG), Barry Porter (Lancaster University),
Fábio M. Costa (UFG) and Iwens Sene Júnior (UFG)

Abstract

Autonomic Computing and related research communities have drawn attention from researchers and industry practitioners who seek techniques and tools to build large-scale, reliable self-adaptive systems. However, building autonomic solutions remains a challenge: i) the upfront effort to develop such systems is very high, making them costly to implement; ii) only specialised parts of the system are made adaptive, limiting its flexibility in handling unknown operating conditions; and iii) state-of-the-art approaches still heavily rely on design-time predictions of operating conditions, making systems execution uncertain when predictions are wrong. To address these challenges, the concept of Emergent Software Systems has been proposed. The Emergent Software approach aims to reduce the upfront effort to create autonomic solutions, and it supports fully adaptive systems able to autonomously learn about the system's structure and operating environment with no predefined knowledge or predictions. This chapter aims to disseminate Emergent Software Systems, presenting the central concept and tools to realise the approach.

1.1. Introduction

Over the last two decades, systems researchers have acknowledged the increasing complexity in developing and managing software systems and the need for new and dynamic solutions [6, 24, 26, 47]. According to Blair [6], this complexity stems from two main reasons: i) the increasing heterogeneity of modern systems, and ii) the increasing volatility ever more present in the operating environment of such systems. In addition, systems have become larger, with millions of lines of code divided into software modules running over distributed infrastructures across the globe (e.g., YouTube), making them difficult to manage and evolve to accommodate new business and user demands.

This makes systems increasingly difficult to manage. In particular, the volatility of the operating environment resulting from the frequent changes that occur in the system is a crucial challenge to overcome. These changes happen across the entire system, and they impact different layers: infrastructure, platform and application. In the infrastructure, the

addition and removal of nodes and adjustments in a node's resource capabilities (e.g., due to costs or system demands) in modern elastic environments, such as the cloud, are very common. These changes in the infrastructure may impact the software modules running on the impacted nodes, which sometimes need to be tuned accordingly. For instance, nodes being removed from the infrastructure due to faults require changes in the software to cope with such situations (e.g., adopting fault-tolerance mechanisms). Likewise, an evolution of the software features may demand changes in the system's overall architecture to improve efficiency as well as in its infrastructure to provide more resources. For instance, the addition of video streaming as a feature in a web-based application may require an increase in network resources and computing resources from the infrastructure.

These are examples of common changes, and given the characteristics of current systems (large scale and heterogeneity), these situations are difficult to handle. Nevertheless, it is not hard to imagine that it is possible to anticipate such scenarios. That gives time for engineers to prepare the system for such changes and make sure that the proper mechanisms are in place when these changes occur. Nowadays, however, due to the scale these systems operate, these examples of changes are often accompanied by difficult-to-predict situations. For instance, content provided by a system may go viral, which suddenly increases user access, demanding more resources and, sometimes, a more optimised version of the system to better handle the new workload. In other cases, due to the complex interaction among the distributed modules of a system, a predefined fault-tolerance mechanism may be ineffective because it simply did not anticipate a series of successive failures throughout the system.

The above examples make evident that it is crucial to address the volatility (i.e., the frequent changes) of modern operating environments in contemporary systems. The illustrated situations demand fast and accurate decision making that may affect many different layers of the system. In the literature [26, 40, 47], the self-adaptive and autonomic computing communities have argued that pushing the responsibility of decision making to adapt systems when facing changes to the machine itself is the best way forward.

Self-adaptive and autonomic systems are software systems able to change their behaviour or structure to accommodate changes with minimum or no human interference in the process [10]. These systems require the definition of an adaptation logic, responsible for deciding how and when adaptation occurs, and an adaptation mechanism responsible for reifying changes to the system. As the volatility of modern systems increases, the demand for self-adaptive systems follows suit. Moreover, considering that the decisions on how adaptation should be performed must be fast and accurate, relying on engineers to manually analyse the current system status and design the new system behaviour to cope with changes is a slow and unsuitable process. Self-adaptation solutions, in turn, are a perfect fit to address the increasing volatility of modern operating environments. That is because they take over the responsibility to make decisions at runtime and quickly perform online adaptations with no human interference.

However, the development of such adaptive systems presents its own challenges. First, the upfront effort of creating such systems is high. This, in turn, makes the development of self-adaptive systems costly. The effort is put into capturing the adaptation logic

and ensuring that the system will not enter a malfunctioning state as a result of adaptation. Also, the adaptation mechanisms should encompass all adaptation possibilities, and support seamless runtime adaption with minimum impact on system's performance. Second, given that it is costly to create self-adaptive systems, a common strategy is to make only specialised parts of the system self-adaptive. This vastly limits the flexibility of the systems in handling changes, particularly the unexpected ones. Lastly, current approaches heavily rely on design-time prediction to build such systems. This inevitably leaves the system incapable of handling unforeseen/uncertain conditions and often underperforming when these predictions are inaccurate. As a response to that, the concept of Emergent Software Systems have been proposed and investigated in the literature [37, 42, 43, 44, 45].

Emergent Software Systems is an approach to facilitate the development of self-adaptive systems. It applies lightweight component-based models (e.g., [7, 11, 36]) in tandem with reinforcement learning [49] algorithms to support the creation of self-adaptive systems. The component-based models enable the composition of software from a collection of small (e.g., the size of a class in a Java program) and highly reusable components. These models also enable seamless runtime adaptation of the system by replacing a set of components with others. Considering that the system is entirely composed of these small components, the entire system is made to adapt. The reinforcement learning algorithm, in turn, enables the system to start with no predefined domain-specific knowledge and learn as it executes which software composition (defined by the set of small components) is the most suitable composition for the operating environment on which the system is currently operating. Thus, the reinforcement learning algorithm allows Emergent Software Systems to autonomously build their own understanding of the system structure and their operating environment, eliminating the need for predictions.

This chapter presents an approach that aims to facilitate the creation of self-adaptive systems, named Emergent Software Systems. The goal is to motivate the adoption of the Emergent Software System approach to build self-adaptive software able to cope with the demands of contemporary systems. To achieve this, we introduce the main concepts of self-adaptive and autonomic computing systems, as well as the state-of-the-art in the field, along with the shortcomings of current approaches (Sec. 1.2). Then, based on the flaws of current approaches, we motivate and introduce the concept of Emergent Software Systems (Sec. 1.3). We introduce the concept along with the challenges involved in realising it. Next, we introduce our implementation of the concept and a framework that supports the creation of Emergent Software Systems (Sec. 1.5). We discuss the application of the concept to build microservices able to evolve their internal architectural composition and to cope with the dynamism of current operating environments (Sec. 1.6). We make available a collection of practical exercises and code examples for the reader to gain hands-on experience with the development of Emergent Software Systems. Finally, we conclude the chapter by inviting researchers and industry practitioners to join the effort to further develop the concept and contribute to create self-adaptive solutions that are able to cope with the demands of contemporary systems (Sec.1.7).

1.2. Related Work and Background

This section introduces the main concept in self-adaptive, autonomic computing systems. It also presents some of the most relevant and seminal papers in the area. We discuss the current state-of-the-art techniques used to implement self-adaptive systems. We also focus on reinforcement learning algorithms to support runtime learning of the self-adaptive systems adaptation logic. This focus on the learning algorithms is essential for understanding our proposed Emergent Software Systems approach. We conclude the section by presenting the shortcomings of current self-adaptive techniques.

1.2.1. Autonomic Computing

Motivated by the increasing complexity in contemporary systems, IBM introduced the concept of Autonomic Computing [26] in a manifesto in 2001. Their motivation were their vision of future systems being characterised by their large size, their interconnectivity with other large systems, the heterogeneity level and the constant demands to cope with changes. These type of systems require skilled software engineers to install, configure, tune and maintain them. In [26], Kephart and Chess argue that contemporary systems were reaching the limit of the human capacity to manage systems adequately and to timely react to unexpected events. 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 with minimum human interference.

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). These systems aim to autonomously maintain specific properties. These are known as self-* properties. These properties are dimensions of autonomous behaviour incorporated in a system to address a particular system aspect. Autonomous systems may implement the following self-* properties: self-protecting, self-optimising, self-healing and self-configuring.

Self-protecting systems (e.g., [52]) are capable of identifying possible security threats and operating risks for the system well functioning. Furthermore, these systems can adequately handle and prevent malicious users from exploiting possible security threats by, for example, autonomously changing their internal structure. Self-optimising systems (e.g., [22]) are systems that can change their structure to improve some aspects of their performance autonomously, for example, by reducing the systems response time. Self-configuring systems (e.g., [25]) can 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., [1]) is capable of maintaining 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 seminal paper 'The Vision of Autonomic Computing' [26]. 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 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 the global system's goals, representing the interface between humans and autonomous systems. These challenges are still in debate in the related research communities with promising solutions and future directions.

1.2.2. Self-adaptive Systems/Self-organising Systems

The term 'self-adaptive systems' is 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 their 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 for capturing 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; model-driven approaches, which apply models representing system properties, QoS traits, system architectures, system goals and equations that support 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 for propagating changes in the existing software. These mechanisms are classified either as parametric or architectural approaches. Parametric approaches consider the system as a black box, having only dial buttons to influence the system behaviour. This approach is limited compared to architectural-based approaches 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., using a load balancer architecture to maintain the performance or 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, it is essential to create mechanisms to ensure that the system achieves its global goals whilst maintaining non-functional requirements. This is imperative, particularly in situations where multiple autonomous single-goal entities make their own adaptation decisions and interact with each other. In this context, the **coordination of software adaptation** is critical 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 to make decisions to

converge the system towards the global desired behaviour. Some approaches use voting schemes, action predictions, and other consensus schemes.

Typical activities of a self-adaptive and self-organising system involve monitoring the system execution, analysing the monitored data, deciding on and executing the course of action the system needs to maintain its desired properties. The feedback loop conceptual framework was widely adopted to implement self-adaptive systems 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 [3], which stands for Monitoring, Analysing, Planning, Executing. The framework actions are guided by the knowledge (the K in the MAPE-K) defined by experts at the design phase. This knowledge is also expanded by the results of the previous system's actions. In today's autonomous systems, the activities defined in the MAPE-K feedback loop are imperative to realise self-adaptive systems. Even if the classic MAPE-K feedback loop is not applied, some form of it is always used to support reliable adaptation.

1.2.3. Organic Computing

The Organic Computing research initiative started in Germany with the focus to address and explore the self-organisation concept in technical software systems, inspired by neuroscience and molecular biology principles and software engineering [35, 47, 51]. Originated around the same time as the Autonomic Computing paradigm, the Organic Computing initiative also focused on the problem of having multiple instances of interacting autonomous systems, which may lead to conflicts and undesired emergent behaviour affecting the resulting system [47].

In detail, autonomous systems with multiple goals might spontaneously interact with other autonomous systems to achieve and maintain global systems goals. The interaction among autonomous system instances is not a far-fetched futuristic idea. A concrete example given in [47] draws attention to the modern cars and the multiple interacting devices that are required to support the basic cars functions. In this example, devices require data from other devices to provide their functionality, and the orchestration of such devices results in the car and all its available functions. Problems with the interactions among these systems, such as delaying the delivery of information to devices, may lead to miscalculations and compromise service execution. The problems with orchestration and synchronisation that may occur in such scenarios are not the only problem. Another fundamental 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 [41, 33] as well as to address further aspects (self-* properties) of autonomous systems [8, 19].

1.2.4. Reinforcement Learning (RL) for Adaptive Systems

Reinforcement Learning is a learning paradigm in which an agent learns to correlate a set of actions to reward values [49, 46]. An agent, in this context, can be defined as software

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 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 [2, 27].

Several papers on self-adaptive systems that apply reinforcement learning algorithms in the adaptation process, such as [2], or any machine learning algorithm for that matter [54, 30, 16], 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 can only adapt within a predefined set of values/actions, limiting the capacity of reinforcement learning algorithms to learn beyond the actions that experts established in the design phase.

