ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

COMPUTER SCIENCE AND ENGINEERING

Ciclo 37

Settore Concorsuale: 01/B1 - INFORMATICA

Settore Scientifico Disciplinare: INF/01 - INFORMATICA

BEHAVIOUR MATTERS: TOWARDS RELIABLE AND ADAPTABLE SYSTEMS

Presentata da: Lorenzo Bacchiani

Coordinatore Dottorato

Ilaria Bartolini

Supervisore

Mario Bravetti

Esame finale anno 2025

Abstract

In today's technologically driven society, the critical importance of ensuring a predictable behaviour in component-based systems cannot be overlooked. Modern software engineering practices, *e.g.,* application autoscaling and Continuous Integration/Continuous Deployment (CICD), promote effective adaptation to time-varying workloads, code quality and rapid deployment. Despite their effectiveness, these practices cannot guarantee the absence of unexpected events within complex systems. The consequences of misbehaviour, *e.g.,* service unavailability, violation of the Quality of Service, highlight the pressing need for enhanced practices.

In this Dissertation, we address these challenges through three primary objectives. *First*, we propose a new timed modelling/execution language to model the behaviour and simulate the execution of component-based systems. Such language enables the evaluation of system functioning early on in the software development lifecycle, giving DevOps teams the possibility of assessing the impact of their choices, *e.g.,* deployment decisions and scaling policies, at the modelling stage. *Then*, we present orchestration-based architectural reconfiguration techniques targeted at ensuring the system reaches a given goal, *e.g.,* service replication/migration. In particular, leveraging the knowledge of component properties, *e.g.,* functional dependencies, required resources, is crucial: starting from declarative specifications of these properties, we can automatically synthesise correct-by construction orchestrations that guarantee to instil the desired behaviour in the system. Moreover, service replication techniques exploiting these orchestrations overcome the drawbacks, *i.e.,* the "domino effect" caused by uncoordinated scaling, of existing service-level adaptation approaches, *e.g.,* Kubernetes Horizontal Pod Autoscaler. *Finally*, we devise a theoretical machinery for behavioural-based analyses in object-oriented languages endowed with typestates, *i.e.,* protocols attached to classes dictating order of method calls, and we implement it in a typestate-based checker for Java. Our type checker can be easily integrated in CICD pipelines, enhancing them with static analyses to proactively prevent component misbehaviours.

Culture sets you free

Acknowledgements

The day has come, this amazing adventure has come to an end. I could not be happier to have embarked on the PhD journey three years ago. It gave me the opportunity to grow, not only from an academic perspective, but also as a person. I faced my limits, I felt the disappointment of failures and somehow I found a way to overcome all these difficulties. Of course, I did not do all by myself, I had the best people around me.

I first want to thank my parents, Barbara and Pietro, for teaching me how precious culture is and for pushing me to work harder and harder to reach my goals. I hope this small achievement I made could be an inspiration to my little brother (not that little anymore), Alberto, to never settle and always strive for something more.

I'd wish to dedicate this work to the family I have created along the path, *i.e.,* my girlfriend Laura, who shares with me all the joy and sorrow, as well as all the successes and failures I have encountered along the way. This journey would not be so wonderful without her, especially my period abroad. With her perseverance, she was able to set up everything (I was about to give up the idea) to come and spend three incredible months in Lisbon, the best of my life. I deeply thank you Laura for all your love, Ti amo[1].

From an academic perspective, I must begin by mentioning Professor Mario Bravetti. These years, he has been an amazing mentor giving me insightful wisdom and suggestions on research and academic career development. He taught me how to become a confident and independent researcher, providing me with numerous opportunities along the way, for which I am truly grateful. If I were to pursue another PhD, I would choose him as my supervisor again. During these three amazing years I had the pleasure to work with great researchers like professors Maurizio Gabbrielli, Gianluigi Zavattaro, Antonio Ravara and dr. Saverio Giallorenzo. I owe them gratitude for the inspiring piece of advice they gave me along my academic path. I also have to thank my colleagues Giuseppe De Palma and João Mota, for the amazing cooperation during these years and the chats about the glory and misery of a PhD life — other than, from time to time, research.

[1] I love you

Contents

List of Figures

List of Listings

Chapter 1

Introduction

In the early days of software engineering, the monolith architectural style was the traditional approach to software development, used by the most important companies leading the information technology industry. As described in [DGL+17a], a monolith is a software where functions are encapsulated into a single application, whose modules cannot be executed independently. The most significant advantage of the monolithic architecture lies in its simplicity: monolith architectures are much easier to test, deploy, debug and monitor. All data are retained in one database with no need for synchronisation and all internal communications are done via intra-process mechanisms. Hence, they are fast and do not suffer from problems common to inter-process communication, *e.g.,* security issues, resource management. The monolith architecture is a natural and first-choice approach to build an application [BOP22]. However, such inherent monolithic structure renders it challenging to be used in the context of distributed systems, without employing specific frameworks or ad hoc solutions, such as Network Objects [BNOW93], Remote Method Invocation (RMI) [Gro02] or Common Object Request Broker Architecture (CORBA) [OPR95]. Despite the adoption of these approaches, monolith architectures still grapple with the general issues that plague monoliths, like: (*i*) large-size monoliths are difficult to maintain and evolve due to their complexity and tracking down bugs requires long perusals through their code base; (*ii*) monoliths suffer from the "dependency hell" [Mer14], where adding or updating libraries often leads systems not to compile/run or, worse, to misbehave; and (*iii*) monoliths

have limited scalability, since they require the whole application to be replicated, in order to handle increments of inbound requests. However, it is possible that the increased traffic places a strain only on a subset of the modules, making the allocation of some of the newly provisioned resources redundant [DGL+17a, Lau19]. To address the above limitations, companies started adopting Service-Oriented Architectures (SOAs). The reasons for choosing such architectural style are manifold: SOAs structure software applications as highly modular and scalable compositions of fine-grained and loosely-coupled services [DGL+17b, BZ09]. SOAs split monoliths into smaller chunks, thus addressing its difficult maintainability: the code base now becomes smaller, reducing the time needed for evolution and maintenance. The development lifecycle of a SOA requires services to be designed and implemented independently from each other. Therefore, in contrast with the problem of "dependency hell", complexities related to, *e.g.,* the adoption of a new library or updating an older one, are bounded to the scope of a single service. In the context of scalability, by decoupling services, SOA promotes individual and autonomous deployment of components, ensuring that only modules actually becoming bottlenecks are involved in deployment operations. SOA modularity, not only facilitates service-specific monitoring, enabling scaling decisions based on real-time performance metrics, but also makes it possible to migrate services across system districts, *e.g.,* cloud, fog or edge levels. Service-specific monitoring eliminates the waste associated with monolithic replication and optimises resource utilisation, leading to improved scalability and cost-effectiveness. Service migration, instead, brings performance improvement without the additional costs related to service replication. These dynamic approaches ensure resources to be provisioned precisely when and where they are needed, preventing over-provisioning and maximising resource efficiency.

The above properties make SOAs well-suited for combining them with modern software engineering practices like automatic adaptation as well as Continuous Integration and Continuous Deployment/Delivery (CICD) pipelines. Automatic application adaptation, which dynamically scales or migrates resources on demand, further enhances the adaptability of SOA architectures. By automatically provisioning and deprovisioning resources, automatic adaptation ensures that applications always have the necessary capacity to handle fluctuating de-

mand, while minimising costs [Bar18]. CICD pipelines automate each stage of the software development lifecycle, from development to system deployment, ensuring that changes are made quickly and reliably. SOAs modularity and loose coupling features facilitate this automation, enabling developers to deploy changes to individual services, without disrupting the entire application [Mau15, HF10a]. Together, SOAs, CICD and automatic adaptation create a powerful combination for developing and managing modern, scalable, elastic and resilient applications: CICD pipelines automate the development and deployment process and automatic adaptation ensures resources to be provisioned efficiently.

1.1 Research Problem

Modern society is increasingly dependent on large-scale software systems that are distributed, collaborative and communication-centred. Correctness and reliability of such systems crucially depend on components behaving consistently to their intended purpose. The consequences of unexpected events/misbehaviours are severe, including security breaches and unavailability of essential services.

Leading tech companies like Amazon, Google and Microsoft have established CICD as a core pillar of their software development strategies. These organisations recognise the power of CICD to enhance agility, reduce development time and improve the overall software quality. By automating manual tasks and implementing automated testing, CICD enables developers to focus on innovation and creativity, while ensuring that code changes are thoroughly vetted before deployment [HF10a]. While CICD practices have revolutionised the software development process, they did not come without limitations: they excel at ensuring code quality and facilitating rapid deployment, but they are not well suited for producing these large-scale component-based systems. As a matter of fact, a noticeable gap in these software engineering practices lies in the absence of comprehensive frameworks or languages specifically tailored to assess the holistic functioning of a component-based system during its initial design phase. While there are various tools and methodologies available for different aspects of software development, such as testing frameworks for code functionality or modelling tools for architectural design, there remains a notable deficiency in tools that can effectively evaluate the overall system be-

haviour early on, during the design phase. This absence poses significant challenges for engineers and developers, striving to anticipate and mitigate potential issues before they manifest in later stages of development or deployment. Addressing this gap would, not only enhance the efficiency and effectiveness of software development processes, but also contribute to the creation of more robust and reliable systems. A significant example, where a framework/language to assess the behaviour of the system as a whole is paramount, is *automatic adaptation*.

Automatic adaptation is a process entailing a set of component interactions that enables systems, designed accordingly to SOA principles, to adapt and respond to changes in the inbound workload. Automatic adaptation is a multifaceted approach that encompasses two complementary strategies: application autoscaling and service migration.

In the literature, there are two approaches to application autoscaling: vertical and horizontal autoscaling. The former, also known as scaling up, refers to adding more resources (CPU, memory and storage) to an existing machine. It is the most straightforward approach, but it is limited by the most powerful hardware available on the market [Wil12]. Given its limited nature, the vertical approach to application autoscaling is not studied in the context of this Dissertation. In contrast, horizontal autoscaling, also known as scaling out, refers to adding more service replicas and distributing the workload among them. This approach can be challenging because it has an influence on the application architecture, but can offer a virtually infinite amount scaling operations [Wil12]. The most popular tool available on the market is the Kubernetes Horizontal Pod Autoscaler[1] (HPA), a default and ready to use autoscaling feature. It depends on manually setting up some threshold values, *e.g.,* target CPU utilisation, minimum and maximum number of service replicas [AKR19].

Service migration usually exploits edge computing and moves a service from a cloud-based environment to the edge-based one. Edge computing is a distributed computing paradigm that brings computation closer to the edge of the network, where data are generated and consumed. One of the most applied technique to this approach to automatic adaptation is the data locality principle, *i.e.,* moving services towards the source of data they need is cheaper than moving data to

[1] `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`

services. Service migration can improve latency, responsiveness and security for applications that require low latency or real-time data processing. As a matter of fact, moving services to the edge significantly reduces latency as well as the amount of data that need to be transferred to the cloud [SZL+22, YLH+18].

Nevertheless automatic adaptation is an extremely powerful tool, it hides significant dangers. On the one hand, defining threshold values and scaling policies is crucial for a correct resource provisioning and significantly influence both performance and costs. Thus, the early evaluation and fine-tuning of the implemented thresholds and policies become paramount to guarantee that the system behaves as intended, *e.g.,* it complies with a certain Quality of Service. On the other hand, services may have intricate interdependencies that require meticulous management, involving the execution of a set of additional operations, when adding a new replica or migrating an existing service. The management of such intricate interdependencies requires DevOps teams to manually devise a program such that its execution correctly deploys the service under consideration. However, DevOps teams have no guarantees that such program terminates its execution without failures, leading to scenarios where adaptation actions are enacted without actually reaching the expected outcome. Thus, it becomes crucial to devise reconfiguration approaches leveraging *correct-by construction* orchestrations, *i.e.,* orchestrations successfully ending without failures, that guarantee to instil the intended goal within the system. Moreover, having the guarantee that an orchestration successfully ends, makes it possible to devise more complex scaling approaches capable of replicating the whole architecture simultaneously. Architectural reconfiguration, as we will see, paves the way to new scaling approaches overcoming the state of the art, *i.e.,* the Kubernetes HPA algorithm. As a matter of fact, the mainstream horizontal autoscaling approach (the one adopted by the HPA) suffers from an inherent problem: as it primarily concentrates on replicas at the level of individual services, it encounters the "domino effect" of uncoordinated scaling, *i.e.,* individual services scaling one after the other (cascading slowdowns) due to localised workload monitoring [HFG+19].

The scenario depicted so far is just one side of the coin: we only focused on interaction among services. As a matter of fact, unexpected events can also occur at a lower level, within interactions among the components of a single service. These

unexpected events lead to uncaught misbehaviours caused by, *e.g.,* dereferencing null pointers [Hoa09] or using objects wrongly (reading from a closed file, closing a socket that timed out [2], etc...). Despite CICD practices encompass a set of testing framework for code functionality, they are inadequate at guaranteeing the adherence to an expected behaviour. The reason lies in the lack of high-level structuring abstractions for describing component behaviour, which, as we will see in this Dissertation, are crucial to enhance static analyses and prevent misbehaviours. As put by Dijkstra [Dij72]:

> program testing can be used to show the presence of bugs, but never to show their absence.

To corroborate Dijkstra's statement, practical experience has shown that these problems are, not only often subtle and difficult to find, even with the aid of automated testing procedures, but they can also have severe consequences. The detection and resolution of these problems can be time-consuming and costly. DevOps teams must spend valuable time analysing code, debugging problems and re-executing CICD pipelines, disrupting service availability. To further exacerbate the situation, such errors often manifest as system malfunctions, leaving behind a trail of exposed sensitive data and potential security breaches that could have devastating consequences for businesses, including financial losses, reputational damage and legal liability.

Summarising what we depicted so far, we identify the following three challenges as leading problems of our research.

- challenge `C1` concerns the evaluation of system behaviour as a whole in the early stages of software development, *i.e.,* at modelling level, fostering a development approach where DevOps teams can analyse the consequences of their choices early on;

- challenge `C2` aims at introducing architectural reconfiguration approaches, leveraging *correct-by construction* orchestrations. Such orchestrations are built upon declarative specifications of, *e.g.,* component requirements and deployment constraints. Moreover, the knowledge of components behaviours

[2] `https://github.com/redis/jedis/issues/1747`.

is crucial to overcome the drawbacks of the state of the art of service-level adaptation, *i.e.,* the approach proposed by the HPA, avoiding the "domino effect" of uncoordinated scaling;

- challenge `C3` focuses on endowing CICD practices with the ability of catching misbehaviours due to wrong internal service interactions, *e.g.,* dereferencing null pointers or using objects wrongly.

The obtained research results in this Dissertation could be used, not only in the context of the challenges highlighted above, but also in other areas, *e.g.,* IoT.

Goal of the thesis. This Dissertation addresses challenges `C1`, `C2` and `C3` by employing high-level structuring abstractions for complex behaviour, enabling the decoupling of desired behaviour from its implementation. To achieve this, we leverage the formalisation of behavioural description of components, *e.g.,* required resources, dependencies, method invocation order, interactions and other relevant aspects. As we will see, such formalisation makes it possible to, not only perform automated analyses ensuring components behave as desired, but also improve the state of the art of autoscaling approaches. In conclusion, the following constitutes the thesis statement: *elevating flat SOA component descriptions to formal behavioural specifications is the first step towards reliable and efficient systems.*

Besides the above challenges per se, in the context of this Dissertation, we also solved all the technical issues, making our solutions, as highlighted above, applicable to a wider range of areas.

1.2 Outline of the Dissertation

The remainder of this Dissertation is divided into seven main Chapters, each addressing distinct motivations and issues that underpin our investigations, culminating in the presentation of our findings. These results, generated throughout the course of the PhD studies, are now showcased in an expanded format to provide comprehensive insights. The Dissertation is structured as follows:

- in Chapter 2, we lay a solid foundation for this Dissertation, introducing the

core concepts and frameworks that underpin our approach to ensuring reliable and secure component interactions in component-based architectures;

- in Chapter 3, we introduce a novel integrated timed modelling/execution language tailored for microservice systems. This language enables the anticipation of system functioning evaluation during the system design phase, thereby fostering an approach that enhances system scalability, reliability and performance. This novel integrated language addresses challenge `C1`;

- in Chapter 4, we introduce two novel orchestration-based architectural reconfiguration techniques. Our techniques follow different approaches to architectural reconfiguration: service migration and service autoscaling. In particular, we leverage *correct-by construction* deployment orchestrations to reconfigure an architecture. In particular, our technique for service autoscaling overcomes the drawbacks of the state of the art of service-level adaptation. Straightforwardly, our orchestration-based architectural reconfiguration techniques address the needs introduced by challenge `C2`;

- in Chapter 5, we present the current state of the art of typestate-based type checking tools and theory. Here, we discuss our theoretical work to overcome existing limitations, *i.e.,* typestate-based analysis of polymorphic code, support for arrays of typestate-endowed objects, and the implementation of these concepts in a typestate-based type checker for Java (JaTyC). In particular, such theoretical work makes typestate-based analysis applicable to real-world scenarios, *e.g.,* CICD pipelines, thus addressing challenge `C3`;

- in Chapter 6, we formally define a subset of the Java programming language combined with the typestate-based elements presented in Chapter 5. We then present a formal representation of the type checking process entailed by JaTyC (introduced in Chapter 5), designing all the type checking rules required to analyse language constructs. Together with Chapter 5, this Chapter addresses challenge `C3`, formally supporting the existing typestate-based type checking process;

- in Chapter 7, we summarise the contributions and conclude the Dissertation.

Chapter 2

Background

This Chapter lays a solid foundation for the Dissertation, introducing the core concepts and frameworks that underpin its approach to ensuring reliable and secure interactions in component-based architectures.

We start exploring the concepts of behavioural types and subtyping, providing a comprehensive understanding of these mechanisms and their critical role in capturing and reasoning about the dynamic behaviour of software components. Building on this theoretical foundation, we turn our attention to the Checker Framework [Con23], a robust and widely-used open-source tool, designed to augment the Java type system. By providing a suite of pluggable type checkers, the Checker Framework empowers developers to define and enforce custom type rules tailored to their application requirements.

The discussion moves toward an actor-based object-oriented language, suitable for modelling complex distributed systems. In particular, we analyse in depth the Abstract Behavioral Specification (ABS) language [JHS+12], a formal means to precisely describe component behaviour for automated analysis, verification and evaluation.

Then, our attention pivots toward microservice architectures within the broader context of Service-Oriented Architectures (SOAs). Consequently, we explore the unique characteristics of microservice architectures and their implications on component interactions. To tackle the complexities of managing and orchestrating microservices, we introduce foundational technologies tailored for containerised

applications, *e.g.,* Docker and Kubernetes. These tools furnish a robust platform for deploying, scaling and managing microservice-based systems, thereby facilitating efficient and dependable operations within cloud-native environments.

2.1 Behavioural Types and Subtyping

The design of programming languages demands meticulous attention to error prevention. A set of prohibited errors, encompassing both untrapped and trapped ones, forms the cornerstone of well-behaved programs. By meticulously eliminating these errors, programs, not only operate safely, but also reduce debugging time and minimise the occurrence of unexpected behaviours. Traditionally, type systems have concentrated on validating computation results, ensuring they conform to some predefined specifications. However, with the emergence of newer and more complex systems, the need for a more comprehensive and structured approach, also encompassing the behaviour of computations, has become paramount.

Behavioural contracts/types [LP07, BZ07, GH05, CDSY17, KDPG18] emerge as a powerful solution to this challenge. They provide a formal framework for specifying and analysing the behaviour of components, transforming flat component descriptions into a graph structure of invoke/receive actions. This formalisation facilitates automated analyses, enabling the verification of component interactions and the guarantee of critical properties, *e.g.,* deadlock freedom and the absence of null-pointer exceptions. In the context of service-oriented computing, where components create an intricate and dynamic network of interactions, behavioural contracts/types offer unparalleled insights into the behaviour of these interactions. They enable developers to reason about the behaviour of components, ensuring that their interactions are consistent, predictable and safe. The adoption of behavioural contracts/types promotes a programming language design that gives priority to, not only the correctness of computation results, but also the integrity of component behaviour, paving the way for more reliable, secure and maintainable software systems.

Typestates. To understand the notion of *typestate*, let us consider the following scenario: a programmer defines a type, *e.g.,* Box, that offers two operations: in,

which sets the item inside the Box, and out, which extracts such item (and consequently removes it). Notice that, the in operation never fails: in case the item is already set, it just updates the item; the out operation, instead, fails, in case no item is set. As the perceptive reader may notice, the Box type has an invariant property: sequences of out operations are prohibited. If the programmer is not careful enough and calls a sequence of out operations, the program compiles, but it exhibits unexpected behaviours.

Following the definition of Strom *et al.* [SY86], the above program is defined as *nonsensical*. The term "nonsensical programs" refers to syntactically well-formed programs with a semantically undefined sequences of statements. Nonsensical program executions pose one of the most insidious forms of software errors. Their unpredictable consequences can wreak havoc, making them particularly challenging to identify and rectify. The underlying erroneous state can remain undetected for extended periods, only to surface later in the behaviour of seemingly unrelated programs, often far removed from the exact statement where the error initially occurred. A compile time mechanism that proactively identifies and prevents nonsensical program executions holds immense value. Beyond simply detecting errors, such a mechanism effectively encapsulates their impact within the erroneous module, minimising the potential for cascading failures and ensuring program integrity. The *typestate* concept is the key to solve the problem.

Typestate is a notion of behavioural types, introduced as refinement of the concept of type in programming languages. Typestates are associated to types and capture the notion of an object of a given type being in an appropriate (or inappropriate) state for the application of a particular operation. Each typestate includes a set of operations that can be safely executed in that state. An object of a given type is, at each point in a program, in an exactly single typestate among the ones associated with its type [SY86]. Therefore, typestates can be considered as finite-state automata, where the application of an operation may or may not cause typestates to change. For example, typestates of the Box type, enforcing the invariant property described above, can be modelled as in Figure 2.1: typestate Empty only allows for the in operations (here, this operation causes the typestate to change), while typestate NonEmpty allows for both in (here, this operation does not cause the typestate to change) and out operations.

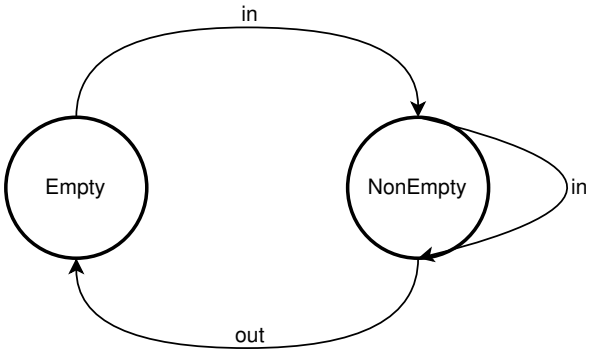


Figure 2.1: Typestates of Box type

The concept of typestate paved the way for the typestate-oriented programming [GTWA14], an extension to the object-oriented programming paradigm, where objects are, not only modelled in terms of classes, but also in terms of changing states. In typestate-oriented programming languages, typestates are treated as a primitive language concept making it possible to perform a supplementary static analysis via typestate checkers. These checkers serve as an additional layer of validation atop an existing programming language, effectively identifying and preventing "nonsensical programs" from entering the codebase.

Session Types. To grasp the concept of *session types*, let us consider the following scenario, taken from Gay and Hole [GH05]: a mathematical server and a client communicating with it. The server offers two mathematical operations, using and returning basic types, *i.e.,* Int and Bool: (*i*) addition of *two* integer numbers; and (*ii*) equality test between *two* integer numbers. The server runs in parallel with a client, which selects among the offered operations. Let us suppose that Alice tries to communicate with this mathematical server, following the protocol informally described in Figure 2.2 [1]. As the keen reader may notice, something catastrophic is going to happen: both Alice and the mathematical server go into deadlock, since the former is waiting for the result of addition, while the latter is waiting for another input, *i.e.,* the second number. Deviations from intended communication behaviour can lead to critical issues, such as the potential occurrence of deadlocks.

[1] In the communication protocol, we do not consider the connection step.

Alice	Mathematical server
1. Select addition.	1. Wait for operation selection.
2. Send an integer number.	2. Wait for number.
3. Wait for the result.	3. Wait for number.
	4. Send result.

Figure 2.2: Communication with the mathematical server

The above scenario highlights the urgent need for a mechanism to formally define and verify these communication protocols, ensuring adherence to the specified sequences of actions and bolstering the reliability of distributed systems. The *session type* concept is the key to solve the problem.

Binary session types[2], here simply referred to as session types, is a typing discipline for communications programming, based on connection-oriented interaction sessions. In networking, a session is a logic unit of information exchange between two or more communicating parties [HVK98, THK94]. The essential concern of a session is to specify the topic of conversation as well as the sequence and direction of the communicated messages [Dd09]. A session type can be considered a formal specification of a two-party communication protocol from the perspective of one party. Session types are defined as a sequence of input and output operations, explicitly indicating the types of messages being transmitted. This structured sequentiality of operations is what makes session types suitable to model communication protocols. However, they offer more flexibility than just performing inputs and outputs [DGS17]: meta-communications for the coordination of branching (*i.e.,* the offering of a set of alternatives), selection (*i.e.,* the selection of one of the possible options at hand) and repetition within a session. We represent the mathematical server $\mathsf{S_{Int}}$ and Alice (in its correct versions) session types as:

$$\mathsf{Alice} : \oplus\{\mathsf{addition} : !Int.!Int.?Int.end, \quad \mathsf{equality} : !Int.!Int.?Bool.end\}$$
$$\mathsf{S_{Int}} : \&\{\mathsf{addition} : ?Int.?Int.!Int.end, \quad \mathsf{equality} : ?Int.?Int.!Bool.end\}$$

[2]For simplicity sake, multi-party session types are not considered in this Dissertation.

Notice that, **?** and **!** represent session type input/output operations, indicating the process is the receiver or the sender, respectively. Instead, **&** and $\oplus$ indicate a *branch* of choices, *i.e.,* a set of alternatives, and *selection*, of a specific choice, *i.e.,* the selection of one of the possible options at hand, respectively. In our specific example, the protocol of Alice states that she can choose between addition and equality, passing as input two integer numbers and receiving an integer number or a boolean value, respectively. The server is prepared to handle addition and equality, expecting as input two integer numbers and producing as output an integer number or a boolean value, respectively.

Understanding the dynamics of communication protocols requires a mechanism that captures the reciprocal roles of interacting participants. The ideas of session types and co-types (session type *duality*) for interaction were first introduced by Honda [Hon93]. Intuitively, session type duality captures the opposite role or perspective in the communication: if there is a session type that defines the expected behaviour of a sending participant in a communication, its dual corresponds to the expected behaviour of the receiving participant and vice versa. Given the session types above, it is clear what duality means: whenever Alice selects a choice, the server offers such choice. The type system uses the mechanism of duality to ensure communication safety (error freedom) for a system of session processes. Thus, it is possible to successfully check a program to guarantee that certain kinds of error do not occur at run-time. The eliminated errors range from disagreements between sender and receiver about the expected type of a message [Mil93] to deadlocks [Kob98].

Subtyping. Subtyping plays a crucial role in the context of behavioural types: it allows for flexible and modular protocol design, by introducing the concept of compatibility. The idea of subtyping is that any value of a certain type can be safely placed in a context expecting a value of some more general type [Dd09]. In the context of session types, if T is a subtype of T', then any channel of type T can be safely used in a context where a channel of type T' is expected. The concept of subtyping has been extended to session types by Gay and Hole [GH05]. The input and output types are respectively contravariant and covariant in the types of values they transmit; labelled branches and select types, in accordance

with the I/O actions, are respectively covariant and contravariant in their label sets. Suppose, now, the mathematical server is extended in two directions: (i) it has a new negation service; and (ii) it extends the equality test to Real numbers. The session type for the new server is:

$$\begin{aligned}\mathsf{S_{Up}} : \&\{&\mathsf{addition} : ?Int.?Int.!Int.end,\\ &\mathsf{equality} : ?Real.?Real.!Bool.end,\\ &\mathsf{negation} : ?Int.!Int.end\}\end{aligned}$$

Assuming that Int is a subtype of Real, it is clear that the system still works without communication errors. When sending, the type of the actual message is allowed to be a subtype of the message type specified by the channel, because it is safe for the receiver to be given a value whose type is a subtype of the expected one [GH05]. Moreover, the additional service negation does not interfere with the communication with Alice, since it is never used in the communication. Thus, we say that $\mathsf{S_{Up}} \leq \mathsf{S_{Int}}$, since it is contravariant in output types and covariant in the branch label set: $\mathsf{S_{Up}}$ offers a superset of alternatives with respect to $\mathsf{S_{Int}}$. Subtyping enhances expressivity of typing with session types since it allows: (i) refinement of participants without invalidating type-correctness of the overall system; and (ii) participants to follow different protocols that are nevertheless compatible according to the subtyping relation [Dd09].

2.2 Java Checker Framework

The Checker Framework represents a significant advancement in the landscape of Java development, transcending the role of a traditional type system to actively prevent errors rather than simply detecting them. By leveraging a modular architecture, the Checker Framework extends the language capabilities to identify and eliminate a wide range of potential issues, including null pointer exceptions, unintended side effects, SQL injections, concurrency errors and mistaken equality tests. This proactive approach to code correctness empowers developers to write more robust and reliable software. Furthermore, the Checker Framework allows developers to contribute to its own evolution by creating custom checkers [BBG+22b].

Listing 2.1: Java implementation of type Box

```
1  import org.checkerframework.checker.nullness.qual.Nullable;
2
3  public class Box {
4      private @Nullable Object item = null;
5
6      public void in(@Nullable Object obj)  {
7          this.item = obj;
8      }
9
10     public boolean hasItem() {
11         return obj != null;
12     }
13
14     public Object out() {
15         Object to_return = obj;
16         obj = null;
17         return to_return;
18     }
19 }
```

This feature allows developers to tailor error detection to the specific needs and domains of their applications, effectively defining their own unique safety nets. For instance, the widely adopted Nullness Checker exemplifies this customisability, ensuring the absence of null pointer exceptions, by treating objects and null as distinct types by default [Con23]. Developers have the flexibility to introduce nullable variables using the Nullable annotation, as shown in Listing 2.1, allowing for fine-grained control over potential null references. This example uses the Box type described before, which may store a reference to another object or the null value. The in method stores a new object and replaces the previous one; the out method extracts the object: it returns the currently stored object and nullify the Box reference to such object; the hasItem method checks if item is not null. The Nullness Checker reports an error on line 17, indicating that the item variable may possibly be null. It is a real error that could happen if either out is called before calling in or a sequence of out operations is performed. Unfortunately, there is no way to specify that we expect the nullness of item to be checked before with hasItem, creating a scenario where we need to use a verbose defensive programming style: always check that item is not null inside the out body. This gives an example where detecting data-errors are not enough and motivates the inclusion of behavioural information in types: this would enforce that methods are called

in the correct order and would avoid false positives as in Listing 2.1, by pruning execution paths that do certainly not occur.

2.3 The ABS Executable Specification Language

The Abstract Behavioral Specification (ABS) language, introduced by Johnsen *et al.* [JHS+10], serves as an actor-based framework for crafting executable models of distributed object-oriented systems. Designed to align closely with programmers thought processes, ABS maintains a syntax reminiscent of Java and a control flow akin to real-world implementations. One notable feature is its capability to compile models into various languages such as Erlang, Java and Haskell. Moreover, ABS boasts a formally defined semantics, following the style of foundational languages, empowering developers to abstract away many implementation intricacies, undesired at the modelling level. These may include the concrete representation of internal data structures, method activation scheduling and communication environment properties. ABS operates at the concurrent object level, providing a comprehensive framework to model concurrent control flow and communication within models. This integration seamlessly merges functional expressions, imperative object-based programming and Concurrent Object Groups (COGs) with cooperative scheduling. In this scheduling approach, processes inside actors voluntarily yield CPU control at predefined execution points, promoting efficient resource utilisation and enabling effective coordination among concurrent activities. Conceptually, each COG is associated with a processor, fostering a distributed environment characterised by asynchronous and unordered communications. Objects communicate through asynchronous method calls, facilitating the initiation of activities in other objects, without disrupting the flow of execution. ABS further enriches communication capabilities by treating futures as first-class citizens, thereby enabling flexible and efficient asynchronous communication patterns. Asynchronous method calls play a pivotal role in triggering concurrent activities, initiating new method activations (referred to as processes) within the recipient object. This approach allows for the seamless integration of active and passive behaviours within ABS models. Active behaviour, typically defined by an optional `run` method asynchronously called after object initialisation, complements passive behaviour trig-

gered by synchronous or asynchronous method calls. Consequently, each object maintains a dynamic collection of processes awaiting execution, stemming from method activations. Among these, only one process per COG is active at any given time, with the remaining processes suspended within a process pool. Process scheduling is nondeterministic yet cooperative, regulated by processor release points. Hence, the degree of concurrency correlates directly with the number of COGs introduced in the model [JHS+10]. Consider the following code snippet (see Listing 2.2) as an illustrative example. In this scenario, a Server class listens for requests from clients. If no requests are pending, it waits for new ones (line 10); otherwise, it proceeds to serve the first arrived request (line 11). Subsequently, the server dispatches the received command to an executor (line 15) and awaits its result (line 16) before forwarding the response to the client (line 18). Additionally, the server includes a method to enqueue new requests following a first-in-first-out (FIFO) policy. It is worth noting that, for simplicity, the implementations of Client and Executor classes are abstracted away. The Server class uses asynchronous method calls (line 15), denoted by o!m(), to serve requests, allowing objects of this class to interleave the execution of the run method with the request method. Furthermore, the Server class exemplifies an active behaviour, as the run method is asynchronously executed upon object instantiation, without requiring explicit invocation.

Timed ABS. *Timed ABS* is an extension of the ABS core language that introduces a notion of *abstract discrete time*, expressing the amount of *time units* elapsed since system start. Such an extension makes it possible to evaluate time-related behaviour of distributed systems, by compiling Timed ABS programs into, *e.g.,* Erlang, and executes them. Timed ABS has also *probabilistic* features that allow modellers to create uniform distributions. Timed ABS introduces the notion of Deployment Component as a *location* where a COG can be deployed. Deployment Components are first-class citizens of Timed ABS. They may be passed around as arguments to method calls and they support a number of methods. Deployment Components may be created dynamically, depending on control flow or statically in the main block of the model. Thus, Timed ABS is expressive enough to model that, *e.g.,* new Deployment Components are created by a provider, Deployment

Listing 2.2: Asynchronous call and active behaviour example

```
type Request = Pair<Client, String>;

class Node(Executor executor) implements Server {
    List<Request> requestQueue = list[];

    Unit request(Request req) {requestQueue = appendright(requestQueue, req);}

    Unit run() {
        while(True) {
            await !isEmpty(requestQueue);
            Request req = head(requestQueue);
            requestQueue = tail(requestQueue);
            Client c = fst(req); //first element of the pair
            String command = snd(req); //second element of the pair
            Fut<String> futRes = executor!execute(command);
            await futRes?;
            String res = futRes.get;
            c.response(res);
        }
    }
}
```

Components are requested from a provider by a resource-aware and scalable application. As stated by Johnsen *et al.* [JSTT15], a Deployment Component is "an abstraction from the number and speed of the physical processors available to the underlying ABS program by a notion of concurrent resource". Simply put, a Deployment Component corresponds to a single virtual machine, which executes ABS code. As virtual machines, ABS Deployment Components are associated with several kinds of resources:

- the Speed resource type models execution speed: it models the amount of *computational resource* per time unit a Deployment Component can supply to the hosted COGs. Intuitively, a Deployment Component with twice the number of Speed resources execute twice as fast. Speed resources are consumed when execution in the current process reaches a statement that is annotated with a Cost annotation. In the example presented in Listing 2.3, the skip statement consumes 5 Speed units from the Deployment Component where the COG executing such statement was deployed. If not enough computational resource is left in the current time unit, then the instruction terminates its execution in the next one;
- the Bandwidth resource expresses a measure of transmission speed, consumed

Listing 2.3: Cost annotation example

```
Time t1 = now();
[Cost: 5] skip;
Time t2 = now();
```

during method invocation and return statements. Notice that, no Bandwidth is consumed if sender and receiver are deployed on the same Deployment Component. As can be seen in Listing 2.4, similarly to Speed, Bandwidth consumption is expressed via the DataSize annotation, thus executing o!process consumes 2 * length(datalist) Bandwidth units;

Listing 2.4: DataSize annotation example

```
Time t1 = now();
[DataSize: 2 * length(datalist)] o!process(datalist);
Time t2 = now();
```

- the Memory resource type serves as a proxy for deployment complexity, indicating the capacity for creating and managing COGs within a component. Unlike Bandwidth and Speed, Memory does not directly impact the ABS model simulated timing behaviour;

- the Cores resource type expresses the number of CPU cores available on a Deployment Component. Used for static deployment decisions, it has no direct impact on the timing behaviour of simulations.

Technically, all untimed ABS models are valid in timed ABS: an ABS model that contains no time-influencing statements will run without influencing the clock and finish at time zero.

2.4 Microservices

As for SOAs, microservices preach for the same advantages, such as dynamism, modularity, distributed development and integration of heterogeneous systems.

However, SOAs are built on the idea of fostering reuse, *i.e.,* a *share-as-much-as-possible* architecture style, whereas microservice architectures seconds the idea of a *share-as-little-as-possible* architecture style [BDD+20]: the focus is on replaceability, autonomy, self-management and lightweightness [New15]. Microservices are independently and conceptually deployable components providing a physical module boundary, even allowing for different microservices to be written in distinct programming languages and be managed by various teams. As defined in [DGL+17a, BDD+20], a microservice is a cohesive, independent process that supports interoperability, by communicating through lightweight messages (often HTTP APIs). The term "cohesive" indicates that a service implements only functionalities strongly related to the concern that it is meant to model, following the Single Resposibility Principle: *there should never be more than one reason for a microservice to change.* The *share-as-little-as-possible* and independent deployability properties of microservices make them a single business capability that is delivered and updated independently and on its own schedule. Thus, changes do not have any impact on other microservices and on their release schedule. However, to truly harness the power of independent deployment, one must utilise very efficient integration and delivery mechanisms [DGL+17a]. As a matter of fact, by using automated continuous delivery pipelines and modern containerisation tools, it is possible to deploy an updated version of a service to production in a matter of seconds, which proves to be very beneficial in an open, dynamic and ever changing business environment. To give an example of a microservice architec-

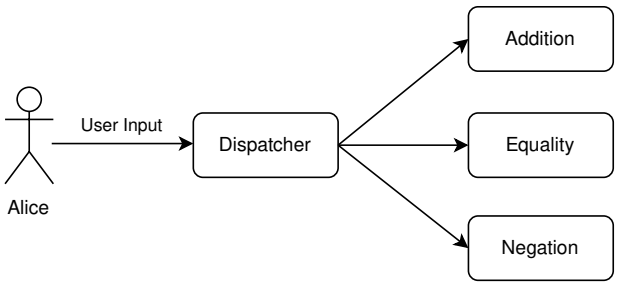


Figure 2.3: Microservice architecture example

ture, let us consider the mathematical server described in Section 2.1. Following the Single Responsibility Principle, as the mathematical server offers 3 functions

(*i.e.,* addition, equation and negation), developers can consider decomposing it into 3 microservices, each handling a single function. As can be seen in Figure 2.3, there is an additional microservice to complete the architecture: the Dispatcher. Such a service implements the logic to check the user input well-formedness and routes requests to the proper microservice. The developers of the architecture above can focus separately on implementing the basic microservice functionalities.