This limitation of parametric tuning approaches also affects some architectural-based approaches such a [27, 32, 18], because, 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 [27], Kim and Park demonstrate the application of Q-learning, a well-known and widely used reinforcement learning algorithm, to architectural change robots in a 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 to apply Q-learning to self-adaptive systems. The paper ends with evaluating the application of reinforcement learning to robot simulations, showing that a robot that applies such an 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 states for the machine learning algorithm. As papers such as [2, 27] 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, limiting the potential and generalisation of reinforcement learning approaches in self-adaptive systems.

We argue 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 models and reinforcement learning algorithms.

1.2.5. Critique of Existing Approaches

Although the main concepts and research areas that focus on self-adaptive software solutions have been introduced, we have not pinpointed the shortcomings of the most relevant work that describes the different existing techniques to create self-adaptive solutions. In this section, we present these studies, dividing them into different categories. The self-adaptive systems research area is vast and multi-disciplinary, having multiple research communities proposing different solutions with different terminologies. To address this issue and present a more complete picture of the state-of-the-art of such techniques, we attempt to group these relevant works based on the similarities of their proposed approach. As a result, we created the following categories: i) rule-based and policy-driven approaches, ii) mathematical models, and iii) bio-inspired approaches. Note that these categories do not contain all relevant approaches. For a better attempt at a complete picture of the different approaches, refer to [42].

Rule-based and policy-driven approaches implement the simplest and most straightforward method to represent the systems adaptation logic. Due to their 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 into 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 [23], and other approaches, such as in [28], the software engineers describe adaptation rules in terms of logic expressions, which allow a small level of reasoning in the adaptation process.

These policy-driven approaches are very effective when all system states are well known before and the deployment environment is fairly static. For that reason, these approaches were used in early examples of self-adaptive solutions. Due to the **high levels of dynamism** 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. Suppose the system suffers from fluctuations in the workload or unexpected failures in the underlying structure. In that case, these systems cannot react accordingly and accommodate the changes on time, leading the system to a malfunctioning state or degraded performance.

Mathematical models consists of approaches that capture their adaptation logic in the form of mathematical functions. Sometimes differential equations are used to establish a relation between input and output. This input-output mapping is used to capture the systems state and connect it to a particular action that changes the systems state and therefore controls the adaptation process. An example of such approach is described in [31]. Lu *et al.* present mathematical models that guide the assignment of classes of HTTP requests to threads responsible for handling these requests in a web server. The goal is to enable an adaptive assignment of requests to threads to guarantee delays below a predefined value as workload characteristic changes. Although this approach supports a dynamic request-to-thread assignment according to workload variation, its adaptation

logic is determined by a predefined set of fix equations, which are highly complex to elaborate and does not evolve to accommodate unexpected events.

A slightly different and better approach to this is shown in [18]. Elkhodary *et al.* propose a learning approach to generate mathematical functions that correlate the presence of software features in the system with the satisfaction levels of the system's 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 the system's deployment, as it is exposed to new operating conditions, it is capable of fine-tuning the functions coefficients to incorporate new operating conditions. Although this approach reduces the complexity of defining equations to guide adaptation by generating the model autonomously, it only tunes the model to accommodate new conditions. If new software modules are added, or completely different conditions are encountered, the system will require offline re-training.

Bio-inspired approaches encode behaviours found in nature into 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 [15]. A great variety of papers explored the application of such methods in a variety of problems such as vehicle routing [20], graph colouring [12] and project scheduling [34]. Another popular example of this approach is evolutionary computing [17]. 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 are used to directly provide systems with self-adaptive capabilities. For example, Ding *et al.* [14] describe an algorithm that encodes the behaviour of Neuroendocrine Immune System to support autonomous composition and adaptation of web services. Another example is described in [40], where Ramirez *et al.* used evolutionary computing to predict operating conditions the system will be facing and how the diverse available software configurations would best suit different conditions maintaining desired QoS levels. These approaches are complementary to Emergent Software Systems, as they can be used in online learning to find optimal architectural compositions.

1.3. The Concept of Emergent Software Systems

This section introduces the concept of Emergent Software Systems and discusses the main challenges of realising the approach. First, we motivate the need for Emergent Software Systems, then we introduce and described the concept in detail. We conclude the section by presenting the main challenges inherent in the concept regardless of current technology's limitations. Note that we do not discuss the implementation of the concept in this section. That is to allow the reader to focus on the ideas behind the concept rather than implementation details. We present an implementation of the concept in Sec. 1.5.

1.3.1. Motivation

The main motivation for considering the advancement of self-adaptive solutions is the increasing dynamism of current operating environments. As previously described, this dynamism requires systems capable to react fast and accurately to new and often unexpected operating conditions. This scenario makes the advancement of self-adaptive systems crucial when considering the development and management of future systems.

An analysis of the start-of-the-art approaches to realise self-adaptive systems (Sec. 1.2) shows that current approaches share one or more of the following characteristics that prevent the advancement of self-adaptation systems: i) significant human-dependency in their design; ii) inability to autonomously evolve their adaptation logic in face of unexpected conditions; and iii) adaptation mechanisms are only applied to very specific parts of the systems. The Emergent Software System concept is introduced to enable the creation of self-adaptive system that addresses these three main concerns.

As previously described, the realisation of self-adaptive solutions requires the development of an adaptation logic, adaptation mechanisms and often a coordination strategy (in distributed contexts) to ensure seamless runtime adaptation. Analysing the most relevant work in this domain with regards to adaptation logic creation and evolution, a research gap was identified [42]. There is a 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 [10]). Furthermore, the creation of systems able to autonomously develop their adaptation logic would reduce the engineering efforts of developing autonomous solutions, facilitating the development of self-adaptive systems.

Considering the adaptation logic, the state-of-the-art approaches implement different techniques including policy-driven (e.g., Grace, *et al.* [23]), system representative models (e.g., SASSY [32]) and mathematical models (e.g., Netkv [53]). Most 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 the operating environment (e.g., FUSION [18]). This approach, however, can not evolve its adaptation logic if new software is added, requiring a re-execution of its offline training approach to generate the new adaptation logic. Finally, Georgé *et al.* [21], 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 approach is not able to 'remember' optimal solutions for previously seen situations, having to search for solutions even when previously seen conditions occur.

Therefore, the concept of Emergent Software Systems is a response to the identified gap in the literature considering the development of self-adaptive systems. The concept facilitates the development of self-adaptive systems because it does not require predictions or domain-specific knowledge. The resulting system is entirely adaptive with no extra effort than writing code (e.g., matching code to a feature model). Lastly, the emergent system can learn and evolve its adaptation logic as it executes and new conditions arise.

1.3.2. Definition

Emergent Software Systems is an approach to facilitate the development of self-adaptive systems. In detail, this novel concept consists of autonomously composing systems from small reusable software components, according to metrics and events (numbers and labels that represent the 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 user-desired goals. Next we present a definition of Emergent Software System as it was first introduced in [42].

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 the form of interfaces that 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 process of connecting software units is the result of software architectural composition, forming a single instance of software in a single node.

The presence of variations of software units enables a variety of architectural compositions. This happens locally, in individual machines, but the same process can also be used to compose distributed systems with emergent software running on a distributed infrastructure. In this distributed setting, each resulting software assembly influences how an executing software on a given node interacts with others. The interaction of multiple emergent systems running on distinct nodes constitutes 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 allows the resulting system to be assembled to achieve G in different ways, enabling the system to maintain the 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 the satisfaction levels of G. On the other hand, the events 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 essential to enable a fair comparison among the metrics of multiple existing compositions, ensuring that the perception of the system health of a given composition is compared to that of another composition in a fair way, given that both were exposed to the same operating conditions.

Problem Statement: The problem that emergent systems have to solve is to learn 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 com-

position by analysing metrics within the same operating condition, 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.

To conclude, the Emergent Software System paradigm differentiates itself from other approaches to self-adaptive systems by incorporating an online learning approach to compose software systems. This strategy is a step beyond current approaches. It 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 increasing 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.

1.3.3. Emergent Software Systems Challenges

The definition of the Emergent Software Systems concept imposes some challenges when implementing it. This section discusses the main challenges and their impact on the realisation of Emergent Software Systems and their adoption to create large-scale self-adaptive systems. These challenges were first introduced in [38].

Combinatorial Explosion: The Emergent Software vision is fundamentally based on a combinatorial learning process. These systems are built from 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 results from combining a set of individual software architectures. Considering that adding one component to the repository exponentially increases the number of possible architectures, this property has a cascading effect in the distributed scenario when considering the amount of participating nodes in the system. The global system architecture is the overall combination of the micro-architectures of all participating nodes. To illustrate how quickly 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 ten valid compositions running on another node. The load balancer is responsible for forwarding requests to both web servers. This system has 25,000 valid compositions ($50 * 50 * 10$). If we add another web server node with 50 compositions, the number of global compositions increases to 1,250,000 ($50 * 50 * 50 * 10$). This is an inherent property of emergent systems and a fundamental problem to be addressed in order to show the feasibility of the emergent systems paradigm and guarantee its consolidation.

Coordination: Emergent systems are responsible for i) composing the system using software units, ii) collecting perception data generated by the components at runtime and

iii) learning about the collected information. In a distributed setting, these systems can operate in different locations with different "personalities". The system can operate by controlling arbitrary groups of nodes in the system. They can control the operation of either individual nodes, in a completely decentralised fashion, or groups of any number of nodes, up to all system nodes, thus providing degrees of centralised control. Additionally, emergent systems can adopt different personalities, for example, by acting in an entirely selfish manner according to a group's local interests and ignoring the rest of the system, or by acting in an altruistic way, making local decisions to benefit global system's interests.

Behavioural Mismatch: In a distributed systems scenario, systems hosted in different nodes might implement behaviours that do not match other interacting nodes. For example, one system might encode different schemes of encryption algorithms, making the nodes incompatible with each other. A trivial solution for this problem is to manually define rules or impose constraints regarding aspects of the system behaviour. As discussed before, such a solution would be 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.

Relative Fitness Landscape: In emergent systems, fitness landscapes 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 architectural composition with the highest fitness value, meaning that it is the optimal available architectural composition for the observed operating environment. Other valid architectural compositions are formed with lower fitness values as components are removed or replaced from the architecture with the highest fitness value. In the context of an emergent software system, 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 choice of a new composition as the most suitable one when the environment changes. This scenario characterises emergent software systems as having **dynamic fitness landscapes**. In [8], Cakar *et al.* define two types of dynamic fitness landscapes, one influenced by external stimuli and the other influenced by the system's constituent components. Cakar *et al.* refer to changes in the fitness landscape caused by the system's internal elements as **self-referential fitness landscapes**.

Emergent software system's fitness landscapes suffer from external stimuli and 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 crucial challenge to realise emergent systems.

Monitoring Data Quality: 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 in turn trigger unnecessary adaptation or the selection of suboptimal architectures. In addition, as a related challenge, the absence of 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 essential aspects of the environment that impact 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 directly impacts the emergent system's ability to locate an 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.

1.4. Successful Projects

As a result of realising the Emergent Software System concept, we built a number of real-world systems to evaluate our approach. In this section, we provide an overall description of the three main projects developed to this date. These projects are open-source and available for download. All instructions to replicate the results are also provided with the code. We also provide references to previous publications where we detail the design, implementation and evaluation of each of these systems, and analyse how they realise the Emergent Software Systems concepts. The tools used to develop these projects are described in Sec. 1.5.

1.4.1. Emergent Web Server

The Emergent Web Server is a fully functional Web server that implements the HTTP 1.0 protocol and appropriately handles HTTP requests. The Web server was entirely developed according to the concept of Emergent Software Systems. It can adapt and evolve its internal composition to better handle requests according to the observed workload characteristics. Its design and implementation are fully described in [37, 38, 45], where it is used as a case study to explore and evaluate the concept. The emergent Web server is also presented in a publicly available video[1] that demonstrates its execution and performance.

The emergent Web server was initially explored with 42 unique ways to compose its internal architecture and was exposed to various workload patterns. The results show that the emergent Web server can converge towards the appropriate composition in a timely manner and learn the best performing composition for different classified workload patterns, being able to recompose its internal architecture to the best performing composition as soon as a previously seen workload pattern is recognised.

Later in this chapter (Sec. 1.5.4.1), the emergent Web server will be used as a target system for the reader to experiment and practice with the concept and tools to create Emergent Software Systems.

[1]To access the video, please visit: https://youtu.be/BBNvbC6w_3Y

1.4.2. Distributed Emergent Web Server

The Distributed Emergent Web Server is an extension of the emergent Web server. It also consists of a fully functional Web server that implements the HTTP 1.0 protocol and can serve content on the Web. The difference between this project and the emergent Web server is that it can delineate parts of its architecture and relocate them to other machines in a distributed infrastructure.

This project was used to explore learning in a distributed setting. The description of the project and the results are described and presented in [39]. The paper also presents the open-source code of the project to enable replication of its results. Besides the learning approach evaluated in the context of a distributed emergent Web server, this project also explores the potential of abstracting distributed compositions as actions to adapt the Web server into different architectural compositions to be learned at runtime.

The distributed emergent Web server can learn, at runtime, and with no predefined domain-specific knowledge, which Web server composition is the most suitable for the observed operating environment. Besides that, the distributed emergent Web server can also learn whether or not it requires running on multiple machines to handle the incoming workload volume.

1.4.3. Emergent Microservices

The Emergent Microservices project introduces a novel approach for the implementation of microservices using the concept of Emergent Software Systems. The project describes the use of the approach to create a fully functional real-world microservice that can appropriately handle incoming requests in a self-adaptive fashion.

The creation of Emergent Microservices aims at enabling microservices to adapt and evolve their internal architectural composition to autonomously cope with workload volume changes and workload pattern changes. The emergent microservice can evolve its internal architectural composition to better handle incoming requests at runtime, with no human interference.

The results of implementing and evaluating the approach are described in [43]. Rodrigues Filho *et al.* show that an emergent microservice can detect and adequately adjust its composition to accommodate changes in the operating environment. The paper also discusses the roles of engineers and machines in the design and evolution of future microservice-based systems. A more detailed description of the emergent microservice is presented later in this chapter (Sec. 1.5). We also use this project as a practical exercise to familiarise the reader with the Emergent Microservices concept and the use of Emergent Software Systems tools to build autonomic systems.

1.5. Perception, Assembly, Learning (PAL) Framework

This section discusses the realisation of the Emergent Software Systems concept. We present a framework that facilitates and supports the creation of emergent systems following the definition given in Sec. 1.3. This framework was introduced in [42].

Emergent Software Systems can autonomously compose software architectures from small and reusable software components according to the observed operating envi-

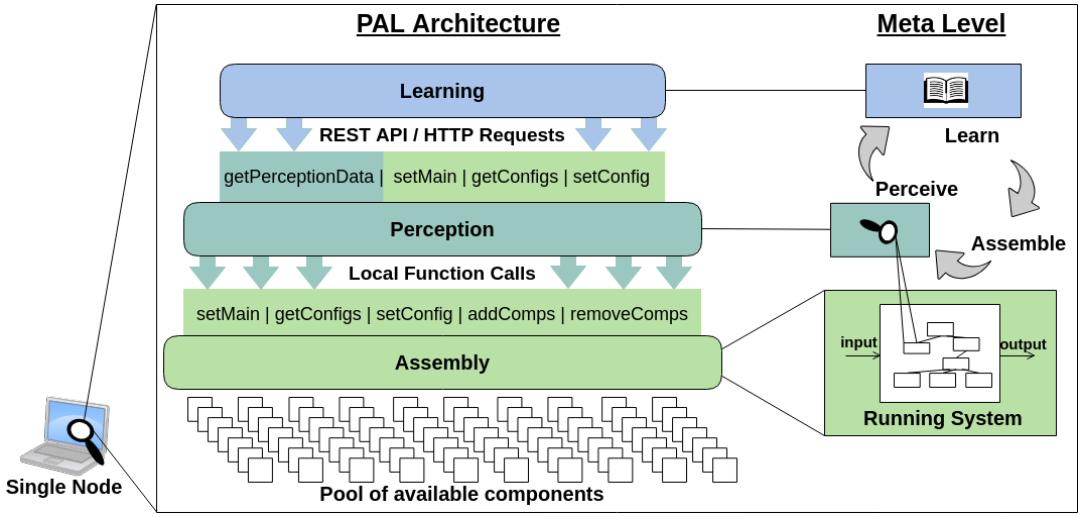

Figure 1.1. The PAL framework architecture [42].

ronment. To realise this concept, we built a framework with three main modules: **Perception**, **Assembly** and **Learning** (Fig 1.1).

The **Assembly** module searches for components in a repository, creates an in-memory representation of all available architectural compositions the system can be assembled into, and supports seamless adaptation from one composition to another at runtime. The **Perception** module, in turn, generates and adds proxy components to the system's architecture to monitor the system's health status and operating environment. Finally, the **Learning** module leads the entire autonomous software composition process through reinforcement learning approaches.

The interaction among the three modules is depicted in Fig. 1.1. The framework is organised in a multi-tier architecture in which the upper layers use the functions of the lower-level layers to implement their own functions and provide higher-level functionalities. The multi-tier modular architecture ensures that each participating module implements a well-defined set of functionalities to the upper level. Although the main modules are designed to be generic and useful in any application domain, this modular structure allows each module to be replaced by a variant, when applicable, to explore the specificities of particular domains. This characteristic makes the framework flexible, facilitating, for example, the experimentation and comparison of a variety of learning algorithms in the Learning module.

The three modules have well-defined roles in the realisation of the Emergent Software System concept. Their interaction results in a feedback loop that keeps the system constantly observing its health status and operating conditions and learning the optimal available system's composition for the observed environment. The following sections focus on the implementation details and the functionalities provided by the PAL framework.

1.5.1. Assembly Module

This section provides details on the Assembly module, covering the component-based model used to enable the concept of Emergent Software Systems and the functionalities

provided to the upper layers. The definition of Emergent Software Systems is based on the composition of fully functional software architectures from small components. To assemble components into functioning local architectures, the Assembly module searches for components in the repository, gathers information about them (and their variants) and assembles them into a fully functional system with composition options. The Assembly module uses functions provided by a component-based runtime to execute component replacement and composition at run time whilst abstracting the processes that support software architecture composition and (re)composition used by the Learning module to realise the reinforcement learning process. The following sections present the API functions provided by the Assembly module and describe the role of a component-based model in the autonomous composition process enabling architectural adaptation by component replacement.

Component-based models are essential to support the concept of Emergent Software Systems, 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 defined when coding components, the component-based model eliminates the need to use extra models to label the system code and define which 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 [18]) are often used to represent how pieces of code interact with each other, and how they could be replaced to support adaptation. Such feature models force developers to detail the relationship between different parts of the system after implementing the system itself, increasing development effort. On the other hand, the component-based model is a way to eliminate that need, integrating essential information for component adaptation directly in the component's code. This characteristic reduces development effort during system design, facilitating software adaptation throughout the entire system rather than limiting it to specific parts that were previously modelled.

This work uses the Dana[2] component model described in [36]. Dana is a general-purpose programming language that inherently provides a component-based model and runtime support for component adaptation in fine-grained complex modular structures. Component-based models require the definition of interfaces and components. Interfaces define function signatures (i.e., function names, return types and the list of parameters with their respective data types). Each interface expects *at least one* component that implements all functions defined in it. Components can *provide* implementation for multiple interfaces and may *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 it is by means of replacing those variants in an executing software that the adaptation process occurs. This "provides-requires" policy is a central part of a component-based model, allowing the language

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

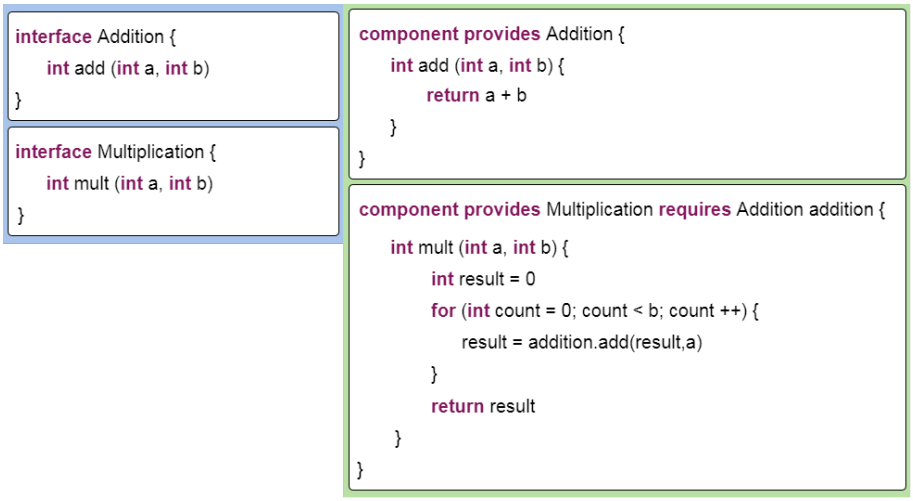

Figure 1.2. Example of interfaces and components. Note that Multiplication requires an external interface (Addition) to complete its implementation. The code is written using the Dana programming language syntax [42].

runtime to connect different components to create fully functional software. Software evolution or adaptation is realised by replacing components in the resulting software architecture at runtime without interrupting the system's services. Moreover, as part of the online replacement of a component, the system needs to examine the component's dependencies (i.e., its "requires" declaration) and perform further changes in its architecture to add any components that are required to support the new one.

An example of the Dana language syntax and its component-based model is show in Fig. 1.2. The figure shows two interfaces: *Addition* and *Multiplication*. Each interface may comprise a number of function definitions, where each function is defined by its name, return type, and a list of parameters with their types. The figure also shows two components that provide the implementation for the two interfaces. 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 the use of these interfaces and components to create a **Calculator** program, if multiple components provide the *Addition* interface, then any of those components can be connected to the *Multiplication* component to make it work. 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 way. Therefore, in this example, there exists a variety of architectural options to realise the **Calculator** program, each option implementing the operation *Multiplication* differently, 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 abstractions provided by the language support, through single function calls, the realisation of software adaptation or composition, involving: `loading` new components into memory, `pausing` an old component's ex-

ecution, `transferring` component states (when applicable – i.e. *replacing* statefull components) and `connecting` a new component with its required components. Dana transparently executes the tasks mentioned above without further actions from the developer. The Assembly module supports these functions with low-level operations to determine and control the composition and adaptation process at runtime. A list of the functions provided by the Assembly module is detailed below.