In the context of a microservice architecture, there are two approaches to establishing complex and elaborate communication behaviour among microservices: *orchestration* and *choreography*. Orchestration refers to the coordination and management of multiple microservices to achieve a specific business function or workflow. It involves defining the sequence of actions or tasks that need to be executed across different microservices to fulfill a particular request or process, *e.g.,* autoscaling. Orchestration typically involves a central orchestrator component that controls the flow of communication and data between the microservices, ensuring that they work together harmoniously to achieve the desired outcome. Choreography, instead, refers to a communication style where individual services work independently and coordinate with each other through events. Each service in a choreography understands the overall workflow and responds to relevant events published by others.

2.5 Technologies for Containerised Applications

In the rapidly evolving landscape of modern software development, containerisation has emerged as a foundational technology, revolutionising the way applications are developed, deployed and managed. At the heart of this transformation lie two key technologies: *Docker* and *Kubernetes* (K8s). In the following Section, we explore the inner workings of these technologies, exploring their core concepts, architecture and capabilities in detail. We examine how Docker simplifies the process of containerisation, enabling developers to package and distribute applications with ease. Likewise, we explore how Kubernetes empowers organisations to harness the full potential of containerisation, providing powerful tools for orchestrating and managing containerised workloads in dynamic, cloud-native environments. Through a comprehensive understanding of these technologies, developers

and organisations alike can unlock new opportunities for innovation, agility, and scalability in the ever-evolving landscape of modern software development.

Docker. Docker is an open source container technology with the powerful ability to build, ship and run distributed applications. Although container technologies, *e.g.,* [SPF+07, Fur14, Iva17], have been around for a long time, Docker is currently one of the most successful tool for application containerisation, since it comes with new powerful abilities that earlier technologies did not possess. As a matter of fact, Docker provides interfaces to simply create and control containers, where applications can be run without modification, no matter the underlying hardware. Moreover, Docker cooperates well with orchestration tools, *e.g.,* Kubernetes, which provide an abstract layer of resources management and scheduling over Docker.

The underlying virtualisation solution, called Docker engine, is a lightweight and portable packaging tool. At its core lies the Docker Daemon, a background process responsible for overseeing Docker objects such as containers, images, volumes and networks. Integral to the Docker Engine is the container runtime component, which executes containers, providing the necessary environment for applications to run, independently of the host system. Docker Images, immutable templates containing application code, dependencies and configurations, serve as the building blocks for containers, fostering portability and consistency across different environments. Additionally, Docker Engine encompasses networking and storage drivers, facilitating communication between containers and external resources, as well as ensuring the persistence of container data.

In addition to the foundational components described above, Docker Compose [ISH21] and Docker Swarm [SK16] are integral parts of the Docker ecosystem, extending its capabilities for orchestrating and managing containerised applications at scale. Docker Compose simplifies the management of multi-container Docker applications, allowing users to define complex, multi-service applications in a single YAML file. With Docker Compose, developers can specify the services, networks and volumes required for their application along with their configurations and dependencies. This declarative approach streamlines the deployment process, enabling users to define and launch their entire application stack with a single command. Docker Compose facilitates collaboration and reproducibility, as developers

can share their application configurations in version-controlled YAML files, ensuring consistency across development, testing and production environments. Docker Swarm, instead, provides native clustering capabilities for Docker, allowing users to orchestrate and manage a cluster of Docker hosts as a single virtual system. Docker Swarm simplifies the deployment and scaling of containerised applications across multiple hosts, providing built-in features for load balancing, service discovery and high availability. With Docker Swarm, users can deploy their applications seamlessly, leveraging on its integrated scheduling and orchestration capabilities to distribute containers across the cluster, based on resource availability and constraints. Docker Swarm fosters resilience and scalability, enabling users to scale their applications horizontally, adding or removing nodes dynamically.

Kubernetes. Kubernetes is a powerful container orchestration platform that simplifies the deployment, scaling and management of containerised applications. At its core, Kubernetes leverages container technologies, such as Docker, to encapsulate applications and their dependencies, ensuring consistency across different environments. These containers are organised into pods, the fundamental units of deployment in Kubernetes. Pods are the smallest, most basic deployable objects in the ecosystem, encapsulating one or more tightly coupled containers that share storage and networking. They represent the fundamental units of deployment, enabling developers to create modular and scalable application architectures. A pod can consist of multiple containers that work together to perform a specific task, *e.g.,* a main application container and one or more helper containers (sidecars for logging or monitoring). Each pod in Kubernetes has its own unique IP address, which allows containers within the pod to communicate with each other. Pods are managed using YAML files as the one presented in Listing 2.5. In particular, the description in Listing 2.5 defines a Kubernetes pod, named example-pod, running a Nginx instance. The pod specification includes resource requests and limits for the pod, ensuring efficient resource allocation and management within the Kubernetes cluster. Specifically, the pod requests 64MB of memory and 250m (milli CPUs, *i.e.,* a fraction of a CPU core, where 1000 milli CPUs correspond to one full CPU core), with limits set to 128MB of memory and 500m CPUs. This configuration ensures that the pod has the necessary resources to run effectively, while

Listing 2.5: Example of pod definition

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

preventing it from consuming excessive resources, contributing to the stability and performance of the environment.

Pods are scheduled onto nodes within a cluster, which can include both physical and/or virtual machines, where they share resources and execute application code. The cluster, comprising a control plane (master) and worker nodes, orchestrates the deployment and management of pods across the infrastructure. Within the control plane, the Kubernetes Scheduler plays a pivotal role in orchestrating the deployment of pods onto nodes of the cluster. This component is responsible for evaluating resource requirements, node availability and affinity rules to make intelligent scheduling decisions. It operates through a multi-step process, beginning with filtering nodes based on their capacity to meet the resource requirements of the pod under examination. Following this, it assigns a priority score to each node, based on factors like resource availability and affinity/anti-affinity rules, to control how pods are placed onto nodes based on node properties and constraints. The scheduler, then, selects the node with the highest priority score and binds the pod to it, taking into account considerations such as node locality and fault tolerance. Throughout this process, the scheduler continuously monitors the cluster state and adjusts pod placements as necessary to maintain optimal resource utilisation and application availability [HBB17].

Complementing the Kubernetes Scheduler is the Horizontal Pod Autoscaler (HPA), a feature that enables automatic scaling of single pods based on observed

CPU utilisation or other custom metrics. The HPA dynamically adjusts the number of pod replicas to match the desired user-defined target, ensuring that the application can handle varying levels of traffic and workload. The HPA calculates the desired number of pods according to this formula:

$$desiredReplicas = \left\lceil currentReplicas \cdot \frac{currentMetricValue}{desiredMetricValue} \right\rceil$$

By automatically scaling pods in response to changes in demand, the HPA improves resource utilisation, maintains application performance and reduces operational overhead. This capability empowers organisations to achieve efficient resource management and cost optimisation in dynamic and rapidly evolving environments, ultimately enhancing the agility and scalability of containerised applications deployed on Kubernetes.

Chapter 3

A Modelling/Execution Language for Microservice Systems

This Chapter contains contributions from the following work of ours: [BBG+24b]

Drawing inspiration from service-oriented computing, the microservices architecture revolutionises software systems, facilitating highly modular and scalable compositions of fine-grained, loosely-coupled communicating components [DGL+17b]. These properties align with modern software engineering practices like continuous delivery/deployment [HF10b] and autoscaling [Amaa]. In particular, autoscaling, the ability to dynamically modify the system architecture during execution, is crucially important for addressing adaptation needs, *e.g.,* fluctuating peaks of user requests. This process, composed by a set of actions, instills a precise behaviour within the system. Thus, ensuring that the system behaves as expected, becomes paramount. As these systems comprise numerous interconnected services, even a minor glitch or malfunction in one component can disrupt the entire application. Particularly, in these non-trivial microservice systems, the challenge lies in ensuring, not only correctness of the deployment process, *i.e.,* the deployment of microservices finishes without errors, but also cost-optimal and resource-efficient distribution of components across available virtual machines (VMs). The autoscaling process, if not devised carefully, can become extremely dangerous: (*i*) if scaling policies are not correctly tuned and meticulously evaluated, the system will deploy an incorrect amount of resources, causing either a waste of resources (in case of

overestimations) or a system overloading (in case of underestimations); and (*ii*) if the deployment process encounters errors, services are not replicated and the system will, eventually, be overloaded. Thus, *it is crucial to evaluate system functioning early on at the design phase*, fostering a safer and more efficient system development process. *In the context of this Dissertation* and, more precisely, within this Chapter, we introduce a novel *integrated timed architectural modelling/execution language* [BBG+24b] based on a *timed extension* of the SmartDeployer tool [BGM+19, BGM+20] for the Abstract Behavioral Specification (ABS) language (refer to Section 2.3) [JHS+10].

ABS is an actor-based timed object-oriented language, suitable for designing, verifying and evaluating concurrent/distributed systems. In particular, it allows for modelling and simulation by exploiting its twofold nature: it is both a process algebra (with probabilistic/timed formal semantics) and a programming language (compiled and executed, *e.g.,* via Erlang backend).

SmartDeployer exploits dedicated ABS code annotations expressing *architectural properties* of: the modelled distributed system (global architectural invariants and allowed reconfigurations), its VMs (their characteristics and resources, expressed as properties of Deployment Components representing them) and its software components/services (accounting for architectural dependencies and invariants). During compilation, SmartDeployer verifies the satisfiability of these annotations against the desired target configuration requirements, modelled using the Declarative Requirement Language (DRL) [dMZ19] and architectural invariants. Once validated, SmartDeployer synthesises deployment orchestrations to build the system architecture and its specified reconfigurations. Similarly, it generates undeployment orchestrations to revert these changes. SmartDeployer uses ABS itself as an orchestration language, making deployment and undeployment ABS code accessible as callable methods. Consequently, these methods can be invoked by the ABS code of services, enabling simulated runtime adaptation. By integrating modelling and execution capabilities within a unified language, we proactively address performance concerns during the design phase. This proactive approach enables early analysis of system behaviour, *e.g.,* deployment and scaling decisions, ensuring that potential issues are anticipated before implementation. SmartDeployer performs checks during compilation to verify the feasibility of de-

ployment orchestrations, which are subsequently executed at runtime to guarantee the system ability to achieve the specified reconfigurations outlined through DRL. For instance, in scenarios with fluctuating workloads, such reconfigurations might involve scaling computational resources through service replication. Thus, we ensure that the system remains adaptable to both positive and negative peaks in user requests, while still meeting the defined Quality of Service requirements. SmartDeployer tackles the challenge of synthesizing deployment orchestrations based on a declarative representation of reconfiguration requirements. This task, often referred to as the *optimal deployment problem*, has been demonstrated to be solvable for microservices exclusively [BGM+19, BGM+20]. SmartDeployer provides an interface with ABS, reading ABS annotations with DRL declarations and injecting code of synthesised (un)deployment orchestrations into the initial annotated ABS program. To do this, it relies on a pluggable external solver producing as output the synthesised architectural configuration (cost-optimal distribution of components over the available VMs), which is, then, translated by SmartDeployer into (un)deployment orchestrations expressed as timed ABS code. Notice that, being the solver *pluggable*, Zephyrus2 can be replaced with any other (not necessarily constraint-based) solver, which takes as input a DRL declaration and produces an architectural configuration.

However, when it comes to considering timing aspects, from a technical viewpoint, SmartDeployer implicitly handles them by simply copying Deployment Components (DC) properties from its annotations, *i.e.,* statically assigning the speed and startup_time properties to each DC instance. As we will see later in this Chapter, these static assignments cause SmartDeployer to synthesise deployment orchestrations (ABS programs) that are incorrect from a timing viewpoint. To overcome and correct this limitation, *in the context of this Dissertation*, we introduce the *Timed SmartDeployer* tool [BBG+24a] that fully integrates, also correctly accounting for time aspects, ABS with annotation-based specification of architectural properties. In particular, Timed SmartDeployer, as can be seen in Figure 3.1, generates *timed deployment orchestrations* that also explicitly manage time aspects of the execution: they use ABS timed primitives to dynamically set DC speeds (based on actually used CPU cores) and overall startup time for the architectural reconfiguration. As we will see, the timed features of orchestrations are essential

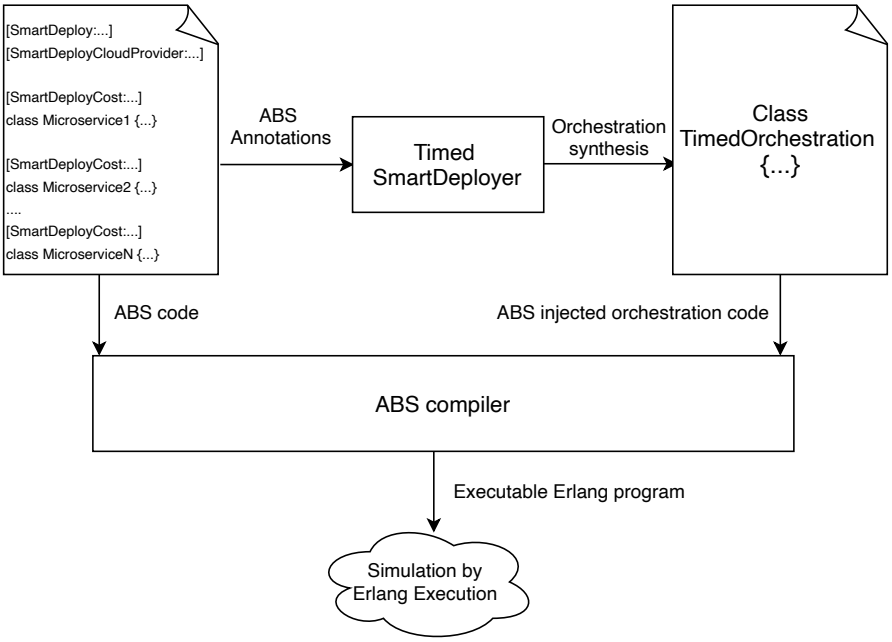


Figure 3.1: Modelling/execution language toolchain

to model, in a throughput-consistent way, adaptation actions.

Finally, we prove the expressiveness of our modelling execution language and its capacity to precisely reproduce the behaviour of real-world microservice systems. In particular, we consider, as a running example, a well-known realistic microservice application, *i.e.,* the *AcmeAir* microservice system, taken from [TS21]. In such an application scenario, we model the system, microservice properties and workload, following the work done by Incerto *et al.* [IPT23]. We simulate the system adaptability under a time-varying sinusoidal load [IPT23] and analyse its behaviour. Concerning the AcmeAir microservice system itself, its model is built by considering static aspects of the architecture (annotations) and ABS code modelling the behaviour of services. The obtained code fully exploits the expressive power of ABS, *e.g.,* using both its timed and probabilistic features.

The Chapter is structured as follows. In Section 3.1, we briefly introduce the approach of [BGM+19, BGM+20] to the automated deployment of microservice applications, while in Section 3.2 we present the AcmeAir microservice architecture that we use as a running example. In Sections 3.3 and 3.4, we describe the SmartDeployer tool and the external solver it relies on, *i.e.,* Zephyrus2. In Sec-

tion 3.5, we present our new Timed SmartDeployer tool and how we use it to consistently model service throughput. In Section 3.6, we present the expressive power of our modelling/executable language, producing an algebraic model of the AcmeAir microservice architecture. In Section 3.7, we prove the reliability and precision in reproducing the behaviour of AcmeAir microservice architecture. Finally, in Sections 3.8 and 3.9 we, respectively, present some related literature and conclude this Chapter.

3.1 Automated Deployment of Microservices

In [BGM+19, BGM+20], Bravetti *et al.* present a formal framework for component-based software systems, addressing the challenge of automated deployment. They define this process as the synthesis of deployment orchestrations, aimed at allocating instances of software components on virtual machines (VMs) to achieve a desired target system configuration. Central to their approach is the formalisation of the deployment life-cycle for each component type, represented as a finite-state automaton. Within this model, each state corresponds to a distinct deployment stage, characterised by a set of *provided ports* (connections exposed by a component for use by others) and *required ports* (connections needed by a component from others to function at that stage). More specifically, Bravetti *et al.* [BGM+19, BGM+20] focus on microservices, a type of component with a deployment life cycle comprising two primary phases. Firstly, the creation phase involves the *mandatory* establishment of initial connections via *strongly required ports/strong dependencies* with other available microservices. Subsequently, the binding/unbinding phase encompasses the establishment of *optional* connections, denoted as *weakly required ports/weak dependencies*, to other available microservices. These two phases facilitate the management of circular dependencies among microservices, offering a structured approach to their deployment.

The concepts of strongly and weakly required ports are, not exclusive to academic research, but also integral to contemporary deployment technologies like Docker Compose [ISH21]. Docker Compose serves as a language for defining multi-container deployments, allowing users to articulate various relations among containers. For instance, users can employ directives such as depends_on or exter-

nal_links to establish different relations between containers. In Docker Compose, these relations can either enforce a specific startup order among containers, akin to strong dependencies, or they can allow for a more flexible startup sequence, resembling weak dependencies. This parallels how the combination of strong or weak dependencies influences the orchestration of microservice deployment, demonstrating the practical relevance and applicability of these concepts in modern and realistic deployment scenarios.

Furthermore, in their work, Bravetti *et al.* [BGM+19, BGM+20] extend their consideration beyond the mere functionality of deployments to encompass resource and cost awareness. This involves modelling memory and computational resources, such as the allocation of virtual CPU cores (referred to as vCores in Azure or simply as virtual CPUs in platforms like Amazon EC2 and Kubernetes) [HBB17]. Specifically, the authors enhance both microservice specifications and virtual machine descriptions, incorporating details regarding the resources they require and provide, respectively. By integrating these resource metrics into their deployment framework, Bravetti *et al.* [BGM+19, BGM+20] enable a more nuanced and efficient orchestration of deployments, taking into account factors beyond strong and weak requirements.

A microservice *deployment orchestration*, formulated within an orchestration language, embodies a structured program with primitives designed to handle two primary operations: (*i*) the creation or removal of specific microservices along with their strongly required bindings; and (*ii*) the addition or removal of weakly required bindings between microservices. This orchestration process is pivotal for transitioning from an initial microservice system configuration to the desired target one. Given an initial microservice system, a set of available VMs and a new target system configuration (corresponding to the set of microservices to be deployed), the *optimal deployment problem* is the problem of finding the deployment orchestration that: (*i*) satisfies core and memory requirements; (*ii*) leads to a new system configuration, where the target microservices are deployed; and (*iii*) chooses the solution that optimises resource usage, if more than one is available. A typical optimisation objective in this context is cost minimisation, wherein the goal is to select the deployment orchestration that minimises the aggregate cost of the virtual machines used for microservice deployment. By formulating and solving this

optimisation problem, stakeholders can effectively manage resource allocation and operational expenses, while achieving the desired system configuration.

While Di Cosmo *et al.* [DCZZ12] demonstrated that permitting components to have arbitrary deployment life cycles renders the optimal deployment problem undecidable, Bravetti *et al.* [BGM+19, BGM+20] countered this complexity by focusing on the simplified life cycle of microservices as described earlier, characterised by the creation and binding/unbinding phases. By narrowing the scope to these two distinct phases, the authors made significant strides in rendering the problem decidable. Specifically, Bravetti *et al.* [BGM+19, BGM+20] introduced a constraint-solving algorithm that effectively addresses the optimal deployment problem within the context of microservices. This algorithm provides a solution comprising the new system configuration, including the microservices earmarked for deployment, their allocation across virtual machines and the necessary bindings to be established among their strong and weak required and provided ports. This approach offers a practical and computationally feasible method for orchestrating microservice deployments, while accommodating the inherent complexities of the system dynamics.

3.2 The AcmeAir Microservice System

In Figure 3.2, we present the AcmeAir microservice system, as outlined in [IPT23]. AcmeAir offers users access to nine distinct endpoints: *Auth, ValidateId, ViewProfile, UpdateProfile, QueryFlights, BookFlights, UpdateMiles, GetRewardMiles* and *CancelBooking*. Following the modelling approach of Incerto *et al.* [IPT23], we adopt the strategy of deploying AcmeAir with the utmost granularity, assigning each endpoint to its dedicated microservice type and making them communicating via simulated synchronous HTTP requests. This granular deployment scheme facilitates independent scaling of individual microservices, as necessitated by workload variations. Each service type is equipped with its dedicated load balancer, which distributes inbound requests among the set of microservice instances, whose number can change at runtime.

Before going into the details of the application of our approach for automated deployment and scaling of microservice applications, as discussed in the subse-

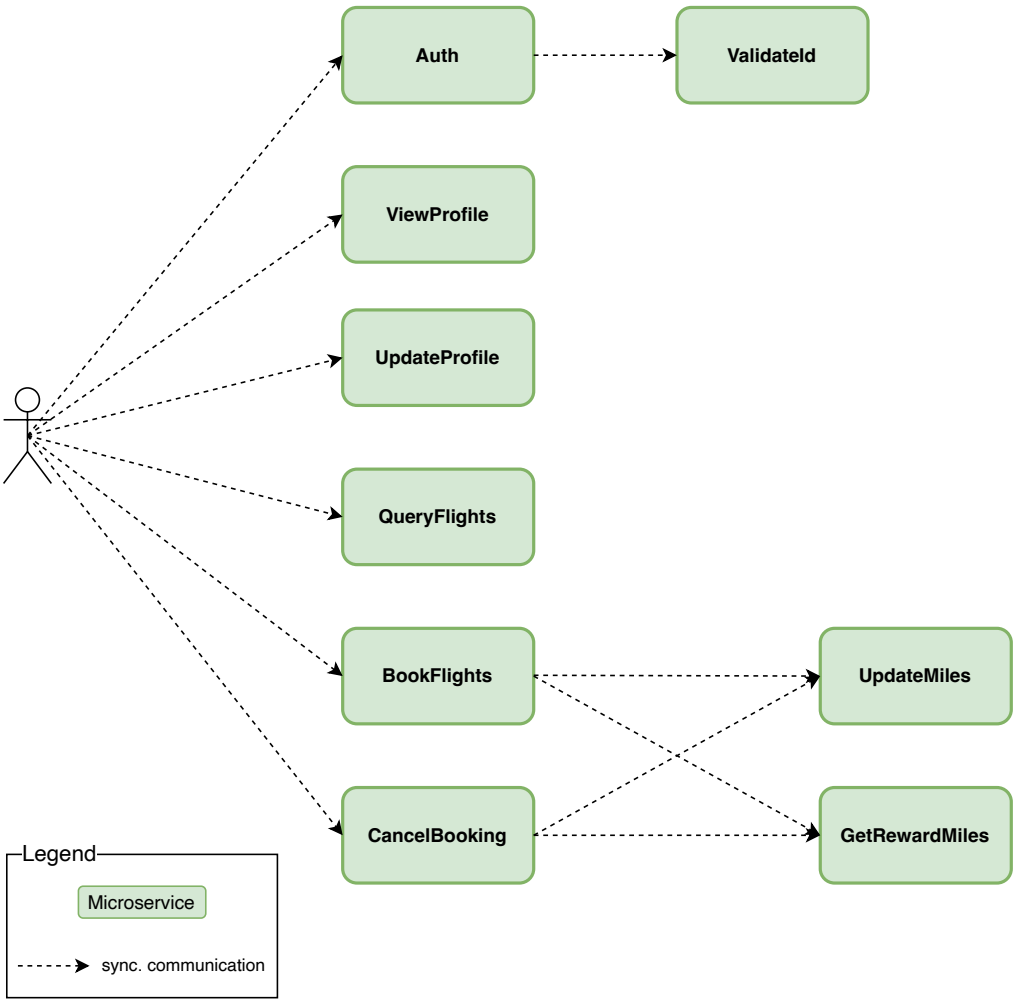


Figure 3.2: AcmeAir microservice architecture [IPT23]

quent Section (see Section 3.6), let us briefly introduce our representation of cloud resources. We adopt a virtual CPU core-centric perspective, considering them both as resources provided by machines and as requirements for microservices. Specifically, in our scenario, we assume microservices to be deployed on Amazon EC2 virtual machines, *i.e., c4.large*, *c4.xlarge*, *c4.2xlarge* and *c4.4xlarge*, each respectively furnishing 2, 4, 8 and 16 virtual CPU cores, referred to as vCPUs in Amazon EC2 (following the terminology of Azure vCores). It is noteworthy that, in our modelling, computational resources supplied by VMs (and demanded by microservices) are represented using virtual cores, each with a fixed speed deter-

mined by the cloud provider. This abstraction serves to decouple the underlying hardware from what users interact with, enabling cloud providers to optimise the utilisation of physical processors efficiently. By delegating the mapping of virtual cores and the scheduling of instructions to the runtime (the VM/OS), providers can maximise hardware utilisation.

3.3 SmartDeployer

In this Section, we introduce the SmartDeployer tool [BGM+19, BGM+20]. This tool operates during ABS compile time, statically addressing the optimal deployment problem outlined at the conclusion of Section 3.1. Its primary function is to synthesise deployment orchestrations capable of achieving a specified target system configuration. SmartDeployer derives its input from dedicated ABS annotations embedded within the compiled ABS program. Subsequently, it generates its output as ABS code, specifically executing deployment or undeployment actions. These generated orchestrations are seamlessly integrated into the initial annotated ABS program, enhancing its functionality and ensuring the attainment of the desired system configuration.

SmartDeployer ABS annotations. The JSON based ABS annotations from which SmartDeployer extracts its input are:

- [SmartDeployCost : JSONstring] class annotation. This annotation is associated with an ABS class that embodies a specific microservice type. Within this annotation, various attributes are delineated, including the functional dependencies such as provided ports, as well as weak and strongly required ports. Additionally, it encompasses specifications regarding the computational resources required by the microservice, such as the number of cores and the amount of memory needed for its operations;
- [SmartDeployCloudProvider : JSONstring] global annotation. The annotation delineates essential properties related to virtual machines hosting microservices, including, *e.g.,* Cores, Bandwidth, Memory, Speed and StartupTime. Additionally, it encapsulates information regarding the cost-per-hour associated

with the creation and utilisation of DCs within the orchestrated deployment execution;

- [SmartDeploy : JSONstring] global annotation. It specifies the desired properties and constraints that should be adhered to within the deployment orchestration. These may include:

  – the id property, which sets the name for the class that is going to include the ABS code of the synthesised orchestration;

  – the cloud_provider_DC_availability property, which fixes the maximum number of VMs the orchestration can allocate.

  Some of these properties may be represented as JSON values, where the content is a declarative specification. This specification typically comprises a formula expressed in a language grounded in first-order logic. For instance:

  – the specification property, which contains the declarative specification of the desired configuration we want to reach, in DRL. A value for this property (taken from our running example) can be:

```
AuthLoadBalancer = 1 and Auth = 1 and
forall ?x in DC: (?x.AuthLoadBalancer > 0
  impl (sum ?y in obj: ?x.?y) = 1)
```

  declaring that 1 instance of AuthLoadBalancer and Auth services must be deployed and that, if a virtual machine hosts the AuthLoadBalancer service, such service must be deployed alone, *i.e.,* that virtual machine does not host other services;

  – the bind preferences property, which is used to specify preferences about *weak* bindings among service instances (again, using the DRL language). A value for this property (taken from our running example) can be:

```
    forall ?x of type Auth in '.*' :
      forall ?y of type AuthLoadBalancer in '.*' :
        ?x used by ?y
```

  meaning that each instance (variable ?x) of the Auth service has to be bound to each AuthLoadBalancer service instance (variable ?y). Given

that there exists only 1 instance of the AuthLoadBalancer service in the system, this just means that each instance of the Auth service has to be bound to its load balancer. More precisely, the in keyword is used to set the scope for the considered services. In this case, the considered ones are only those located inside the DCs whose names are declared after in. Such a declaration can be made with a regular expression like $'.*'$ (meaning any string), *i.e.,* the service can be located in any DC.

Synthesised (un)deployment orchestration. The SmartDeployer tool generates the desired *(un)deployment orchestration* as output, which is essentially an ABS program. This program is injected into the initial annotated ABS codebase, augmenting it with a set of instructions expressed in the orchestration language, *i.e.,* ABS itself. Upon execution, this newly synthesised timed orchestration drives the system towards the deployment/undeployment of a configuration that satisfies the specified properties and constraints. Notice that, being such an orchestration automatically synthesised via a constraint-solving technique, it is *correct-by construction* and guarantees the correct deployment of the desired microservices.

Internal details. As elaborated earlier, SmartDeployer seamlessly integrates with ABS by interfacing with ABS annotations containing DRL declarations and incorporating the generated (un)deployment orchestration code into the original annotated program. To achieve this functionality, SmartDeployer exploits a pluggable external solver, *i.e.,* the Zephyrus2 constraint solver [ÁCJ+16], which facilitates the synthesis of deployment orchestrations based on the specified constraints and requirements. The external solver produces as output the synthesised architectural *configuration* (cost-optimal distribution of components over the available VMs), which is, then, translated by SmartDeployer into (un)deployment orchestrations expressed as ABS code.

3.4 The Zephyrus Deployment Engine

In Section 3.3, we described how SmartDeployer extracts deployment information from ABS code. This information encompasses two main categories. The *class an-*

notations, which articulate the requirements of objects representing the resources and dependencies of microservice instances. These annotations elucidate the essential characteristics necessary for the proper functioning of the microservice instances. The *global annotations*, which outline the available computing resources and the desired deployment properties. These properties serve as guidelines for the deployment process, dictating the allocation of microservices across the available computing resources. The deployment engine undertakes the processing of these annotations, thereby automating the synthesis of a microservice architecture. This synthesis involves strategically allocating various microservices onto the available computing resources. Notice that, this allocation process is meticulously executed, considering both local factors (such as individual microservice dependencies) and global constraints (such as the minimisation of total allocated resources).

The deployment engine, which is currently used in SmartDeployer (and its timed version), is Zephyrus2 [ÁCJ+16]. Zephyrus2 is a tool for the optimal deployment of software components over virtual machines that exploits SMT (Satisfiability Modulo Theories) and CP (Constraint Programming) technologies. More precisely, Zephyrus2 expects in input three different kinds of deployment information:

- a description of the components that can be deployed (which includes the consumed computing/memory resources as well as the functionalities required/provided from/to other components);
- a description of the virtual machines where components can run (including offered resources and other information, *e.g.,* their cost);
- the specific requirements on the component-based software architecture to be computed and deployed over the available virtual machines.

The last item could also include objective functions to optimise, *e.g.,* the request to minimise the total cost of the used virtual machines. Zephyrus2 produces as output a description of the components to deploy, their allocation over the available virtual machines and their bindings that reciprocally require/offer functionalities. The computed deployment configuration satisfies the constraints specified as input.

Zephyrus2 computes its output as a solution to an optimisation problem encoded in MiniZinc [NSB+07], a solver independent language for modelling constraint satisfaction and optimisation problems. The interested reader can find in [ÁCJ+16] details about how Zephyrus2 produces the MiniZinc specification of the deployment problem and how it exploits state of the art tools to solve such problem. Here, we simply give an idea of how the translation of deployment requirements into MiniZinc constraints work, presenting a couple of simple examples.

As a first example, we consider the allocation of memory to the components. Let us consider the constraint

$$\bigwedge_{v \in VM} \sum_{C \in CompTypes} \mathsf{inst}(C, v) \cdot C.mem \leq v.mem$$

where VM denotes the set of all the available virtual machines, $CompTypes$ the possible component types, $\mathsf{inst}(C, v)$ the number of component instances of type C on the virtual machine v, $C.mem$ the memory consumed and $v.mem$ the memory available. This constraint ensures that we cannot allocate an amount of components such that the memory required is more than the available one.

As an additional example, we show how it is possible to require the deployment to minimise the total cost. The constraints enforcing that all virtual machines v host at least one component (bounding used(v) to be true) is expressed as follows:

$$\bigwedge_{v \in VM} \Big(\sum_{C \in CompTypes} \mathsf{inst}(C, v) > 0 \Big) \Leftrightarrow \mathsf{used}(v)$$

To minimise the total cost, we can minimise the following objective function:

$$\min \sum_{v \in VM,\, \mathsf{used}(v)} v.cost$$

where $v.cost$ is the cost of the virtual machine v.

3.5 Timed SmartDeployer

Unlike SmartDeployer, its timed extension produces deployment orchestrations that additionally encompass dynamic management of overall Deployment Component (DC) speed and startup time (computational resources per time unit, see Section 2.3), based on the number of DC virtual cores that are actually used by some microservices, after enacting the synthesised deployment orchestration [BBG+24a]. As a matter of fact, the original SmartDeployer implicitly handles timing aspects by simply copying DC properties from annotations, causing static assignments of speed and startup time to each DC instance. The first causes microservices, deployed in a DC with unused cores, to unrealistically proceed faster, as if they could exploit the computational power of unused cores. The second causes the overall startup time to be the *sum* of that of individual DCs (since in the orchestrations DCs are sequentially created). In particular, the solution to the speed problem is to dynamically evaluate, during the orchestration execution, the number of cores actually being used and adjust the speed to: speed - core_speed · unused_cores. The solution to the startup time problem is to dynamically set such a time to the *maximum* of DCs startup time. The above is realised automatically synthesizing timed orchestrations, whose language, *i.e.,* the ABS language, additionally includes (with respect to untimed ones) two primitives *explicitly* managing timing aspects

- one to decrement the speed of a DC: *decrementResources(...)*;
- one to set overall the startup time of created DCs: *duration(...)*.

Solving the speed problem is fundamental: such dynamic management, based on cores actually being used, guarantees that microservices will always access the same amount of VM speed, no matter where it is deployed. To fully understand the importance of accounting for timing aspects during orchestration synthesis, let us consider the following scenario. We set the an ABS time unit to be 1 second and VMs to supply 5 speed_per_core (encoded with speed_per_core()). According to the empirical measurements of Incerto *et al.* [IPT23], the throughput of an actual implementation of the *QueryFlights* service is 17.6 (encoded with query_tput()) requests per second. In the ABS code, to model service throughput, we make use of the *Cost* instruction tag (see Section 2.3). *E.g.,* in the case of the *QueryFlights*,

which requires 6.498 CPU cores (recall, 1 CPU core is equal to 1000 millicores, thus 6.498 CPU cores corresponds to 6498 millicores) to be deployed, we obtain the throughput of 17.6 req/s as presented in Listing 3.1, where the method request(...) is executed at each request.

Listing 3.1: *QueryFlights* implementation

```
class QueryFlights(...) implements QueryFlightsInterface {
  Result request(Rat start) {
    [Cost: speed_per_core() * time_unit_to_sec() * 6498 / query_tput()] skip;
    Rat stop = timeValue(now());
    prometheus!push("latency", "QueryFlights", stop - start);
    prometheus!push("completed", "QueryFlights", 1);
    balancer!removeMessage();
    return Success;
  }
}
```

Due to our SmartDeployer timed extension, the amount of VM speed used by *QueryFlights* is always $5 \cdot 6.498$ ($\mathsf{speed\_per\_core} \cdot \mathsf{cores\_required}$), independently of the VM where it is deployed, *i.e.,* *QueryFlights* can use up to $5 \cdot 6.498$ computational resources per time unit. The *Cost* tag above causes each request to consume $\mathsf{speed\_per\_core} \cdot 6.498 \cdot 1/\mathsf{throughput}$ computational resources (recall 1 ABS time unit corresponds to 1 second). Therefore, since $\mathsf{throughput}/1$ is the *QueryFlights* throughput expressed in requests per time unit, this realises the desired (deployment independent) service throughput.

3.6 Modelling the AcmeAir System

We employ our modelling and execution language to comprehensively analyse the performance of the AcmeAir architecture, as illustrated in Figure 3.2. Our modelling approach extends beyond merely capturing the static elements of the case study architecture (modelled as ABS annotations). As a matter of fact, it also incorporates dynamic facets, integrating automatic generation of deployment orchestrations (from static annotations), the code representing microservices and their adaptive behaviour in response to varying inbound workloads. Regarding microservices, we adopt a modelling approach where each type is represented as an ABS class, with each object of that class functioning as a replica. In our

Microservice	**Throughtput (req/s)**	**CPU Cores**
Auth	9.6	3.650
ValidateId	19.1	1.877
ViewProfile	10.7	3.582
UpdateProfile	8	2.734
QueryFlights	17.6	6.498
BookFlight	9.8	3.155
UpdateMiles	36.3	5.096
CancelBooking	13.7	2.032
GetRewardMiles	31.4	4.380

Table 3.1: AcmeAir microservice throughput

modelling schema:

- each class incorporates the *Cost* annotation (see Sections 2.3 and 3.5), which is crucial to model the microservice throughputs detailed in Table 3.1;
- communication between microservices is modelled as simulated HTTP synchronous requests, as in [IPT23].

The AcmeAir ABS algebraic model is publicly available at [Bac24c].

3.6.1 Automated Deployment of the AcmeAir System

Even if the AcmeAir system is composed by few microservices, they have a quite complex network of dependencies, *e.g.,* each load balancer strongly depends on exactly one instance of the microservice type it handles, the *BookFlights* service depends on the *UpdateMiles* and *GetRewardMiles* load balancers. Notice that, the strong dependency from a load balancer to exactly one instance of the microservice type it handles is needed to ensure that load balancers have always at least one instance connected (consequently, our modelling schema avoid deploying useless load balancers). To ensure the correct initial deployment and functioning of the system, we use our Timed SmartDeployer tool to automatically synthesise a deployment orchestration capable of ensuring the attainment of our goal. To this purpose, we declaratively define the specification of the configuration we want to

reach as follows. We first specify, for each load balancer, their strong dependencies using the SmartDeployCost annotation presented in Section 3.3. An example of strong requirements can be found in Listing 3.2: the declarative specification states that the *CancelBooking* load balancer has strong dependencies towards an instance of the microservice type it handles and the Prometheus service.