The Assembly Module API:

- `bool setMain(char compName[])`: This function is responsible for starting the assembly 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.
- `String[] getConfigs()`: This function returns a list of the available architectural descriptions. These descriptions are string-based 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 currently executing software composition.
- `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 future 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 the required interfaces. For each required interface, the Assembly module searches the repository for components that provide it. Suppose the Assembly module locates multiple components providing the same interface. In that case, it gathers the component variants (i.e., the multiple components providing the same interface). It attaches them to the interface creating an adaptation point, i.e., a point

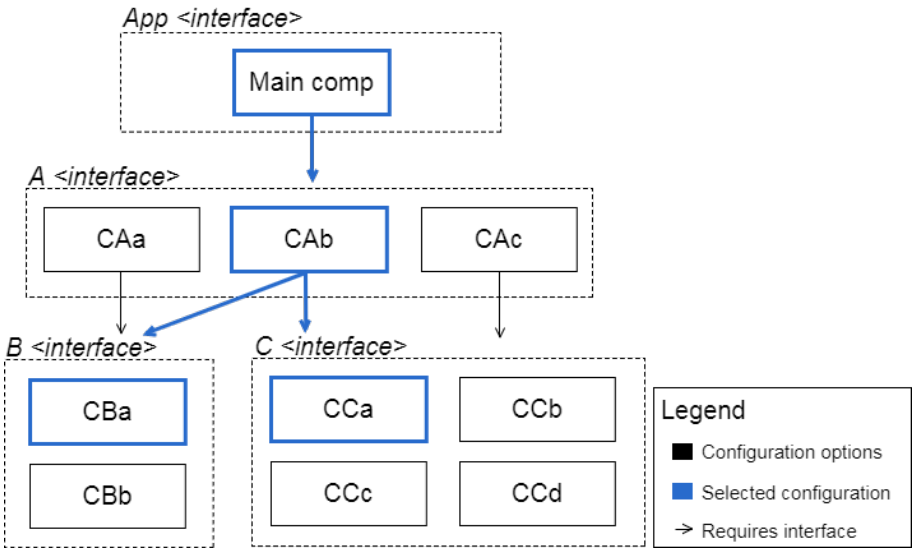

Figure 1.3. An example of a generic architecture represented by the Assembly module. This results from executing the Assembly module function *setMain* [42].

where there are multiple components to choose from. These adaptation points are represented in a tree, as shown in Fig. 1.3 for hypothetical 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. It continues with loading components and their variants for the required interfaces until the entire architectural tree is complete 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 in the selection of one component in each required interface on the tree (an example is shown in Fig. 1.3). The components marked in blue in the figure are part of the currently executing composition. After loading components and their variants, the Assembly module selects and runs a random architectural composition. The example in Fig. 1.3 presents a total of 14 available architectural compositions. As show above, the Assembly also module provides functions to add new components (which were not in the repository when `setMain` was first executed) and 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 compositions. Furthermore, the Assembly module provides functions to return a list of architectural descriptions of the available compositions, a function to return the description of the currently executing composition, and a function to change from one composition to another. These functions work by acting on information about the available components, as structured in a tree (as exemplified in Fig. 1.3) and stored in the Assembly module.

The Assembly module provides architectural descriptions to enable external modules to reason about the running software structure and change the running composition. This text-based description provides information about the components that make up a specific composition, along with their relationships, i.e., how the components are connected. The architectural descriptions are helpful to the PAL framework for two main

reasons: i) they enable external modules to interact with the architecture variations built by the Assembly module; and ii) they enable external modules to infer and reason about information on the architectures. Thus, architectural descriptions facilitate the interactions between the other framework modules (mainly Perception and Learning) and the Assembly module. Since a description contains all the details to reassemble any architectural composition, they enable, for example, the Learning module to reason about the available architectural compositions to understand the impact of specific components in the architecture. Furthermore, the descriptions support the creation of different implementation versions of the Assembly module, in case a domain-specific version of the Assembly module may perform faster in special scenarios, enabling these different versions to internally represent compositions in different data structures whilst maintaining a consistent format for referencing valid software compositions.

An example of the architectural description is presented bellow:

$$|MainComp, CAb, CBa, CCa|MainComp : A : CAb, CAb : B : CBa, CAb : C : CCa|$$

This is the architectural description of the executing composition illustrated in Fig. 1.3. The description has two main parts: the list of components and the components relationships. On the left-hand side of the description, there is a list of all participating components in the architecture. In the example, the list of components is $MainComp, CAa, CBA, CCa$. The component relationships part of the description (on the right-hand side) describes how the participating components are connected. The first relationship, $MainComp : A : CAb$, tells that $MainComp$ is connected to CAb through the interface A.

1.5.2. 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 provides, through a RESTful API, both its own functions and the Assembly module functions to external modules interacting with Perception. This facilitates access to functions that support the learning process, such as functions to change to a new software composition, to get a list of possible software compositions, to add new components or remove components, and to get the executing composition description (i.e., Assembly functions), as well as functions to allow insertion, removal of monitoring components (proxies) to extract monitoring information from the executing system (i.e., Perception functions). This section is divided into two subsections: the first one describes the perception data (events and metrics), which represent the system's health status and classify operating environments. The second subsection describes a particular type of component (proxies) that are autonomously generated and inserted into the software architecture to collect metrics and events from executing compositions.

1.5.2.1. Perception Data (Events and Metrics)

The perception data (Events and Metrics) are data types through which the system represents its health status and operating conditions, which is essential information for the

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

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

Figure 1.4. Metric and Event data types in the Dana programming language [42].

system to learn about its internal composition and execution environment. The learning algorithm uses these data to classify environments and determine the level of satisfaction of the system's goal, serving as the base for the system to learn and make design choices. The metrics are used to represent aspects of system health status (e.g., performance, security level, and so on), while events are created to store the values of environment features (e.g., input patterns, hardware characteristics). In tandem, the Learning module uses both events and metrics to realise its reinforcement learning approach. As an essential characteristic, both data types (see Fig. 1.4) were designed to capture information regardless of the application domain, enabling the PAL framework to learn about any target 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 represents the aspect of the system that is being monitored. This field may be set as *'response time'* or *'the 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 corresponding value associated with the metric. For example, for *'response time'* the system may store a value in milliseconds representing the time that the system takes to process incoming requests. Additionally, metrics are used to represent system status and express the 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. `name` and `preferHighValue` are often manually defined by a 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, we focus on showing the feasibility of the Emergent Software Systems approach and not to explore the autonomous optimisation of multi-goal systems.

Events are used to represent different features of the operating environment. The event data type represents features with the following attributes: `name`, `type`, and `value`. The `name` attribute stores the name of the feature; for example, in cases where it is im-

portant to characterise request patterns, a possible event type could be named as *'request type'*. The attribute `type` defines the data type of the environment feature. In our request pattern example, this 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, it stores 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 system operation. The `name` values are often manually defined at design time, while the `type` and `value` are collected from the executing target system.

After collecting events and metrics from the executing system, other important attributes can be inferred: the number of times a certain attributed was perceived (e.g., number of requests for text files). This could be essential information that assists the system in determining whether an event is recurring or not, which might be an essential aspect to characterise the operating environment. Another critical piece of information is the time when the metrics and events were collected. Time-stamped data enables the system to establish a timeline with information on how the system and the operating environment behave in a time frame. The collection of events and metrics is done by proxies, which are components that are autonomously generated to collect (and time-stamp) information about the system.

1.5.2.2. Proxy Components

Proxy Components are essential elements in the Perception module. They are responsible for extracting events and metrics from the system and operating environment. A proxy component complies with the same policy of other components in a component-based model. Thus, with regards to the Assembly module, it is indistinguishable from other components in terms of code structure, syntax, and connection to other components. The only difference is that proxies are annotated to allow the Assembly module to locate them and extract the collected metrics and events and to avoid connecting two or more proxy components.

A special characteristic of 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 another that provides the interface (e.g. component B). Then the proxy can intercept function calls from A to B and extract information from their interaction. Fig. 1.5 illustrates a proxy component inserted in the system.

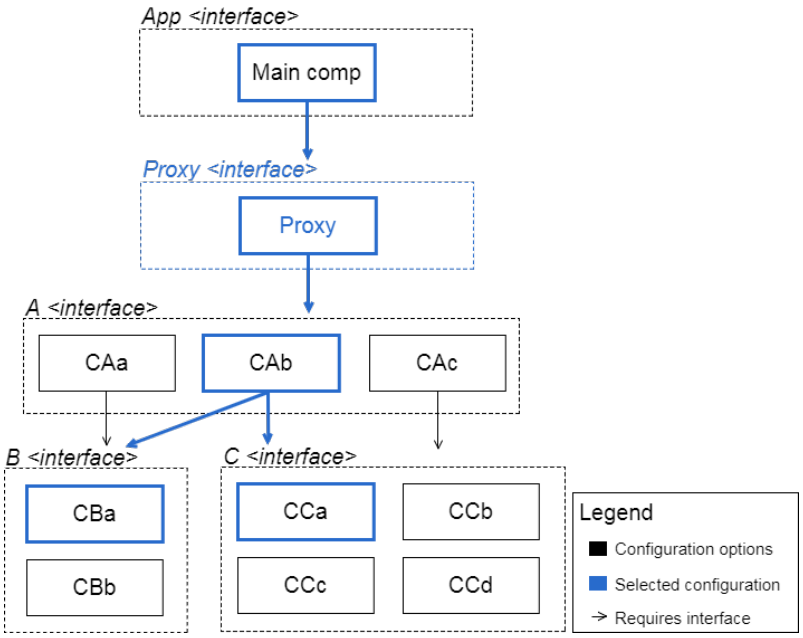

Figure 1.5. Proxy component inserted in the system's architecture [42].

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 monitored component to extract information for the system. Thus, it is possible to separate the code responsible for monitoring a specific component from the component itself, not requiring component developers to code functions to monitor the component, avoiding it becoming 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 to enable their monitoring.

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. Proxy generation is further detailed next. Furthermore, considering that a Proxy component is generated to monitor target interfaces rather than components, and that multiple components might require the same interface throughout the system, a Proxy Expression Language was designed to allow the Learning module to express precisely what interface to place the proxies in the system structure. The Proxy Expression Language is also described in detail in what follows.

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 the components to be monitored. The Learning module triggers the proxy generation process. Once it obtains the system goals and the available architectural compositions, the Learning module triggers the proxy component generation process. This generation process requires as input the target interface and components responsible for generating 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-collector components. For example, to collect the response time in milliseconds, 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 to collecting event information. An example 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 components described above 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 generated in the proxy are timestamped once created and stored in an object named `Container`. This object is responsible for averaging the metrics and events of the same type, updating the 'count' variable to express the number of occurrences of a specific event or metric, and making them available to the Perception module, which, in turn, formats them into a JSON string as **perception data** to send to the Learning module upon request. Note that averaging event and metric values is essential to reduce the amount of data stored in the proxy component.

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 can monitor all component variants that provide such feature, independent of individual component implementations. Furthermore, suppose that the system adapts its architectural composition, from an implementation of the monitored component to one of its variants. In that case, the Perception module can replace the component without replacing the proxy, reducing the number 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 components in the software architecture. As a result, when placing a proxy to monitor an interface, the Perception module actually needs to place multiple proxies to monitor *every* component that requires that interface throughout the architecture, thus impacting the performance of the system. The Proxy Expression Language was designed to provide fine-grained control over the process of proxy placement, while avoiding the need for the Learning module to indicate what component (instead of interface) should be monitored every time an architectural change occurs.

The Proxy Expression Language (PEL) is a tool to precisely express where to place proxy components in the architectural composition. The Learning module uses this language to ensure that the generated proxy components are kept in place, monitoring the intended interface even as the software compositions are constantly changing. The Perception module is responsible for interpreting the expression and making a list of architectural compositions without proxy components, associating these compositions with

their equivalent compositions with proxy components added at the right places (interfaces or components) 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 the list of equivalent compositions and decides. Suppose the new composition matches one of the compositions in the list. In that case, the Perception module changes the software composition to the corresponding new composition with the proxy component in the right place according to the expression. This strategy enables the Learning module to work with architectural descriptions with no added proxy components. The Perception module is responsible for applying the PEL expression provided by the Learning module once (before it starts the experimentation process) and transparently adding proxy components to the appropriate places 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 for only one specific component, or for all components that require an interface, or any level between the two, for example, by placing a proxy component to monitor an interface but only if a specific component requires that interface (as opposed to every component that requires that interface). The Learning module can thus create expressions as generic or as specific as needed, 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 software architecture parts gives a clearer perspective of the system status and its operating environment. However, it adds overhead to the processing and collection of events and metrics, impacting system performance. Thus, an essential challenge in adding proxies to the software architecture is identifying the number of proxy components and their optimal placement to reduce the number of proxies whilst maximising the quality of collected data. Considering that, in an Emergent Software Systems architecture, a component calls functions on lower-level components (as shown in the component graph of Fig. 1.3), by placing proxies at the highest possible level, the monitoring proxy can calculate the time a stack of function calls takes to execute. This is not a general rule and may vary according to 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*.

1.5.2.3. The Perception Module API

The Perception Module API is presented below:

- `void addProxy(char exp[])`: This function receives as parameters an expression in the Proxy Expression Language (PEL) format that determines where the proxy should be inserted into the software architecture. At the end of this function's execution, the user should expect the monitoring proxy to be inserted into the systems architecture;
- `void removeProxy(char exp[])`: This function removes inserted monitoring proxies. The function receives as parameters the same expression used to add the proxy in the system and uses it to remove the proxy;

- `char[] getPerceptionData()`: This function returns the collected metrics and events from the executing system in a JSON format;

These are the main functions provided by the Perception module. Note that, due to the dependency of the Perception module from the Assembly module (the Perception modules uses the Assembly module functions to implement their own), any component that interacts with the Perception module will have access to both the Perception module functions and the Assembly module functions.

1.5.3. Learning Module

The Learning module is responsible for guiding the learning process, which is crucial to the realisation of Emergent Software Systems. This module uses the lower level modules (Perception and Assembly modules) to trigger the deployment and composition process and to request Proxy generation by selecting Metrics and Events generators according to the system goals. It controls the entire online learning process, triggering the exploration and exploitation phases, classifying the operating environment and identifying the most suitable architectural composition. This section describes the learning process (exploration and exploitation phases), showing three learning strategies: i) the *Baseline* approach, ii) the *Feature-based* approach, and iii) two *Multi-armed bandit* approaches. Furthermore, this section also describes an environment classification algorithm.

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 the assembled collection of components (the system's behaviour) and the system's perception of its performance in each identified operating condition. The Learning module executes its main task by requesting the Assembly module to change the system's architectural composition in order to experiment with the identified changes in the operating environment. It then observes its performance and the operating conditions through the Perception module.

1.5.3.1. Learning Algorithms for Emergent Software Systems

The generic approach to realise learning in Emergent Software Systems involves three main tasks. Firstly, the system must be able to characterise and classify features in its 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). Secondly, after finding the most suitable architectural composition for the perceived environment, the system exploits the optimal composition whilst observing the environment and the its own performance to detect any changes and trigger learning again, in case it detects unforeseen changes. Finally, 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 already known to be good [49]. This third task is essential because the emergent software framework operates on live software, and sub-optimal performance has real consequences.

Performing online reinforcement learning is highly challenging. The software is not in control of its operating environment and cannot know in advance when it may reliably compare any two software compositions against the same operating condition. Moreover, there are complex interactions between the exploration process itself and the environment, as selecting a "good" composition may impact the system performance and change the perceived environment. The explored learning approaches are reinforcement learning [49] algorithms tailored to our particular problem space. The learning approaches presented in this section continually discover optimal assemblies by exploring the search space while simultaneously classifying observed features of the operating condition into labelled environments. The *Baseline* approach explores all available compositions before opting for the best performing option. This approach is presented as a baseline to compare with other learning strategies because it is guaranteed (when operating conditions are maintained during exploration) to find the optimal global composition. The *Feature-based* approach, on the other hand, explores the search space analysing features instead of individual compositions. Thus it reduces the search space by focusing on representative compositions of available features. This approach relies on domain-specific assumptions and is scalable to support learning in large search spaces (i.e., with a large number of system composition options). Finally, the *Multi-armed bandit* approaches differ from the above strategies by adopting better strategies to balance the exploration and exploitation phases. They also consider fluctuations in the performance metrics collected from the live system, making these approaches more robust. If a drastic fluctuation on a performance metric of the system occurs, the baseline and the feature-based approaches may converge towards a suboptimal system composition.

The Baseline Algorithm applies a standard "exploration activity" to both characterise the current environment and identify the best composition for that environment. 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., after system startup), (ii) the current environment characteristics deviating outside the expected ranges from existing experience, or (iii) current system performance deviating beyond its expected range.

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), based on previous investigation, 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 phase. The selected action continues to be monitored and analysed for its suitability every w_t time units. A change

in the operating environment is detected 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. This detected change in the operating environment also triggers a change in the learning behaviour, forcing it to start exploring again. The algorithm waits for $w_t * 3$ of consistently observed behaviour to avoid frequent oscillation between exploitation and exploration phases before changing its current course. In case (i), if the detected event pattern has been previously seen, the best matching composition is simply selected. In all other cases, new exploration activity is triggered. This process of exploration/exploitation repeats continually, where the amount of exploration reduces as fewer new environmental conditions are seen. Note that this learning algorithm, 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.

The Feature-based learning strategy is an alternative solution for the Learning module. 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 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 any component of the other features, which might not be true for every application.

This strategy maintains the essence of the reinforcement learning algorithm, i.e., the entire process described in the baseline strategy still applies. The only part of the algorithm that the feature-based strategy changes is the selection of the next composition to be tested. In the baseline approach, the algorithm tests them all. The feature-based approach navigates through the software architecture from top to bottom, deciding on the component variants it encounters for each feature (interface) with multiple component variants. Once it determines the best component variant for a feature, it traverses the tree downwards considering only the chosen component variant, eliminating the branches that correspond to 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 the baseline approach.

The description of the *Feature-based* approach as well as an evaluation of its per-

formance is detailed in [42]. The algorithm considers a tree-like structure that represents all architectural compositions for the software, as illustrated in Fig. 1.3. Considering a tree-like structure representing the available architectural compositions, the algorithm selects the first interface (down from the root) 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 to bottom, determining the best component variant only for the required interfaces with more than two alternative components implementing them. The algorithm tests all interfaces (that have component variants) required by a selected component at a higher level, ignoring the interfaces required by components that were not selected at any level of the tree. The selection of component variants in each required interface is carried out by executing architectural compositions containing the component variants and selecting the variant in the best performing architecture. 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 containing all selected components the "optimal" composition.

A quick performance comparison between the *baseline* and *feature-based* algorithms shows that the feature-based strategy converges much faster. Considering the abstract architectural compositions of Fig. 1.3, the *baseline* 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 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 baseline approach. The worst case scenario, however, is not as significant, having a difference of only 30 secs (and not guaranteeing global optimality). However, if we consider that, for every exploration phase the baseline approach always executes in 2.3 secs (considering w_t = 10 secs), the feature-based search is still very advantageous. A more expressive scenario, using a real system as example, is described and explored in [42], showing the advantages of the *Feature-based* search approach.

So far, the presented algorithms were specifically developed to realise the Emergent Software Systems concept. Next, we present two pre-existing algorithms, describing their use to build Emergent Software Systems.

Multi-armed Bandit is a well-known problem in statistics with many proposed algorithms to solve it [5, 9, 48, 50]. The solutions consist in balancing the exploration and exploitation phases. This balance is widely studied and very important in reinforcement learning strategies. In the multi-armed bandit problem, we consider multiple slot ma-

chines, just like those found in casinos, that give a particular reward whenever their arms are pulled. Each machine has a probability distribution function over the presented reward, meaning that each time an arm is pulled, a different reward may be presented. Some machines may be more inclined to have a higher probability of presenting high rewards. The problem consists in discovering which machine is most likely to yield the highest average reward. The probability distribution function of each machine is not previously known. The solution should explore the available machines to get their rewards and learn their probability distribution functions. However, as the solution gathers information on the rewards of some machines, it should also *exploit* the machine (or set of machines) which are known to yield high rewards up to that point.

Modelling the Emergent Software Systems learning problem using the multi-armed bandit model is useful for two main reasons. Firstly, it allows the use of well-studied algorithms known to provide the solution. The advantage of using such algorithms is that they have theoretical guarantees regarding convergence towards the optimal solution. Although the theoretical convergence occurs in infinite time, it is guaranteed that the algorithm will eventually converge towards an optimal solution. Secondly, the fluctuations over the reward signal match the fluctuations of collected metric values from running systems that are never fixed on a specific value but rather slightly fluctuates over a range. That means that when an executing system is queried for its execution metrics (the metrics that allow the evaluation of its performance) the metric readings are never a fixed number, it always varies slightly as it is queried.

In the context of emergent software systems, pulling an arm means that an action is being performed on the running system to change its architecture from one composition to another. The bandit algorithms explore the available systems compositions at runtime to find the composition that yields the best performance. Considering that the system is in production, the algorithm must carefully balance the exploration (i.e., finding the best-performing composition) and exploitation (i.e., taking advantage of the best-performing composition) phases. This balance is crucial to avoid overly affecting the performance of the running system while it handles incoming user requests.

The two main bandit algorithms used to realise Emergent Software Systems are **Upper Confidence Bound (UCB1)** [4] and **Thompson Sampling** [9]. This section describes the working mechanisms of these two solutions and how they are used in the context of realising emergent systems.

Upper Confidence Bound (UCB1) is a well-known and widely used algorithm to solve the multi-armed bandit problem. The algorithm works as follows: it starts by taking action (i.e., pulling an arm), then it waits until it receives a reward, it normalises the reward, and then it calculates what action should be taken next based on its equation, for every single available action. The action that yields the highest value as a result of calculating the equation becomes the following action to be taken. The equation is described and shown below:

$$r_k + \sqrt{\frac{2 \ln n}{n_k}}$$

The variable k indicates a certain action. It ranges from $0 < k \leq$ *total number of*

actions, each action in an emergent systems context is a unique architecture composition. The variable n indicates the total number of times the algorithm performed actions (i.e., selected any architecture composition). The n_k term indicates the number of times action k was selected. The first part of the equation defined by r_k represents the average reward of choosing action k. The squared root part of the equation determines the algorithm's confidence level on a certain action k. The algorithm calculates the result of the equation for every single available action. The action that yields the highest value from the equation becomes the algorithm's next action. This process repeats while the system is executing, and at some point in time, when the algorithm converges, the action that yields the highest average reward will always be chosen from that point on. After convergence, the average reward value for every action becomes too large, and the squared root part of the equation (i.e., the confidence level of the algorithm on each action) becomes too small that it cannot force the algorithm to take any other action, expect the best one.

The confidence level is inversely proportional to the squared root part of the algorithms' equation. That means that as the algorithm gains confidence in a specific action, that second part of the equation yields a minimal value, and therefore it interferes less on the action that will be chosen. On the other hand, if actions are neglected (i.e., not chosen for a long while), the confidence level of the algorithm to take such actions drops, meaning that the resulting value of the squared root part of the equation increases, forcing the algorithm to choose the action. In this particular regard, the equation dictates the primary mechanism on which each UCB1 algorithm operates, making it balance between exploring and exploiting the available actions as it increases the algorithm's confidence in the returned reward of a specific action.

We refer the reader to [39] for further detail on the approach, including an example and the evaluation results of its use in the Learning module of the PAL framework.

Thompson sampling is another algorithm that solves the multi-armed bandit problem. The difference between this algorithm and UCB1 is in the mechanism of balancing the exploration and exploitation phases. Instead of being guided by the UCB1 equation, each action in the Thompson sampling approach is selected with the probability of it being the best action given the history of the algorithm's actions selection.

In detail, the algorithm considers a probability distribution over the reward of each action with densities given by bell curves. That means that the centre of the bell curve is history of the average reward on that action, and the spread of the curve is the level of uncertainty (i.e., high uncertainty is a result of few attempts at a specific action). For an action to be selected with Thompson sampling, the random sample from its bell curve must be higher than the corresponding samples from all other actions. This is true if either the centre point is high or if the spread is large, corresponding respectively to the high average observed rewards or high uncertainty.

As a result, the algorithm selects the most likely actions or actions that are likely to perform well, but the algorithm has less information (i.e., the algorithm is less confident about). The rest of the algorithm is very similar to the UCB1; the algorithm selects an action, observes its reward, normalises its reward, determines which action should be taken next according to its probability of yielding high rewards, then selects the next action and repeats. At some point, as the system executes, the algorithm gathers enough

information on the probability distribution of each action, so that it knows with a high degree of certainty which action will yield the highest reward.

For an example of the use of this approach to create an emergent web server, please see [37].

1.5.3.2. 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 is very challenging. That is mainly because the executing architectural composition may distort the perceived conditions. These distortions create the illusion of changes in the environment. For example, the system may perceive a workload volume increase, when in reality, it is the running architecture that can process more requests in less time. This phenomenon is described in Sec. 1.3 and it is referenced as self-referential fitness landscapes. This section describes the environment classification algorithm used in tandem with the *Baseline* and *Feature-based* learning approaches.