Listing 3.2: *CancelBooking* load balancer strong requirements

```
sig : [
  {kind : require, type : "PrometheusInterface"},
  {kind : require, type : "CancelBookingInterface"}
]
```

Notice that, having load balancers with a strong dependency on exactly one instance of the microservice type it handles together with the already existing architectural strong dependencies (see Figure 3.2), *e.g., BookFlights* depending on the *UpdateMiles* and *GetRewardMiles* load balancers, allow us to create a simpler declarative specification (used by Timed SmartDeployer to synthesise the orchestrations for the initial system deployment). As a matter of fact, as can be seen in Listing 3.3, such specification does not explicitly include all the system components, *e.g.,* it misses the *ValidateId*, *GetRewardMiles* and *UpdateMiles* services, but, instead, it just requires the specification of some load balancers. Nonetheless,

Listing 3.3: Instance requirement specification

```
  specification :
    "BookFlightsLoadBalancer = 1 and QueryFlightsLoadBalancer = 1 and
        CancelBookingLoadBalancer = 1 and
     UpdateProfileLoadBalancer = 1 and AuthLoadBalancer = 1 and
         ViewProfileLoadBalancer = 1"
```

starting from the declarative specification in Listing 3.3, the synthesised deployment orchestration installs all the services depicted in Figure 3.2 and their load balancers. The reason is the following. All the load balancers have a strong dependency towards an instance of the microservice type they handle. Thus, to satisfy these requirements, Timed SmartDeployer deploys the load balancers and additionally installs an instance of the microservice type they handle (even if not explicitly specified). Given the architectural topology depicted in Figure 3.2 other strong requirements come into play, *e.g.,* the *Auth* strongly depends on the load

balancer of the *ValidateId* service. Thus, again, to satisfy such requirement, Timed SmartDeployer deploys the *ValidateId* load balancer, additionally installing an instance of the *ValidateId* service (even if not explicitly specified).

Notice that, in case we deploy more than one instance per microservice type, we need to ensure that each such instances are connected to their load balancers. To this purpose, we leverage the weak requirements to model such connections, expressing them as bind preferences (using the same logical expression as the one presented in Section 3.3).

3.6.2 The Local Scaling Algorithm

To endow the AcmeAir system with the ability to adapt to time-varying workload, we design, using our timed integrated modelling/execution language, the algorithm used by the Kubernetes Horizontal Pod Autoscaler (see Section 2.5). In particular, each microservice (type) has a dedicated monitor and it is locally replicated, creating new instances every time scaling needs are detected. The monitor code, presented in Listing 3.4, works as follows. We use a scaling condition on monitored

Listing 3.4: HPA algorithm ABS implementation

```
Rat total_req = await prometheus!getV("total request", serviceName);
Rat workload = total_req / monitoring_window();
Rat base_throughput = lookupUnsafe(serviceThrougputs(), serviceName);
Rat instances = await prometheus!getV("instances", serviceName);
Rat supp = base_throughput * instances;
if ((workload + k_big()) - supp > k() || supp - (workload + k_big()) > k()) {
    Int target = ceil(float((workload+ k_big())/base_throughput));
    if(target > instances) this!scaleUp(target - instances);
    else if(target < instances) this!scaleDown(instances - target);
}
```

inbound workload involving two constants[1] called K, to leave a margin under the guaranteed service throughput and k to prevent sequences of scale up and down actions. The condition for scaling up is $(inbound\_workload + K) - throughput > k$ and the one for scaling down is $throughput - (inbound\_workload + K) > k$. The algorithm first applies the above scaling conditions, with base_throughput being the microservice throughput and instances the number of deployed instances (initially

[1] In Listing 3.4, these constants are represented as functions returning a fixed value.

set to baseN, *i.e.,* the initial number of deployed instances) and updated in case of scaling needs. Following the Kubernetes mathematical calculation, we have

$$desiredReplicas = \left\lceil currentReplicas \cdot \frac{currentMetricValue}{desiredMetricValue} \right\rceil.$$

In our algorithm, we set $currentReplicas = instances$, $currentMetricValue = \frac{workload+K}{instances \cdot base\_throughput}$ and $desiredMetricValue = 1$ (meaning that the supported workload is at least equal to the inbound one). Thus, simplifying the Kubernetes formula, we get

$$\left\lceil \frac{workload + K}{base\_throughput} \right\rceil,$$

which is implemented in Listing 3.4 (see line 7), where the function $\lceil x \rceil$ (encoded as ceil in Listing 3.4) denotes the *ceil* function, which yields the smallest integer greater than or equal to x. Finally, the local scaling performs (un)deploy operations to reach the target number of instances: if scale down occurs, the system keeps installed at least baseN instances.

Notice that, base_throughput takes the values specified in Table 3.1 (stored in serviceThrougputs() and accessed via the lookupUnsafe ABS function); workload and instances are retrieved via the prometheus object that emulates the functionalities of the well-known monitoring system Prometheus [RV15], storing and providing access to essential metrics, *e.g.,* number of deployed instances and workload. The wrapper object encapsulates the timed deployment orchestrations applied for service replication. Finally, the scaleUp and scaleDown methods (for brevity not reported in Listing 3.4) are used to deploy/undeploy the amount of replicas passed as parameter, *i.e.,* the target replicas number minus the current one (or vice versa).

3.7 Executing the AcmeAir System

We now show the capacity of our modelling/executing language to precisely reproduce the behaviour of real-world systems, simulating our running example via the Erlang backend. In our simulation, we set:

- the initial system configuration to deploy 1 instance for each microservice type (as in [IPT23]);

- an ABS unit to be 1 second;
- the monitor to perform its check every 5 seconds;
- the number of users to be updated every 5 seconds;
- the simulation to last 2000 seconds (as in [IPT23]);
- users to periodically execute their behaviour in a uniformly distributed time interval of $[0, 3]$ seconds, where the interval includes natural numbers only;
- the StartupTime of Deployment Components to 0 as the virtual machine used in [IPT23] are immediately ready to work.

To test the accuracy of our ABS model, we address the sinusoidal inbound workload, described by [IPT23], by introducing a dedicated *generating service*, presented in Listing 3.5.

Listing 3.5: AcmeAir workload generator

```
class WorkloadGenerator(...) implements WorkloadGeneratorInterface {

    Rat time = 0;
    List<UserInterface> users = list[];
    Int userID = 0;

    Unit run() {
        while(time <= simulation_duration()) {
            await duration(generation_window());
            Rat sin_shape = this.sin_taylor(time%period, sin_shape_accuracy());
            sin_shape = sin_shape * mod + shift;
            Int u = round(sin_shape);
            if(u > length(users)) this!addUsers(u - length(users));
            else if(u < length(users)) this!removeUser(length(users) - u);
        }
        this!removeUser(length(users));
    }
}
```

In particular, the generating service we implement periodically, *i.e.,* every generation_window() time units (corresponding to 5 seconds), adds/removes users according to the chosen sinusoidal load pattern. Notice that, such load pattern is not directly implementable, since ABS lacks of a built-in sin function. Thus, to accomodate this, we approximate the sin function using a Taylor series, with mod and shift parameters (see Listing 3.5), borrowed from the methodology outlined

by Incerto *et al.* [IPT23], to fine-tune the workload. Users are implemented as an

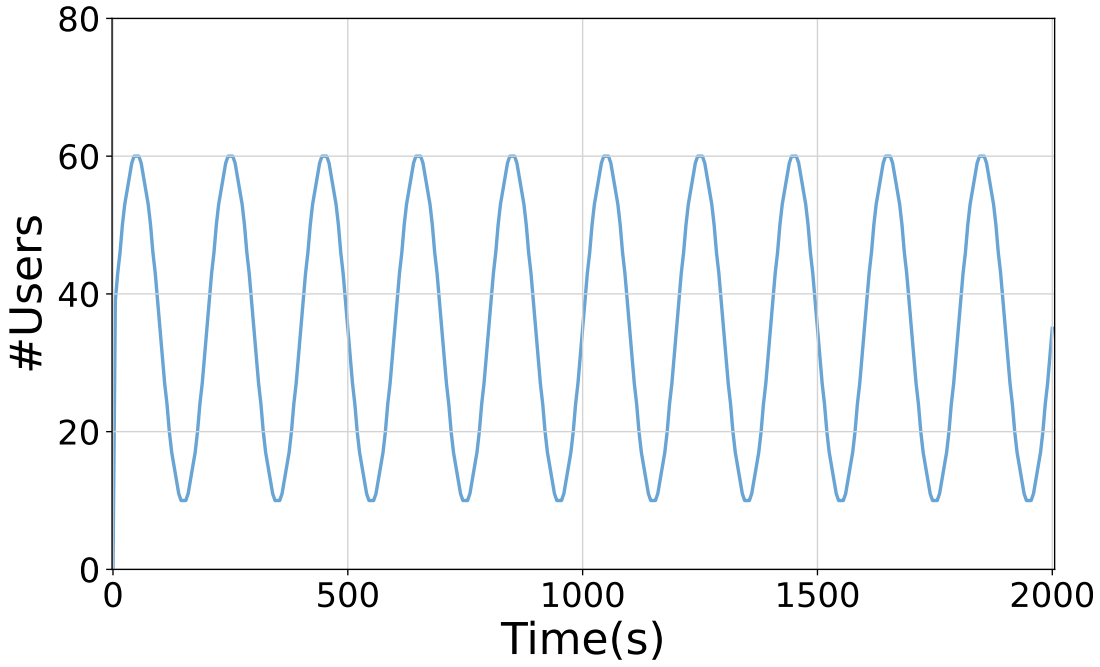
Listing 3.6: ABS sin approximation

```
Rat sin_taylor(Rat x, Rat terms) {
    x = pi() * x / 100;
    Rat result = 0;
    Int power = 1;
    Int sign = 1;
    while(terms > 0) {
        result = result + (sign * pow(x,power) / factorial(power));
        power = power + 2;
        sign = sign * -1;
        terms = terms - 1;
    }
    return result;
}
```

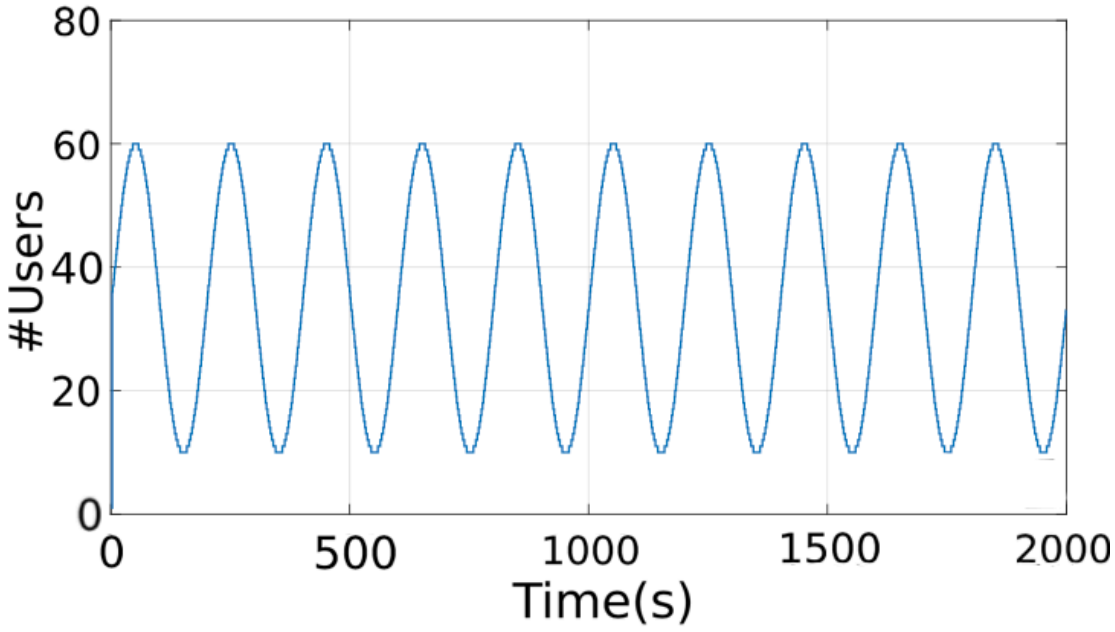
active class synchronously interacting with all AcmeAir endpoints in a sequential manner. Notably, as detailed in [IPT23], certain microservices such as *ViewProfile* and *CancelBooking* may experience more than one request during a single execution of user behaviour.

We now discuss the results obtained, simulating the execution of the AcmeAir system using our timed integrated modelling/execution language. To obtain statistically significant results, we perform 30 independent runs of our simulation and consider the average behaviour. As can be seen in Figure 3.3, the workload we adopt in our simulation (see Figure 3.3a), as expected, precisely reproduces the one used in [IPT23] (see Figure 3.3b): in both cases, the load follows a sinusoidal pattern limited between 10 and 60 users with a period of 200 seconds. Notice that, we are not modelling the autoscaling approach of Incerto *et al.* [IPT23], as their scaling approach is not straightforwardly reproducible with our timed modelling/execution language, thus we cannot directly compare with Incerto *et al.* [IPT23]. As a matter of fact, our focus is on the behavioural pattern the simulated system follows. Given the workload in Figure 3.3, we can reason on the pattern we expect our results to highlight. In particular, due to the sinusoidal nature of the workload, if we were to consider the system throughput and allocated resources trends, we would expect them to resemble such sinusoidal pattern.

Our reasoning is supported by the empirical results shown in Figure 3.4, where we analyse the trend followed by the system throughput. As can be seen, the simulated system behaves as we expect: whenever the number of users grows,



(a) Simulated sinusoidal load



(b) Measured sinusoidal load, from [IPT23]

Figure 3.3: Simulated and measured user generation pattern comparison

the local scaling algorithm starts working and enacts the required scaling actions to manage such time-varying workload, thus increasing the maximum supported throughput. Conversely, if the number of users decreases, the system removes useless resources, consequently, reducing the maximum supported throughput. The

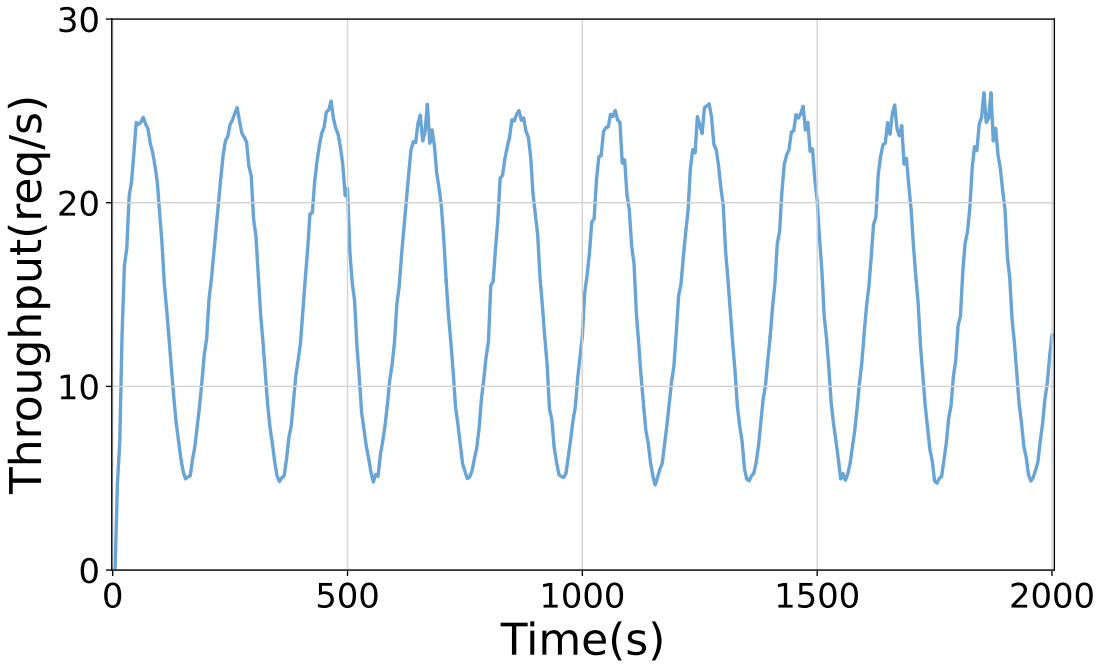


Figure 3.4: Simulated AcmeAir throughput

sinusoidal pattern we discussed so far can be also found in Figure 3.5, where we analyse the number of allocated cores. As can be seen, the graph clearly shows the same trend as the one presented in the throughput experiment, witnessing, again, the consistency of the simulated behaviour with the one we expect.

3.8 Related Work

We presented an integrated approach for the design, specification, automatic deployment and simulation of microservice architectures, based on the ABS language.

Regarding the adoption of executable semantics for simulation, the work done in [BdBdG17] diverges from ours, opting for a real-time Haskell backend. This choice enables direct communication between the simulation and actual services,

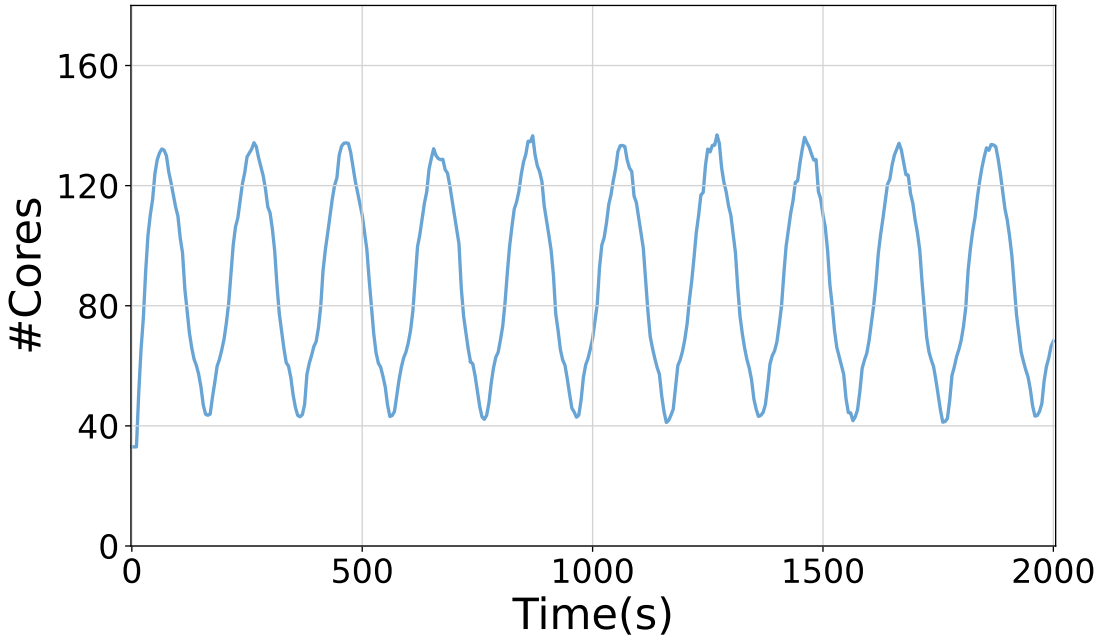


Figure 3.5: Simulated AcmeAir allocated cores

thereby blending external execution with simulation in real-time. Conversely, our methodology does not necessitate communication between the simulated system and external components during simulation, thus avoiding the complexities associated with synchronising real and simulated time.

Another line of work encompasses the integration of timed or stochastic process algebras into the software development lifecycle. This approach aims at analysing the performance of modelled systems, as shown in [BDMIS04, HHK02].

Finally, alternative proposals have surfaced, employing specialised models for cloud deployment specification, such as TOSCA (Topology and Orchestration Specification for Cloud Applications) [OAS] or AEOLUS [DMZZ14]. These models serve to describe the components of a cloud service system and their deployment orchestration process comprehensively. They document the organisational structure of these components and outline the orchestration process necessary for their deployment. The interested reader can find a recent survey of the model-based methodologies used to ensure the correctness of reconfigurations in component-based systems at [CHLR23].

3.9 Discussion

In this Chapter, we presented an integrated approach for the design, specification, automatic deployment and simulation of microservice architectures, based on the ABS language. The basic ingredients of this approach are:

- the ABS language, used to specify the behaviour of microservices;
- deployment annotations added to the ABS code, carrying information like the available computing resources and their costs, the resources consumed by each microservice instance and constraints about the minimum number of instances for each microservice;
- the use of a compile-time deployment engine, *i.e.,* Timed SmartDeployer, able to synthesise optimal deployment orchestrations starting from declarative deployment annotations extracted from ABS code;
- compilation of timed ABS code into executable Erlang programs, capable of simulating the specified system.

We showed the reliability, precision and expressiveness of our timed modelling/execution language implementing a realistic microservice architecture and testing its performance under a time-varying workload. The results show that our approach closely reproduce a coherent behavioural pattern with the one expected from a real-world system execution.

Concerning future work, we plan to expand the modelling/execution capabilities of our language including failures (*e.g.,* network partitioning, computing hardware failures) to precisely evaluate their impact on the deployed system. To this aim, we could evaluate the system following the practice of Chaos Engineering [CR20], simulating failures in ABS and ensuring that the available resources are enough to guarantee a given level or robustness and resilience. Moreover, to improve the portability of our approach, we also plan to base our system modelling using a workflow language/notation that also includes data flow besides standard control flow, such as BPMN [OMG11]. This will make it possible to automatically calculate microservice throughput and its average number of invocation per execution of user behaviour.

Chapter 4

Orchestration-based Architectural Reconfiguration

This Chapter contains contributions from the following work of ours: [BBG+22a, BBG+25, BPS+22b, BPS+22a]

Orchestration-based architectural reconfiguration embodies a sophisticated approach to shaping the dynamics of complex systems, where orchestration serves as the guiding force infusing a predefined behaviour. At its core, orchestration manages the intricate interplay of various components and resources towards a singular goal, weaving coherence and efficiency into the fabric of the architecture. Through meticulous coordination and alignment of disparate elements, orchestration, not only streamlines processes, but it also imparts a sense of purpose, directing the system actions towards predetermined outcomes. This method transcends mere adaptation, as it imbues the system with an inherent understanding of its overarching purpose, driving it towards optimal performance and adaptability. By instilling a given behaviour, orchestration catalyses responsiveness to changing circumstances, fostering resilience and agility in the face of evolving challenges. Furthermore, orchestration promotes autonomy within the system, empowering it with the ability to self-regulate its operations in alignment with the established objectives. In essence, orchestration-based architectural reconfiguration represents a paradigm shift, where the system evolves from a passive entity to an active agent, driven by purpose and guided by orchestrations towards continuous optimisation and enhancement. While orchestration-based architectural reconfiguration tech-

niques hold significant advantages, their efficacy hinges on meticulous design and implementation. Without careful planning and execution, these techniques can falter, leading to inefficiencies, conflicts and even system failures. Let us suppose we want to *replicate* some microservices to offload the existing instances. Again, without careful implementation, our reconfiguration technique could perform an uncoordinated set of actions, not correctly satisfying the interdependencies among microservices. Thus, some of the newly deployed instances may not work properly. Another scenario could be the following. We now want to *migrate* some services from cloud to edge, to reduce the overall system latency. Without careful implementation, our reconfiguration technique might move such services to a Virtual Machine (VM) on edge that has not enough available resources for the hosted services, thus, the migration fail. Hence, there arises a critical need for reconfiguration approaches leveraging *correct-by construction* orchestrations to mitigate risks and ensure reliability, thus reducing the likelihood of human error and enhancing consistency as well as improving performance.

In the context of this Dissertation, we present two different architectural reconfiguration strategies for service *service* autoscaling [BBLZ21, BBG+22a, BBG+25] and *migration* [BPS+22b, BPS+22a]. In particular, we first model and simulate our reconfiguration strategies using the framework presented in Chapter 3 to evaluate their performance at the design phase and subsequently, we provide a real-world implementation to prove their effectiveness in a realistic environment. In both cases, we exploit *correct-by construction* orchestrations to actually perform adaptation actions: in the simulated environment, we use the Timed SmartDeployer tool (see Section 3.5), to automatically synthesise such orchestrations, while, in the real-world one, we use our new *Kubernetes SmartDeployer* (K8sSD) tool [Bac24f], inspired by Boreas [LMTY21] and SmartDeployer, that statically generates *correct-by construction* orchestrations compatible with Kubernetes.

Concerning service autoscaling, we first propose a novel reactive scaling algorithm [BBLZ21] for architecture-level dynamic adaptation, called *global scaling*, that overcomes the drawbacks of the traditional scaling approach, *i.e.,* the "domino effect", caused by uncoordinated scaling actions. Notice that, our novel algorithm crucially leverages the microservice automated deployment methodology, described in Section 3.1: strongly required port can be used to indicate the set of entities

a microservice communicate with, while weakly required ports the connection between a load balancer and the instances it manages. Such methodology is used to automatically synthesise the deployment orchestrations that our algorithm applies to perform architecture-level dynamic adaptation. We then endow our algorithm with *proactive* capabilities using an off-the-shelf machine learning module to forecast the inbound workload, further improving performance. However, predictors are weak against exceptional events, resulting in the application of inappropriate deployment orchestrations (either wasting resources or degrading the Quality of Service). Thus, we also propose a novel *proactive-reactive* [BBG+22a, BBG+25] methodology for mixing the measured workload and the predicted one, in case the predictor fails to correctly forecast the inbound workload. We evaluate the performance of our algorithm on a realistic microservice system: an Email Message Analysis Pipeline [BGM+19, BGM+20]. In particular, we perform system execution using inbound traffic taken from the Enron dataset [KY04]. To highlight the extent of the advantages of our global scaling with respect to the local one implemented in Section 3.6.2 (traditionally used in the literature [Amab, Apa, Doc] and by, *e.g.,* Kubernetes [HBB17]), we empirically test the differences in performance: our results show that our scaling algorithm overcomes the local scaling and avoids cascading slowdowns. Moreover, to show the need for our proactive-reactive algorithm with respect to a purely proactive one, we selectively pick outliers from the Enron dataset and run benchmarks to evaluate its performance.

Concerning service migration [BPS+22b, BPS+22a], we propose an approach to enable the dynamic orchestration of services across nodes of the edge-cloud continuum. Thus, migrated services can be dynamically activated/deactivated on different edge/cloud nodes so that the overall latency is minimised, while maintaining the edge node resource usage under control. We devise our architectural reconfiguration approach, so that the execution of a given orchestration is performed exploiting the data locality principle: maximising the efficiency of data access by moving computation towards data. By organising data and structuring adaptation approaches in a way that minimises the distance between accessed data elements, systems can significantly reduce latency and improve overall performance. To this aim, we validate our platform on a use case proposed by an industrial company (Bonfiglioli S.P.A.). The use case focuses on the seamless al-

location of a *Data Processing* service — in charge of preparing data coming from a production line for a subsequent anomaly detection task — between edge and cloud nodes. We evaluate different policies for the Data Processing service allocation varying the amount of data generated by the production line and measuring the corresponding latency to generate the alarm.

The Chapter is structured as follows. In Section 4.1, we introduce our novel K8sSD, a tool to automatically synthesise deployment orchestrations compatible with Kubernetes. In Section 4.2, we present our new scaling approach, *i.e.,* the proactive-reactive global scaling, presenting both a simulated (via our modelling/execution language, see Chapter 3) and real-world implementation. In Section 4.3, we introduce our novel orchestrator to perform edge-cloud continuum service migration, evaluated via a simulated and real-world implementation. In Section 4.4, we discuss some related literature, while in Section 4.5, we conclude the Chapter.

4.1 A Smart Deployer for Kubernetes

In this Section, we present our new Kubernetes SmartDeployer (K8sSD) tool, inspired to both Boreas [LMTY21] and SmartDeployer (see Section 3.3), to automatically synthesise *correct-by construction* orchestrations compatible with Kubernetes. Such tool, whose code is publicly available at [Bac24f], is implemented in Python and communicate with Zephyrus2 (see Section 3.4) via HTTP requests. The Zephyrus2 tool was originally designed to minimise the cost of application deployment to virtual machines (VMs). While conceptually there is not a big difference between that problem and the placement of service pods on nodes within a Kubernetes cluster, *i.e.,* the optimal deployment problem, in practice we need to perform extensive adjustments and conversions of the data and constraints, before Zephyrus2 can process the placement of Kubernetes pods. Similarly to SmartDeployer, our tool takes as input declarative specifications of deployment constraints and requirements and produces as output a deployment orchestration.

Input. Our tool input is formed by the following `YAML` specifications: (*i*) declarative descriptions of nodes hosting services; (*ii*) deployment requirements already

Listing 4.1: System resources declarative specifications

```
k8s-worker-1:
    resources:
        memory: "4G"
        cpu: "900m"
k8s-worker-2:
    resources:
        memory: "4G"
        cpu: "900m"
---
services:
    name:
        - proxy
```

satisfied (*i.e.,* microservices representing strong dependencies already deployed); and (*iii*) declarative descriptions of services to be deployed.

Concerning (*i*), node descriptions are encoded in `YAML` format, specifying, for each node, the RAM and amount of CPU it supplies. An example can be seen in Listing 4.1: our cluster includes 2 identical nodes supplying 4 gigabytes of RAM and 0.9 CPU cores. Since Zephyrus2 does not support fractional CPU specification (while Kubernetes allows millicores for pod consumption specification), the CPU values must be rescaled of a factor of 1000.

Concerning (*ii*), since the synthesised deployment orchestrations can be used in the context of, *e.g.,* service replication, we also need to account for deployment constraints already satisfied in previous deployment actions. To do that, as can be seen in Listing 4.1, it is enough to declare in the `YAML` specification a field `services`, containing all service app names (matching those specified in the deployment Kubernetes `YAML` file metadata) representing the dependencies already satisfied. This operation is crucial for the orchestration synthesis, to understand why let us consider the following scenario. We already have a running instance of a *Proxy* service and we now want to install a *Backend* instance (see Listing 4.2). Since we are not deploying any instance of the *Proxy* service (because we do not need to replicate such service), the *Backend* strong requirement towards the *Proxy* service cannot be satisfied and, consequently, preventing Zephyrus2 from finding a configuration satisfying the specified requirements.

Concerning (*iii*), K8sSD extracts the information to be preprocessed and sent to Zephyrus2 from standard Kubernetes `YAML` files of kind `Deployment`. To account

Listing 4.2: Service declarative specification

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: backend
5  spec:
6    selector:
7      matchLabels:
8        app: backend
9    replicas: 3
10   template:
11     metadata:
12       labels:
13         app: backend
14     spec:
15       affinity:
16         podAffinity:
17           requiredDuringSchedulingIgnoredDuringExecution:
18           - labelSelector:
19               matchExpressions:
20               - key: app
21                 operator: In
22                 values:
23                 -  frontend
24             topologyKey: "kubernetes.io/hostname"
25       containers:
26       - name: backend-container
27         image: k8s.gcr.io/pause:2.0
28         resources:
29           requests:
30             cpu: "300m"
31             memory: "500M"
32 ports:
33   required:
34     strong:
35       - name: "proxy"
36         value: 1
```

for strong dependencies (see Section 3.1), we extend the Kubernetes deployment language. As can be seen in Listing 4.2 (lines 32-36), the `required.strong` field indicates the strongly required services. Notice that, `name` must match the strong dependency name specified under `metadata`, while `value` indicates the amount of instances to strongly connect. *i.e.,* To better understand how the tool works, let us consider the example presented in Listing 4.2, where we declare the specifications of a pod running a *Backend* service. Here, the first constraint `backend = 3` (see line 9) imposes Zephyrus2 to search for configurations with exactly 3 *Backend* instances. Kubernetes allows to define two types of intra pod affinities, *i.e.,* "hard" ones that specify rules that must be met for a pod to be scheduled and "soft" ones

that specify preferences that the scheduler will try to enforce, but will not guarantee. Currently, we only consider the "hard" affinities, since these are the ones that cannot be violated and restrict the possible admissible configurations. An example of pod affinity can be found in Listing 4.2 (lines $16-24$), where we require the `backend` service to be deployed in a node running an instance of the `frontend` one. We also support pod anti-affinity, where we require the specified service to be deployed in a different node with respect to the one the specification refers to. Notice that, the keyword `requiredDuringSchedulingIgnoredDuringExecution` states that the constraints are "hard" ones, meaning that they must be satisfied during service placement. Following SmartDeployer (see Section 3.3), K8sSD encodes the affinity requirement as `(forall ?x in locations: (?x.backend > 0 impl ?x.frontend > 0))`, while anti-affinity as `(forall ?x in locations: (?x.backend <= 1))`. In lines $32-36$, we specify strong dependencies: the *Backend* service strongly depends on a single instance of the *Proxy* one. This specification produces a constraint to ensure a specific deployment order. As a matter of fact, in the synthesised orchestrations, service deployments follow a topological order, *i.e.,* services with dependencies come after the ones they depend on.

Output. K8sSD produces as output: (*i*) a `YAML` file specifying, for each node, the RAM and amount of CPU left, after executing the synthesised orchestration; and (*ii*) an orchestration deploying the services specified as input, either in a platform independent declarative language, *i.e.,* `YAML`, or in a specific programming language, *i.e.,* the `Python` programming language.

The synthesised orchestrations are executed by means of *Pulumi*[1], an Infrastructure as a Code (IaaC) platform that enables developers to define, deploy and manage cloud infrastructures using common programming languages (*e.g.,* Python) and declarative notation, *i.e.,* the `YAML` language. The advantage of using programming languages over declarative notations is that it allows developers to choose a language they are familiar with, making the synthesised orchestrations intuitive and easy to read (their execution relies on the dedicated Pulumi client library for that language). On the other hand, the `YAML` language is a declarative notation, making it possible to model orchestrations in an abstract way, without including

[1]https://www.pulumi.com

execution details specific to some programming languages (these orchestrations are executed via a dedicated deployment engine).

We first present an example of deployment orchestration using `YAML` as orchestration language. For brevity sake, we omit the `properties` field as it contains the information specified in Kubernetes `YAML` files. The orchestration presented in Listing 4.3 deploys two services, *i.e.,* the *Backend* and *Proxy*. Thanks to the `dependsOn` annotation, which models strong dependencies, such orchestration first deploys the *Proxy* service, then the *Backend* one (the latter depends on the former). The peculiarity of defining orchestrations using the `YAML` language, is that there is no reminiscence of programming languages: the produced orchestration only contains details directly related to the deployment process. Moreover, the synthesised orchestration contains, for each specified service, the target hosting node (see the `nodeName` field), computed such that the resource usage is optimal, *i.e.,* minimise the amount of used virtual machines.

Listing 4.3: `YAML` orchestration example

```
name: my-k8s-app
resources:
    proxy-0-pod:
        name: proxy-0-pod
        options: {}
        properties:
            ...
            nodeName: k3d-k3s-default-agent-0
        type: kubernetes:core/v1:Pod
    backend-0-pod:
        name: backend-0-pod
        options:
        dependsOn:
            - ${proxy-0-pod}
        properties:
            ...
            nodeName: k3d-k3s-default-agent-1
        type: kubernetes:core/v1:Pod
runtime: yaml
```

We now present an example of deployment orchestration using Python as orchestration language. As can be seen in Listing 4.4, we use the `dependsOn` Pulumi annotation to express strong dependencies. The `pod_backend_0` specifies as value of the `dependsOn` annotation the service `pod_proxy_0`, instructing the Pulumi engine to wait for the `pod_proxy_0` service to be up and running, before

deploying `pod_backend_0`. The topological ordering enacted by K8sSD is crucial: if we were to switch the definition of the services in Listing 4.4, the orchestration would produce an error during execution. It is important to notice that, the synthesised orchestration contains, for each specified service, the target hosting node (see `nodeName` in line 5), computed such that the resource usage is optimal, *i.e.,* minimise the amount of used virtual machines.

Listing 4.4: Python orchestration example

```
1  from pulumi import automation as auto
2  def pulumi_prog():
3    pod_proxy_0 = k8s.core.v1.Pod("pod-proxy-0", metadata={...},
4      spec={"containers": [...], "nodeName": "k3d-k3s-default-agent-0"})
5    pod_backend_0 = k8s.core.v1.Pod("pod-backend-0", metadata={...},
6      spec={"containers": [...], "nodeName": "k3d-k3s-default-agent-1"},
7      opts=pulumi.ResourceOptions(depends_on=[pod_proxy_0]))
8  def deploy():
9    stack = auto.create_or_select_stack(stack_name="dev-1",
10     project_name="example", program=pulumi_prog)
11   stack.up()
```

4.2 Proactive-Reactive Global Scaling

Modern cloud architectures use microservices as their highly modular and scalable components, which, in turn, enable effective practices such as continuous deployment and horizontal (auto)scaling. Although these practices are already beneficial, they can be further improved by exploiting the interdependencies within an architecture (interface functional dependencies), instead of focusing on a single microservice. Architecture-level dynamic deployment orchestrations for service replication bring significant advantages with respect to the traditional local scaling technique: they eliminate the *"domino effect"* of unstructured scaling, *i.e.,* single services scaling one after the other (cascading slowdowns) due to local workload monitoring, as done in, *e.g.,* Kubernetes [HBB17].

In the context of this Dissertation, we first propose a novel technique for architecture-level dynamic adaptation, called *global scaling* [BBLZ21], that overcomes the drawbacks of the traditional scaling approach. The global scaling algorithm leverages the knowledge of functional dependencies between microservice

requests and it reaches, via architecture-level reconfigurations, a target system Maximum Computational Load (MCL), *i.e.,* the maximum supported frequency for inbound requests. The idea is that, in a *reactive* approach, *i.e.,* by monitoring at run-time the inbound workload, our algorithm causes the system to be always in the reachable configuration, with the least amount of deployed microservice instances, that better fits such workload. Global reconfigurations are targeted at guaranteeing a given increment/decrement of the system MCL.

We then endow our approach with *proactive* capabilities using an off-the-shelf machine learning module to forecast the inbound workload, further improving performance. However, predictors are weak against exceptional events, resulting in the application of inappropriate deployment orchestrations. Thus, we also contribute a novel *proactive-reactive* algorithm [BBG+22a, BBG+25] to mix the measured workload with the predicted one. Our algorithm casts the comparison as the capacity of the system to deal with a given workload (*i.e.,* system MCL), obtained from its current system reconfiguration. Hence, we have a way to estimate both over- and under-scaling of proactive global scaling, given by the distance with respect to the system MCL induced by the actual traffic.

4.2.1 A Proactive-Reactive Global Scaling Platform

In the context of this Dissertation, we introduce a novel platform that DevOps can use to perform proactive and/or reactive global scaling. The platform depicted in Figure 4.1 includes an external microservice architecture (labelled with G, $\mathsf{M}_1, \mathsf{M}_2$, M_3) and internal elements (*i.e.,* orange boxes) building the scaling approach used to adapt the system. Since the platform sees microservices as instance parameters, we abstract away from their behaviour. In Figure 4.1, we distinguish three flows: (*i*) $\rightarrow$ showing the inbound workload; (*ii*) $\dashrightarrow$ modelling the run-time execution of an adaptation process; and (*iii*) $\Leftarrow$ indicating the synthesis of deployment orchestrations. We now describe each element of the platform.

Deployment Orchestration Engine. This component receives deployment orchestrations and executes them to perform system reconfigurations. It is a loosely-coupled component, taken from existing solutions, *e.g.,* Kubernetes [HBB17].

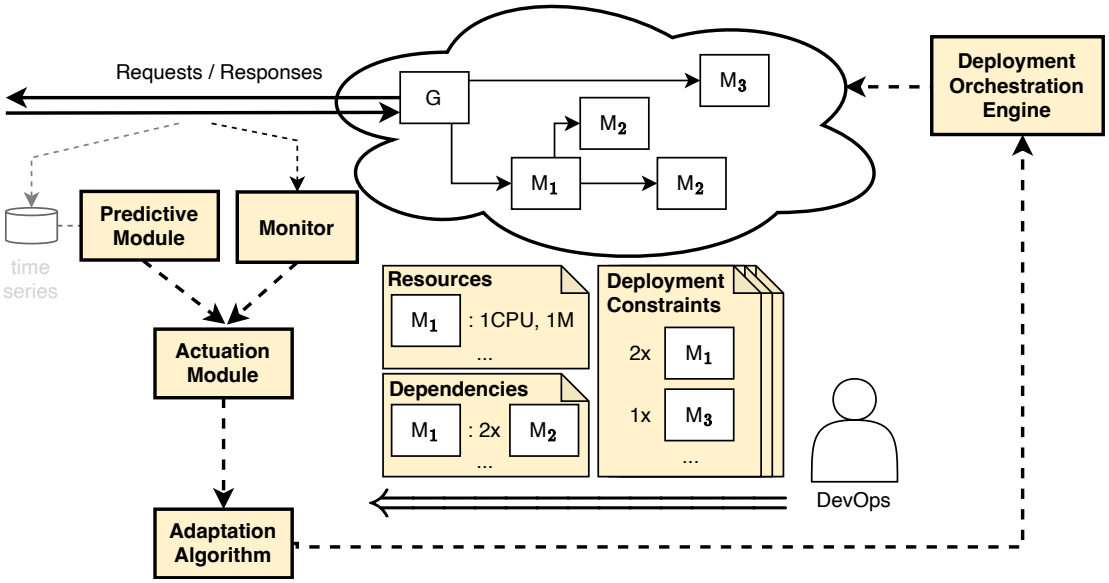


Figure 4.1: Architectural view of the proactive-reactive global scaling platform

Adaptation Algorithm. The *Adaptation Algorithm* implements the strategy, deciding the deployment orchestrations to apply, to cope with inbound workload. Such module needs two inputs to work. The first input, depicted with $\Leftarrow$, represents deployment orchestrations statically computed via, *e.g.,* K8sSD (see Section 4.1). These orchestrations are such that they satisfy the user (*DevOps* in Figure 4.1) specifications, *i.e., Resources*, *Dependencies* and *Deployment Constraints*. The second input, represented by $\dashrightarrow$, is the workload the system has to support after the adaptation process.

Monitor. The monitor tracks the traffic flowing on the architecture within a prefixed *time window* and checks the possible occurrence of a *workload deviation*, *i.e.,* the difference between the monitored workload and the globally supported one. When such a condition occurs, the *Monitor* sends to the *Actuation Module* the amount of measured workload.

Actuation Module. The *Actuation Module* plays a pivotal role in our platform: it allows the seamless coexistence of the reactive modality with the proactive one. In particular, the *Actuation Module* computes the amount of workload given as in-

put, *i.e.,* the *target workload*, to the *Adaptation Algorithm*. We distinguish among three modalities: (*i*) *reactive* mode, if the target workload is the one measured by the monitor (the predicted one is discarded); (*ii*) *proactive* mode, if the target workload is represented by the predictions from the *Predictive Module* (the measured one is discarded); and (*iii*) *proactive-reactive* mode, if the target workload is computed combining signals from the *Monitor* and *Predictive Module*.

Predictive Module. The *Predictive Module* acts independently of the actual inbound traffic forwarding the prediction to the *Actuation Module*. For instance, the *Predictive Module* can use a static model, *e.g.,* forecasting traffic peaks at predetermined times or sophisticated techniques to have more accurate predictions of traffic fluctuations. In Figure 4.1, we represent the input of the *Predictive Module* with the greyed-out arrow receiving information from the traffic flow and stores it into a time series dataset for further usage, *e.g.,* training tasks.

4.2.2 The Email Message Analysis Pipeline

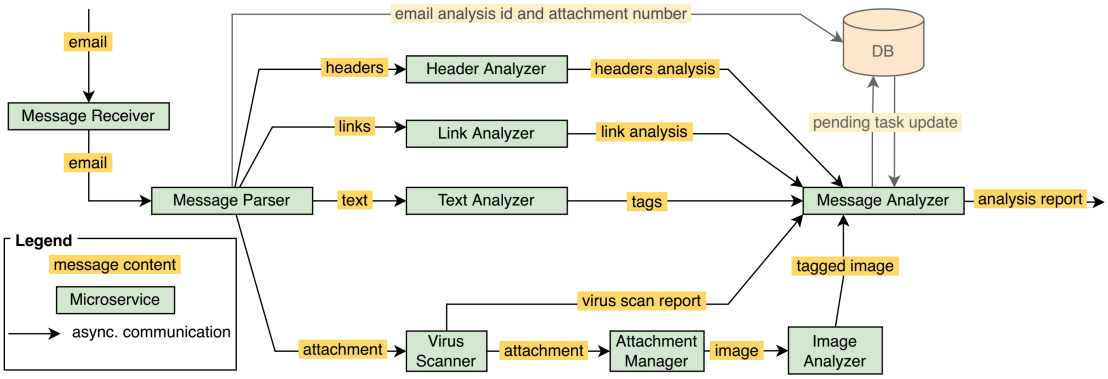


Figure 4.2: Microservice architecture of the Email Message Analysis Pipeline

We now present the specific microservice architecture we use in the proactive-reactive global scaling platform to test its effectiveness. In particular, we use the Email Message Analysis Pipeline [BGM+20, BGM+19] depicted in Figure 4.2. The architecture separates and routes the components found within an email (headers, links, text, attachments) into distinct, parallel sub-pipelines with specific tasks, *e.g.,* checks the format of the email, tags its content, detect malicious attachments.

The *Message Receiver* microservice is the entry point of the architecture and acts as a proxy, receiving and triggering the analysis of incoming emails. The *Message Receiver* forwards an inbound email to the *Message Parser*, which performs some preliminary validity checks. If the message is well-formatted, such service first stores a pending-analysis task under a unique identifier for the current email in a companion database, *i.e.,* the *DB* service. Notice that, the *DB* maintains the status of all pending analyses in the system and it is an element external to the architecture (represented by the faded part at the top of Figure 4.2). After storing the pending task, the *Message Parser:* (*i*) splits the parsed email into four components, *i.e.,* header, links, text and attachments; (*ii*) tags them with the unique identifier of the pending-analysis task; and (*iii*) sends the four components to their corresponding sub-pipelines. From the top, the first and second sub-pipelines, respectively, take into account the headers (*i.e., Header Analyzer*) and the links (*i.e., Link Analyzer*) contained in the email. The third sub-pipeline includes a *Text Analyzer* that tags the body of the message and performs a sentiment analysis. The last sub-pipeline handles attachments and it is the most complex in the architecture. The first microservice in the attachment sub-pipeline is a *Virus Scanner*, which checks each attachment for the presence of malicious software. If an attachment results malicious, it is deleted and reported as dangerous to the *Message Analyzer*, as described later. Safe attachments are forwarded to an *Attachment Manager* for further analyses. The *Attachment Manager* inspects each attachment to identify its content type (*e.g.,* image, audio, archive) and routes it to the appropriate part of the sub-pipeline. Notice that, in Figure 4.2, we just exemplify the concept with an *Image Analyzer* that tags the content of each image and checks the absence of explicit content. All sub-pipelines forward the result of their (asynchronous) analyses to the *Message Analyzer*, which collects them in the *DB*. After all analyses belonging to the same pending task are completed, the *Message Analyzer* combines them and reports the result of the analysis process.

4.2.3 Microservice MF and MCL

The MF of a microservice type is determined by the role it plays in the architecture, *e.g.,* in our running example, by the email part it receives. For example, assuming

an email has 2 attachments on average, the *Virus Scanner* microservice receives, for each email entering the system (*i.e.,* request to the initial *Message Receiver* microservice), a mean of 2 requests. Thus, microservice MF is determined by the established *functional dependence between requests* to such microservice type and requests entering the system.

Therefore, concerning our running example, we base the calculation of the MF of its microservice types, on the following average estimation of the structure of emails entering the system: (*i*) a single header; (*ii*) a set of links (treated as a whole); and (*iii*) $\mathsf{N}_{\mathsf{attach}} = 2$ attachments (individually sent to the attachment sub-pipeline), each having an average size of $\mathsf{size}_{\mathsf{attach}} = 7\mathsf{MB}$ and containing a virus with probability $\mathsf{P}_{\mathsf{V}} = 0.25$.

Given the emails average structure descrived above, MFs are calculated as follows. *Header Analyser, Link Analyzer* and *Text Analyzer* have $\mathsf{MF} = 1$ since emails have a single header, a set of links treated as a whole and a single text body. As attachments are individually sent, each one generates a specific request to the *Virus Scanner*, therefore such service has $\mathsf{MF} = \mathsf{N}_{\mathsf{attach}}$. The MF of microservices following the *Virus Scanner* in the pipeline corresponds to the number of virus-free attachments, computed as $\mathsf{MF} = \mathsf{N}_{\mathsf{attach}} \cdot (1 - \mathsf{P}_{\mathsf{V}})$. Finally, the MF of the *Message Analyzer* is the sum of the email parts (1 header, 1 set of links, 1 text body and $\mathsf{N}_{\mathsf{attach}}$ attachments).

The MCL of a microservice type is the maximum number of requests an instance of that type can handle in a second. It is computed as follows:

$$\mathsf{MCL} = 1/(\frac{\mathsf{size}_{\mathsf{req}}}{\mathsf{data\_rate}} + \mathsf{pf})$$

where: (*i*) $\mathsf{size}_{\mathsf{req}}$ is the average request size of microservices in MB; (*ii*) $\mathsf{data\_rate}$ is the microservice rate in MB/s to manage requests, determined accounting for microservice required cores (taken from server data in [Raw]); and (*iii*) pf is a penalty factor expressing additional time microservices need to manage requests, *e.g.,* those performing computationally expensive tasks. We compute microservice $\mathsf{size}_{\mathsf{req}}$ as follows. For microservices handling attachments, but *Message Analyzer*, we have: $\mathsf{size}_{\mathsf{req}} = \mathsf{N}_{\mathsf{attach\_per\_req}} \cdot \mathsf{size}_{\mathsf{attach}}$ where $\mathsf{N}_{\mathsf{attach\_per\_req}} = \mathsf{N}_{\mathsf{attach}}$ for microservices receiving entire emails, while, for the others, $\mathsf{N}_{\mathsf{attach\_per\_req}} = 1$. For *Header*

Analyser, Link Analyzer and *Text Analyzer*, we consider $\mathsf{size}_{\mathsf{req}}$ to be negligible, thus (since $\mathsf{pf} = 0$) their MCLs are infinite. Concerning *Message Analyzer* request size, we compute the average size of the MF requests an email entering the system generates (since we consider only attachments to have a non-negligible size), *i.e.,*

$$\mathsf{size}_{\mathsf{req\_MA}} = \frac{\mathsf{N}_{\mathsf{attach}} \cdot (1 - \mathsf{P}_{\mathsf{V}}) \cdot \mathsf{size}_{\mathsf{attach}}}{\mathsf{MF}}.$$

As we will see, MCL and MF microservice type are important properties, since they are used to calculate the minimum instance number of that type to guarantee an overall system MCL, denoted by $\mathsf{sys\_MCL}$. Formally, being $\lceil x \rceil$ the ceiling function taking as input a real number and returning the least integer greater than/equal to x, we have

$$\mathsf{N}_{\mathsf{instances}} = \lceil \frac{\mathsf{sys\_MCL} \cdot \mathsf{MF}}{\mathsf{MCL}} \rceil$$

Notice that, the MF is implicitly modelled by the method call sequence required to analyse an email (see Figure 4.2). The MCL is explicitly modelled in the ABS code, as described in Section 3.5, using the `Cost` annotation. In the real-world implementation there is no need to explicitly design the MCL of services, as it is already modelled by, *e.g.,* service implementation, available resources.

4.2.4 Architectural Scaling of Microservices

A common scenario where the automated deployment of microservices, described in Section 3.1, can be straightforwardly used is *autoscaling*: strongly required ports can be used to indicate the set of entities a microservice communicates with, while weakly required ports the connection between a load balancer and the instances it manages. One crucial prerequisite for configuring the deployment of a microservice architecture is ensuring that each microservice is defined with a strongly required port towards the subsequent microservices in the pipeline. For instance, let us consider the *Message Parser*, which strongly requires connections with the *Header Analyzer*, *Text Analyzer* and *Link Analyzer*, as depicted in Figure 4.2. These ports should not be directly connected to instances of the respective microservices, but rather to their corresponding load balancers. Subsequently, each load balancer fea-

tures a weakly required port, which must be connected to all available instances of the corresponding microservice type, enabling the load balancer to efficiently distribute incoming requests among them. This design choice is underpinned by several reasons. Firstly, by establishing strongly required connections to a microservice proxy, it becomes feasible to deploy load balancers of *Header Analyzer*, *Text Analyzer* and *Link Analyzer* beforehand. Then, the instances of *Message Parser* can be deployed subsequently, as they can readily connect to the load balancer they strongly require. Finally, the connection of the load balancers to their instances can be established through the weakly required port, completing the deployment setup. This approach ensures a streamlined and efficient deployment process, facilitating the orderly establishment of connections among microservices within the architecture.

One of the key advantages of employing strongly and weakly required ports lies in their ability to facilitate dynamic adaptation. This framework enables seamless incorporation of new microservice instances to address increased workload demands. When encountering a surge in workload, for example, a new microservice instance can be swiftly introduced by promptly connecting it to the strongly required load balancer of the next microservice in the pipeline. Subsequently, the corresponding load balancer establishes a binding with the newly created instance via the weakly required port. Conversely, when removing a microservice instance, the process unfolds in the reverse order. Initially, the binding between the load balancer of the instance slated for removal is severed. Subsequently, the concerned instance can be safely deallocated. This orchestrated approach ensures a smooth and controlled adjustments to the deployment process, effectively accommodating fluctuations in workload, while maintaining system integrity.

If we generalise the above process, from adding/removing one microservice to an arbitrary amount, we can dynamically adjust the entire architecture to meet resource demands effectively. Consider a scenario where increased workload necessitates the deployment of three new instances of *Message Parser* and two new instances each of *Header Analyzer*, *Text Analyzer* and *Link Analyzer* to handle the influx of traffic efficiently. When utilising autoscaling mechanisms [Amaa], scaling in/out decisions are typically made locally by individual services. Consequently, the scaling out process occurs sequentially: firstly, the *Message Parser*

service, positioned at the forefront of the pipeline and thus directly impacted by the surge in traffic, scales out. Subsequently, the *Header Analyzer*, *Text Analyzer* and *Link Analyzer* services follow suit, as they start to experience increased invocations triggered by the newly added instances of *Message Parser*. However, with a holistic understanding of microservice dependencies, we can leverage the insight that multiple services may require scaling out simultaneously. This enables a global adaptation strategy, wherein services such as *Message Parser*, *Header Analyzer*, *Text Analyzer* and *Link Analyzer* can be scaled out concurrently, thereby enhancing system elasticity and responsiveness to varying workload demands.

4.2.5 Calculation of Scaling Configurations

In this Section, we show the mathematical process we follow to build system configurations, which the statically synthesised deployment orchestrations have to reach. Notice that, such orchestrations are obtained via Timed SmartDeployer in the case of the ABS simulation and using K8sSD (see Section 4.1) in the case of the real-world implementation. We start with a base system configuration $\mathbf{B}$, which guarantees a system MCL of 60 emails/s. In Table 4.1, we present the number of instances for each microservice type, calculated according to the formula in Section 4.2.3. We treat *Header Analyser, Link Analyzer* and *Text Analyzer* as if they had an infinite MCL, thus they are never replicated. We also consider four incremental configurations, each one adding a number of instances to each microservice type (see Table 4.1) with respect to the base system configuration. Those incremental configurations are used as target configurations for automatic (un)deployment orchestration synthesis to perform run-time architecture-level reconfiguration. As shown in Table 4.2, Δ configurations are used, in turn, to build (summing them up element-wise as arrays) the incremental configurations $\mathsf{Scale1}, \mathsf{Scale2}, \mathsf{Scale3}$ and $\mathsf{Scale4}$ that guarantee an additional system MCLs.

The reason for not considering our Scales as monolithic blocks and defining them as combinations of the Δ incremental configurations is the following. Let us suppose the system is in a $\mathbf{B} + \mathsf{Scale1}$ configuration and the increase in incoming workload requires the deployment of $\mathsf{Scale2}$ and the undeployment of $\mathsf{Scale1}$. Without Δ configurations, we would need to perform an undeployment of $\mathsf{Scale1}$

Microservice	**B**	$\Delta 1$	$\Delta 2$	$\Delta 3$	$\Delta 4$
Message Receiver	1	+1	+0	+1	+1
Message Parser	1	+1	+0	+1	+1
Header Analyzer	1	+0	+0	+0	+0
Link Analyzer	1	+0	+0	+0	+0
Text Analyzer	1	+0	+0	+0	+0
Virus Scanner	1	+1	+2	+1	+2
Attachment Manager	1	+0	+1	+0	+1
Image Analyzer	1	+0	+1	+0	+1
Message Analyzer	1	+1	+2	+1	+2

Table 4.1: Base **B** (60 $\frac{emails}{s}$) and incremental Δ configurations

Scale 1 (+60 $\frac{emails}{sec}$)	**Scale 2 (+150 $\frac{emails}{sec}$)**	**Scale 3 (+240 $\frac{emails}{sec}$)**	**Scale 4 (+330 $\frac{emails}{sec}$)**
$\Delta 1$	$\Delta 1 + \Delta 2$	$\Delta 1 + \Delta 2 + \Delta 3$	$\Delta 1 + \Delta 2 + \Delta 3 + \Delta 4$

Table 4.2: Incremental Scale configurations

followed by a deployment of Scale2. With Δ configurations, instead, we can simply additionally deploy $\Delta 2$.

For each microservice type, the number of additional instances considered in Tables 4.1 and 4.2 for each Scale configuration is calculated as follows. Given the additional system MCL, the number $\mathsf{N}_{\mathsf{deployed}}$ of already deployed instances of that microservice type, its MF and MCL, we have:

$$\mathsf{N}_{\mathsf{instances}} = \left\lceil \frac{(\mathsf{base\_MCL} + \mathsf{additional\_MCL}) \cdot \mathsf{MF}}{\mathsf{MCL}} - \mathsf{N}_{\mathsf{deployed}} \right\rceil$$

4.2.6 Reactive Global Scaling Algorithm

In the following Section, we thoroughly describe our reactive global scaling algorithm and its implementation. For brevity sake, in the presentation process, we only show its real-world Python implementation, nonetheless the ABS implementation is, as the reader expects, equivalent to the one reported here.

In the reactive global scaling algorithm, differently from Kubernetes [HBB17], we have a single monitor that periodically executes the scaling algorithm. Such an algorithm scales the system as a whole based on the overall workload and request functional dependencies. As can be seen in Listing 4.5, we use the same scaling

condition presented in Section 3.6.2, but as workload we use the one entering the system and as throughput the system MCL (*i.e.,* `mcl` in the code).

Listing 4.5: Monitor code

```
if workload - (mcl-self.k_big) > self.k or (mcl-self.k_big) - workload > self.k:
  deltas = self.scaler.calculate_configuration(workload + self.k_big)
  mcl, _ = self.scaler.process_request(deltas)
```

The `calculate_configuration` method is the heart of our algorithm: it computes the system configuration to cope with the target workload passed as input. Such configuration, *i.e.,* `deltas` in Listing 4.5, is expressed in the form of a `List`, where index i represents Δi and the i-*th* element is the number of Δi applications.

Listing 4.6: Global scaling: calculate configuration

```
def calculate_configuration(self, target_workload):
 config = self.base_config.copy()
 deltas = np.zeros(len(self.scale_components))
 mcl = self.estimate_mcl(self.base_config)
 candidate_config = config
 while not mcl - target_workload >= 0:
   for i in range(len(self.scale_components)):
     candidate_config = config + self.scale_components[i]
     deltas[i] += 1
     mcl = self.estimate_mcl(candidate_config)
     if mcl - target_workload >= 0:
       break
   config = candidate_config
 self.curr_config = config
 return deltas
```

The code presented in Listing 4.6 uses the constant `scale_components`: it is an array (in our case it contains 4 elements, those presented in Table 4.2) storing, at each position, an array representing a `Scale` configuration (*i.e.,* specifying, for each microservice, the number of additional instances to deploy). The code exploits the variable mcl, containing the current system MCL (assumed to be initially set to the one of the **B** configuration, see Table 4.1). At first, the code applies the scale up/down conditions described in Listing 4.5. Then it loops, starting from the **B** configuration copied in variable `config` (an array that stores, for each microservice, the number of instances we are currently considering) and selects the Scale configurations to add to `config` until a configuration `c` such that its system MCL satisfies $\mathsf{mcl} - \mathsf{K} - \mathsf{target\_workload} \geq 0$ is found (notice that, the

`target_workload` we pass as input is already increased by K, see Listing 4.5). The system MCL of a configuration `c` is calculated via `estimate_mcl`, which yields

$$\mathsf{min}_{0 \leq i \leq \mathsf{length(c)}-1} \;\; \mathsf{c[i]} \cdot \mathsf{MCL_i/MF_i}$$

with $\mathsf{MCL_i/MF_i}$ denoting those of the *i-th* microservice. The algorithm is composed by two nested loops: the outermost `while` loop is in charge of keeping the algorithm computing the target configuration, *i.e.,* the one capable of coping with the inbound workload incremented by K; the innermost `for` loop selects the first Scale configuration that, added to `candidate_config`, yields a candidate configuration, whose system MCL satisfies the scaling condition above (see Listing 4.5). If no configuration is found, the global scaling algorithm just selects the last (the biggest) Scale configuration (Scale4 in our case), thus implementing the following invariant: if N Scale reconfigurations are applied and increasingly sorted by system MCL increment, they guarantee the produced system configuration is either $\mathbf{B}$ or $\mathbf{B} + (\mathsf{n} \cdot \mathsf{ScaleN}) + \mathsf{scale}$, for some $\mathsf{scale} \in \{\mathsf{Scale1}, \mathsf{Scale2}, \ldots, \mathsf{ScaleN}\}$ and $\mathsf{n} \geq 0$.

The invariant property of our algorithm guarantees that multiple deployments of the same Scale configuration are not allowed, except for ScaleN. The reason is the following. The biggest configuration ScaleN should be devised such that the workload rarely yields to additional scaling needs. Even if a sequence of ScaleN occurs, the system would be sufficiently balanced. As a matter of fact, differently from smaller Scale configurations, ScaleN is assumed to add, at least, an instance for each microservice with finite MCL (as for Scale4 in our case). The invariant property of our algorithm is fundamental to avoid the following scenario. Let us suppose that the system is currently in a B+Scale1 configuration and the workload grows such that it requires the deployment of another Scale1. If we naively apply Scale1, some microservices are not replicated, *e.g., Attachment Manager* and *Image Analyzer*. If the workload keeps growing at the same pace, such that the difference between the supported and the measured ones keeps requiring the application of Scale1, eventually the *Attachment Manager* and *Image Analyzer* become the bottlenecks of the system (since they are never replicated by Scale1). Thus, the system performance begins to degrade, despite scaling actions being executed.

Once the `calculate_configuration` method has computed the Δ scales re-

Listing 4.7: Global scaling: apply Δ scales

```
def apply_scales(self, deltas):
  increments_to_apply = deltas
  if self.current_delta is not None:
    increments_to_apply = deltas - self.current_delta
  self.apply(increments_to_apply)
  self.current_delta = deltas
  self.mcl = self.estimate_mcl(self.curr_config)
  return self.mcl, increments_to_apply
```

quired to reach the target system configuration, we use the `apply_scales` method in Listing 4.7 to apply them. In particular, this method performs an element wise difference between the previously computed Δ scales and those currently applied. If such difference is positive, the method `apply` commands Kubernetes to asynchronously perform a deployment, otherwise an undeployment.

4.2.7 Proactive Global Scaling

A straightforward improvement of our global scaling approach is the adoption of techniques to predict in advance workload peaks. As we will see, such techniques can further soften the impact of such events, leading to better performance. Here, we show the steps we follow to build our proactive global scaling implementation, using a state of the art data analytics technique [KMND20]. We apply the data analytics steps to the Enron dataset [KY04], made public by the Federal Energy Regulatory Commission during investigations concerning the Enron corporation (version of May the 7th, 2015). The dataset contains 517431 emails from 151 users, without attachments, distributed over a time window of about 10 years (starting from 1995).

Descriptive and Diagnostic Analytics. We perform the cleaning procedure of the Enron dataset for classification tasks and then we extract the attributes to predict the number of incoming emails at a given time. First, we extract the *datetime* attribute for each email in the dataset and then we sum the number of emails in the desired monitored time unit, *i.e.,* one hour, for each month of the year, day of the month and day of the week. Thus, we generate five new attributes: *month*, *day*, *weekday*, *hour* and *counter*, *i.e.,* the prediction target, for each dataset

instance. This gives us a representation of the email flow in the system at a given hour. The intuition for such a pre-processing is simple. The phenomena of increase or decrease in the flow of emails that occur in a company depend on factors, such as the specific time of the working day (*e.g.,* peak in the early hours versus the night hours), the month (*e.g.,* monthly, bimonthly), the day of the month (*e.g.,* salary) or the day of the week (*e.g.,* weekdays versus holidays).

Predictive Analytic. In this step, we use an off-the-shelf machine learning technique, specifically MLP (Multi-Layer Perceptron), which is capable, in contrast with purely linear models, *e.g.,* linear regression, of exploring nonlinear patterns and increase prediction performance, while containing complexity (about 7000 parameters) and resource usage (about 1ms inference time). We categorise the numerical variables using the standard one-hot encoding technique to prevent our model from attributing wrong semantics to these variables (*e.g.,* month 12 is "greater than" month 1), resulting in a data representation of 70 attributes plus the *counter*, *i.e.,* the prediction target.

Then, we followed the traditional training process for machine learning. We partitioned the cleaned preprocessed data into three sets: one for training the neural network model, one for validating its hyperparameters (the parameters of the training process and network architecture) and one for testing the accuracy of the model. We use this last set to compute the error rate of the model.

The neural network we use in the training process consists of three fully-connected layers. We applied the Rectified Linear Unit (ReLU) nonlinear activation function to the output of each layer. Each level compresses the input into a smaller representation, going from 70 to 64 attributes, in the first level and from 64 to 32 attributes, in the second level. Finally, the 32 attributes are linearly projected into a single value, corresponding to the target of the regression, *i.e.,* the counter attribute. To compute the error rate, we adopt the Mean Squared Error (MSE) as loss function. To optimise the network parameters we use Adaptive Moment Estimation (Adam). We performed the training process with a learning rate of 0.1 and an exponential decay scheduler with gamma 0.9.

After the training, given a time slot, *e.g.,* the tuple hour (0–24), the predictor forecasts the number of emails incoming therein.

Prescriptive Analytic. Since we are implementing a proactive scaling approach, the prescriptive step is straightforward. The global scaling algorithm computes the scale configurations to apply, instead of using monitor signals, as done in the reactive version (see Section 4.2.6).

4.2.8 Proactivity and Reactivity: A Mixing Algorithm

Predictors are inherently weak against exceptional events and relying solely on them for scaling decisions can lead to issues. For instance, if the predictions underestimate the workload, the scaling algorithm will fail to adapt the target system to the current workload, producing unacceptable performance. Conversely, if the predictions overestimate the workload, the scaling algorithm will deploy unnecessary microservice instances, leading to a waste of money. To address these issues, we design a proactive-reactive approach that mitigates the impact of inaccurate predictions and prevents inappropriate scaling decisions. Specifically, *in the context of this Dissertation*, we introduce a novel algorithm, based on a weighted sum, to linearly combine the predicted and measured workloads (using reactive signals from the monitor). For brevity sake, we only present the real-world Python implementation of our algorithm, nonetheless the ABS one is fully equivalent.

Our algorithm is based on using the system MCL to cast the comparison as the capacity of the system to deal with a given workload, defined by its current scaling configuration. Hence, we have a way to detect both over- and under-estimations of predictions, driven by the distance from the system MCL (of the scaling configuration) induced by the measured traffic. We do not directly compare the predicted and measured workloads at a given monitoring window: the interaction between message queues and scaling times makes it difficult to reliably estimate the accuracy of the predicted scaling configuration with respect to traffic fluctuations.

We consider statically-defined scores s_i for each architectural reconfiguration Δ_i, computed accounting for the increment in system MCL. For each Δ_i, we have a differential system MCL increment of: $\Delta MCL_1 = 60$ for Δ_1 and $\Delta MCL_i = 90$ for Δ_i with $2 \leq i \leq 4$. Given ΔMCL_i, we compute:

$$s_i = \frac{\Delta MCL_i}{\sum_{j=1}^{4} \Delta MCL_j}.$$

Notice that, this yields $\sum_{i=1}^{4} s_i = 1$.

At each monitoring window m, we proceed as follows. In step 1, as shown in Listing 4.8, we compute, for each index i, the difference $\texttt{diff}_i$ between the Δ_i, needed to cope with the predicted workload at $m-1$ and those for the measured one at m (`pred_conf` and `actual_conf` below, respectively). Then, as can be

Listing 4.8: `compute_diff` code

```
def compute_diff(self, pred_conf, actual_conf):
 diff = []
 for i in range(len(pred_conf)):
   diff.append(pred_conf[i] - actual_conf[i])
 return diff
```

seen in Listing 4.9, we compute $w \in [0, 1]$, used to linearly combine the predicted and measured workloads to determine the target value that drives scaling actions. Since $|\mathsf{diff}_i| > 1$ only happens in exceptional cases (*e.g.,* the predicted workload

Listing 4.9: `compute_weight` code

```
def compute_weight(self, pred_conf, actual_conf):
 curr_weight = 0.0
 diffs = self.compute_diff(pred_conf, actual_conf)
 for i in range(len(pred_conf)):
   curr_weight += abs(diffs[i] * self.scores[i])
 return min(curr_weight, 1)
```

is so distant from the measured one that it induces a scale configuration with a difference of Scale4), we compute:

$$w = min\left(\sum_{i=1}^{4} s_i \cdot |\mathsf{diff}_i|, 1\right).$$

We keep track of w values computed in the last 3 monitoring windows with the function $h = \{(1, w_{m-2}), (2, w_{m-1}), (3, w_m)\}$: w_m is the weight computed in the current window and w_{m-2}, w_{m-1} are the preceding ones. The pairs $(1, w_{m-2})$ and $(2, w_{m-1})$ are added only if the system was already running at those times. This mechanism is implemented in Listing 4.10, where we append the new weight passed as parameter to `errors` (*i.e.,* a `List` of weights), possibly removing the oldest one.

Listing 4.10: `store_weights` code

```
def store_weights(self, weight):
 self.errors.append(weight)
 if len(self.errors) > self.error_limit:
  self.errors.remove(self.errors[0])
```

In step 2, we compute the distance

$$dist = \frac{\sum_{(i,w)\in h} w \cdot i}{\sum_{(i,\_)\in h} i}$$

between the measured and predicted workloads (where $w \cdot i$ indicates that the most recent w is the most influential one) at the monitoring window m. The closer the distance (computed in Listing 4.11) is to 1, the less accurate the prediction is.

Listing 4.11: `store_distance` code

```
def compute_distance(self):
 num = 0.0
 den = 0.0
 for i in range(len(self.errors)):
   num += self.errors[i] * (i + 1)
   den += i + 1
 return num/den
```

In step 3, in the `mix` method (see Listing 4.12), we linearly combine the predicted and measured workload (*i.e.,* `predicted` and `measured` below) using the result of `compute_distance` to find the load the system has to cope with.

Listing 4.12: `mix` code