After collecting information about the explored architectural compositions, the Learning module receives a list of *Events* and *Metrics* attached to each executed composition. Based on that information, the system classifies the operating environment. This approach minimises the effects of environment distortion by classifying environments using ranges. The algorithm creates ranges for each collected Event and Metric, with minimum values defined by the lowest perceived values for the event or metric and maximum values defined by the highest value registered by the compositions, as illustrated in Fig. 1.6.

Fig. 1.6 illustrates the classification of two operating environments obtained from the tests with the emergent Web server; these results are detailed in [45]. Consider a Web server program and the operating environment as the patterns of requests handled by the server. The first environment consists of large-size text file requests with low variations, i.e., the majority of text files requested were repeated files. The second environment, on the right hand side of the figure, 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 no extra event type and value (e.g., within 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 three iterations (the threshold to trigger exploration or architectural changes), the system compares the values it perceives with previously defined environments, in case these values are within all the ranges established

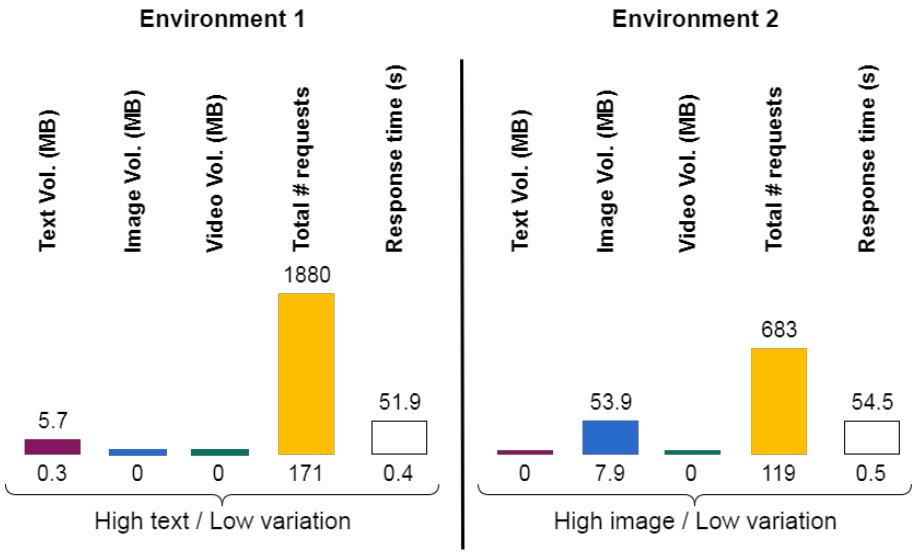

Figure 1.6. Example of classified environments using ranges of collected *Event* and *Metric* values. The first environment consists of requests of large-size text files with low variation. The second environment consists of large-size image files with low variation [45].

in classified environments, the system changes to the composition attached to the new environment. Otherwise, the system triggers exploration.

This range-based 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 it is demonstrated in [45]. 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, as with any current machine learning approach, this approach suffers when essential features of the environment are not defined as Events, leaving, for example, two or more distinct environments to be classified as the same. As previously mentioned, the implementation of the Emergent Software Systems learning approach is very challenging, with several open issues to be addressed. Therefore, further research is required to better explore and overcome the range-based environment classification issues.

1.5.4. Practical: Using the PAL Framework

This section covers the main ideas and concepts of a set of practical exercises that aim to familiarise the reader with the presented framework. Note that all the details of the practical exercises along with the code for the PAL framework and the detailed instructions on how to install all the necessary software to run the exercises are available on Github[3]. The framework and the target system were developed using the Dana programming language. All code presented in this section is written in the Dana programming language[4].

[3]https://github.com/robertovrf/sbrc21_minicurso
[4]https://projectdana.com

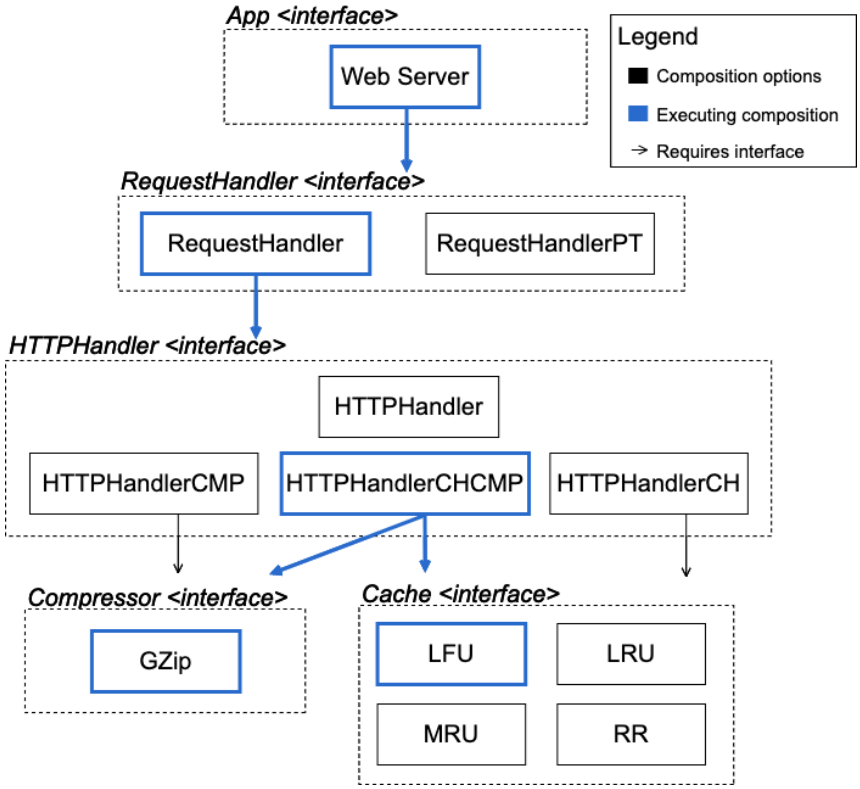

Figure 1.7. Emergent web server architecture.

The practicals presented in this section aim at providing hands-on experience to the reader on creating emergent systems. We begin by describing the target system on which practical will be based. We introduce the components used throughout the practical and describe the module that allows the target system to learn to self-adapt without human interference or predefined domain-specific knowledge.

1.5.4.1. Emergent Web Server

This section focuses on the emergent Web server as the target system. The Web server is developed using the Dana programming language following its component-based model. The Web server consists of four primary interfaces that define its main functionalities. There are component variants for some of these four interfaces, i.e., components that implement the same interface but in a slightly different manner. These component variants enable the recomposition of the web server architecture, maintaining its main functionalities but behaving in a way that may impact the Web server's performance when subjected to different request patterns.

Fig. 1.7 shows the Web server architecture. The outer dotted boxes represent the interfaces, while the inner boxes represent the components that implement the respective interface. The main interfaces are the *RequestHandler*, *HTTPHandler*, *Compressor* and *Cache*. The *RequestHandler* interface defines how the Web server handles TCP requests. For that particular interface, there are two components that implement that functional-

ity. The first component, also named *RequestHandler*, creates a thread to handle each TCP request. The component variant, named *RequestHandlerPT*, maintains a pool of threads instead and assigns incoming TCP connections to an existing thread to handle the incoming request.

The interface *HTTPHandler* defines the functions necessary to handle HTTP requests. For that interface, there are four component variants: *HTTPHandler*, which implements the functions to handle HTTP requests with no additional optimisations; *HTTPHandlerCMP*, which compresses the response before sending it to the client; *HTTPHandlerCH*, which caches the response after sending it to the client; *HTTPHandlerCHCMP*, which compresses the response before sending it to the client and caches the compressed response in memory.

The remaining interfaces, *Compressor* and *Cache*, define the functionality of both compression and caching. Both interfaces are used by the components that implement the *HTTPHandler* interface and require compression and/or caching. For the compressor interface, the *GZip* component implements the gzip compression algorithm. For the cache interface, in turn, each component variant implements a different cache replacement algorithm (to replace cached items when the cache is full and a new item needs to be cached). The cache implementations are: Least-Frequently Used (*LFU*), Least Recently Used (*LRU*), Most Recently Used (*MRU*) and Random Replacement (*RR*).

The set of components illustrated in Fig. 1.7 are used to compose a functioning Web server capable of serving files located in the *htdocs* folder. To compose a fully functional Web server, one component for each of the interfaces must be part of the application architecture. The adaptation of the Web server consists of swapping at runtime one component to a variant component that implements the same interface. For instance, we enable the adaptation of a Web server composition that uses the *HTTPHandler* component variant that implements the *HTTPHandler* interface to the *HTTPHandlerCH* that caches the response after sending the response to the client. In the new composition, the Web server performs better when the workload pattern has many repeated requests that result in the same response.

In the Github repository that hosts the code for the practicals, the reader will find the Web server code in the repository folder and the instructions to run it with a specific fixed composition. In the next section, we describe a component that enables the reader to interact with the Web server and its various compositions.

1.5.4.2. Interactive Emergent System (IES)

The Interactive Emergent System (IES) is a command-line tool that provides access to the PAL framework. Its goal is to allow the user to interact with the framework modules through the command line to inspect and control a running emergent system. In this part of the practical, we expect the reader to run IES and use it to compose the emergent Web server (explored in Sec. 1.5.4.1), change it from one composition architecture to another as it executes, and collect response time information from the executing Web server.

To execute IES, it is necessary to provide, as parameters, the path to the root com-

ponent of the Web server. A detailed description of how to execute IES to start the web server is provided in the companion Github repository. Once the IES starts, it calls the Assembly module function *setMain()*, which triggers the Assembly module to search for the components to compose the Web server, including its several architectural compositions. After the Web server is loaded and one of its compositions is executing, the IES presents a command prompt that enables interaction with the framework functions.

The user can call any function of the ESS framework modules through the IES command prompt. For instance, the user can list all Web server compositions found by the Assembly module (through the Assembly function *getAllConfigs()*) and change the executing Web server architecture to any of the available compositions in the list (*setConfig(config)*). The user is also able to query the running Web server (*getConfig()*) and collect information on the performance of its current composition (by calling the Perception function *getPerceptionData()*). Note that the Web server executes normally during the IES execution, and clients can issue HTTP requests to it at any time. Requests to the Web server can be issued from a regular web browser or through a client program that is also available in the repository.

We expect the reader to explore all functions provided by the PAL framework modules and familiarise themselves with the capabilities provided by the framework. Next, we explore the execution of the PAL framework to realise the emergent Web server, showing the potential of the emergent software systems framework to autonomously compose systems and learn the most suitable composition for different operating conditions.

1.5.4.3. Emergent Web Server

This section presents the third and practical exercise, which provides hands-on experience executing the emergent Web server. This exercise consists in executing the PAL framework targeting the Web server and experimenting with different operating conditions for the target system as it learns the best composition for each operating condition. We expect the reader to execute the framework and observe how it operates. We welcome the reader to develop different operating conditions (by creating different clients for the Web server) or to create new components to allow for further architecture variants. After such additions to the experiment, we expect the reader to test the framework to see if it locates the best composition for the new setting.

In this practical, the reader should execute the emergent Web server using the Interactive Emergent System tool. After the Web server is loaded, the reader should start a client program that generates a workload for the server. As the web server handles the requests, the reader should start the learning process and observe the system as it learns the optimal composition with no predefined domain-specific knowledge or human interference. Once the learning converges, the reader can choose a different client that generates a different workload pattern, in order to see if the learning can detect the change and converge to the new optimal composition.

We also encourage the reader to interact with the IES tool and manually find the optimal best composition for a given workload pattern generated by an available client program. This will give the reader an idea of how challenging it is for the learning al-

gorithm to perform the same task. We also encourage the reader to think of different learning approaches that could be used to make the learning module converge faster to an optimal composition by writing and experimenting with new learning strategies. A detailed description to execute this practical is in the Github companion repository.