```
def mix(self, measured, predicted, pred_conf, actual_conf):
   curr_weight = self.compute_weights(pred_conf, actual_conf)
   self.store_weight(curr_weight)
   react_score = self.compute_distance()
   pred_score = 1 - react_score
   target = (react_score * measured) + (pred_score * predicted)
   return target
```

4.2.9 Executable Model and Real-World Implementation

In this Section, we start introducing the ABS algebraic model of the proactive-reactive global scaling platform (shown in Figure 4.1), designed using our integrated timed modelling/execution language (detailed in Chapter 3). Following this, we proceed to illustrate the concrete implementation of our platform, shedding light on the underlying adopted technologies. Our main focus lies in first evaluating the performance of our novel proactive-reactive global scaling platform early on at design level, to determine if the effort to implement its real-world version worth it; then we proceed to the evaluation of the real-world implementation. The executable model of the global scaling platform is available at [Bac24e], while the real-world implementation at [Bac24a, Bac24b].

Executable model. We now present, for each element of the platform depicted in Figure 4.1, its ABS counterpart describing how we model it. As done in Section 4.3.4, we adopt our modelling/execution language to design our global scaling approach and test its performance early on. We first design and model the Email Message Analysis Pipeline (see Section 4.2.2) as a set of ABS communicating classes. Together with the code implementing the system behaviour, we define the microservice characteristics, *e.g.,* strong/weak ports and required resources, via Timed SmartDeployer annotations. The *deployment orchestration engine*, *i.e.,* the component in charge of executing deployment orchestrations, is implicitly represented by the Erlang backend we use to run ABS programs (recall, our deployment orchestrations are ABS programs). The *Adaptation Algorithm* and *Actuation Module* are modelled as ABS classes implementing the algorithms described in Sections 4.2.6 and 4.2.8. Notice that, in our executable model, we use our Timed SmartDeployer tool to automatically synthesise the deployment orchestrations enacting the global reconfigurations (*i.e.,* those defined in Section 4.2.5). Thus, we need to define annotations expressing the characteristics of the VMs used in our simulations and deployment constraints, *e.g.,* number of instances per microservice type (see *Resources* and *Deployment Constraints* in Figure 4.1). The *Monitor* implements the scaling condition presented in Section 4.2.6 and it is modelled as an active class (see Section 2.3) periodically executing its behaviour.

Differently from the platform depicted in Figure 4.1, in our ABS executable model, we do not have any counterpart modelling the predictive module presented in Section 4.2.7. Instead, we statically inject in our simulations the predictions (generated via our predictive module) related to the Enron corpus dataset. The reason is the following. The predictions use physical time, while ABS simulations logical time. Since there is no direct mapping among these time values, if we were to mix them, we would generate simulations with inconsistent timing behaviour.

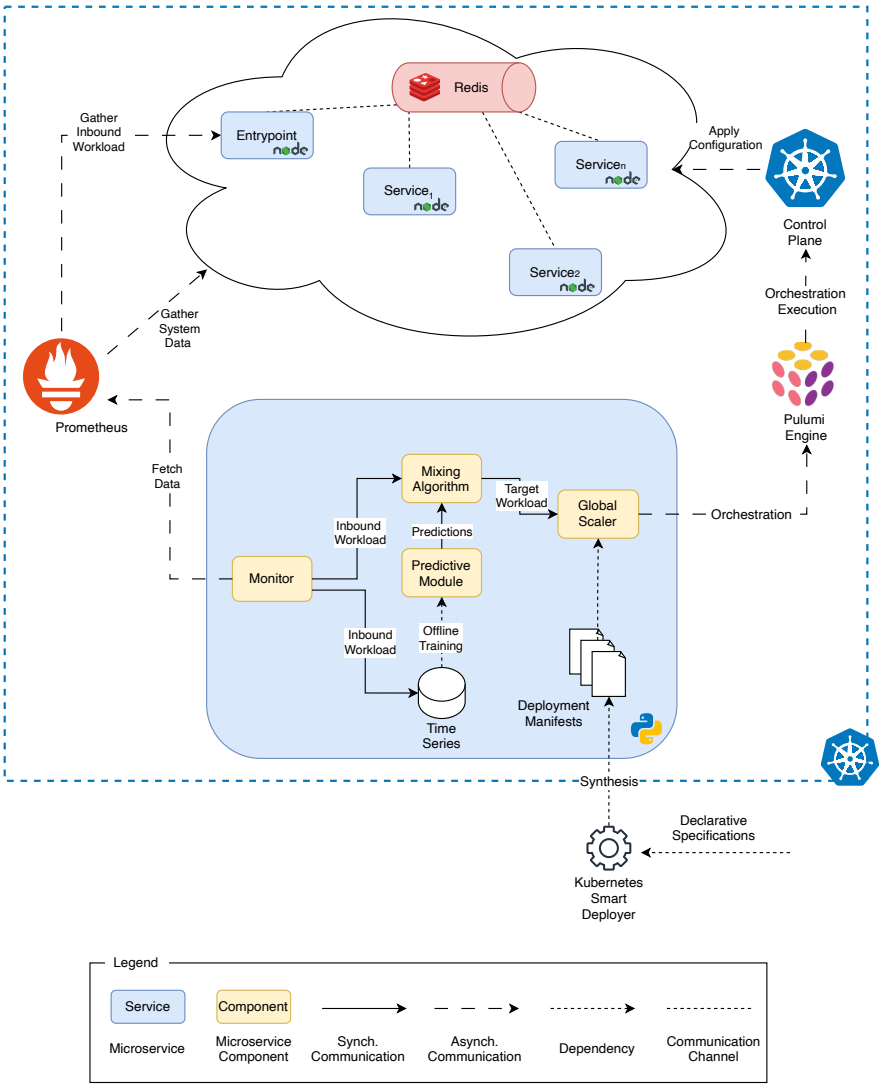


Figure 4.3: Implementation of the global scaling platform

Real-world Implementation. As can be seen in Figure 4.3, the implementation of the proactive-reactive global scaling platform requires the adoption of a wide range of technologies. We automatically build the Kubernetes cluster hosting the platform and manage its deployment (together with the one of the Email Message Analysis Pipeline, depicted within the cloud in Figure 4.3) via Terraform [Has], an Infrastructure as a Code (IaaS) tool, defining and managing infrastructure resources via declarative configuration files (see [Bac24b] for examples).

To record and monitor system metrics, we use the Prometheus tool [RV15]. Prometheus collects and stores metrics as time-series data, recording information with a timestamp and optional key-value pairs called labels. It offers a powerful query language for real-time data analysis, enabling users to create flexible queries.

As can be seen Figure 4.3, since the *Monitor*, *Adaptation Algorithm* (Global Scaler in Figure 4.3), *Actuation Module* (Mixing Algorithm in Figure 4.3) and *Predictive Module* are tightly coupled and communicate synchronously, we strategically place them within the same Kubernetes pod as a single service, implemented using the Python programming language. Notice that, this is just a design choice to avoid both the creation of too many services and introduce an unnecessary network communication overhead, but, if the user wants, it is possible to deploy them separately. Periodically, the *Monitor* component fetches data from Prometheus, forwards them to the mixing algorithm and stores them for future training operations of the *Predictive Module*. Once the *Global Scaler* has decided the Scales to apply, it exploits the Pulumi engine to execute the corresponding orchestrations (Pulumi, in turn, leverages Kubernetes to perform the actual changes to the system configuration). Notably, in our real-world implementation to synthesise the deployment orchestrations we use our novel *K8sSD*.

The Email Message Analysis Pipeline is implemented using Typescript [BAT14] (*i.e.,* a strongly typed version of JavaScript) and uses Redis [Red24], an in-memory data store, as database to store crucial information, *e.g.,* pending tasks, the result of email analyses. Notice that, Redis also handles the communication among services: we use its Stream feature, to manage communication among microservices and model queues of fixed-length size. Redis Stream is a data structure that allows users to manage and process streams of messages in a log-like manner, making it well-suited for real-time data processing and building message queues. A key

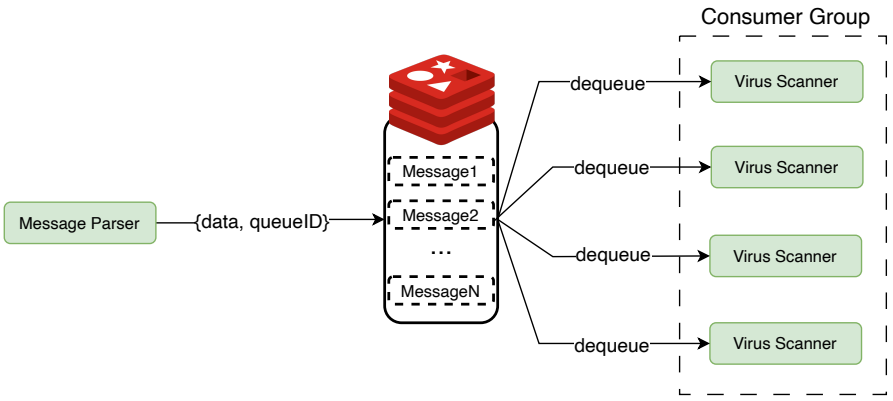


Figure 4.4: Microservice communication via Redis Stream

feature we crucially use in our implementation is the *consumer groups*: it allows multiple consumers (*i.e.,* several microservice instances) to process messages in parallel. An example of how this mechanism works is depicted in Figure 4.4: the *Message Parser*, acting as a producer, enqueues a message in a specific queue, *i.e.,* the *Virus Scanner* one. Redis ensures that each message is delivered to exactly one consumer, *i.e., Virus Scanner* instances in Figure 4.4, in the group. Each instance of the *Virus Scanner* reads messages independently allowing their parallel processing and analysis. Unlike the ABS implementation, this setup does not use dedicated load balancers for each microservice type. Redis acts as a single global load balancer uniformly distributing messages among consumers of queues. Being that all instances of a given type are idempotent (they consume messages at the same rate), the real-world implementation behaves as the ABS one: all microservice instances receive the same amount of requests.

4.2.10 Experimental Settings and Evaluation

In our benchmarks, to test the effectiveness of our proactive-reactive global scaling algorithm, we employ the experimental configuration outlined in Table 4.3.

In both simulated and real-world environments, we deploy a cluster composed by 10 virtual machines, each one equipped with 2 vCPUs and 4 GB of memory. In the real-world scenario, these virtual machines are provisioned by Digital Ocean and we use Kubernetes as orchestration engine; in the simulated environ-

Nodes	10
Node resources	$2vcpu$ with 4 GB of RAM
Request window	1s
Monitoring window	10s
ABS time unit	1 time unit = 30ms
Dataset	Enron Corpus dataset [KY04]

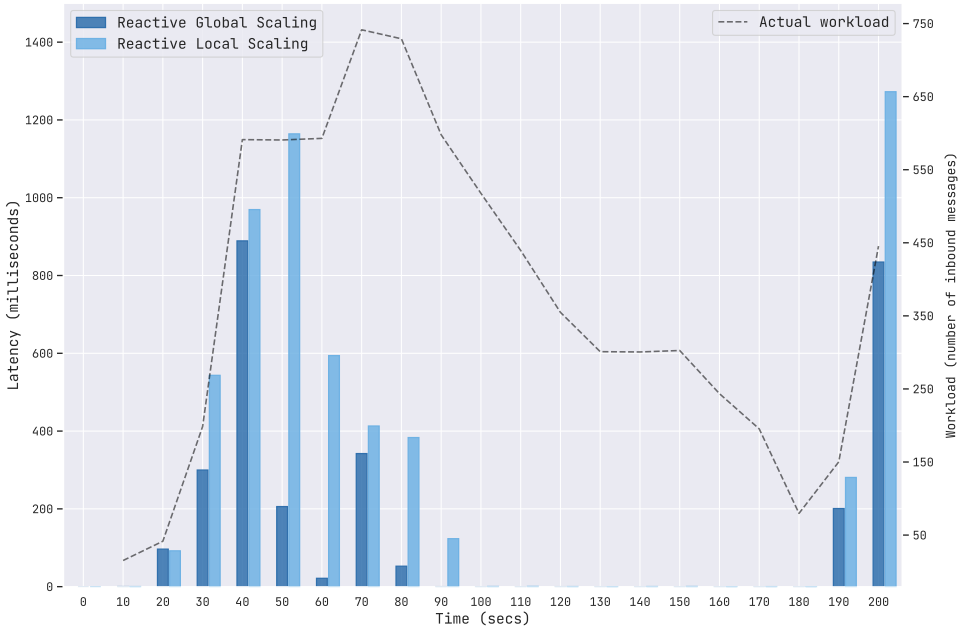
Table 4.3: Experimental settings

ment virtual machines are modelled as Deployment Components and the engine is represented by the Erlang backend (in charge of executing the simulation and consequently, the synthesised orchestrations). In both environments, we implement a workload generator service evenly distributing requests, as specified in the Enron dataset, at each second. Notice that, the Enron dataset uses, as time scale, hours, while our workload generator uses seconds. Such a discrepancy is intentional: for scalability testing, we treat emails within the Enron dataset as if they were emails per second, to simulate high-load conditions and assess system performance under stress. We apply the same reasoning to our predictor: it forecasts the amount of emails expected in the next hour to ensure predictions to be consistent with the dataset hourly time scale, but we treat them as emails per second.

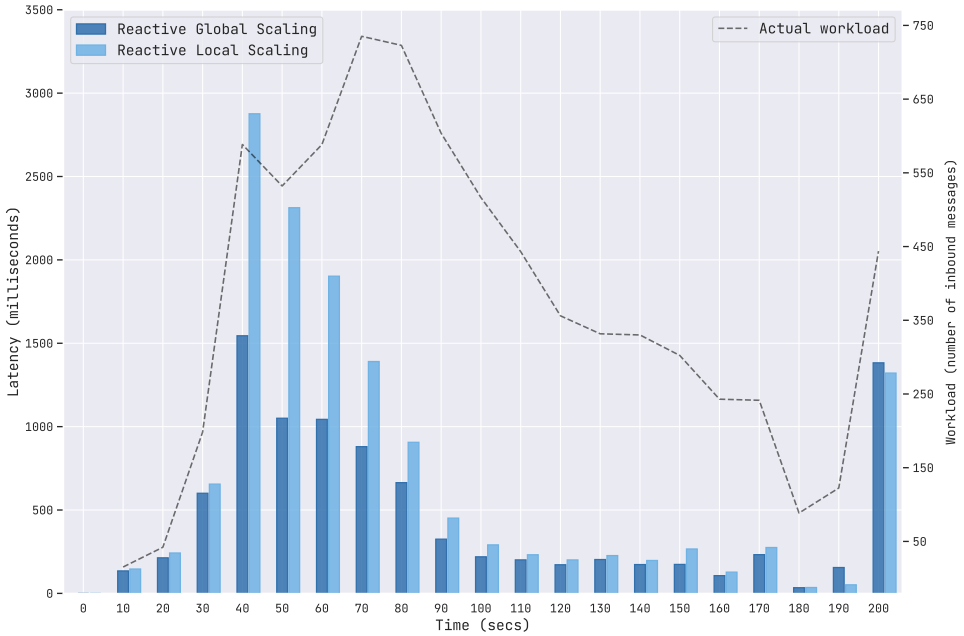
To ensure that our benchmark results are statistically significant, we conduct a total of 25 independent runs for each benchmark, with each run lasting approximately 200 seconds. This approach helps us achieving a statistically significant set of data that can be analysed to extract meaningful insights. Specifically, we focus on several key performance metrics, including latency, message loss and the number of deployed instances. We calculate the average values for these metrics to evaluate benchmarks. The graphs resulting from our analysis may exhibit some discrepancies in the workload used, however this is expected given the inherent differences between simulated and real-world environments. Both the simulated benchmarks and their real-world counterparts use the same workloads, which allows for a direct comparison between the two. In the simulated environment, we have the advantage of perfectly capturing the inbound workload, as the simulation can be controlled with high precision (being that it runs as a single program). Conversely, in the real-world setup, the measurements are gathered via Prometheus,

a monitoring and alerting toolkit, subject to network uncertainties. These uncertainties, *e.g.,* varying network delays, can lead to variations in measurements.

Reactive global vs reactive local scaling. The first benchmark we run is the comparison between the reactive version of the global (see Section 4.2.6) and local scaling (see Section 3.6.2). For each benchmark and metric, we show the results obtained from the simulated environment and the real-world one. In Figures 4.5a and 4.5b, we respectively evaluate the latency (considered as the average time the system needs to completely analyse an incoming email) in the simulated and real-world environments. As can be seen, in both environments, the extent of improvement brought by our global scaling algorithm is significant, outperforming the mainstream scaling approach. The reason is that our algorithm is capable of adapting the whole architecture as soon as a workload burst is detected at the entrance of the system, avoiding the "domino effect", which, instead, affects the local scaling. The results in Figures 4.5a and 4.5b are also confirmed by the message loss comparison: as can be seen in Figures 4.6a and 4.6b, our algorithm always loses far fewer messages, than the local one, testifying its capacity to faster adapt the system. Finally, the last benchmark of this group concerns the number of deployed instances. This comparison is crucial to understand the reason why our approach performs better with respect to the local one. As can be seen in Figures 4.7a and 4.7b, it is clear the local scaling suffers from the "domino effect": whenever the inbound workload grows, the number of deployed microservice instances grows linearly over time, causing a severe delay in the adaptation process, due to local monitoring of workload. Hence, with respect to global adaptation, where microservices in the target configuration are deployed together, the number of instances grows slower and performance worsen.

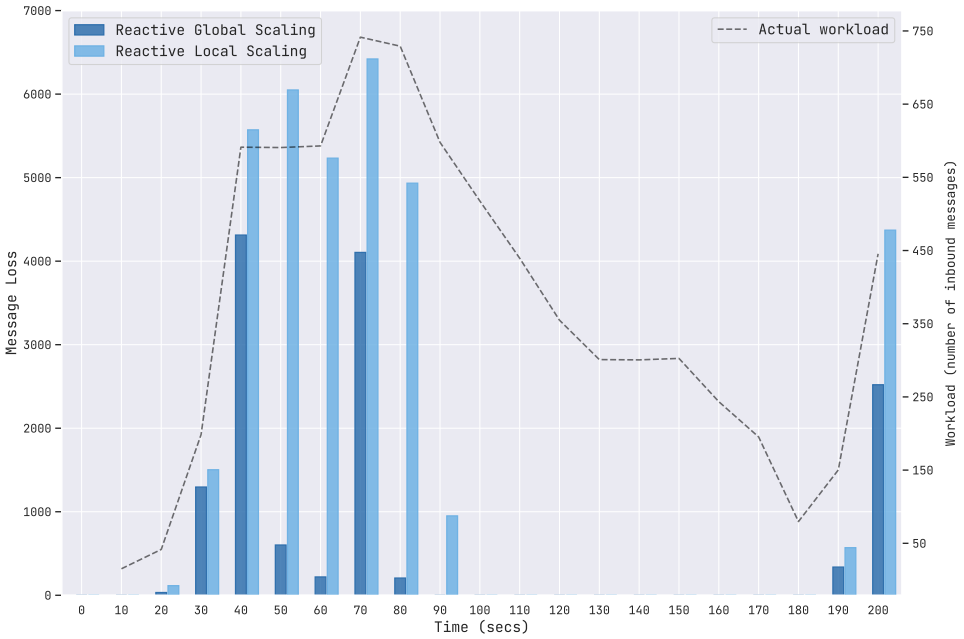


(a) Simulated execution

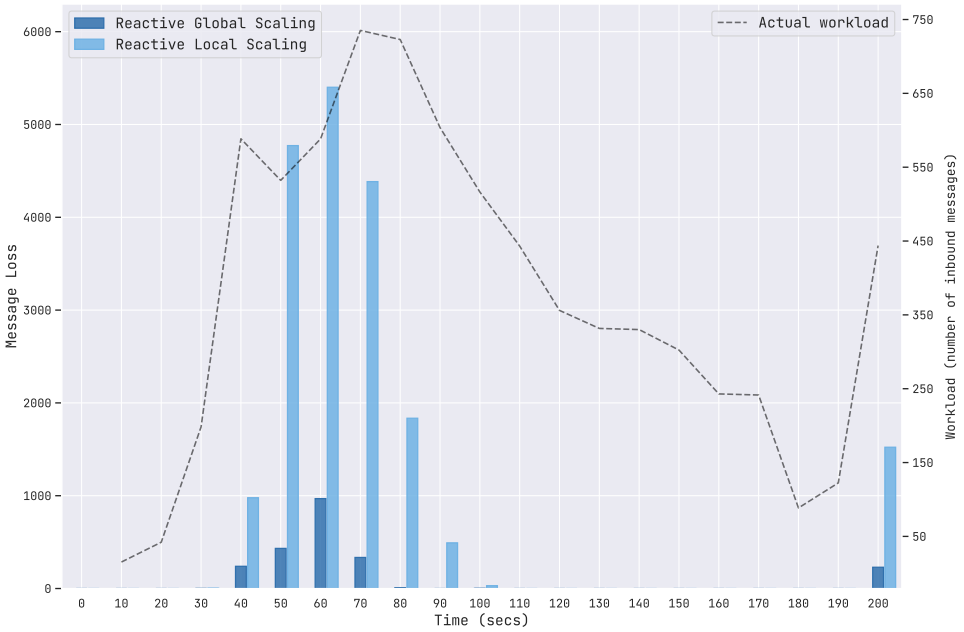


(b) Real-world execution

Figure 4.5: Reactive global and local scaling: latency

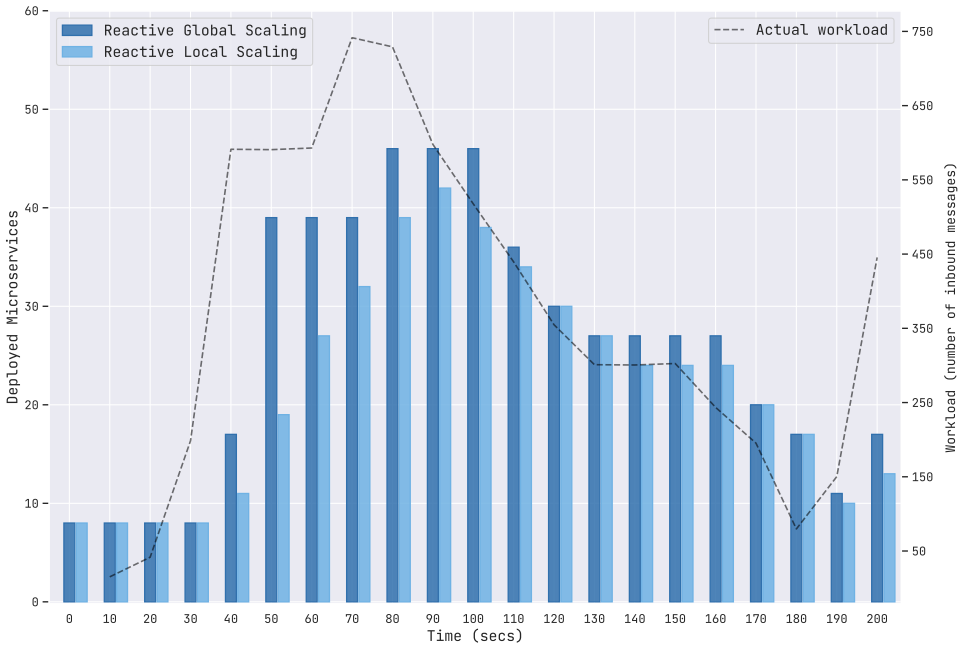


(a) Simulated execution

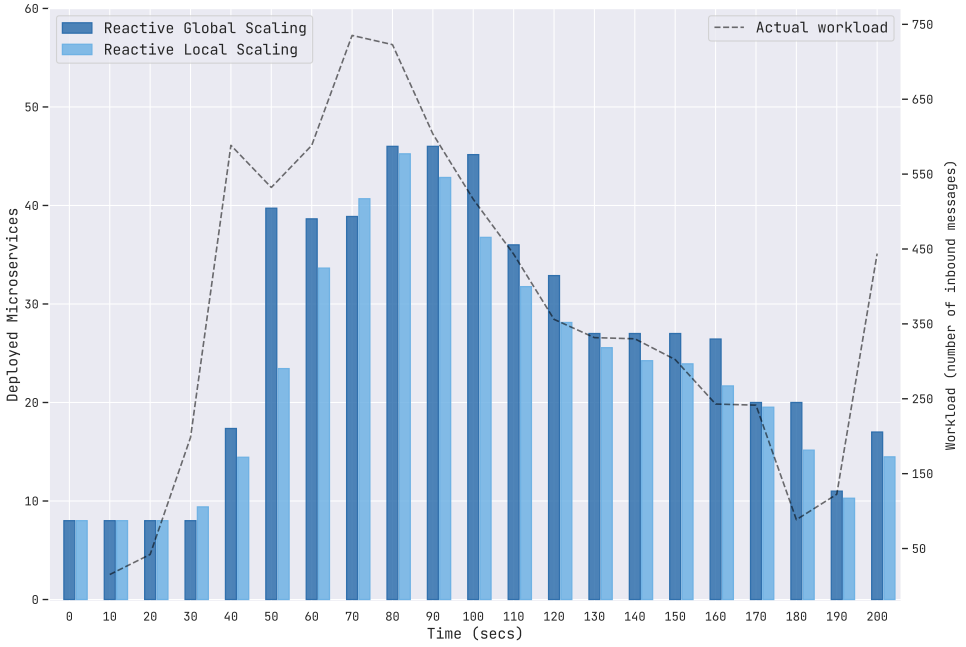


(b) Real-world execution

Figure 4.6: Reactive global and local scaling: message loss



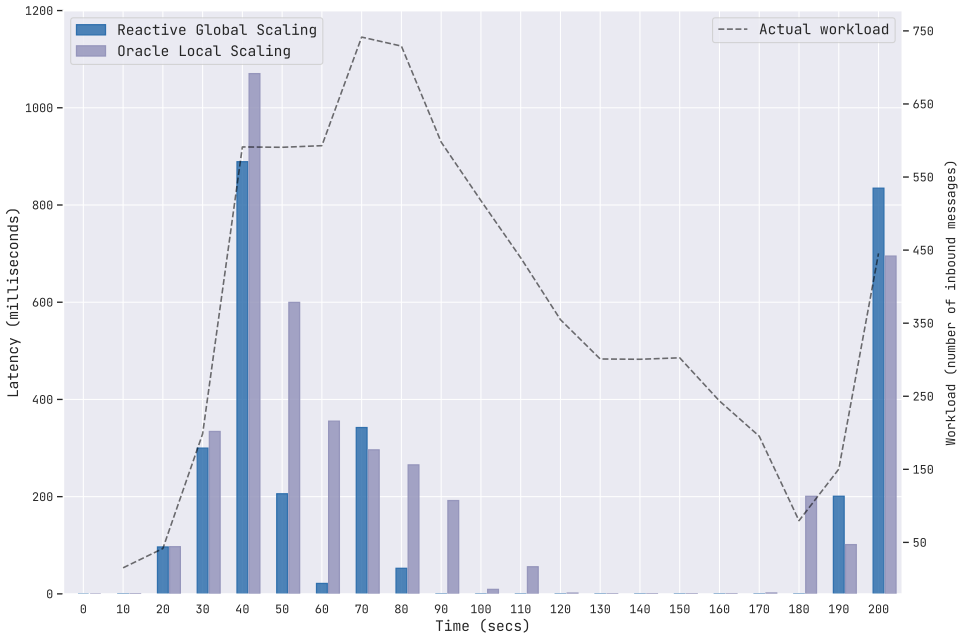
(a) Simulated execution



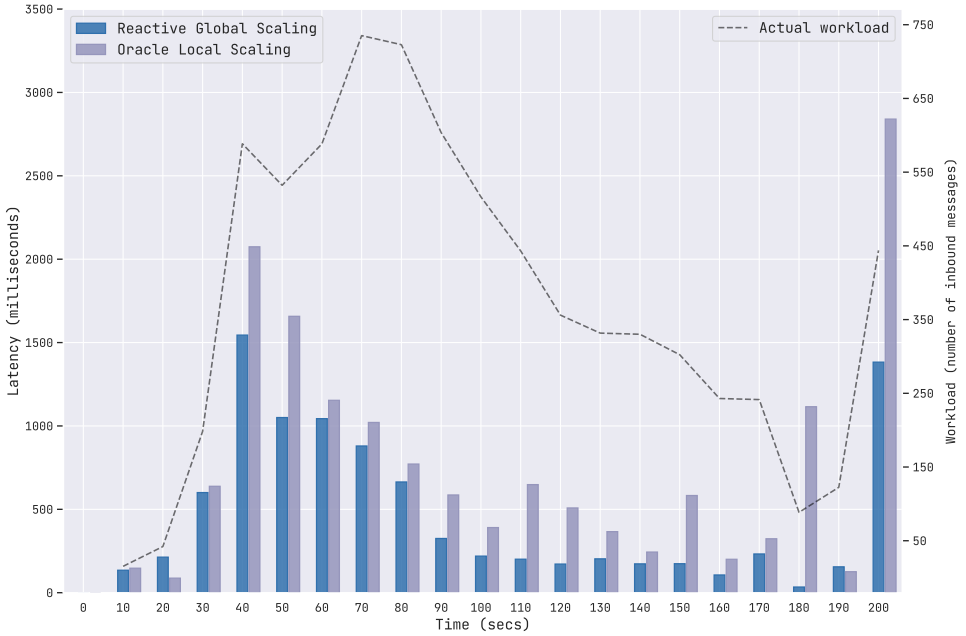
(b) Real-world execution

Figure 4.7: Reactive global and local scaling: deployed instances

Reactive global vs oracle local scaling. In the following benchmarks, we compare the reactive global scaling against the local one endowed with an oracle, *i.e.,* a perfect predictor, capable of knowing in advance the exact amount of workload entering the microservices. Despite the local scaling knowing in advance the exact number of requests in each microservice, our approach still performs better. As can be seen in Figures 4.8a and 4.8b, our approach has significantly better performance, always keeping latency under acceptable values. A similar trend is observed when considering the message loss comparison (see Figures 4.9a and 4.9b). Once again, our approach demonstrates a clear advantage over the mainstream method, as it results in significantly fewer message loss, ultimately providing superior performance. The implications of reduced message loss are critical, especially in systems where reliability and timely communication are essential for maintaining performance and meeting performance expectations. The reason behind this improvement is clear when we consider Figures 4.10a and 4.10b, which take into account the number of deployed instances: despite proactivity (*i.e.,* the capacity of predicting workload peaks), the local scaling still suffers from the "domino effect". Proactivity, as can be seen comparing Figures 4.10a and 4.10b with Figures 4.7a and 4.7b, merely shifts the problem one monitoring window backward, instead of completely eliminating it. While proactive approaches attempt to anticipate and address issues before they arise. Such approaches, whenever applied to local scaling, often fail to fundamentally resolve the underlying limitations, as evidenced in the figures. This side-by-side comparison highlights that, although proactivity may smooth the problem, it does not provide a long-term solution to the "domino effect" drawback. On the other hand, the comparison further underscores the significant advantages of exploiting functional dependencies: by leveraging them, we can completely eliminate the adverse effects associated with the local scaling.

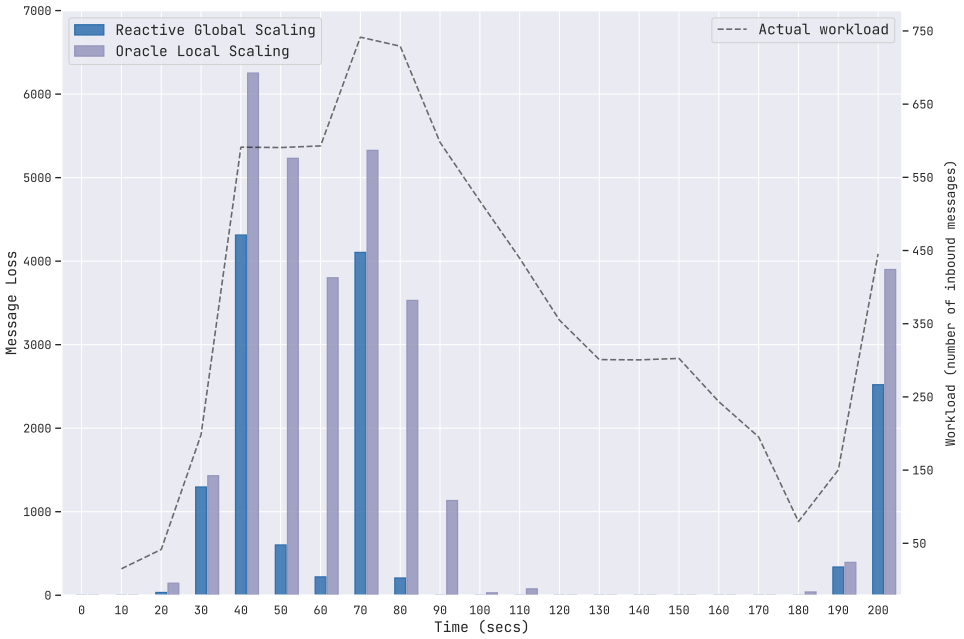


(a) Simulated execution

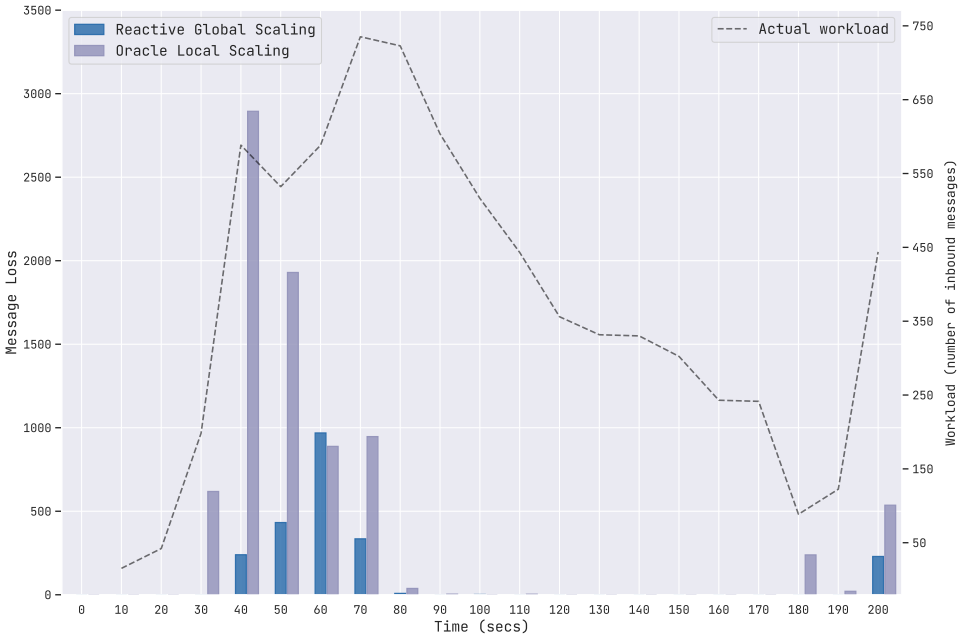


(b) Real-world execution

Figure 4.8: Reactive global and oracle local scaling: latency comparison

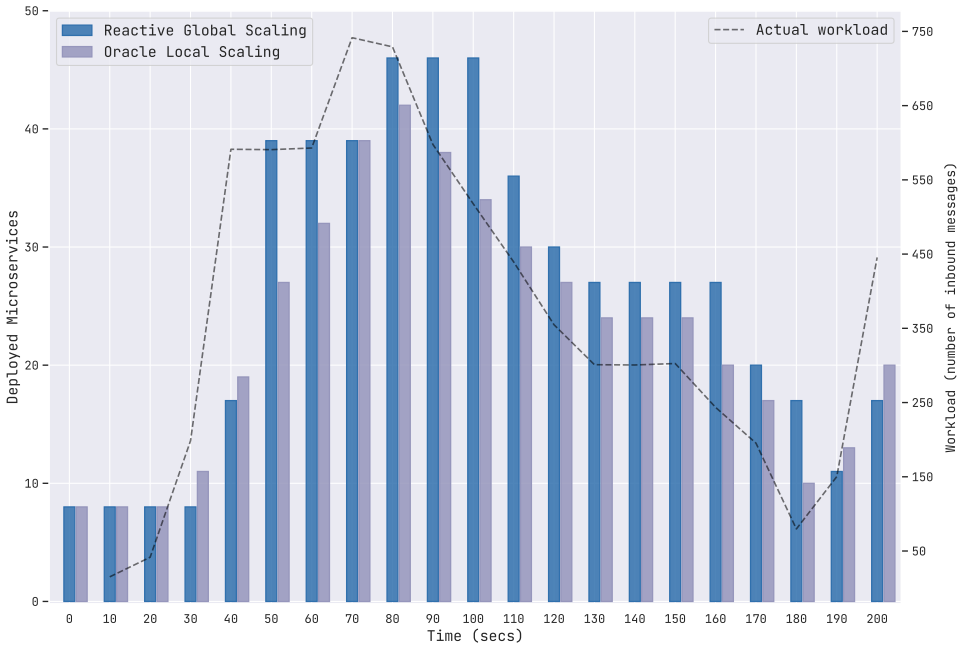


(a) Simulated execution

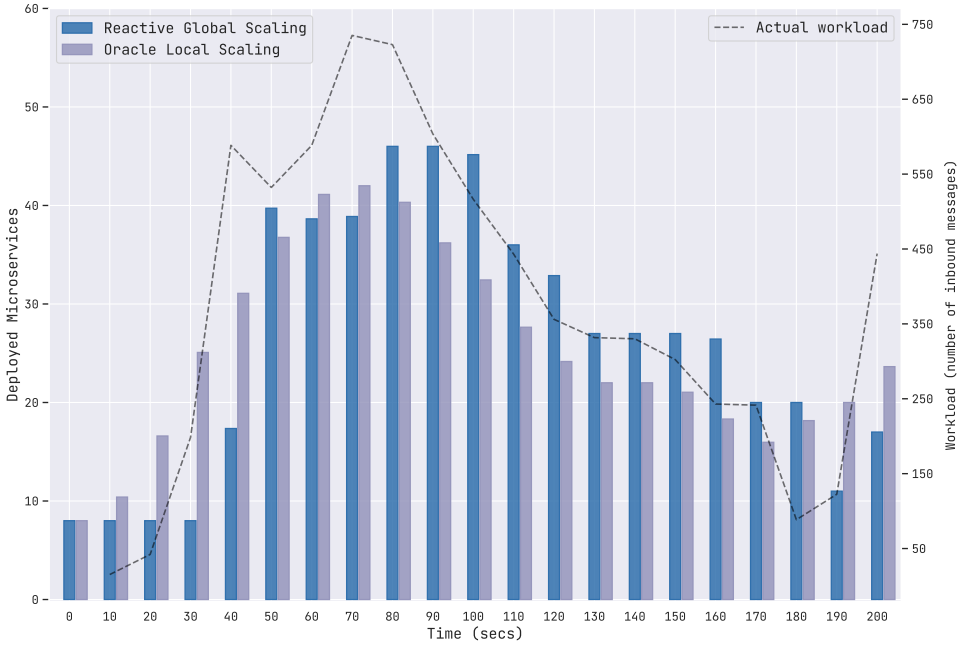


(b) Real-world execution

Figure 4.9: Reactive global and oracle local scaling: message loss



(a) Simulated execution

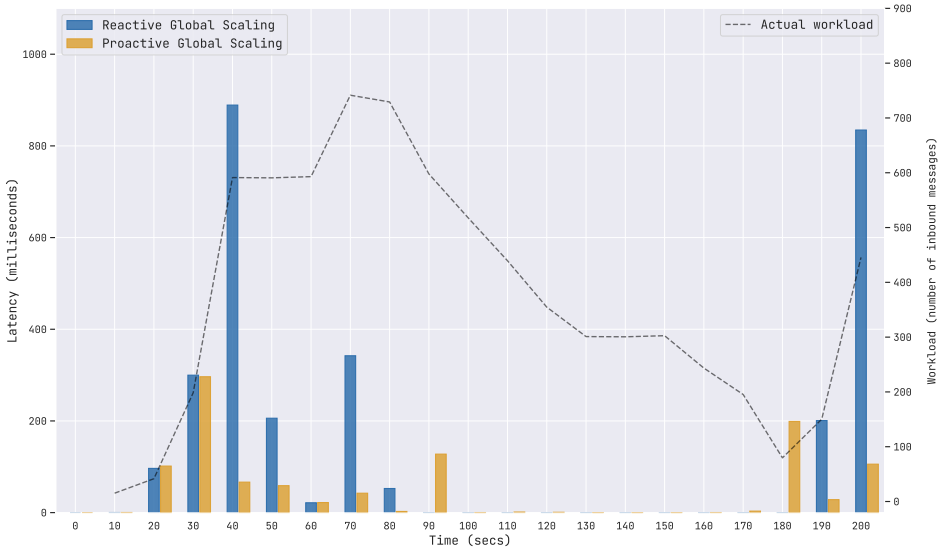


(b) Real-world execution

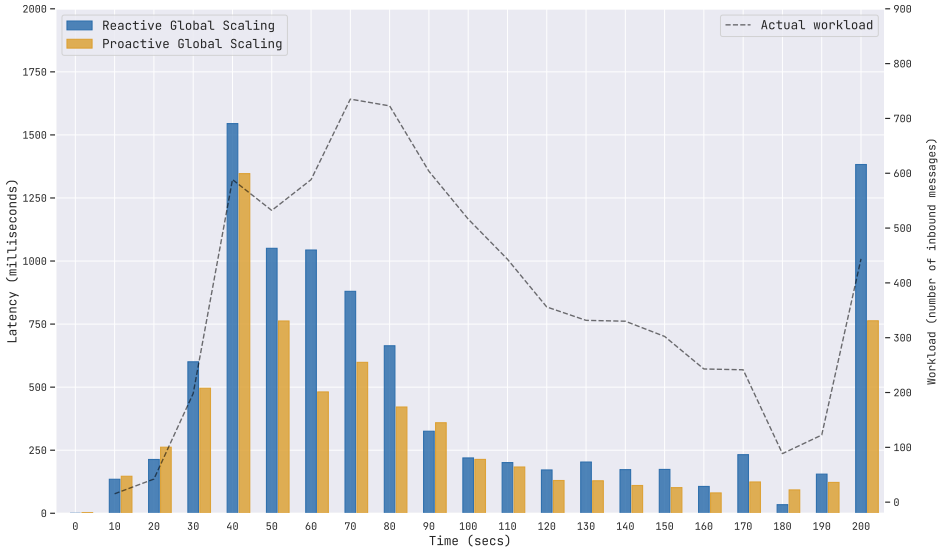
Figure 4.10: Reactive global and oracle local scaling: deployed instances

Proactive vs reactive global scaling. In the following experiment, we aim to compare the effectiveness of our proactive global scaling algorithm with the reactive one, assessing the magnitude of the improvements it offers. The proactive global scaling algorithm, as detailed in Section 4.2.7, leverages a machine learning-based workload predictor to anticipate scaling requirements before they arise. This predictive capability enables the system to implement scaling actions in advance, preparing it to accommodate peaks in workload before they occur. By contrast, the reactive algorithm only responds to scaling needs after they have manifested, which can lead to delays and inefficiencies in managing system performance. The results of this comparison, illustrated in Figures 4.11a and 4.11b and Figures 4.12a and 4.12b, demonstrate the tangible benefits of proactive global scaling. Specifically, the proactive approach significantly reduces both latency and message loss, thereby ensuring a higher and more consistent performance. The results show a marked improvement in performance metrics, underscoring the effectiveness of anticipating and addressing workload demands before they impact system operation. This proactive strategy, not only enhances system responsiveness, but also contributes to a more reliable and robust service experience. The reason behind such an improvement can be observed in Figures 4.13a and 4.13b: the proactive approach always anticipates the reactive one in deploying the instances required to handle the workload peaks.

Notice that, contrary to what the reader may think, anticipating workload peaks does not consume more resources with respect to waiting for them to occur (as in the reactive version). As a matter of fact, the anticipation of scaling actions occur in both directions, *i.e.,* adding and removing resources, thus the resources used in the proactive approach are equivalent to those used in the reactive one.

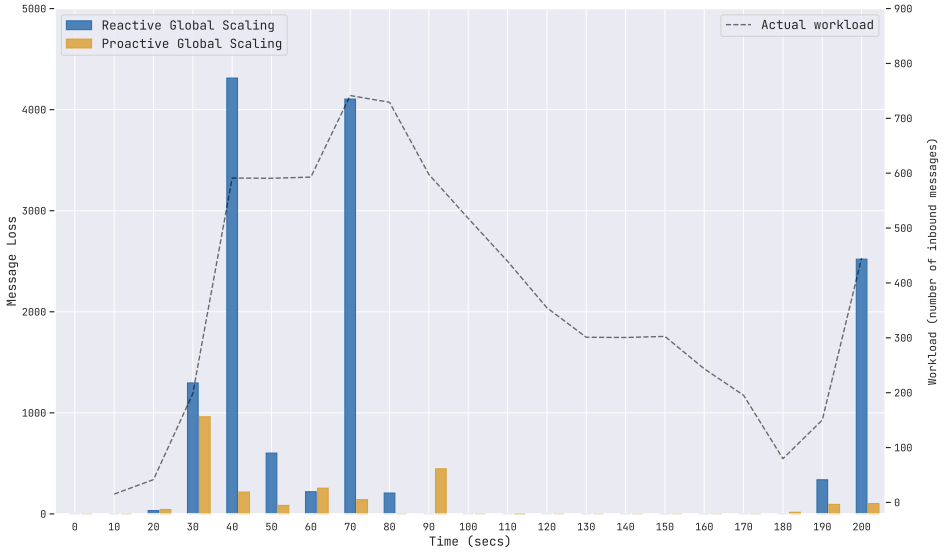


(a) Simulated execution

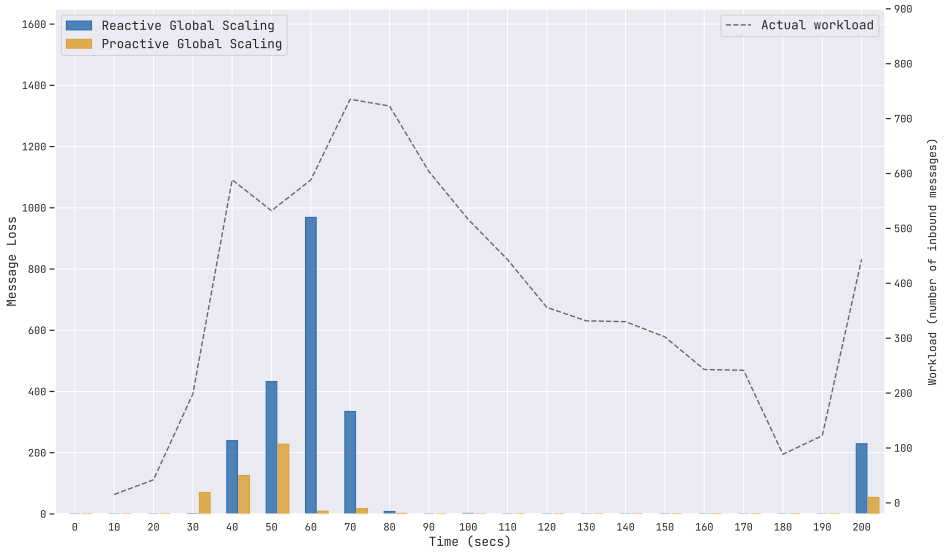


(b) Real-world execution

Figure 4.11: Proactive and reactive global scaling: latency

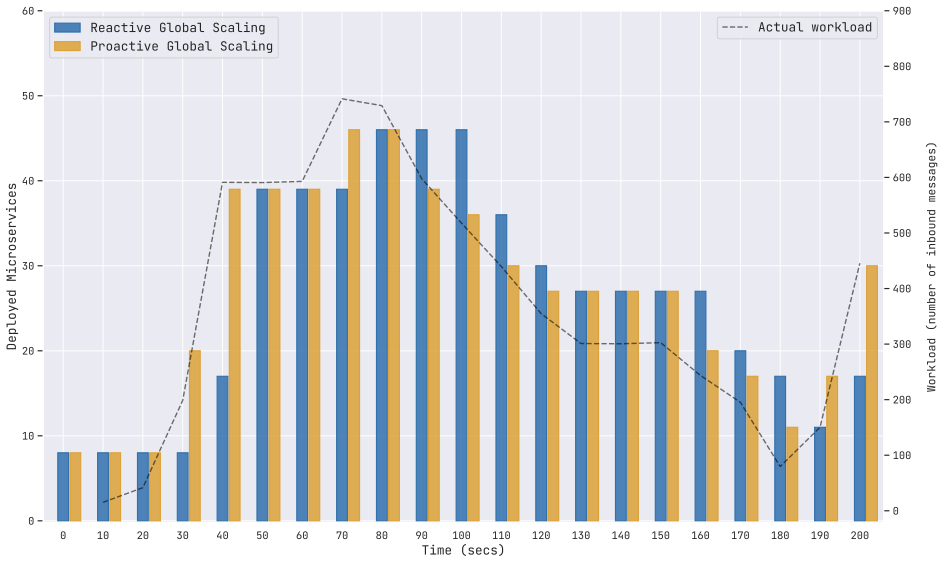


(a) Simulated execution

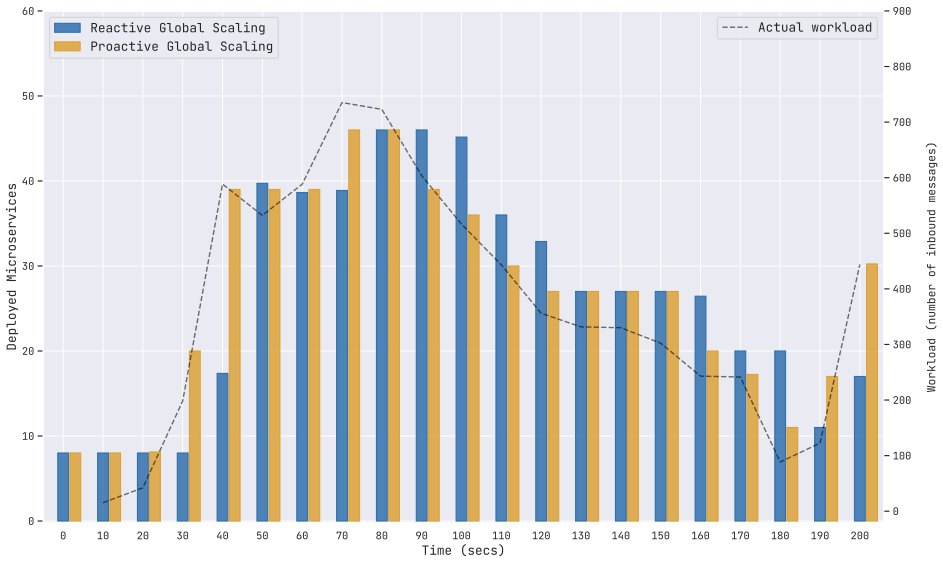


(b) Real-world execution

Figure 4.12: Proactive and reactive global scaling: message loss



(a) Simulated execution



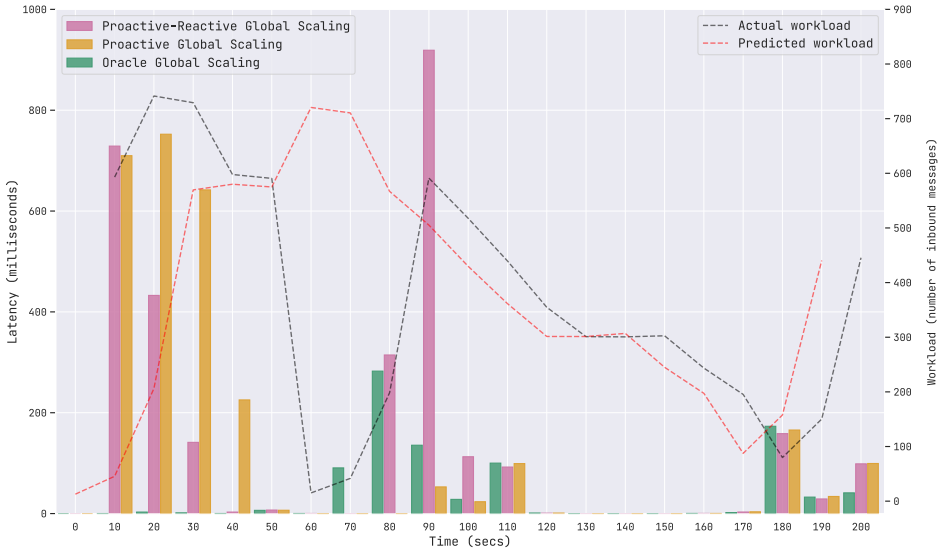
(b) Real-world execution

Figure 4.13: Proactive and reactive global scaling: deployed instances

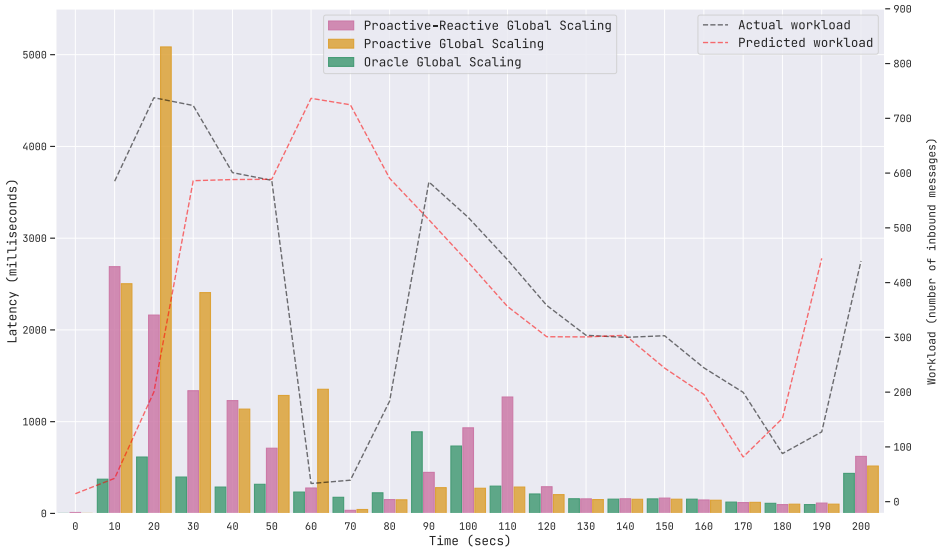
Proactive vs proactive-reactive global scaling. The last benchmark we run concerns testing the performance of our proactive-reactive global scaling algorithm. To do that, we selectively pick outliers from the Enron dataset, in order to have a workload our predictor struggles to forecast. Here, we endow the global scaling with an oracle, *i.e.,* a perfect predictor, capable of perfectly predicting the inbound workload at the entrance of the system, to serve as benchmark for optimal performance (especially, concerning the number of deployed instances). Notice that, it may happen that, even though the oracle knows in advance the exact amount of workload entering the system, the latency and message loss are not always zero. The reason is that the calculation of the scaling configurations, presented in Table 4.2, assumes an average structure of emails, *e.g.,* two attachments per email, which does not always correspond to the actual one.

As can be seen in Figures 4.14a and 4.14b, our algorithm for mixing the measured and predicted workloads is crucial: while the proactive approach is not capable of adapting to the initial undetected peak, the proactive-reactive one, as soon as it detects a difference between the measured and predicted workloads, resorts to the mixing algorithm to compute the adjusted workload the system has to support. The same behaviour can be observed Figures 4.15a and 4.15b: the proactive-reactive approach has a significant reduced message loss with respect to the proactive one.

Dually, as shown in Figures 4.16a and 4.16b, our mixing algorithm is capable of avoiding the waste of resources due to predictions overestimating the actual workload. As a matter of fact, as can be seen between 50-90s, the proactive-reactive approach, after having detected the unexpected workload pit, it adjusts the number of deployed instances according to the newly computed target workload, also accounting for the measured one. On the contrary, the proactive approach, by just considering the workload predictions, installs unnecessary microservice instances, producing a significant waste of resources.

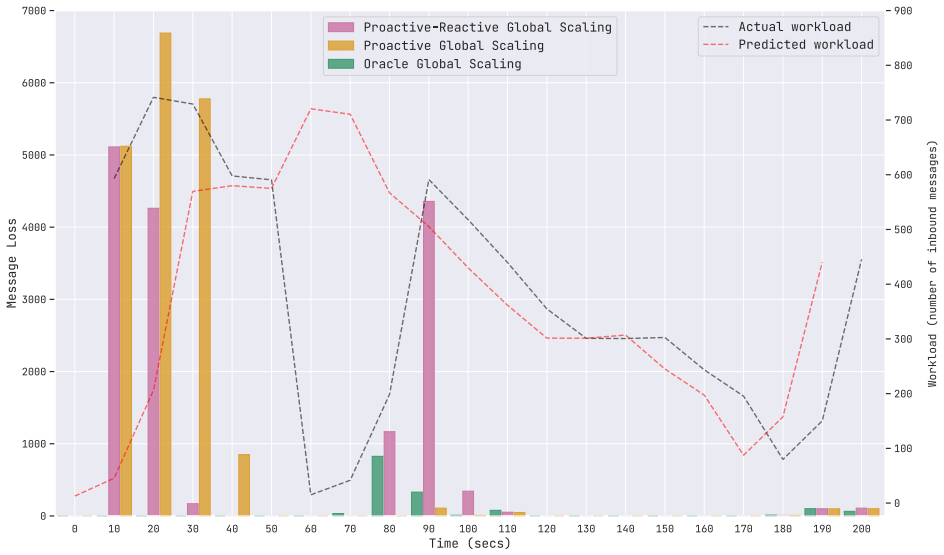


(a) Simulated execution

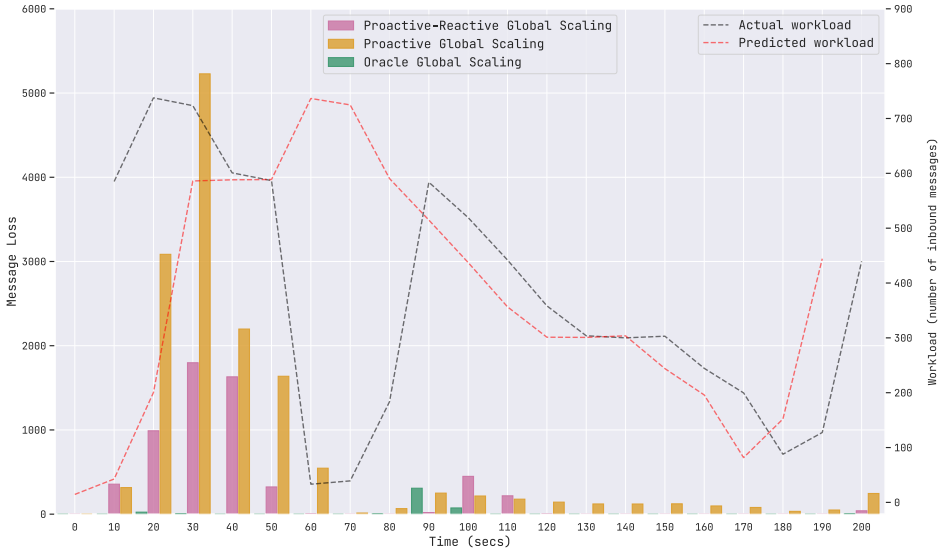


(b) Real-world execution

Figure 4.14: Proactive and proactive-reactive global scaling: latency

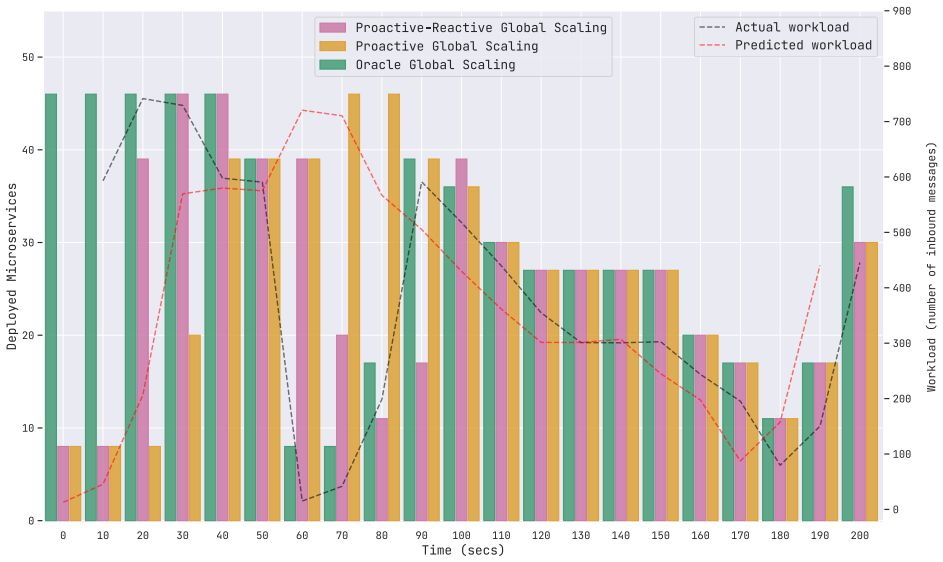


(a) Simulated execution

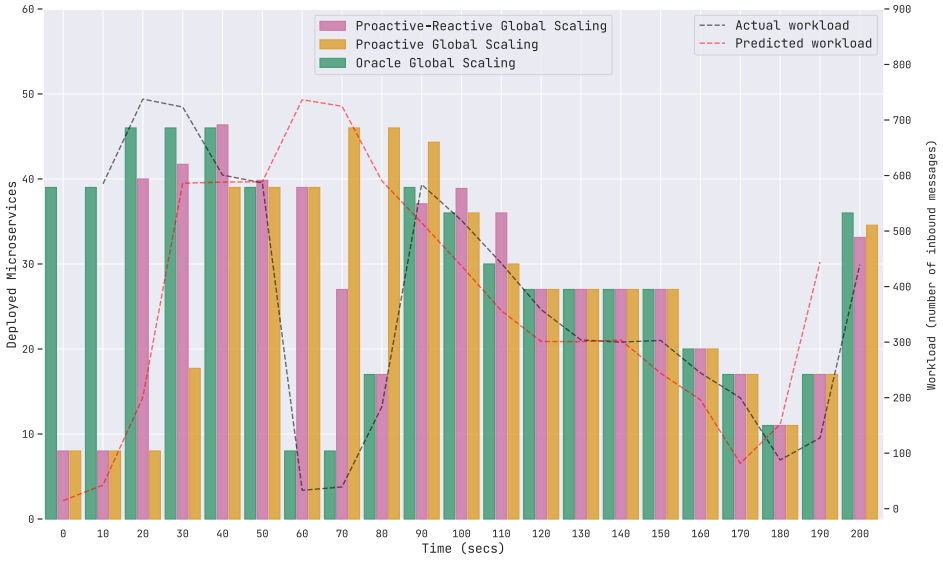


(b) Real-world execution

Figure 4.15: Proactive and proactive-reactive global scaling: message loss



(a) Simulated execution



(b) Real-world execution

Figure 4.16: Proactive and proactive-reactive global scaling: deployed instances

4.3 Edge-Cloud Continuum Service Migration

In Industry 4.0 scenarios, the Internet of Things (IoT) and cloud computing are considered key technological enablers of industrial process monitoring and automation [XYGG18]. Among the enabled applications, early fault detection of assembly line machines plays a crucial role to minimise operation downtime and maximise production efficiency. Most of the reported deployments of industrial fault detection systems are based on a cloud-centric approach, *i.e.,* the sensory data collected from the line machines are transferred to cloud infrastructures, where they are stored and analysed *e.g.,* via machine learning tools. While this approach can ensure service scalability, it does not fit the requirements of emerging, time-critical industrial applications that introduce strict Quality of Service (QoS) requirements in terms of reliability and latency of the operations. This is the case, for instance, of industrial robots or unmanned/guided vehicles used for transportation of tools and products [FSZS18]. Supporting low latency operations in time-critical Industry 4.0 scenarios is becoming a major research challenge and pushing the adoption of novel solutions in two complementary domains. On the communication side, standards like Time Sensitive Networking (TSN) and the 5G [AMS+22] are envisioned to minimise data acquisition latency, while ensuring deterministic delivery of messages. On the processing side, edge computing solutions [QCZ+20] have been largely investigated as a viable alternative to reduce the processing latency and save bandwidth. Generally speaking, edge computing is a relatively new paradigm that attempts to offload computational tasks to devices nearby IoT data sources: example of edge devices may include micro-controllers (also referred to as the extreme edge), micro-computers, Base Stations (BS) or servers, all sharing the common feature of being geographically close to data sources. To do that, edge computing solutions involve multiple components [XLL+20]: (*i*) edge *caching*, *i.e.,* techniques allowing to store portions of data on edge devices; (*ii*) edge *intelligence*, *i.e.,* techniques aimed at extracting knowledge from the cached data, often adapting existing machine learning techniques to be executed on hardware constrained edge devices; and (*iii*) edge *offload*, *i.e.,* assign tasks to other devices in case a single edge node does not have enough resource to execute them. Regarding the latter, recent work enlarges such concept through the emerging paradigm of

the edge-cloud continuum, which refers to the possibility of seamlessly allocating resources and workloads to edge/fog/cloud nodes, based on resource utilisation or QoS metrics [SWVDT21]. At the same time, while several components of the continuum, *e.g.,* the task allocation policies, have been investigated, few work reports real-world deployments in industrial scenarios.

In the context of this Dissertation, we attempt to fill such gap describing the design and implementation of a software architecture for Industry 4.0 scenarios, enabling edge-cloud continuum operations of machine monitoring and fault detection. The architecture has been deployed within the SEAmless loW lAtency cLoud pLatforms (SEAWALL) project founded by the BIREX[2] consortium, a competence center for the Industry 4.0 recently established in Bologna, Italy. The project involved academic researchers and companies, the latter being technical service providers or end-users. The proposed architecture implements an *orchestration-based architectural reconfiguration technique* to perform dynamic orchestration of workloads among the nodes of the edge-cloud continuum, *i.e.,* service migration, and address time-aware data processing support. As done in Section 4.2.4, starting from declarative descriptions of, *e.g.,* component characteristics and deployment requirements, we automatically synthesise the deployment orchestrations our technique needs to perform the architectural reconfiguration. Services to process Data can be dynamically activated/deactivated on different edge/cloud nodes to minimise the overall latency, while the edge servers are not overloaded. To this aim, we validate our platform in a use case proposed by an industrial automation company (Bonfiglioli S.P.A.). The use case focuses on the seamless allocation of the service in charge of preparing data coming from a production line for subsequent anomaly detection tasks between edge and cloud nodes. We evaluate different allocation policies varying the amount of data generated by the production line and measuring the corresponding latency to generate the alarm.

4.3.1 The Industry 4.0 Use Case

The SEAWALL platform is designed as a solution to a case study proposed by Bonfiglioli S.P.A, a world leader manufacturer in industrial automation, mobile ma-

[2]https://bi-rex.it/en/

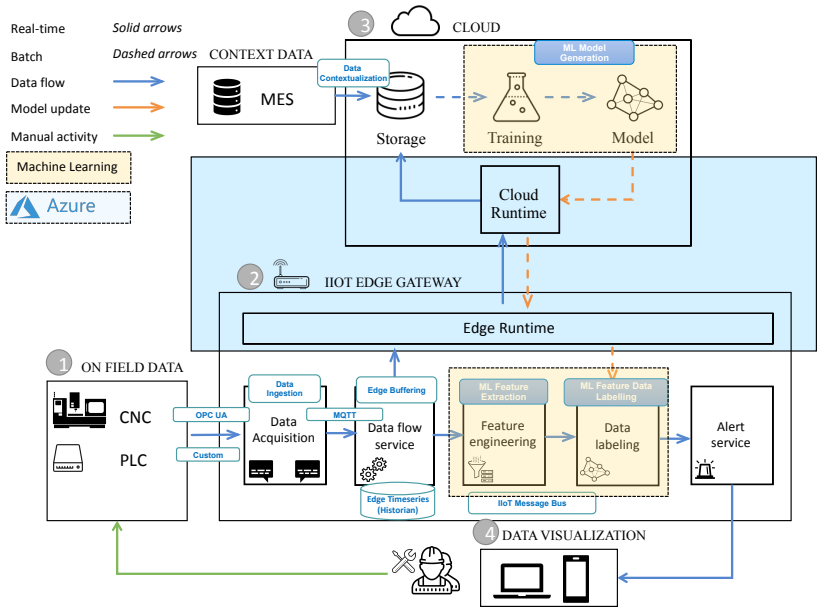


Figure 4.17: Bonfiglioli industrial automation architecture

chinery and wind energy. It concerns, as can be seen in Figure 4.17, an edge-cloud continuum system to control and perform anomaly detection over CNC (Computer Numerical Control) or PLC (Programming Logic Controller) machines. Such machines produce various kinds of data, transmitted following specific protocols, *e.g.,* OPC UA, describing their current working state. The data must be stored to perform data analyses, including the production of anomaly detection models, based on machine learning techniques. Due to the heavy amount of storage and computational resources needed to store the data and train the machine learning models, this part of the system is expected to be deployed in the cloud. The machine learning models are periodically re-trained also considering the most recent data and then used in the context of a control loop that, due to low latency constraints, is expected to be deployed on the edge. The control loop is triggered by an alerting service that periodically uses the machine learning model to classify the most recent data acquired from the production line and communicates the outcomes of the predictions to the operators. The use case cannot be considered as a strict real-time IoT system like the robotic/unmanned vehicle applications mentioned in Section 4.3, due to the usage of machine learning techniques (which

are generally delay-tolerant), best-effort networking solutions and because of the nature of the system output, which is mainly informative and does not involve automatic actions. However, there is still the need of minimising the latency of the data alerting process. The realisation of the system has been developed adopting edge-cloud solutions offered and managed by a public cloud provider.

This system has been already successfully experimented on one machine. Nevertheless, the company found out that the large scale adoption of the current architecture is not possible for several limitations. We mention below some of such limitations:

- a unique machine learning classifier deployed in the edge cannot manage an intensive flow of data, possibly produced in parallel by several machines connected to the same edge gateway;
- the implementation of several independent control loops (one for each machine or one for each flow of data possibly generated by different sensors in the same machine) cannot be managed due to the limited edge storage and parallel computing resources;
- the current system does not support dynamic automatic scaling of the computing resources at the edge level;
- the adoption of an edge-cloud solution managed by public cloud providers imposes further limitations: it is not possible to dynamically migrate components from edge to cloud and vice versa, depending on customer-defined load-balancing rules.

In the next Sections, we will present the design, implementation and experimental validation of an alternative architecture that attempts to overcome the serious limitations of the existing deployment, exploiting our architectural reconfiguration strategy [BPS+22a].

4.3.2 Low Latency Edge-Cloud Continuum Architecture

The microservice architecture proposed in Figure 4.18 includes two main layers, *i.e.,* the *cloud* and *edge* layers, with each macro-functionality mapped to a sin-

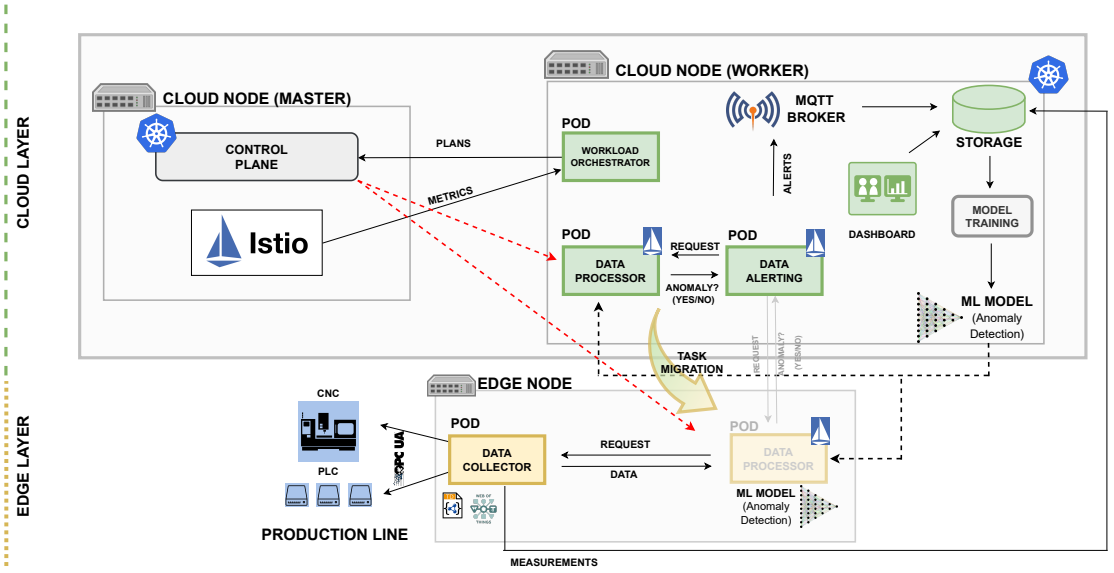


Figure 4.18: Low latency edge-cloud continuum architecture

gle component. More in detail, at the edge layer we find all services in charge of producing and collecting data. Data are generated by a *production line* that can be considered as the extreme edge layer of the architecture and consists of a set of industrial machineries equipped with heterogeneous sensors, monitoring the working conditions of the machineries themselves. We mapped the entire production line to a W3C Web Thing to take benefit of the new W3C Web of Things (WoT) standard [KML+20] in terms of interoperability. As a result, we abstract away from the specific machinery in use and from the sensor types, since new kind of machineries can be easily added, by providing the WoT mapping interface. An edge node, close to the data source, hosts two services: a static one, *i.e.,* the *Data Collector*, gathering data through the OPC UA protocol and a dynamic one, *i.e.,* the *Data Processor*, that, instead, is moved between the cloud and edge layers depending on the specific needs. The *Data Processor* can be invoked to retrieve the latest measurements from the *Data Collector* and executes anomaly detection tasks to discover possible misbehaviours of the production line. The *Data Processor* is constituted by a pre-trained machine learning model. Notice that, we do not propose any new machine learning technique for anomaly detection, since the focus is on the dynamic allocation of the workload of the processing task. As a result,

the SEAWALL platform is general enough to support multiple anomaly detection algorithms, assuming that the code for training and inference tasks is provided and properly connected to the *Data Processor* interface. In the cloud layer, we find the *Data Alerting* service, which is in charge of periodically triggering the *Data Processor*, in order to get the response of the anomaly detection task. Furthermore, the cloud layer hosts: (*i*) a storage service that acts as data lake of alerts and raw sensory data from the production line; (*ii*) an IoT dashboard showing the current machineries conditions and triggered alarms; and (*iii*) a workload orchestrator service, whose task is to migrate the *Data Processor* from cloud to edge and vice versa, under certain conditions. The dotted lines in Figure 4.18 highlight the possibility of re-training the machine learning model when new data are available on the storage service and transferring back the updated parameters to the model. Notice that, under normal conditions, the *Data Processor* is deployed in cloud, given the higher availability of resources with respect to the edge node. However, different policies can be easily implemented in the workload orchestrator, in order to adapt the entire system to the specific use case.

4.3.3 Latency and Size-based Policies

At the very core of our architectural reconfiguration technique for service migration we find the migration policies implemented in our *Workload Orchestrator*. Notice that, for brevity sake, we present only the real-world implementation (developed using the JavaScript programming language) of such policies, nonetheless the ABS (simulated) version is fully equivalent.

These policies can handle an arbitrary number of services and ensure that exactly one service is deployed within the edge node at any time. As we will see in the code excerpts below, migration operations are executed via the `moveToCloud` and `moveToEdge` functions, which encapsulate the necessary logic to run migration orchestrations, based on the provided service index. For each service to migrate, both policies retrieve the perceived latency and received data size (referred to as `latencies` and `bytes` in the code below). The policies then evaluate whether to enact migration operations based on these metrics, by looking for, in the case of the latency-based policy, the service with the highest latency or the highest

received data size, in the case of the size-based policy. The thresholds `lat_th_edge`, `lat_th_cloud` and `size_th` are used to guide these migration decisions. These thresholds can be adjusted to accommodate the specific constraints of the system under examination.

Concerning the latency-based policy, as can be seen in Listing 4.13, we look for the service perceiving the greatest latency (stored in the `max` variable): if such latency is greater than `lat_th_edge`, we migrate it to the edge. Notice that, to ensure that exactly one service is deployed on the edge at any given time, we make use of the `zone` array (*i.e.,* an array keeping track of the position of each service). If the orchestrator finds out that a service, different from the one we are currently migrating, is already deployed on edge, we first free the edge node and, consequently, migrate the service under consideration (lines 1-12). If the service

Listing 4.13: Latency-based policy