1.6. Emergent Microservices

This section introduces the concept of Emergent Microservices [43]. This concept stems from the direct application of the Emergent Software Systems concept to build microservices able to autonomously evolve their internal architecture to accommodate changes in the workload pattern. We present the concept of microservices and motivate the need for emergent microservices. We then explore the concept in the domain of smart cities, which is known for having highly volatile execution environments. We conclude the section by presenting a set of hands-on exercises to familiarise the reader with the Emergent Microservice concept. The code for this practical is also available in the GitHub companion repository[5], along with detailed instructions to run the example. In the practical, we also expect the reader to extend the example by creating new emergent microservices and testing the concept in different domains and operating conditions.

1.6.1. Microservices

Before we dive into the concept of Emergent Microservices, we first introduce the general concept of microservices. Microservices has gained popularity in the industry and has been used as building blocks to create large-scale and flexible systems. Due to its increasing popularity, the concept has become very important in modern distributed systems and has been extensively used to build systems in various domains (e.g., Web-based systems, video streaming services, supporting platforms for IoT-based systems).

In short, microservices can be defined as self-contained small services that implement a single functionality of a larger system [29]. As a result, instead of implementing a system as a massive, highly interconnected monolith, the system can be broken down into smaller self-contained, easier-to-manage and highly reusable services. These characteristics make microservice-based systems highly maintainable, easier to evolve, highly reusable and adaptive, which are essential characteristics to create modern systems.

Each microservice is self-contained, which means that they can be deployed and run independently of other parts of the system. A microservice provides a RESTful API, with its endpoint URI, HTTP method, expected parameters, and expected result. We show an example of a microservice API function documentation below:

```
HTTP Method: GET
URI: https://189.234.22.12:80/collector/get_data
HTTP status code: (200) OK,
                  (400) Bad Request or
                  (500) Internal Server Error
Response description: Collected data (JSON format)
Response example:
```

[5]https://github.com/robertovrf/sbrc21_minicurso

```
{"resources": [ {
  "uuid": "ae9cf502-5ed2-47d4-914c-c1caec1c41c4",
  "capabilities": {
      "environment_monitoring": [{
          "temperature": "38.313",
          "humidity": "38.313",
          "date": "2016-06-21T23:27:35.000Z"}]
}}]}
```

Microservices are often deployed on cloud-based infrastructures, packaged inside containers that are managed by container-orchestration systems. The container-orchestration systems are fundamental to manage microservices in real deployments. They offer support for elasticity both horizontal and vertical, and system recovery when the container fails, triggering the creation of a new instance. These are important actions to cope with the dynamism of the system's operating environment, which constantly changes and demands the system to adapt.

1.6.2. Emergent Microservices

Microservice supporting tools, such as container orchestration systems, are well equipped to handle workload volume variation. To cope with the increasing user demands, they may create new replicas of the microservices, or increase the amount of computing resources available to the container running the microservice. However, they are not able to efficiently cope with variations that also involve changes in the workload pattern.

Changes in the workload are common in current systems. In general, these changes may affect the workload volume (e.g., the number of requests issued in a time frame), which may increase or decrease according to user demand, and the workload pattern (e.g., from requests for text-based data to requests for image-based data). Changes in the workload pattern often impact microservice performance depending on the type of functionality or data being returned. In that sense, some workload patterns require the evolution of the microservice's internal implementation to better accommodate the new pattern. For instance, if there is an increase in the size of text-based data returned by the requests to a microservice, compression of the returned data may be an effective way to improve the service's performance, enabling it to better accommodate the new workload.

Such improvements in the service implementation required by a change in the workload pattern are often conducted by the developer. The process typically consists in having an expert analyse the service logs to determine the characteristics of the new workload pattern and how they impact the service's current implementation. Then, developers are required to implement, test and deploy a new version of the service. This process is entirely human-dependent and often inefficient. Emergent Microservice [43] is a concept that aims to support the autonomous evolution of a microservice's internal composition as a response to workload pattern changes. It represents a more efficient way to evolve a microservice's internal architecture at runtime, automating the process described above. We argue that this is not only convenient but crucial to support the development of future distributed systems.

The Emergent Microservice concept is realised by applying the Emergent Software Systems framework to a component-based implementation of microservices. To facilitate microservice implementation, we developed an emergent microservice framework on top of components that implement the Web server. The framework is illustrated in Fig. 1.8 and is composed of a set of components that implement the business logic of the microservice, along with utility components that assist their implementation. Examples of utility components are implementations of common data structures (e.g., hash map, lists, queue, trees) and database drivers (e.g., MongoDB, MySQL).

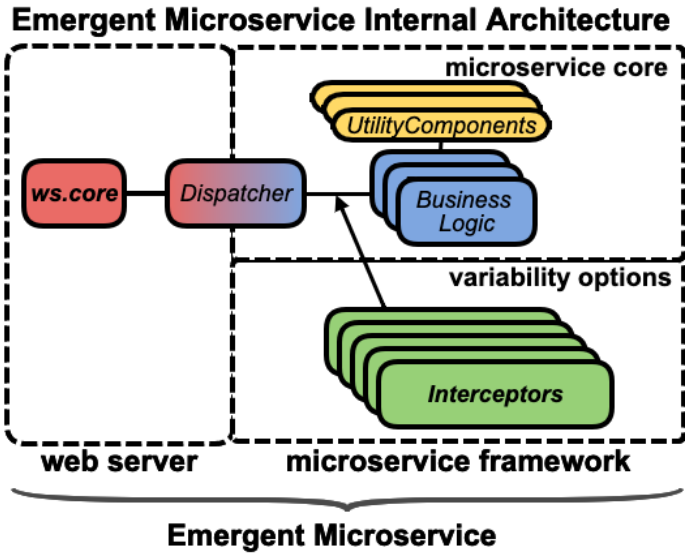

Figure 1.8. Internal architecture of an emergent microservice, showing the main components and how they connect. (Figure adapted from [43]).

.

Apart from the business logic and utility components, interceptor components comprise an essential part of the emergent microservice's framework. They intercept the requests sent to the business logic components and alter their processing to change a specific non-functional concern. For instance, a microservice that constantly accesses a database to return historical unchanged data can improve its performance by caching the data in memory. In that context, a cache interceptor can be transparently inserted into the microservices internal composition to boost its performance when needed.

Interceptors are thus a transparent method of adding variants to the microservice's internal composition. This allows the development of dynamically adaptive microservices without extra effort from the developers. They are only required to provide the core implementation of the microservice by implementing their business logic components, possibly making use of the utility components. A set of predefined interceptors are provided, adding standard variants to the microservice's internal composition. The emergent software systems framework uses these variants to compose the microservice and learn at runtime the most appropriate composition for it.

Interceptors, however, are not the only way to add variation to the emergent microservice's internal architecture. Any component variant added to the framework might be picked up by the PAL framework and used to generate different internal architectural

compositions for the microservice. If there are different ways to implement the microservice's business logic, implementing the microservices functions differently is also a way to create different microservice architectural variants.

Once the components and their variants are implemented, the deployment process of an emergent microservice is similar to the process of deploying regular microservices. Often these microservices are deployed on a cloud computing environment, using containers and container-orchestration technologies. The process often involves containerising the microservices components and their variants as well as the PAL framework. As the container starts, the PAL framework assembles the first microservice composition and starts learning, as incoming user requests are handled by the microservice, the best composition for the observed request pattern. As a regular container deployment managed by a container-orchestrator, the microservice is also susceptible to be replicated as the volume of workload increases and the service performance hits a predefined threshold. This process of configuring container-orchestration systems to manage the microservice container is better described in the practicals (Sec. 1.6.4).

1.6.3. Case Study: Emergent Microservices in the InterSCity Platform

The Smart Cities domain was chosen as a use case to illustrate the Emergent Microservice concept due to the highly volatile execution environments that it entails. In particular, we focus on the InterSCity platform [13]. It is an open-source microservice-based platform to support smart cities applications. The platform aims at abstracting the interaction between applications and city resources, providing a layer capable of scaling to handle the application and data processing demands of large cities. The platform architecture is composed of six microservices, as shown in Fig. 1.9.

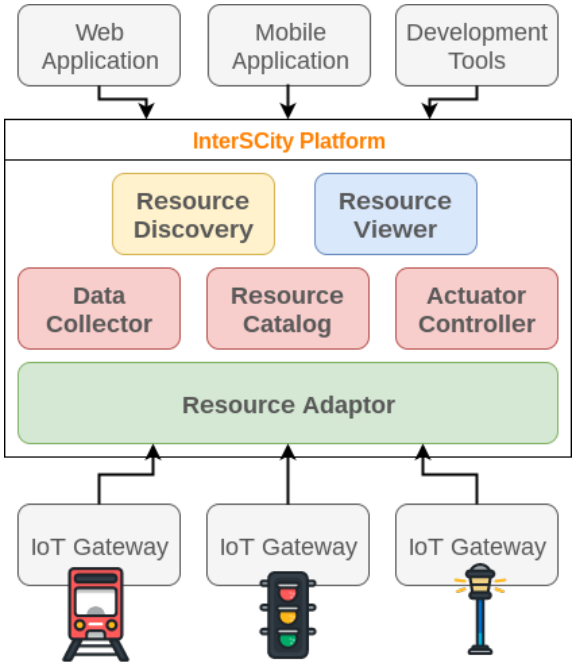

Figure 1.9. InterSCity platform microservices (https://interscity.org).

Each of the InterSCity microservices has a specific role. *Resource Adaptor* is the microservice responsible for receiving incoming requests from devices, redirecting those requests to the appropriate microservice. *Resource Catalog* is responsible for storing information about all devices available in the city. *Resource Discovery* provides information about specific devices. *Data Collector* is responsible for providing applications with access to all data collected from city devices, as well as providing the devices with a gateway to push their collected data. *Actuator Controller* is the microservice that provides access to the city actuator devices. Finally, *Resource Viewer* is responsible for showing visual representations of available city resources.

We selected the Data Collector (DC) microservice to redesign it as an emergent microservice for this case study. DC was chosen for being a crucial microservice in the platform, responsible for handling all data access generated by city applications. Thus, it plays a crucial role in the global performance and scalability of the system.

The functionality of the DC microservice is provided in terms of four operations:

- Retrieve all historical data in the database;
- Retrieve all historical data collected from a specific city resource (e.g., a bus line information);
- Retrieve the most recent data collected;
- Retrieve the most recent data collected from a specific city resource (e.g., the latest location of a specific bus);

To realise the emergent DC, we implement these operations on the business logic components of the emergent microservice framework (described in Sec. 1.6.2). The MySQL drive component is used as a utility component to provide the microservice with access to a database. As interceptors, we make available components that implement caching, compression, and both caching and compression (in the same component). They are used to generate the DC internal composition variants.

The architecture of the DC microservice, with all its components and variants is shown in Fig 1.10. The *Dispatcher* component forwards incoming HTTP requests to the *DataCollector* component for processing. This component is responsible for implementing the core functionality of the DC microservice. The interceptors, in turn, are the components that add variants to the DC microservice and enable it to have its behaviour optimised according to the pattern of incoming requests.

In total, the interceptors enable the creation of four unique microservice internal architectural compositions. In the default composition of the DC microservice, the *Dispatcher* is directly connected to *DataCollector*, and the microservice performs as in its original, non-emergent implementation. The second composition is obtained when the *Cache* interceptor is plugged between the *Dispatcher* and *DataCollector* components. This interceptor caches all responses sent back to clients. It increases the microservice performance when clients frequently request the same sets of data – caching the response in memory increases the microservices performance because it prevents the service from

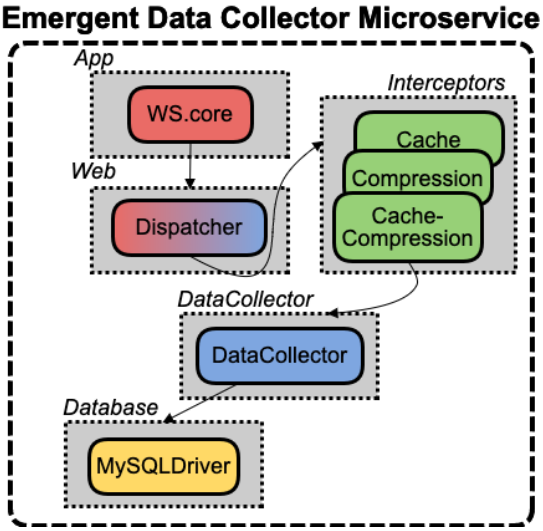

Figure 1.10. Data Collector Emergent Microservice. (Figure adapted from [43]).

connecting and retrieving data from a remote database. The compression composition, in turn, is obtained by placing the *Compression* interceptor in between the *Dispatcher* and *DataCollector* components. This interceptor compresses all responses before sending them back to the client. This composition may increase the microservice's performance when the returned data is large and highly compressible. Under these conditions, response compression may dramatically decrease the network transmission time of the response. Finally, the *CacheCompression* interceptor caches the compressed returned response. This composition improves the microservice performance in scenarios where clients request repeated data that are large and highly compressible.