```
1  var max = Math.max(...latencies)
2  const index = latencies.indexOf(max)
3  if(max > lat_th_edge) {
4      if(zone.filter(z => z == "edge").length == 0) {
5          moveToEdge(index);
6      }
7      else {
8          const edge_index = zone.indexOf("edge");
9          if (edge_index != index) {
10             moveToCloud(edge_index);
11             moveToEdge(index);
12         }
13     }
14 }
15 else if(zone[index] == "edge" && max < lat_th_cloud) {
16     moveToCloud(index);
17 }
```

with the highest latency is already on the edge side and such latency is less than `lat_th_cloud`, the orchestrator moves such service back to the cloud.

Differently from the latency-based policy, in the size-based one (whose code can be seen in Listing 4.14), we look for the service receiving the highest amount of data and migrate it on the edge (if the edge node is free); otherwise we first move the service currently on the edge back to the cloud and, consequently, deploy the service under consideration on the edge. Again, this policy ensures that at most one service is placed on edge at any time (see line 5-11).

Listing 4.14: Size-based policy

```
var max = Math.max(...bytes)
const index = bytes.indexOf(max)
if (max > size_th) {
    if (zone.filter(z => z == "edge").length == 0) {
      moveToEdge(index);
    }
    else {
        const edge_index = zone.indexOf("edge");
        if (edge_index != index) {
          moveToCloud(edge_index);
          moveToEdge(index);
        }
    }
} else if (zone[index] == "edge" && bytes[index] < size_th) {
  moveToCloud(index);
}
```

4.3.4 Executable Model and Real-World Implementation

Within this Section, we start presenting the formal model of the architecture presented in Figure 4.18, crafted using our integrated timed modelling/execution language (detailed in Chapter 3). Following this, we proceed to illustrate the concrete implementation of this platform, shedding light on the underlying adopted technologies. Our main focus lies in first simulating and executing an operational model akin to Figure 4.18, then validating the obtained results using our real-world implementation.

Executable model. We leverage our integrated timed modelling/execution language (see Chapter 3) to precisely capture the behaviour of the platform illustrated in Figure 4.18. Through this modelling approach, we thoroughly analyse the performance of our architectural reconfiguration technique under various migration policies. For interested parties, both the SEAWALL algebraic model and the data analysis code are openly accessible at [Bac24d].

In the process of implementing our ABS algebraic model, we proceed as follows: (*i*) we devise static aspects of the architecture (annotations) to be inputed to Timed SmartDeployer (see Section 3.5), in order to automatically synthesise deployment orchestrations for the initial system deployment and subsequent migration actions; and (*ii*) we implement, via ABS code, the behaviour of services. During the modelling phase, we abstract away low-level details such as communi-

cation protocols and data generation. Instead, we focus solely on modelling the services composing the architecture, including the *Data Processor*, *Data Collector*, *Data Alerting* and the *Workload Orchestrator*. Unlike the ABS algebraic models presented in Section 3.6 where reconfiguration is enacted via service replication considering service throughput, here reconfiguration actions are represented by service migration operations, based on network-related metrics, *e.g.,* transmission speed and data volume. Thus, we leverage the ABS DataSize annotation (referenced in Section 2.3) to accurately model bandwidth usage, consistently marking ABS instructions modelling network communication among services.

However, capturing a realistic communication behaviour with ABS concepts like Deployment Component, `Bandwidth` and `DataSize` annotations, poses a challenge due to the uncertain nature of network communications. Achieving this realism necessitates the analysis of real-world data representing transmission speeds on both cloud and edge sides, without executing any migration actions. An intriguing observation arises when comparing transmission speeds of edge and cloud virtual machines, as depicted in Figure 4.19. Despite both sides having equivalent resources, the edge side exhibits faster transmission speeds. This discrepancy can be attributed to the proximity between data and the node hosting the computation using such data. When both data and services reside within the same node, the distance is effectively zero, resulting in expedited data reception. Notice that, network communication within the same node uses a virtualised instance of the Ethernet adapter, *i.e.,* vEthernet, which is influenced by the underlying network infrastructure. This explains why latency on the edge side is not zero.

In our communication modelling, we assume that the transmission speed at the edge is solely determined by the available bandwidth provided by the hosting virtual machine, without being significantly impacted by other external factors, *e.g.,* the distance between nodes. To quantify the relationship between latency and data sizes, we exploit the data presented in Figure 4.19 (taken from [BPS+22a]), comprising 10 latency measurements for each data size. From this dataset, we derive the actual bandwidth consumed on the edge side as the ratio of the *i-th* data size and the corresponding average latency $\mathsf{avg\_lat}_{\mathsf{i}}^{\mathsf{edge}}$, observed over these measurements on the edge side. This computation yields to a practical estimation of the effective bandwidth utilisation for different data sizes on the edge.

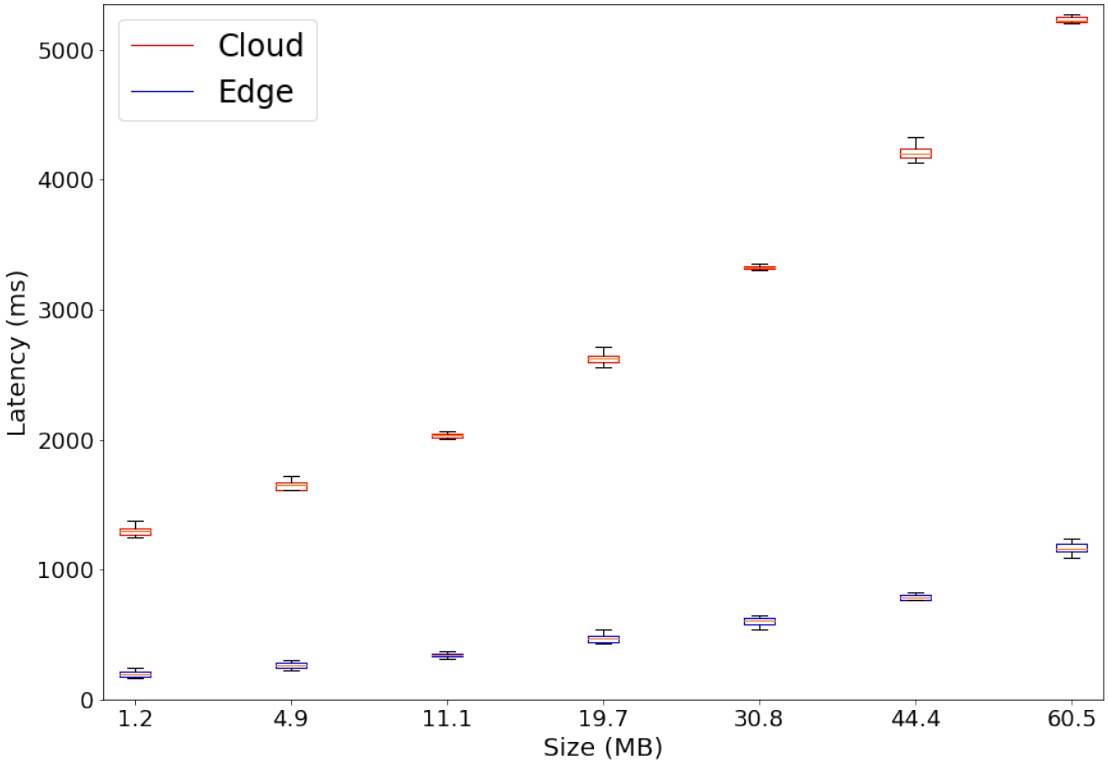


Figure 4.19: Transmission speed analysis [BPS+22a]

In our simulation, to align the communication behaviour to the one highlighted in Figure 4.19, we proceed as follows. We define $\mathsf{b}_{\mathsf{tot}}$ as the total bandwidth a Deployment Component can supply, s_i as the *i-th* size of transmitted data and $\mathsf{b}_\mathsf{i}^{\mathsf{edge}}$ as the *i-th* edge bandwidth usage computed as

$$\mathsf{b}_\mathsf{i}^{\mathsf{edge}} = \frac{\mathsf{s}_\mathsf{i}}{\mathsf{avg\_lat}_\mathsf{i}^{\mathsf{edge}}}.$$

We calculate the scaled data size values $\mathsf{scaled}_\mathsf{i}$, inputed to the `DataSize` annotation, as follows:

$$\mathsf{scaled}_\mathsf{i} = \mathsf{s}_\mathsf{i} \cdot \frac{\mathsf{b}_{\mathsf{tot}}}{\mathsf{b}_\mathsf{i}^{\mathsf{edge}}}.$$

This formula allows us to adjust the size values proportionally, ensuring that the resulting transmission times accurately reflect the observed transmission speeds.

Notice that, the $\mathsf{scaled_i}$ values are such that

$$\frac{\mathsf{scaled_i}}{\mathsf{b_{tot}}} = \frac{\mathsf{s_i}}{\mathsf{b_i^{edge}}},$$

independently of the value assigned to $\mathsf{b_{tot}}$: this is crucial to define a modelling methodology that is independent from the total bandwidth supplied by the virtual machines used in the real-world implementation, since it is difficult to precisely retrieve such information. Moreover, our modelling methodology is general enough and applicable to any cloud vendors.

However, $\mathsf{scaled_i}$ values do not take into account the cloud side slowness, caused by factors such as distance from the data source. Thus, we need to enrich our analysis with a *slowdown factor* computed as

$$\mathsf{fact_i} = \frac{\mathsf{b_i^{edge}}}{\mathsf{b_i^{cloud}}},$$

where $\mathsf{b_i^{cloud}}$ is computed following its edge counterpart methodology. Thus, given $\mathsf{fact_i}$, we compute $\mathsf{avg\_lat_i^{cloud}} = \mathsf{fact_i} \cdot \mathsf{avg\_lat_i^{edge}}$. To inject the previously computed slowdown factor in our simulation, we proceed as follows. Considering that, for the *i-th* data size, the `DataSize` annotation already produces a delay equal to $\mathsf{avg\_lat_i^{edge}}$, we need to account for the remaining delay of $(\mathsf{fact_i} - 1) \cdot \mathsf{avg\_lat_i^{edge}}$. Thus, technically, we simulate such an additional delay as computing time, exploiting the `Cost` annotation, passing as input $(\mathsf{fact_i} - 1) \cdot \mathsf{avg\_lat_i^{edge}} \cdot \mathsf{speed}$ (where `speed` is the Deployment Component speed).

Real-World Implementation. We present the primary technologies used in the implementation of the architecture detailed in Section 4.3.2. The backbone of our system, *i.e.,* the *production line*, is encapsulated as a Thing Description (TD), facilitating its exposure as a Web Thing. This integration is orchestrated through the *Data Collector* service, which acts as a conduit between the digital and physical words. Leveraging the OPC UA protocol, the *Data Collector* seamlessly interfaces with physical machinery, ensuring smooth data acquisition. Noteworthy is its adaptability to diverse data acquisition sources, thanks to the compliance with any WoT-compliant protocol. To achieve this versatility, we use

Node-wot[3], a robust NodeJS framework designed explicitly for the Web of Things (WoT). *Node-wot* empowers our system to consume and produce Web Things in accordance with the W3C WoT Scripting API specifications [4], thereby ensuring interoperability and seamless integration with the broader IoT ecosystem. Both the *Data Processor* and *Data Alerting* services are developed in Typescript, leveraging the NodeJS runtime[5]. They expose REST APIs, facilitating seamless data interaction. For message queuing, we rely on the open-source Eclipse *Mosquitto* MQTT broker[6]. Data storage is handled by an *InfluxDB* instance[7]. Additionally, the dashboard module is implemented as an Angular web application. To ensure flexibility and scalability, the entire architecture is containerised. Each service is encapsulated within a Docker container and managed by the Kubernetes controller running on the master node. This containerisation strategy enhances deployment efficiency and facilitates orchestration in dynamic environments. We maintain ongoing performance monitoring of the *Data Alerting* service with the assistance of the Istio framework[8]. Our focus lies particularly on alert generation latency, which encompasses both data retrieval from the *Data Processor* module and the execution latency of the machine learning model. These metrics serve as vital inputs for our custom *Workload Orchestrator* service, guiding resource allocation decisions for the *Data Processor*, whether on the cloud or edge side. This adaptive resource management ensures optimal performance and responsiveness of the system in dynamic environments. Currently, our migration policy triggers the offloading of the *Data Processor* service to the edge node once certain thresholds are hit. These thresholds are adjustable by the system administrator through a straightforward dashboard interface. Importantly, our orchestrator is designed to accommodate various migration policies with ease. For instance, it can seamlessly implement offloading strategies based on different factors, *e.g.,* the volume of data received by the *Data Processor* service. The output of our *Workload Orchestrator* is operationalised through the application of statically synthesised deployment or-

[3]https://github.com/eclipse/thingweb.node-wot
[4]https://www.w3.org/TR/wot-scripting-api/
[5]https://nestjs.com/
[6]https://mosquitto.org/
[7]https://www.influxdata.com/
[8]https://istio.io/

chestrations, automatically generated by the tool discussed in Section 4.1. These orchestrations are then executed to dynamically manage the activation and deactivation of pods across different nodes. Notably, this orchestration process occurs without the need for any active code migration (*i.e.,* it occurs by deactivating services on edge, activating them on cloud and vice versa), ensuring smooth and efficient resource allocation.

4.3.5 Experimental Settings and Evaluation

To test the efficacy of our edge-cloud continuum approach, we employ the configuration outlined in Table 4.4. In both simulated and real-world environments, we deploy a cluster comprising two nodes, each one equipped with 2 vCPUs and 4 GB of memory. In the real-world scenario, these nodes are provisioned by Digital Ocean and we use Kubernetes as orchestration engine. In the simulated environment, these nodes are modelled as Deployment Component and the orchestration engine is represented by the Erlang backend (in charge of executing the simulation and the synthesised orchestrations). Specifically, one node is located in London, serving as the representation of the cloud layer, while the other node is situated in San Francisco, mimicking the edge layer. This geographical distribution ensures a diverse and realistic testing environment, allowing us to assess the impact of our reconfiguration technique across different locations. Notice that, in the simulated environment such geographical distribution is modelled following the communication behaviour analysis described in Section 4.3.4. In the real-world scenario, there is an additional node, *i.e.,* the master one, where we deploy the Kubernetes control plane. To test the performance of our architecture, we deploy the *Data Alerting* in the cloud layer, the *Data Collector* in the edge one and allow the placement of the *Data Processor* on both.

The *Data Alerting* service operates with a request interval set to 3 seconds, whereas *Workload Orchestrator* conducts its activities at intervals of 10 seconds. To correctly model the timing behaviour in our ABS simulation, we first define an ABS time unit to be equivalent to 50 milliseconds and then we convert the above described time intervals from seconds to time units.

In a realistic industrial scenario, we confront the challenge of managing multiple

Nodes	3
Node locations	London, San Francisco
Node resources	$2vcpu$ with $4gb$ RAM
Request payload	3500 B
Request window	3s
Monitoring window	10s
ABS Time Unit	1 time unit = 50ms

Table 4.4: Experimental settings

independent pipelines. Deploying multiple services to the edge layer poses the risk of overloading edge nodes. In contrast, cloud nodes offer the advantage of elasticity, allowing for easier scalability, by adding nodes to the cluster as needed. In such a scenario, it becomes crucial to assess *Workload Orchestrator* policies that strategically deploy only one selected *Data Processor* service to the edge, while retaining the remainder in the cloud. This approach aims to mitigate the risk of overburdening edge nodes, while taking advantage on the computational power of the cloud. To gauge the adaptability of our architecture in handling multiple pipelines, we deploy three independent pipelines. Each pipeline follows a distinct profile for the inbound workload. In particular, the workload is computed as $\mathsf{size_i} \cdot \mathsf{size_i} \cdot \mathsf{req\_payload}$, where $\mathsf{req\_payload}$ is $\mathsf{3500B}$ and $\mathsf{size_i}$ assumes the following ordered sequences of values:

- $[80, 60, 20, 140, 80, 60, 40, 120]$ for pipeline 1;
- $[60, 40, 120, 80, 60, 20, 140, 80]$ for pipeline 2;
- $[20, 140, 80, 60, 40, 120, 80, 60]$ for pipeline 3.

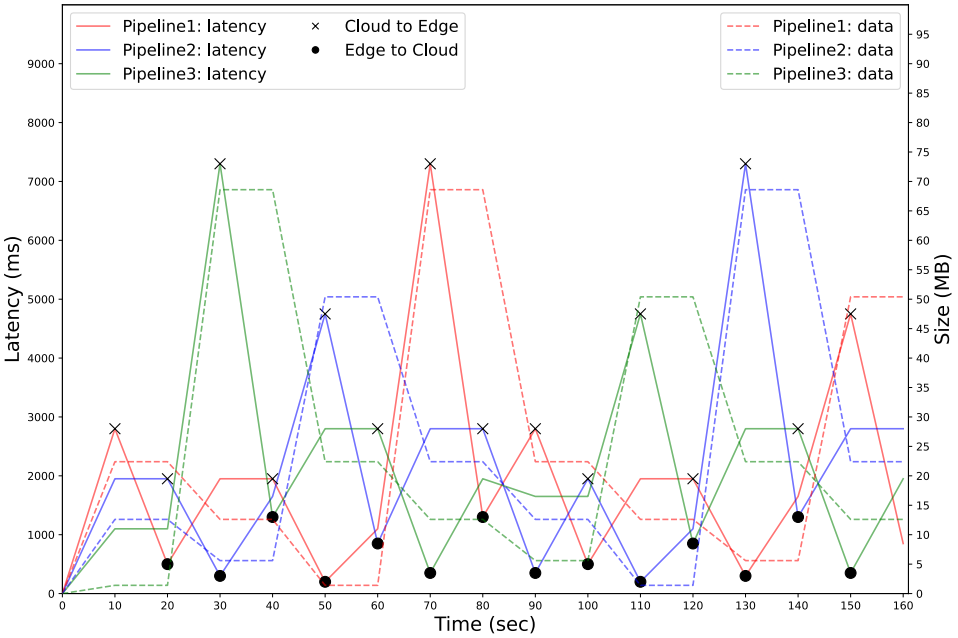
Notice that, even if we adopt the same workloads in the simulated and real-world environment, the data size perceived by the *Data Processor* differs between the two environments. The reason lies in the data optimisation/reduction occurring in the real-world network communication: since data are originally in a text-based format (*i.e.,* JSON) and encoded into a binary format (*e.g.,* Buffers or MessagePack), the resulting binary data are smaller [VK22].

To each pipeline, we impose the constraint that at most one *Data Processor* service can operate in the edge layer at any time. This constraint ensures a fair

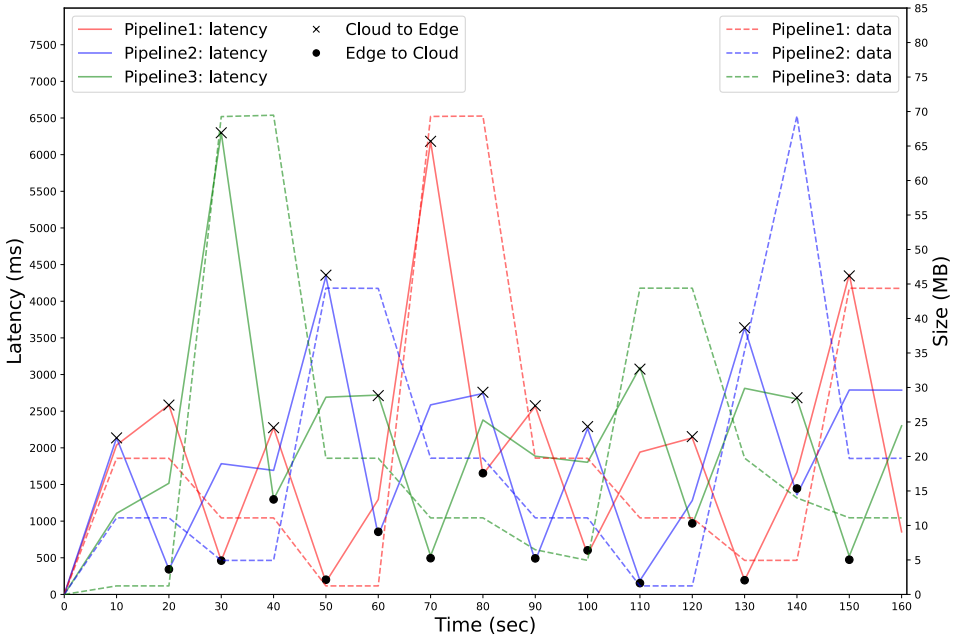
assessment of the system performance under realistic conditions.

We start with the three *Data Processor* services in the cloud layer and, every 10 seconds, the latency and workload size are queried. We compare the two policies described in Section 4.3.3: the first moves to the edge layer the service with the highest latency, meanwhile the second relocates the service with the highest incoming workload size. Given that, according to the Bonfiglioli requirements, it is acceptable to have a latency grater than 1s and strictly less than 2s, we design our *Workload Orchestrator* to migrate the *Data Processor* service in the cloud until specific conditions are met: (*i*) when the latency-based policy is applied, the service remains in the cloud as long as the latency falls below a threshold of 2 seconds (thus, we set `lat_th_edge` to 2s), while remains in the edge as long as the latency is greater than 1s (thus, we set `lat_th_cloud` to 1s); (*ii*) as can be seen in Figure 4.19, since a 11 MB size corresponds to a latency of 2s, when the size-based policy is applied, the service stays in the cloud as long as the request size is below a 11 MB threshold (thus, we set `size_th` to 11 MB).

Latency-based policy. In Figures 4.20a and 4.20b, we present the results related to both the ABS and real-world implementations of the latency-based policy. Swapping points are highlighted with `x` and •, describing migration from cloud to edge and vice versa, respectively. As expected, as soon as the latency related to a service exceeds the defined threshold, the orchestrator immediately migrates it to the edge, significantly reducing the time to receive data. This policy, as shown in Figures 4.20a and 4.20b, incurs the problem of swapping *Data Processor* services at each step. To understand the reason behind this problem, let us consider the following example. Suppose there are two *Data Processor* services DP_i and DP_j with a perceived latency of 4 and 3 seconds and a constant received data size of 13 and 14 MB, respectively. In the monitoring window m_w, the DP_i service is the one with the highest latency and it is moved to edge. Consequently, thanks to the data locality principle, DP_i, in the monitoring window m_{w+1}, is the one with the lowest latency, while DP_j still perceives a latency of 3 seconds. Thus, in the monitoring window m_{w+2}, DP_j and DP_i are swapped, returning back where we started and so on in the subsequent windows. This behaviour can be seen in Figures 4.20a and 4.20b, *e.g.,* at $t = 40s$ the orchestrator moves DP_3 to the cloud despite the



(a) Simulated execution



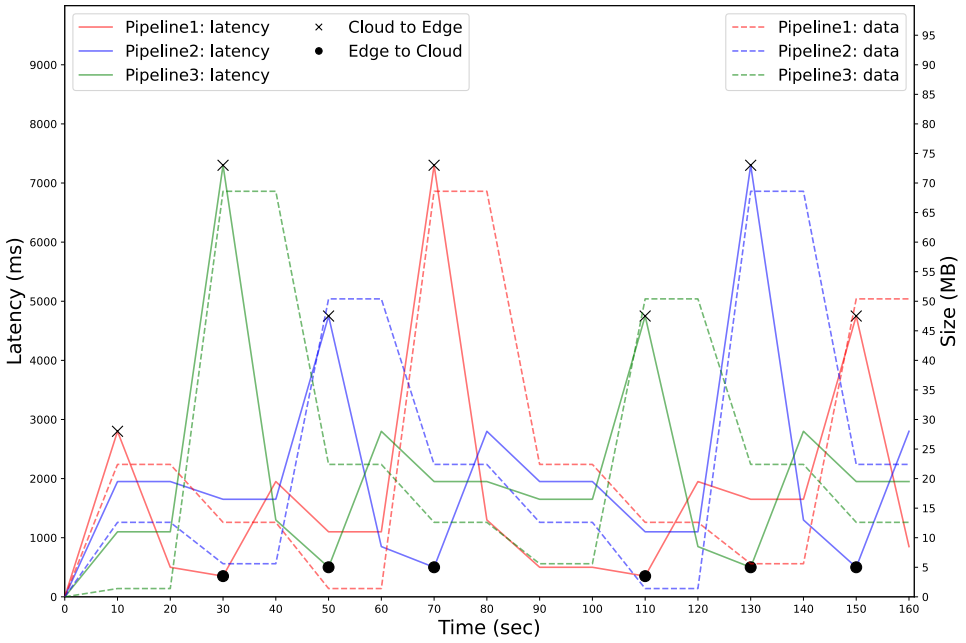
(b) Real-world execution

Figure 4.20: Latency-based policy performance

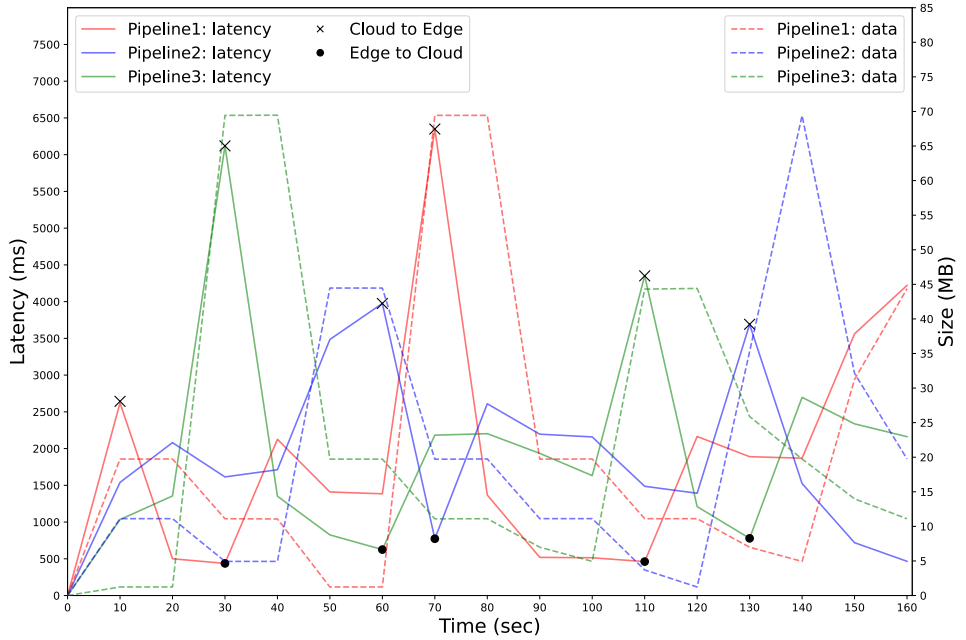
fact that it has the highest request size (and consequently the highest latency if kept there).

Size-based policy. In Figures 4.21a and 4.21b, we present the experimental results related to the size-based policy. As expected, once the size of the data being transmitted to a particular service exceeds the predefined threshold, the orchestrator promptly reacts, migrating the service to the edge and leading to a significant reduction of the transmission latency. This demonstrates the effectiveness of the policy in managing data flow efficiently. The behaviour observed in this policy is particularly noteworthy, as it exhibits greater stability and involves fewer migrations compared to the other policy. This stability is primarily due to the fact that the decision-making process is based on the workload size, which remains consistent regardless of fluctuations in other factors. More specifically, while latency may increase or decrease depending on whether the *Data Processor* is located on a cloud node or an edge one, the perceived size of the workload remains unaffected by the geographical location of the processing unit. This consistency in workload size makes it an ideal metric to determine the optimal timing to trigger service migration, ensuring the system remains stable and avoids unnecessary migration actions. By using workload size as the main trigger, the policy effectively balances system performance and the overhead costs associated with service migration.

When comparing the two policies (both in the ABS simulation and the real-world deployment), as illustrated in Table 4.5, it becomes evident that the size-based policy significantly reduces the number of swaps with respect to the alternative approach. This reduction is achieved by only initiating migrations when there is a substantial increase in workload size within a pipeline. Consequently, the system avoids reacting to minor fluctuations, which could otherwise lead to unnecessary migrations, causing, *e.g.,* network overhead. In both the ABS and real-world implementations, the policies respond to spikes in latency, aiming to optimise overall system performance. The comparable latency flows observed in both the implemented policies suggest that leveraging the data locality principle is a good strategy for service migration approaches.



(a) Simulated execution



(b) Real-world execution

Figure 4.21: Size-based policy performance

	Latency-based (ABS)	**Latency-based (real-world)**	**Size-based (ABS)**	**Size-based (real-world)**
Pipeline 1: swaps	11	11	5	4
Pipeline 1: avg latency	1.9s	1.8s	1.7s	1.8
Pipeline 2: swaps	10	10	4	3
Pipeline 2: avg latency	2s	1.8s	2s	1.8s
Pipeline 3: swaps	8	8	4	4
Pipeline 3: avg latency	2s	2s	2s	1.9s

Table 4.5: Comparative analysis of swaps

4.3.6 Refining System Simulation: Delayed Triggers

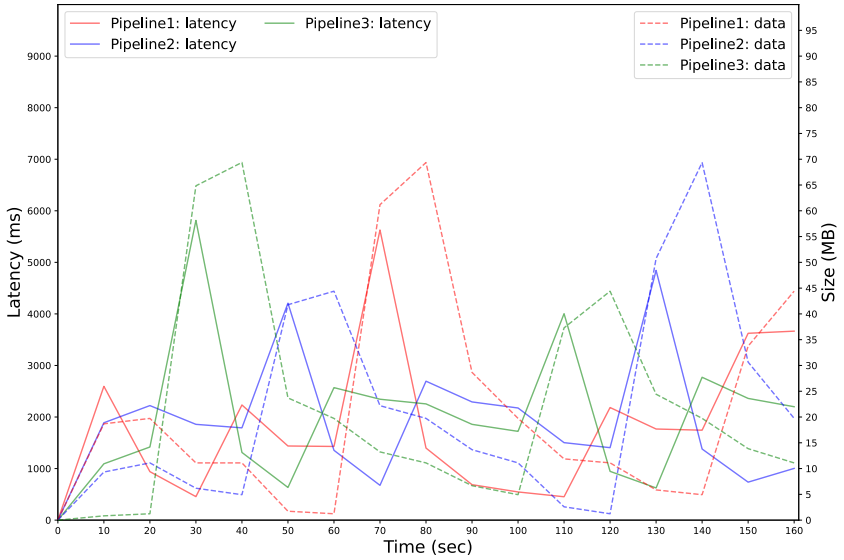


Figure 4.22: Average performance of 25 independent runs of the *real-world* system, under the size-based policy

Despite having similar behavioural patterns (*i.e.,* the simulation performs migration actions when the real-world implementation does), the reader may notice a (not that) subtle difference: the real-world system execution is affected by unexpected delays, causing some triggers to be received in the wrong order. An example is the following: according to the implementation of the low latency edge-cloud

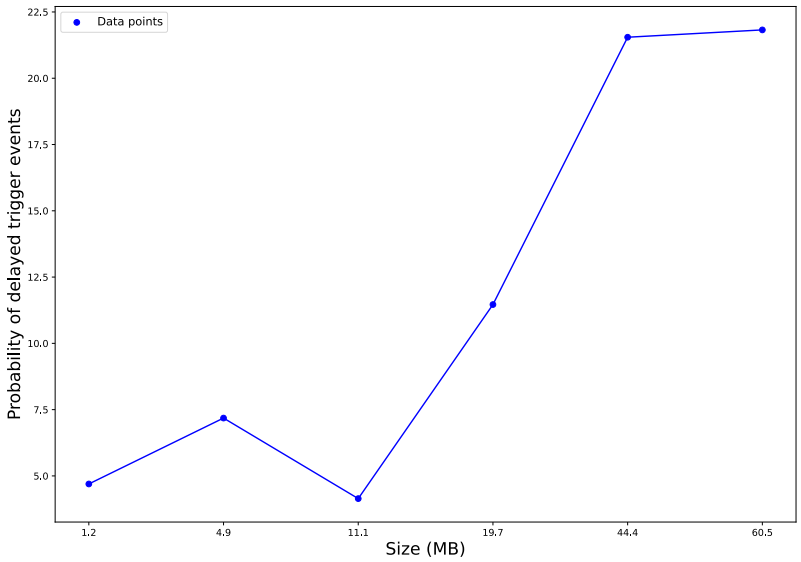


Figure 4.23: Probability of delayed trigger events

continuum architecture [BPS+22a], the trigger to update the data size is fired by the *Workload Orchestrator* every two monitoring windows. The reason behind this choice concerns experimental reproducibility: the workload we use is controllably updated by a single entity, thus making it reproducible across independent runs, without the need of synchronisation mechanisms. Being that the data size changes every two monitoring windows, we expect that the one perceived by the *Data Processor* changes accordingly. However, as can be seen in Figure 4.22, which presents the average performance of 25 independent runs of the real-world system under the size-based policy, some plotted data sizes present an unexpected behaviour. At time 30s-40s, the data sizes are expected to be the same, however, as can be seen in Figure 4.22, the ones perceived in the pipeline 3 clearly diverges from one to another. In the simulated environments no unexpected behaviours occur: code statements not tagged with `Cost` and `DataSize` annotations (see Section 2.3) are executed instantly, *i.e.,* logical time does not advance.

To perform a realistic evaluation of our migration policies for architectural reconfiguration within the simulated environment, we need to probabilistically

model such delayed triggers. To this purpose, we first define a methodology to recognise delayed triggers for data size updates. Given $\mathsf{size_i}$ and $\mathsf{size_j}$ data sizes, such that we expect $\mathsf{size_i} = \mathsf{size_j}$, a delayed trigger occurs whenever,

$$\frac{\min(\mathsf{size_i}, \mathsf{size_j})}{\max(\mathsf{size_i}, \mathsf{size_j})} < 0.75 \wedge |\mathsf{size_i} - \mathsf{size_j}| > 1.2,$$

i.e., the similarity between $\mathsf{size_i}$ and $\mathsf{size_j}$ is lower than 0.75 and the difference between the size values is greater than 1.2 MB. The similarity provides a normalised measure of the smaller value with respect to the larger one, indicating how close the smaller is to the larger in a scale-invariant way. The constraint on the difference between the size values is needed to rule out delayed triggers that do not influence the migration behaviour, *i.e.,* delayed triggers that are not relevant. As can be seen in Figure 4.19, the data size 1.2MB corresponds to a delay of 1.2s in cloud nodes and 0.2s in the edge ones. If the *Data Processor* service is placed in an edge node, a data size less than or equal to 1.2MB would anyway cause the migration to cloud (delayed triggers causing data size values to be lower than 1.2MB do not influence the behaviour). If the *Data Processor* service is placed in a cloud node, a data size less than or equal to 1.2MB would not produce any migration action.

To probabilistically evaluate the occurrence of delayed triggers, we produce a dataset containing all the instances for each data size (see Table 4.4) and the corresponding amount of delayed triggers. In particular, we consider data from 85 independent runs of the real-world system, taken from [BPS+22a]. We compute the probability of delayed trigger of *i-th* data size as

$$\frac{\mathsf{tot\_delay_i}}{\mathsf{tot_i}},$$

producing the distribution presented in Figure 4.23[9].

After integrating the probabilistic behaviour, *i.e.,* the possible occurrence of delayed triggers, in our simulated environment, we perform 25 independent runs to evaluate the average behaviour of the simulated system against the one presented in Figure 4.22. As can be seen in Figure 4.24, the produced average be-

[9]To generate such distribution, we use `poly1d`, see `https://numpy.org/doc/stable/reference/generated/numpy.poly1d.html`.

haviour closely follows the real-world one presented in Figure 4.22, showing both the expressive power of our modelling/execution language and the precision of our analyses.

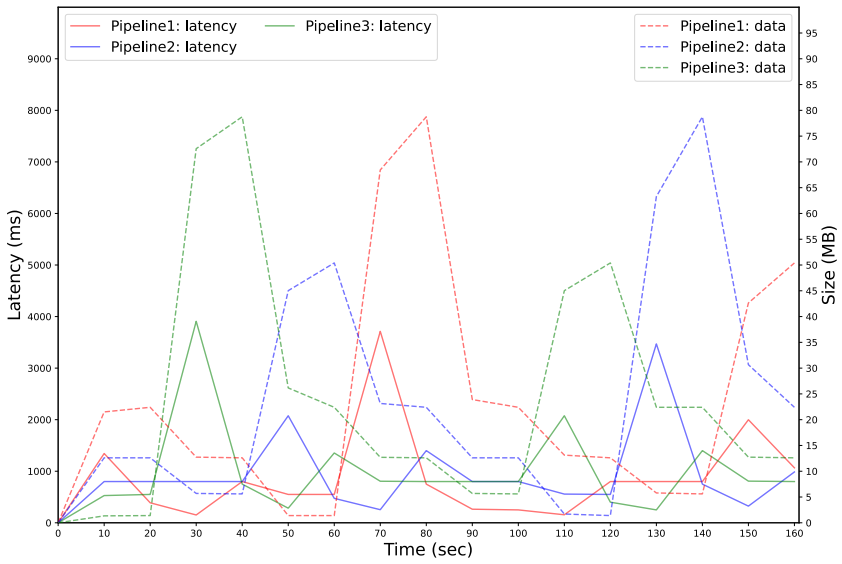


Figure 4.24: Average performance of 25 independent runs of the *simulated* system, under the size-based policy

4.4 Related Work

The main contribution of this Chapter regards the introduction of two different architectural reconfiguration strategies for service service *autoscaling* [BBLZ21, BBG+22a, BBG+25] and *migration* [BPS+22b, BPS+22a]. In this section, we review the current state of the art, starting with the approaches for autoscaling.

Service autoscaling. Local scaling focuses on adjusting the number of instances at the level of a single microservice. These approaches can be reactive (triggered by specific events) or proactive (aimed at preventing undesired events). Recent examples of reactive local scaling include Bayesian optimisation techniques [YCZ22]

and fuzzy logic [LBT18]. Proactive local scaling often involves prediction techniques to create early scaling mechanisms, using probabilistic modelling frameworks or time series analysis techniques, such as k-means [DGVV12] and neural networks [MA20, PCLH21a]. Researchers have also proposed hybrid approaches that mix reactive and proactive elements to improve system behaviour and manage unexpected traffic fluctuations [QCB18, MA20, BLV+19]. Industry solutions from major cloud vendors like Amazon and Google typically follow reactive scaling approaches based on user-defined thresholds, with recent additions of predictive capabilities exploiting historical data for automatic adaptation [Bar18, Mic, Goo].

Going towards global scaling, SmartHPA [ATWS24] is a Horizontal Pod Autoscaler for Kubernetes that adapts according to the resources available to the infrastructure. SmartHPA uses decentralised autoscaling under resource-rich infrastructures and a hierarchical approach under resource limitations so that the auto-scaler allocates and deallocates microservice replicas based on their relative load. While the hierarchical approach considers some aspects of the global state of the system (*e.g.,* microservice replicas vs load), it does not perform a coordinated scaling of the architecture, as found in global scaling. Global scaling involves coordinating the scaling of multiple interacting microservices. Previous work in this area includes decidability results for optimal deployment of microservices [BGM+19, BGM+20]. Other approaches, such as those proposed by [USC+08, GCW19], rely on performance models, but suffer from limitations, *e.g.,* delayed system capacity assessments and restrictions to specific architectures. Recent studies highlight the potential of machine learning techniques combined with performance-aware approaches in improving microservice autoscaling efficiency. For example, GRAF uses a graph neural network to proactively allocate resources, while minimising CPU usage and meeting latency requirements, outperforming traditional autoscalers in resource savings and latency convergence [PCLH21b, PCLH24]. Similarly, MS-RA, a self-adaptive, requirements-driven solution, shows superior performance compared to Kubernetes Horizontal Pod Autoscaler, achieving good performance with fewer resources [NNSN24]. The Polaris framework introduces a performance-aware autoscaler that uses high-level latency requirements, showing advantages over low-level CPU-based approaches [BBP+22]. Other contributions include Showar *et al.* [BK21], who proposed a scheduling framework to optimise resource allocation

for microservices and Burstaware predictive autoscaling, which leverages burst patterns in workloads to ensure high performance during demand spikes [AIB+22]. PBScaler addresses bottlenecks by adjusting resource allocations in real-time, preventing performance degradation due to resource constraints [XWL+24].

In conclusion, the combination of proactive and reactive global scaling approaches can significantly enhance the scalability and efficiency of microservice architectures.

Edge-cloud continuum service migration. IoT applications are composed of multiple, interacting components enabling the storage, processing and valorisation of sensory data as well as the actuation on the target environment. For this reason, the microservice patterns have emerged as a viable approach to decompose an IoT application into a set of loosely coupled services [SWVDT21]; the latter can be containerised via virtualisation tools such as Docker (see Section 2.5) and deployed at different nodes of the compute continuum thanks to orchestration engines such as Kubernetes (see Section 2.5). The question of how the continuum can help meeting the QoS requests is a novel yet investigated topic for generic microservice-based applications; at the same time, few studies refer to the IoT or to Industry 4.0 scenarios. Efficient workload allocation is the major concern in most of these papers. In [KHA19] the authors assume that an IoT application can be modelled as a set of microservices (called processing elements) forming a directed acyclic graph. Hence, they formulate the processing elements scheduling problem where the goal is to minimise the latency of the whole workflow. Similarly, the framework in [AGS+20] attempts to allocate Web Things (WTs) to nodes of the continuum, taking into account the interdependencies among the WTs, so that the in-network overhead is minimised. The placement of components must find the optimal trade-off between conflicting requirements in terms of QoS requests from users/applications and unpaired resource and cost availability on edge and cloud nodes. For these reasons, multi-objective workload allocation strategies have been proposed [KMKP21], where the placement framework attempts to jointly minimise the service completion time, communication energy consumption as well as storage cost. Another recent study [TIRB22] raises the concern about the dynamic variation of metrics used for allocation policies, *e.g.,* execution time or CPU utilisation,

which can vary significantly over time on different nodes of the continuum and employs machine learning-based solutions to learn workload patterns. In the computing continuum environment proposed by Proietti *et al.* [MB21], each edge cluster contains a scheduler service in charge of receiving requests from clients and taking decisions whether to execute the task locally, on the cloud or to reject it. The decision is based on a reinforcement learning approach, which receives a positive reward for every task completed within a deadline. All the previous studies assume a global centralised IoT workload balance. Vice versa, Zeinab *et al.* [NZDP21] proposes a decentralised load-balancing system for IoT service placement, which aims at reducing the cost of service execution, enabling each edge/cloud node to generate a predefined number of possible service placement plans. A further issue to consider is how to implement the workload allocation. In allocation-only schemes, containers are switched on/off at different nodes, but no software mobility occurs. Vice versa, migration-based strategies enable container transfer among nodes of the continuum, in a stateless or stateful way as discussed in [PVM20].

4.5 Discussion

In this Chapter, we introduce two distinct approaches to orchestration-based architectural reconfiguration: service migration and autoscaling. For each architectural reconfiguration approach, we begin by implementing the systems under consideration using our integrated timed modelling/execution language. This allows us to evaluate the performance of our approaches early in the design phase. Then, after having evaluated the effectiveness of our approaches, we develop the real-world implementations to validate them in realistic scenarios. Both orchestration-based architectural reconfiguration approaches leverage *correct-by construction* deployment orchestrations automatically synthesised starting from declarative specifications of, *e.g.,* component characteristics, deployment constraints. These orchestrations, being correct-by construction are guaranteed to successfully reach their intended purpose, *i.e.,* replicate/migrate services.

In addition to presenting and evaluating innovative orchestration-based architectural reconfiguration techniques, this Chapter also demonstrates the expressiveness and precision of our integrated timed modelling/execution language through

extensive modelling of real-world systems.

Proactive-reactive global scaling. We proposed an innovative global scaling approach that leverages the functional interdependencies among microservices, supported by a proactive-reactive scaling algorithm designed to optimise performance. We testd our approach through a series of benchmarks, conducted in both simulated and realistic environments. The simulations are carried out using our integrated timed modelling/execution language, while the real-world benchmarks requires the implementation of the novel global scaling platform described in Section 4.2.1. Initially, we run benchmarks to evaluate the reactive version of our global scaling approach against the conventional local scaling method. As proved, our global approach, not only surpasses the performance of the reactive local scaling, but it also outperformes an enhanced version of local scaling equipped with an oracle, *i.e.,* a perfect predictor. This demonstrates the inherent superiority of our scaling approach in managing scaling decisions, even in scenarios where local scaling has access to ideal forecasting capabilities. In the final set of benchmarks, we assess the effectiveness of our proactive-reactive scaling algorithm in handling workloads that are particularly challenging to predict. For this purpose, we selectively extracted outliers from the Enron corpus dataset, which our workload predictor struggled to forecast accurately. The results testify the effectiveness of our approach, showing that it can maintain optimal performance even under unpredictable conditions.

Looking ahead, several clear directions for future work present themselves, including the enhancement of prediction techniques and the refinement of algorithmic mixing strategies. For instance, natural language processing techniques could be employed to extract additional features, such as the number of attachments per email, to improve the representation of the regression target (in our case, the inbound requests). This could lead to more accurate predictions and better scaling decisions. To further bolster the resilience of our global scaling technique against potential failures, we plan to introduce dedicated monitoring mechanisms for each microservice type. These monitors would periodically verify that the number of instances matches the expectations, set by the orchestrator implementing our proactive-reactive global scaling algorithm. In the event that

the actual number of instances falls short of the expected count, indicating that one or more failures have occurred, the orchestrator would automatically restore the correct number of instances. This additional layer of monitoring ensures that our system remains robust and reliable, even in the face of unforeseen disruptions.

Edge-cloud continuum service migration. We presented a microservice architecture designed specifically for deployments across the edge-cloud continuum, aimed at addressing the limitations of an Industry 4.0 application, provided by one of the industrial partners of the SEAWALL project. Our innovative architecture, developed using open standards and technologies, overcomes the most critical shortcomings of the previous implementation. Our orchestrator enables flexible customisation of policies for the dynamic mobility of workloads between the edge and cloud layers, allowing for: (*i*) a significant reduction in communication latency through optimised service placement; (*ii*) seamless resource sharing across multiple independent data analysis pipelines; and (*iii*) the elimination of common challenges found in IoT edge-cloud solutions from public cloud vendors, *e.g.,* limited system customisation and vendor lock-in. Additionally, it supports heterogeneous data sources through the adoption of the W3C Web of Things (WoT) standard. Unlike the previous system, which relied on a single classifier deployed at the edge, our architecture can support multiple classifiers (and thus several independent control loops) that can be deployed at either the edge or cloud layers.

We are currently preparing to transfer our architecture to the production line, following its successful testing with synthetic data. Looking ahead, our future work focus on two main areas of extension. First, we plan to enhance our testing by incorporating various sources of synthetic data, potentially distributed across different nodes or regions. In this scenario, we will need to develop a topology-aware workload orchestrator capable of optimally relocating services closer to their respective data sources. Second, we aim to enrich the workload mobility policies supported by our system by exploring the use of machine learning techniques to predict increases or decreases in data generation, enabling us to proactively manage the movement of services towards data sources.

Chapter 5

Typestate Trees for Statically Typed Languages

This Chapter contains contributions from the following work of ours: [BBG+22b, BBG+24c]

Within object-oriented programming languages, *e.g.,* Java, errors, ranging from dereferencing null pointers to misusing resources (*e.g.,* attempting to read from a closed file), can lead to a plethora of issues within software systems. These issues manifest as unexpected behaviours, program malfunctions or outright crashes, undermining the reliability and robustness of the software. Hence, there exists a pressing need to develop sophisticated tools aimed at aiding the software development lifecycle, identifying and rectifying these errors as early as possible. This proactive approach is essential, given the frequency of such bugs, a fact underscored by Wetsman *et al.* [Wet20]. In programming languages, the detection of certain common errors is facilitated by the presence of type systems integrated into type checkers [Car96]. However, despite the advancements in this domain, the scope of errors identified by current mainstream object-oriented languages remains somewhat limited. Notably, within these languages, there is a noticeable absence of static assurances on programs behaving as expected. For instance, a very simple example is the following. Consider the scenario where methods must be invoked in a specific sequence, such as calling hasNext before next in an iterator. A real-world example of a method being called out of order is a bug found[1] in *Jedis*[2], *i.e.,* a Java

[1] https://github.com/redis/jedis/issues/1747
[2] https://github.com/xetorthio/jedis

client for *Redis*, an in-memory database that persists on disk. The error happened when there was an attempt to close a socket that timed out, in other words, there was an operation being available on a state that should not allow that. While such protocols are often delineated in natural language within documentation, they lack static enforcement mechanisms. Consequently, this void in static verification serves as a breeding ground for a myriad of errors, such as attempting to access a variable before it has been properly initialised [BKA11]. While some language frameworks support a refined analysis, they require expert users to provide complex specifications, for example, in separation logic [JSP+11, Rey02, IO01].

In this Chapter, we first introduce an existing tool to help filling this gap, *i.e.,* a Java Typestate Checker [MGR21] (JaTyC), which type checks programs where objects are associated with protocols, *i.e.,* typestates. Java classes are annotated with typestates, which define the behaviour of class instances, in terms of available methods and state transitions. JaTyC makes it possible to statically check:

- **absence of "the billion dollar mistake"** [Hoa09], *i.e.,* null pointer exceptions;
- **protocol compliance**, *i.e.,* objects are used according to their protocols;
- **Protocol completion**, *i.e.,* protocols reaching the end state.

Ensuring these properties is crucial to avoid protocol bugs as in [Wet20], where a mobile application tracing COVID-19 failed to perform a crucial step in the protocol: notify users if they were in close contact with potentially infectious patients, leaving the protocol uncompleted.

In the dynamic landscape of object-oriented programming languages, effective static analysis tools must intricately navigate the complexities of inheritance to truly fulfill their purpose. Inheritance, a fundamental concept in object-oriented programming, enables classes to inherit properties and behaviours from parent classes, fostering code reusability and hierarchical organisation. Therefore, a thorough understanding and incorporation of inheritance and polymorphism are indispensable for static analysis tools, aiming to effectively scrutinise object-oriented programming codebases, ensuring, not only correctness, but also maintainability

and scalability in software development endeavors. This is challenging with typestates: since a class can inherit from another and be used as a type of the superclass (*i.e.,* upcast), it is crucial to ensure that the behaviour specified in the usage protocol of the superclass is also possible in its subclasses. Thus, we need a notion of **subtyping** for protocols akin to the one for session types [HVK98, HLV+16].

In the context of this Dissertation, we extend JaTyC with support for inheritance [BBG+22b], adapting the synchronous subtyping algorithm for session types, inspired by the work of Gay and Hole [GH05], Lange and Yoshida [LY16] and Bacchiani *et al.* [BBLZ21]. Such algorithm is automatically invoked, *starting from the initial pair of states* of the class protocols, whenever a class with a protocol attached presents the keyword `extends`.

Despite the introduction of the subtyping algorithm among protocols, the support for polymorphism is still limited: casting operations are only allowed at the beginning of the protocol (*i.e.,* immediately after objects creation) or at the end of the protocol [BBG+22b]. Thus, such support paves the way to new research challenges: since the typestates of the superclass and those of the subclass can possibily be in a many-to-many subtyping relation, *i.e.,* one typestate of the superclass can have multiple subtypes in the subclass and vice versa, *how can we compute the result of a typestate up/downcast in the middle of a protocol?* Moreover, as we will see later in this Chapter, applying the subtyping algorithm only to the initial pair of typestates of the protocols is not enough: some safe programs could potentially be rejected, due to some uncaught pairs of typestates actually being in the subtyping relation. It is crucial to overcome these limitations to make the typestate approach applicable to real-world scenarios since, as shown by Mastrangelo *et al.* [MHN19], *polymorphism and cast operations are widely used.*

In the context of this Dissertation [BBG+24c], we provide a solution to these problems, applying the subtyping algorithm starting from any pair of typestates and introducing a theory based on a richer data structure, named *typestate tree*, which supports upcast and downcast operations at any point of the protocol, leveraging union and intersection types. The theory is language agnostic and applicable to object-oriented languages statically analysable through typestates, thus opening new scenarios for acceptance of programs using inheritance and polymorphism. All proofs of proofs of theorems, lemmas and corollaries presented in this Chapter

have been *fully mechanised in Coq*[3].

In the next Sections, we adopt the following semantics for subscripts: those in normal font represent variable elements (*e.g.,* parameters), while those in bold font are used to assign different meanings to the same symbol.

In addition, *in the context of this Dissertation*, we enlarge the supported JaTyC language with a new syntactical construct, by providing a preliminary support for linear arrays, *i.e.,* arrays of objects with protocol attached. Notice that, such new syntactical construct, in order to be properly handled, requires a careful reasoning on the impact such an extension has on the existing ecosystem. Thus, to perform such analysis, we first formalise the JaTyC type system and define the subtyping relations among the types JaTyC defines within its type system. Then, we introduce the new type to handle linear arrays in the type system and, to allow users to define linear arrays in programs, we extend the type checking process.

Wrapping up, in this Chapter, we formalise, in Section 5.1, the typestate language used in this Dissertation. We present JaTyC in Section 5.2, describing its main functionalities. Since JaTyC does not support inheritance (and consequently polymorphism), we introduce first, in Section 5.3, the subtyping relation for typestates, then, in Section 5.4, a JaTyC version enhanced with an implementation of the synchronous subtyping algorithm by Gay and Hole [GH05] for typestates. In Section 5.5 we present our novel theoretical work to fully support inheritance and polymorphism. In Section 5.6, we embed typestate trees within JaTyC, thoroughly describing the type checking process for each supported syntactical construct. In Section 5.7, we present our work to include arrays of linear objects in JaTyC and, in Section 5.8, we test JaTyC against of a suite of realistic examples. Finally, in Section 5.10 we conclude the Chapter.

5.1 Typestates

The syntax provided in Definition 2, which closely resembles the one introduced by Bravetti *et al.* [BFG$^+$20], defines the typestate language utilised within the scope of this Dissertation. The meta-variable m ranges over the set of method

[3]The proofs Coq code is publicly available at `https://zenodo.org/record/7712822/files/behavioural-casting-coq.tar?download=1`

identifiers **MNames**. The meta-variable l ranges over the set of output values **LNames**. The meta-variable s ranges over the set of typestate names **SNames**.

Definition 1 (Typestate syntax)**.** *Typestate terms, ranged over by meta-variable u, and states terms, ranged over by meta-variable w, are generated by the following grammar.*

$$\begin{array}{rcl} u & ::= & {}_d\{\widetilde{m : w}\} \mid s \\ w & ::= & u \mid \langle\widetilde{l : u}\rangle \\ d & ::= & \varepsilon \mid \mathsf{drop} \end{array}$$

Typestate terms u can be either *input state* terms ${}_d\{\widetilde{m : w}\}$ or typestate names s. In turn, state terms w can be either typestate terms u or *output state* terms $\langle\widetilde{l : u}\rangle$. Input state terms, denoted by ${}_d\{\widetilde{m : w}\}$, represent sets such as $m_1 : w_1$, $m_2 : w_2, \ldots, m_n : w_n$, where $n \geq 0$ is a natural number and d is an optional *droppable* flag. These states offer callable methods, viewed as input actions or external choices, followed by arbitrary states. The interpretation is that selecting a method m_i, the input state term transits to the state term w_i. Output state terms, denoted by $\langle\widetilde{l : u}\rangle$, represent sets such as $\langle l_1 : u_1, l_2 : u_2, \ldots, l_n : u_n\rangle$, where n is a positive natural number. These states present all possible outcomes of a method call, with values l_1 to l_n, viewed as output actions or internal choices l_i, followed by typestate term u_i. In our setting, only boolean and enum values are considered as outputs.

Concerning the optional droppable flag d: ${}_{drop}\{\widetilde{m : w}\}$ represents an input state term where the protocol can be dropped. Droppable states are useful in scenarios where it is not strictly necessary that an object completes its protocol, *e.g.,* iterating over just a few elements of an iterator. Moreover, we make the assumption that in an output state term $\langle\widetilde{l : u}\rangle$, there exists at least one output. In an input state term ${}_d\{\widetilde{m : w}\}$, instead, the absence of inputs is represented by $\mathsf{drop}\{\}$, signifying the termination state of the protocol, also denoted by end.

To deal with recursive behaviour, typestates use equational definitions over typestate terms.

Definition 2 (Defining equation syntax)**.** *Defining equations, ranged over by meta-variable E, are terms generated by the following grammar.*

$$E \quad ::= \quad s = {}_d\{\widetilde{m : w}\}$$

Typestates are denoted by $u^{\widetilde{E}}$, with $\widetilde{E}$ being a set of defining equations. Similarly, states are denoted by $w^{\widetilde{E}}$, with $\widetilde{E}$ being a set of defining equations.

We expect that each typestate name s used in both w and the body of the $\widetilde{E}$ equations has a unique defining equation in $\widetilde{E}$. Notice that, in the body of an equation E, we consider input state terms only, *i.e.,* disregarding equations like $s = s'$, so that typestate names s do not occur unguarded. In our formal setting, therefore, a protocol is expressed by defining a typestate.

In the following, we denote the set of typestates $u^{\widetilde{E}}$ as $\mathcal{U}$ and the set of states $w^{\widetilde{E}}$ as $\mathcal{W}$. Hereafter, whenever the finite set of equations $\widetilde{E}$ is clear from the context, we consider states w implicitly associated with $\widetilde{E}$. We omit writing ε.

The grammar introduced in Definition 2 provides a formal specification for *protocols* associated with classes. For clarity sake, we first informally introduce a protocol example in the form parsed by JaTyC[4]. For instance, let us examine the protocol defined in Listing 5.1. In this case, we define a protocol for an iterator,

Listing 5.1: Example of protocol associated to a class