We experimented with the emergent DC in a cloud-based environment to validate the concept. The result is illustrated in Fig. 1.11. We executed the emergent DC and two other implementation of DC: *Default* and *Cache-Compression*. The *Default* composition consists of a regular implementation of the DC microservice with no extra components to improve its performance. On the other hand, the *Cache-Compression* implementation consists of adding both a *Cache* and *Compression* components to the DC microservice. This last implementation compresses every response before sending it back to the client, and after sending the response, it caches the compressed response in memory. We subjected all three microservices to two different workloads, *Workload A* and *Workload B*. These workloads have distinct characteristics. *Workload A* consists of repeated requests to a specific small set of data (i.e., a workload where the cache composition would increase the performance of the system). In contrast, *Workload B* consists of requests to completely new data (i.e., a workload that would make cache-based compositions perform badly).

Fig. 1.11 shows that the *Default* composition has the worst performance when subjected to *Workload A*, but it has the best performance when subjected to *Workload B*. The graph also shows that *Cache-Compression* performs significantly better than *Defult* when subjected to *Workload A*, but it performs worse when subjected to *Workload B*. This result is expected due to the characteristics of the experimented workloads and the imple-

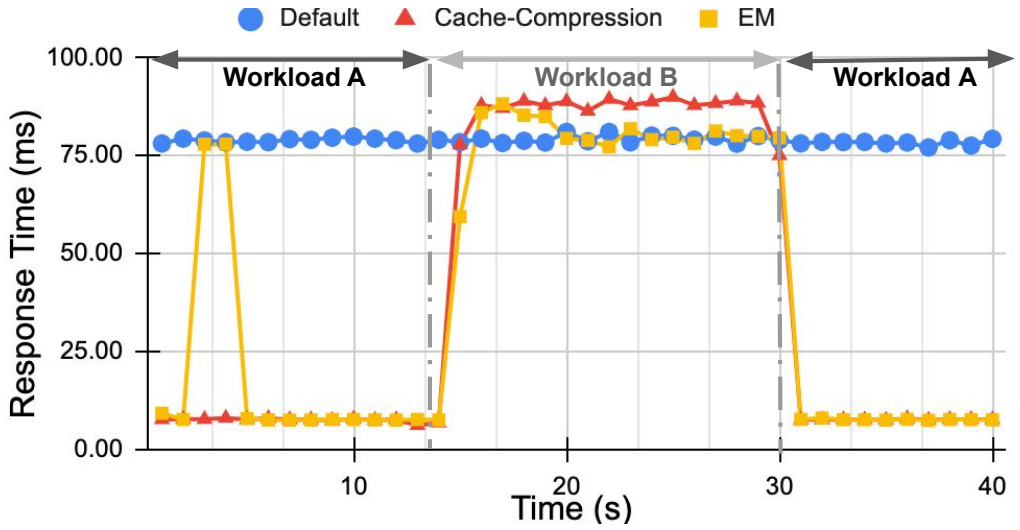

Figure 1.11. Average response time for three versions of the same microservice when subjected to two distinct workloads (A and B). EM learns and converges to the best performing composition for both Workload A (cache-compression) and Workload B (default). At 30s, EM identifies a previously seen workload pattern and is able to immediately adapt its composition.

mentation details of both microservices. The graph also demonstrates the ability of the emergent DC (the EM yellow square-dotted line) that after the learning phase (represented by the spikes on the yellow line) converges towards the best performing composition for both *Workloa A* and *B*. The graph finally shows that when the emergent DC is subjected to an already seen workload (from time 30s onwards on the graph – *Workload A* is executed again), emergent DC 'remembers' what the best composition is for *Workload A* and quickly changes its composition to *Cache-Compression* without the exploration phase.

As a result of creating and evaluating an emergent version of the DC microservice, we demonstrate that a microservice can react and evolve its internal composition when the incoming request pattern changes. Based on the application of the Emergent Software Systems concept and framework to realise Emergent Microservices, we can create microservice-based systems that are not only capable of coping with changes in the workload volume (through the use of supporting tools such as autoscalers on a cloud-based infrastructure), but are also capable of quickly and autonomously reacting in the face of workload pattern changes. Next, we describe a practical exercise that shows the necessary steps to execute the emergent DC microservice and subject it to various workload patterns in order to observe its dynamic self-adaptability.

1.6.4. Practical: Using PAL to Realise Emergent Microservices

After reading the previous sections (Sec. 1.6.2 and Sec. 1.6.3), where we introduce the concept and framework of Emergent Microservices (EM), along with its realisation using the PAL framework, we invite the reader to access the companion repository[6] and download the code and instructions to execute the emergent DC.

In the repository, the reader will find the emergent DC source code, along with

[6]https://github.com/robertovrf/sbrc21_minicurso.

the code for the PAL and emergent microservices frameworks, with instructions to run the system both on the cloud and on a local machine. If choosing to run it on a cloud-based infrastructure, the reader will find a set of scripts and a detailed tutorial to generate the emergent microservice Docker image, as well as the Kubernetes[7] scripts (YAML) to deploy emergent microservices. Otherwise, if the reader chooses to execute the microservices on a local machine, we make available an image of a virtual machine that comes configured with all necessary software. We also make available a set of client programs that simulate user behaviour and define different workload patterns for the reader to observe how the emergent DC learns at runtime, and without human interference, which microservice composition has the best performance.

If the reader has access to a cloud infrastructure, they will also find the instructions to run their emergent microservice in tandem with the Horizontal Pod Autoscaler (HPA). HPA is a supporting tool that observes the execution of a container and, in case the container's resource consumption increases above a predefined threshold, creates further replicas of the executing container. This is the state-of-the-art technique that has been widely used in production to autonomously handle sudden increases in workload volume.

The use of HPA in tandem with emergent microservices offers the potential to deal with both workload volume increase and workload pattern fluctuations. As the workload changes in pattern, the emergent microservice can learn, at runtime, the best performing composition for the detected workload pattern before HPA trigger any creation of replicas. As the workload volume increases, HPA can increase replicas to cope with the new volume. In tandem, emergent microservices can keep all replicas to their best performing composition according to the observed pattern. This, in turn, may lead to a decrease in the number of necessary replicas (i.e., resources usage) to cope with the new volume. Although this is a promising idea, it has not been fully explored in the literature.

The code available in the repository enables the reader to experiment and create their very own microservice. We encourage the reader to write different microservices and different component variants for their own microservices, as well as for the DC microservice, to familiarise with the concept. We also motivate the reader to use our available code to build on our work and solve the challenges that remain to be solved when realising the concept of emergent software systems in general and emergent microservices in particular. We list a set of research challenges and directions that the reader may consider in case they are interested in further exploring this research topic:

- Exploration of new learning approaches to better classify operating environments;
- Exploration of new learning approaches that enable better interaction among emergent microservices and other supporting tools that exist in the industrial microservice ecosystem (e.g., horizontal and vertical autoscalers);
- Explore the composition of large-scale systems using emergent microservices. Building systems with thousands or more emergent microservices;

We hope the community joins us in developing this novel technology that has

[7]https://kubernetes.io.

much potential to support the creation of future Internet systems. We believe that applying the PAL framework to build microservice architecture-based systems is a promising way to support the creation of large-scale self-adaptive systems that can handle the increasing dynamism of modern systems.

1.7. Conclusion

This chapter aimed to disseminate the concept of Emergent Software Systems, motivate the approach and advertise the tools to realise the concept. This chapter also presents the case for applying the concept to create real-world, fully functioning adaptive-systems. In particular, we presented an emergent web server and an emergent microservice. We also presented practical exercises to familiarise the reader with the concept and with the creation of real-world emergent software applications.

We have presented the concept of Emergent Software Systems as an approach to facilitate the development of autonomic computing systems. The concept consists in applying lightweight component-based models and reinforcement learning algorithms to create systems able to learn, at runtime, the most suitable available composition for the observed environment. Using these two technologies to build such systems enables the creation of everyday software systems that are emergent and adaptive. We also enable self-adaptive systems to cope with unknown conditions as our approach is shown to learn at runtime with no predefined domain-specific information.

This concept was successfully used to create real-world systems. It was explored in data centre-based applications such as the emergent Web server, the distributed Web server and emergent microservices. All information about the concept and these projects have been published elsewhere by the authors, as referenced throughout the chapter.

The chapter also presented the challenges and potential research directions to develop the concept further, hoping for a more significant adoption by the self-adaptive/ autonomic systems research community and industry practitioners. We finish the chapter by inviting everyone interested in developing future systems to join the effort to further develop and explore the main ideas that define the concept of Emergent Software Systems and their potential in supporting the creation and management of future systems.

Acknowledgements

This research is part of the INCT of the Future Internet for Smart Cities, funded by CNPq (grant 465446/2014-0), CAPES (grant 88887.136422/2017-00), and FAPESP (grants 14 /50937-1 and 15/24485-9). Dr. Rodrigues Filho would also like to thank FAPESP for supporting his research under grant 2020/07193-2.

References

[1] 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.

[2] 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.

[3] 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.

[4] P. Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[5] D. A. Berry and B. Fristedt. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). *London: Chapman and Hall*, 5(71-87):7–7, 1985.

[6] G. Blair. Complex distributed systems: The need for fresh perspectives. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1410–1421. IEEE, 2018.

[7] 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.

[8] 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.

[9] O. Chapelle and L. Li. An empirical evaluation of thompson sampling. *Advances in neural information processing systems*, 24:2249–2257, 2011.

[10] B. H. C. e. a. Cheng. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[11] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 160–178. Springer, 2001.

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

[13] A. M. Del Esposte, F. Kon, F. M. Costa, and N. Lago. Interscity: A scalable microservice-based open source platform for smart cities. In *SMARTGREENS*, volume 1, pages 35–46, 2017.

[14] 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.

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

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] V. Issarny, A. Bennaceur, and Y.-D. Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 217–255. Springer, 2011.

[25] 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.

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

[27] 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.

[28] 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.

[29] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert. Microservices. *IEEE Software*, 35(3):96–100, 2018.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] C. Müller-Schloer and S. Tomforde. *Organic Computing-Technical Systems for Survival in the Real World*. Springer, 2017.

[36] 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.

[37] B. Porter, M. Grieves, R. Rodrigues Filho, and D. Leslie. RE$^{\text{X}}$: A development platform and online learning approach for runtime emergent software systems. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2016.

[38] B. Porter and R. Rodrigues 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.

[39] B. Porter and R. Rodrigues Filho. Distributed emergent software: Assembling, perceiving and learning systems at scale. In *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 127–136. IEEE, 2019.

[40] 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.

[41] 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.

[42] R. Rodrigues Filho. *Emergent Software Systems*. PhD thesis, Lancaster University, 2018.

[43] R. Rodrigues Filho, M. P. de Sá, B. Porter, and F. M. Costa. Towards emergent microservices for client-tailored design. In *Proceedings of the 19th Workshop on Adaptive and Reflexive Middleware*, pages 1–6, 2018.

[44] R. Rodrigues 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*, pages 1–6, 2016.

[45] R. Rodrigues 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.

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

[47] 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.

[48] S. L. Scott. A modern bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010.

[49] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[50] W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

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

[52] 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.

[53] 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.

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