```
typestate BaseIt {
  HasNext = {
    boolean hasNext(): <true: Next, false: end>,
    drop: end
  }
  Next = {
    Object next(): HasNext
  }
}
```

attached to a class named *BaseIt*, comprising two states: HasNext, the initial state, and Next. In the following, we assume, for simplicity sake, protocols to have the same name as the class they belong (as a matter of fact, in JaTyC, protocols can have different names with respect to their classes). The usage of an iterator, in compliance with this protocol, necessitates invoking the hasNext method prior to calling next, ensuring that there are remaining items to retrieve. Failure to adhere to this protocol results in an IndexOutOfBoundsException being thrown. This iterator may be "dropped" at the HasNext state, as specified by the drop: end

[4]The protocol complete grammar is available at `https://gist.github.com/jdmota/85683e518c56676612e4ba63eaa9b3f2`

transition, defined in the HasNext state. This indicates that one may stop using the iterator if it is in HasNext.

In our formal setting, a protocol is formally represented as $u^{\widetilde{E}}$, while in JaTyC we make the following assumptions: (i) a protocol is defined as $s^{\widetilde{E}}$, so that we always have the initial state name; and (ii) we adopt a restricted typestate syntax, where $w ::= s \mid \widetilde{\langle l : s \rangle}$. Formally, the protocol presented in Listing 5.1 is defined as $\widetilde{\mathsf{HasNext}^{E_{\mathsf{BaseIt}}}}$ with E_{BaseIt} including:

$$\begin{aligned} \mathsf{HasNext} &= {}_{\mathsf{drop}}\{\mathsf{hasNext} : \langle\ \mathsf{true} : \mathsf{Next},\ \mathsf{false} : \mathsf{end}\ \rangle\} \\ \mathsf{Next} &= \{\mathsf{next} : \mathsf{HasNext}\} \end{aligned}$$

5.2 JaTyC: A Java Typestate Checker

JaTyC is a tool to type check Java programs, where objects are associated with protocols, *i.e.,* typestates. JaTyC, a new implementation of Mungo [KDPG16], incorporates critical features, while addressing known issues, *e.g.,* prevention of null pointer errors and analysis of the flow of execution. Notably, it rectifies an issue of Mungo where the *continue* statement was assumed to jump at the beginning of a loop body, potentially resulting in false negatives [MGR21]. JaTyC can be accessed via its GitHub repository[5]. Implemented in Kotlin [JI17], it operates as a plugin for the Checker Framework (refer to Section 2.2).

To underscore the need for JaTyC, let us consider the example of the LineReader Java class, as presented in [MGR21]. This class is responsible for both opening a file and reading it line by line, as illustrated in Listing 5.2. In this context, Status is an enum featuring two possible values: OK, indicating the file has been correctly opened, and ERROR, indicating something went wrong during the file-handling process. The intended protocol is implicitly defined by the sequences of method calls supported and the states reached through those calls. To utilise the *LineReader*, users must invoke the open method, passing the file path. If the call returns ERROR, it indicates that the file could not be opened. Conversely, if it returns OK, users can proceed to read the file. Before invoking the read method, the eof one must be called to confirm that the end of the file has not been reached.

[5] `https://github.com/jdmota/java-typestate-checker`

Listing 5.2: *LineReader* class

```
1  import java.io.*;
2  public class LineReader {
3      private FileReader file = null;
4      private int curr;
5
6      public Status open(String f) {
7          try {
8              file = new FileReader(f);
9              curr = file.read();
10             return Status.OK;
11         } catch(IOException e) {
12             return Status.ERROR;
13         }
14     }
15
16     public String read() throws IOException {
17         StringBuilder str = new StringBuilder();
18         while(curr != 10 && curr != =1){
19             str.append((char) curr);
20             curr = file.read();
21         }
22         if(curr == 10) curr = file.read();
23         return str.toString();
24     }
25
26     public boolean eof() {
27         return curr == -1;
28     }
29
30     public void close() throws IOException {
31         file.close();
32     }
33 }
```

Each read call yields a new line. Upon completing file reading, the close method should be invoked to release resources and close the underlying stream.

Failure to adhere to this contract may result in errors or incorrect results. Attempting to read before calling open will trigger a NullPointerException, since the file field holds a null reference. Similarly, calling the read method after close will raise an IOException since the stream is already closed. Furthermore, continuing to read the file after eof returns true will cause read to return empty strings, falsely suggesting that the file being read contains empty lines. The Java compiler tolerates most of the erroneous behaviours described above, thus, we now illustrate how to enhance Java programs with typestate annotations to identify and reject programs containing such behavioural errors at compile-time.

To enforce a prescribed behaviour to a given Java class, the user must in-

Listing 5.3: *LineReader* protocol

```
1  typestate LineReaderProtocol {
2      Init = {
3          Status open(String): <OK: Open, ERROR: end>
4      }
5      Open = {
6          boolean eof(): <true: Close, false: Read>,
7          void close(): end
8      }
9      Read = {
10         String read(): Open,
11         void close(): end
12     }
13     Close = {
14         void close(): end
15     }
16 }
```

clude the Typestates(...) annotation, containing the (relative) path to the protocol file. For example, given the protocol presented in Listing 5.3, to attach it to the *LineReader* class, the user must incorporate in the class code @Typestates("LineReaderProtocol.protocol"), assuming that the protocol file is in the same folder as the *LineReader* class and its file name is *LineReaderProtocol*. The protocol in Listing 5.3 defines four distinct states: Init, Open, Read, and Close, with an implicit inclusion of the end state, representing the final state of the process. In the initial Init state, only the open method is accessible (line 3). Transition to the Open state occurs upon a successful return of OK from the open method; otherwise, the state transitions to end, where further operations are prohibited. Following file opening, the close method is callable at any point, except if the file has already been closed (lines 7, 11 and 14). Within the Open state, the eof method can be invoked (line 6). If the method returns true, the state transitions to Close; otherwise, it transitions to Read. While in the Read state, the read method is accessible and makes the state transitioning back to Open (line 10).

Protocol compliance and completion. JaTyC guarantees that Java class instances adhering to a typestate, not only obey to the prescribed protocol, but they also undergo protocol compliance and completion, ensuring that crucial method calls are not overlooked and resources are properly released. This ensures robustness and prevents resource leaks within the system. To illustrate an instance of an

incorrect usage, let us examine the *LineReader* example provided in Listing 5.4, sourced from [MGR21]. As specified in the protocol outlined in Listing 5.3, whenever the read method is invoked, the *LineReader* object should be in the Read state. However, the reader object defined in Listing 5.4 is in the Close state once entered the while loop. Thus, the only permissible method at this stage is close, rendering read inaccessible. Furthermore, there is no invocation of the close method anywhere in the code. Consequently, the protocol fails to reach the end state, leaving the resource handling incomplete with the possibility of encountering dangerous scenarios.

Listing 5.4: Wrong usage of *LineReader*

```
public class Main {
  public static void main(String[] args) {
    LineReader reader = new LineReader();
    if(reader.open() == Status.OK) {
        while(reader.eof()) {
            System.out.println(reader.read());
        }
    } else {
        System.err.println("Could not open file");
    }
  }
}
```

Nullness checking. Null pointer errors cause most of the runtime exceptions in Java programs [BKA11]: being able to detect them at compile-time is therefore crucial. Towards that direction, JaTyC offers the following guarantees: (*i*) types are non-null by default (differently from the Java type system), thus method calls and field accesses are performed on non-null types; and (*ii*) false positives (in classes with protocols) are ruled out by taking into account that we invoke methods in a specific order. To allow a type to be nullable, we use the @Nullable annotation.

To better understand how guarantee (*i*) works, *i.e.,* method calls performed on non-null types, let us consider the example in Listing 5.5, where we present two scenarios with methods potentially called on null values. In method $m1$, JaTyC reports an error since the read call could potentially be performed on a null type. Instead, in method $m2$, no errors are reported, since JaTyC enforces a defensive programming style, requiring the programmer to check the null equivalence first.

Listing 5.5: Nullness checking

```
1  public class Main {
2      void m1(@Nullable FileReader reader) {
3          System.out.println(file.read());
4      }
5
6      void m2(@Nullable FileReader reader) {
7          if(reader != null) {
8              System.out.println(file.read());
9          }
10     }
11 }
```

Concerning guarantee (*ii*), *i.e.,* ruling out false null types, let us suppose that a user calls read before open (see Listing 5.2): a null pointer exception will certainly occur, since the field file is obviously null. However, from Listing 5.3, we know that the defined behaviour ensures that open is called before read, thus, file cannot be null at the moment of the read method call and JaTyC does not raise errors. The above behaviour is obtained tagging the field file with the @Nullable annotation, making it possible to implement the method read without the need for defensive programming, *i.e.,* explicitly checking for nullness of file. While JaTyC makes it possible to avoid the verbosity of defensive programming, *i.e.,* checking that item != null, many static analysis tools, *e.g.,* the *Nullness Checker* of the Checker Framework, force the programmer to include such explicit check.

5.3 Typestate Subtyping

State subtyping plays a pivotal role in supporting behavioural casting. In our context, subtypes encompass a superset of methods compared to their supertype counterparts (input contravariance), while offering a subset of the supertype method outputs (output covariance). To define state subtyping, we closely adhere to the framework established by Gay and Hole in their work on session types subtyping [GH05]. Consequently, we define the subtyping relation as a simulation, recognising that protocols can manifest as, possibly, infinite state systems. Additionally, we present a sound and complete algorithm to verify if subtyping between two states holds, *i.e.,* the subtype simulates the supertype.

To begin, we introduce the function unf (see Definition 3) to unfold typestate name definitions. Then, we define subtyping, following standard approaches (see Definition 4).

Definition 3 (Names definition)**.** *The function* $\mathsf{unf} : \mathcal{W} \to \mathcal{W} \setminus \mathbf{SNames}$ *is such that, given a state* $w^{\widetilde{E}} \in \mathcal{W}$*, if it is a typestate name,* $\mathsf{unf}(w^{\widetilde{E}})$ *yields the body of its defining equation; otherwise,* $\mathsf{unf}(w^{\widetilde{E}})$ *yields the given state* $w^{\widetilde{E}}$*. Formally,*

$$\mathsf{unf}(w^{\widetilde{E}}) = \begin{cases} d\{\widetilde{m : w}\}^{\widetilde{E}} & \text{if } w^{\widetilde{E}} = s^{\widetilde{E' \cup \{s = d\{\widetilde{m:w}\}\}}} \\ w^{\widetilde{E}} & \text{otherwise} \end{cases}$$

Definition 4 (State simulation)**.** *A relation* $R \subseteq \mathcal{W} \times \mathcal{W}$ *is a* state simulation*, if* $({w_1}^{\widetilde{E_1}}, {w_2}^{\widetilde{E_2}}) \in R$ *implies the following conditions:*

1. *If* $\mathsf{unf}({w_1}^{\widetilde{E_1}}) = {}_{d_1}\{\widetilde{m_1 : w_1}\}^{\widetilde{E_1}}$ *then* $\mathsf{unf}({w_2}^{\widetilde{E_2}}) = {}_{d_2}\{\widetilde{m_2 : w_2}\}^{\widetilde{E_2}}$ *and for each* $m : w_2'$ *in* $\widetilde{m_2 : w_2}$*, there is* w_1' *such that* $m : w_1'$ *in* $\widetilde{m_1 : w_1}$ *and* $({w_1'}^{\widetilde{E_1}}, {w_2'}^{\widetilde{E_2}}) \in R$*.*

2. *If* $\mathsf{unf}({w_1}^{\widetilde{E_1}}) = \langle\widetilde{l_1 : u_1}\rangle^{\widetilde{E_1}}$ *then* $\mathsf{unf}({w_2}^{\widetilde{E_2}}) = \langle\widetilde{l_2 : u_2}\rangle^{\widetilde{E_2}}$ *and for each* $l : u_1$ *in* $\widetilde{l_1 : u_1}$*, there is* u_2 *such that* $l : u_2$ *in* $\widetilde{l_2 : u_2}$ *and* $({u_1}^{\widetilde{E_1}}, {u_2}^{\widetilde{E_2}}) \in R$*.*

Definition 5 (Subtyping on typestates)**.** *Let* ${w_1}^{\widetilde{E_1}}$ *and* ${w_2}^{\widetilde{E_2}}$ *be states. We say* ${w_1}^{\widetilde{E_1}}$ *is a* subtype *of* ${w_2}^{\widetilde{E_2}}$*,* i.e., ${w_1}^{\widetilde{E_1}} \leq_{\mathbf{S}} {w_2}^{\widetilde{E_2}}$*, if and only if there exists a state simulation* R *such that* $({w_1}^{\widetilde{E_1}}, {w_2}^{\widetilde{E_2}}) \in R$*.*

To better understand subtyping on typestates, let us consider an example with two protocols. First, we consider the protocol presented in Listing 5.1 (related to the *BaseIt* class), with the only difference being that here we assume that HasNext has no drop: end transition. Additionally, we consider the protocol in Listing 5.6 (related to the *RemovableIt* class), modelling the behaviour of an iterator allowing removal of elements. In our setting (recall, a protocol is represented by $s^{\widetilde{E}}$, with s being the initial typestate name), we formally represent the *BaseIt* protocol as

$$\begin{aligned} \mathsf{HasNext} &= \{\mathsf{hasNext} : \langle\ \mathsf{true} : \mathsf{Next},\ \mathsf{false} : \mathsf{end}\ \rangle\} \\ \mathsf{Next} &= \{\mathsf{next} : \mathsf{HasNext}\} \end{aligned}$$

Listing 5.6: *RemovableIt* protocol

```
typestate RemovableIt{
  HasNext = {
    boolean hasNext(): <true: Next, false: end>
  }
  Next = {
    Object next(): Remove
  }
  Remove = {
    boolean hasNext(): <true: Next, false: end>,
    void remove(): HasNext
  }
}
```

and the *RemovableIt* protocol as

$$\begin{aligned} \mathsf{HasNext} &= \{\mathsf{hasNext} : \langle\ \mathsf{true} : \mathsf{Next},\ \mathsf{false} : \mathsf{end}\ \rangle\} \\ \mathsf{Next} &= \{\mathsf{Next} : \mathsf{Remove}\} \\ \mathsf{Remove} &= \{\mathsf{hasNext} : \langle\ \mathsf{true} : \mathsf{Next},\ \mathsf{false} : \mathsf{end}\ \rangle,\ \mathsf{remove} : \mathsf{HasNext}\} \end{aligned}$$

Given the formal definition of the protocols presented above, leveraging Definition 5, an example of typestate simulation (recall Definition 4) follows, where $\mathsf{HasNext}^{\widetilde{E_{\mathsf{RemovableIt}}}} \leq_{\mathbf{S}} \mathsf{HasNext}^{\widetilde{E_{\mathsf{BaseIt}}}}$:

$$\begin{aligned} &\{(\mathsf{HasNext}^{\widetilde{E_{\mathsf{RemovableIt}}}}, \mathsf{HasNext}^{\widetilde{E_{\mathsf{BaseIt}}}}), (\mathsf{Next}^{\widetilde{E_{\mathsf{RemovableIt}}}}, \mathsf{Next}^{\widetilde{E_{\mathsf{BaseIt}}}}), \\ &(\langle \mathsf{true} : \mathsf{Next}, \mathsf{false} : \mathsf{end}\rangle^{\widetilde{E_{\mathsf{RemovableIt}}}}, \langle \mathsf{true} : \mathsf{Next}, \mathsf{false} : \mathsf{HasNext}\rangle^{\widetilde{E_{\mathsf{BaseIt}}}}), \\ &(\mathsf{Remove}^{\widetilde{E_{\mathsf{RemovableIt}}}}, \mathsf{HasNext}^{\widetilde{E_{\mathsf{BaseIt}}}})\} \end{aligned}$$

5.4 Enhancing JaTyC: Inheritance Support

Incorporating the support for inheritance into static analysis for object-oriented programming languages is crucial. This becomes particularly challenging when dealing with typestates, as the relation between classes can influence the behaviour specified in the usage protocols of the superclasses. *In the context of this Dissertation*, we enhance JaTyC capabilities to handle inheritance with the integration of an adapted version of the synchronous subtyping algorithm for session types [GH05, LY16, BBLZ21]. This algorithm serves to ensure that the protocol specified in subclasses aligns with that of their superclasses, thus maintaining

consistency and adherence to the intended behaviour across the inheritance hierarchy. In addition to implementing such algorithm, addressing method overriding and casting becomes imperative. Method overriding necessitates careful consideration to ensure that subclass methods appropriately substitute the superclass ones, without violating the intended behaviour. Similarly, casting operations must be handled accurately to maintain type safety and preserve the integrity of the inheritance hierarchy.

Synchronous subtyping algorithm. The algorithm presented in Listing 5.7 for supporting protocol subtyping draws inspiration from the synchronous subtyping algorithm used for session types [GH05, LY16, BBLZ21]. It is automatically

Listing 5.7: Synchronous subtyping algorithm for typestates

```
typealias SP = Pair<AbstractState<*>, AbstractState<*>>

fun subty(g1: Graph, g2: Graph, currP: SP, marked: Set<SP> = emptySet()) {
  if (currP in marked) return
  val derived = currP.first
  val base = currP.second
  when {
    derived is State && base is State -> {
      val t1 = derived.normalizedTransitions
      val t2 = base.normalizedTransitions
      // Input contravariance
      if(t1.keys.containsAll(t2.keys)) {
        t2.keys.forEach {
          subty(g1, g2, t1[it]!! to t2[it]!!, marked + currP)
        }
      }
    }
    derived is DecisionState && base is DecisionState -> {
      val t1 = derived.normalizedTransitions
      val t2 = base.normalizedTransitions
      // Output covariance
      if(t2.keys.containsAll(t1.keys)) {
        t1.keys.forEach {
          subty(g1, g2, t1[it]!! to t2[it]!!, marked + currP)
        }
      }
    }
  }
}
```

invoked whenever a class with a protocol attached presents the keyword extends. It constructs graphs from the protocols provided as input (starting from the initial pair of states) and traverses them by executing common input/output operations,

marking each encountered pair of states. It is important to note that, in our context, input operations are represented by method calls, while output operations are indicated by the values returned by these calls. Pairs of states are marked under the following conditions:

- both states are input states and satisfy the principle of *input contravariance* (lines 8-17 in Listing 5.7), meaning the subtype can perform a set of input operations greater than or equal to those of the supertype;
- both states are output states and satisfy the principle of *output covariance* (lines 18-27 in Listing 5.7), indicating the supertype can perform a set of output operations greater than or equal to those of the subtype;
- both states are `end` states (lines 8-17 in Listing 5.7), meaning the set of available input transitions is empty.

The algorithm terminates when either all reachable pairs have been marked (indicating subtyping holds) or a pair of states fails to satisfy any of the aforementioned conditions (indicating subtyping does not hold).

Due to Java lack of support for inheritance in enums, all of their values are considered as returnable, necessitating their inclusion in the protocol. Consequently, output covariance always holds, as in our setting, all outputs are invariant.

For example, let us consider the *RemovableIt* protocol in Listing 5.6. It extends the *BaseIt* one (Listing 5.1 without drop: end), adding the typestate Remove with the new method remove. The subtyping algorithm, ensures that the *RemovableIt* is a subtype of *BaseIt*. In particular, the Remove typestate respects input contravariance and output covariance with respect to the HasNext one in the supertype.

Method inheritance. Inheritance enables the reuse of methods from superclasses, *i.e.,* the ability to override some of them, or to add new ones. For instance, let us consider the code snippet depicted in Listing 5.9: it showcases an implementation of an iterator along with the attached protocol outlined in Listing 5.1 (without the inclusion of drop: end). This implementation provides standard methods, *e.g.,* hasNext, next, and remainingItems, which respectively allow one to check if the next element is available, iterate one step forward and count the number of

Listing 5.8: *BaseIt* implementation

```
import java.util.*;
import jatyc.lib.*;

@Typestate("BaseIt")
public class BaseIt {
  private String[] items;
  protected int index;

  public BaseIterator(String[] items) {
    this.items = items;
    this.index = 0;
  }

  public boolean hasNext() {
    return this.index < this.items.length;
  }

  public Object next() {
    return this.items[this.index++];
  }

  public int remainingItems() {
    return this.items.length - this.index;
  }
}
```

remaining elements. Notably, the *RemovableIt* implementation, presented in Listing 5.6, in contrast to the *BaseIt* one, utilises a list as its underlying data structure. As a result, all methods must be overridden to access the collection effectively. Furthermore, in accordance with its protocol, the class depicted in Listing 5.9 incorporates a new method, *i.e.,* remove.

To correctly support inheritance, we need to deal with the following cases:

- a class without protocol extending a class without protocol;
- a class with protocol extending a class with protocol;
- a class without protocol extending a class with protocol;
- a class with protocol extending a class without protocol.

Handling the first and second scenarios is straightforward: the former requires no inspection, since classes, by default, do not have protocols, while the latter is verified using the subtyping algorithm presented in Listing 5.7. In the third case, the subclass inherits the protocol: the usage of overridden methods adheres to the

Listing 5.9: *RemovableIt* implementation

```
import java.util.*;
import jatyc.lib.*;

@Typestate("RemovableIt")
public class RemovableIt extends BaseIt {
  protected List<Object> items;

  public RemovableIt(String[] items) {
    super(items);
    this.items = Util.toList(items);
  }

  public boolean hasNext() {
    return this.index < this.items.size();
  }

  public @Nullable Object next() {
    return this.items.get(this.index++);
  }

  public void remove() {
    this.items.remove(--this.index);
  }

  public int remainingItems() {
    return this.items.size() - this.index;
  }
}
```

inherited protocol, while newly added methods are regarded as *anytime* methods, *i.e.,* methods callable at any moment that do not appear in the protocol. Regarding the fourth scenario, all methods in the superclass are considered as *anytime* and it is mandatory that they remain as such in subclasses. Consequently, these methods cannot be included in the protocols of subclasses. Notice that, any method not included in the protocol is automatically marked as an *anytime* method, akin to the remainingItems method in the *BaseIt* class. To ensure safety, *anytime* methods are restricted to performing read operations or calling others *anytime*.

Casting. The support for inheritance enables polymorphism with the consequent needs for casting operations management. Consider the scenario illustrated in Listing 5.10. Here, we create a *RemovableIt* object and assign it to a variable of type *BaseIt*, thus performing an upcast. Subsequently, we pass this object to the iterate method, which iterates over all items and returns a *BaseIt* in the end state. Finally, we perform a downcast and call the *anytime* method remainingItems, which

Listing 5.10: *Polymorphic code* example

```
1  import jatyc.lib.*;
2
3  public static void main(String[] args) {
4    BaseIt it = new RemovableIt(args);
5    RemovableIt rIt = (RemovableIt) iterate(it);
6    System.out.printf("Left:%d\n", rIt.remainingItems());
7  }
8
9  public static BaseIt iterate(@Requires("HasNext") BaseIt it) {
10   while (it.hasNext()) {
11     System.out.printf("Item:%s\n", it.next());
12   }
13   return it;
14 }
```

correctly returns zero. Notably, the method iterate utilises the Requires annotation from the jatyc.lib package to indicate the states the parameter is expected to be in. Thanks to our subtyping algorithm, despite iterate expecting a *BaseIt* object in the HasNext state and receiving a *RemovableIt* object in the same state, we ensure a safe execution of the method. This is achieved by verifying, prior to executing the method call (at line 5), that $\mathsf{HasNext}^{\widetilde{E_{\mathsf{RemovableIt}}}} \leq_{\mathbf{S}} \mathsf{HasNext}^{\widetilde{E_{\mathsf{BaseIt}}}}$. Thus, if the code is safe with a *BaseIt* object, it remains safe with a *RemovableIt* object.

Limitations. We enhanced JaTyC with a limited support for polymorphism. In particular, casting operations are currently only allowed at the beginning of the protocol (before any method is called) or at its end. This limitation is caused by the fact that states belonging to subclasses and superclasses could possibly be in a *many-to-many* relation. For example, if we run the subtyping algorithm on the HasNext typestate of the *BaseIt* (Listing 5.1 without drop: end), it turns out this typestate is a supertype of both HasNext and Remove states of the *RemovableIt* (Listing 5.6). Similarly, if we downcast from *BaseIt* to *RemovableIt*, we do not know to which state we should cast to (either HasNext or Remove). In addition to the uncertainty of typestate casting results, as we will see later in this Chapter, applying the subtyping algorithm only to the initial typestates of the protocols is not enough: some safe programs could potentially be rejected from JaTyC, due to some uncaught pairs of typestates actually being in the subtyping relation. As the keen reader may notice, *droppable states* are not considered in the subtyping

analysis. As we will see later in this Chapter, these states require relaxing the subtyping algorithm and need a proper formalisation to correctly deal with them.

5.5 Behavioural Up/Down Casting

To emphasise the importance of supporting behavioural castings in the middle of protocols, let us consider an analogy drawn from the automotive industry, where driving dynamics control allows to customise the drive mode[6]; for SUVs, we consider a Comfort and a Sport modalities, where each allows specific features: Eco-Drive and FourWheelsDrive, respectively. Listing 5.11 describe the behaviours of the controllers of a *Car* and a *SUV*, respectively, where class *SUV* extends *Car*.

Listing 5.11: *Car* and *SUV* protocols

```
typestate Car {
  OFF = {
    boolean turnOn(): <true:ON,false:OFF>,
    drop: end
  }
  ON = {
    void turnOff(): OFF,
    void setSpeed(int): ON
  }
}

typestate SUV {
  OFF = {
    boolean turnOn(): <true:COMF_ON,false:OFF>,
    drop: end
  }
  COMF_ON = {
    void turnOff(): OFF,
    void setSpeed(int): COMF_ON,
    Mode switchMode(): <SPORT:SPORT_ON,COMFORT:COMF_ON>,
    void setEcoDrive(boolean): COMF_ON
  }
  SPORT_ON = {
    void turnOff(): OFF,
    void setSpeed(int): SPORT_ON,
    Mode switchMode(): <SPORT:SPORT_ON,COMFORT:COMF_ON>,
    void setFourWheels(boolean): SPORT_ON
  }
}
```

Both vehicle types share two foundational states: OFF, signifying a powered-

[6]BMW drive Sport Mode vs Comfort Mode, https://www.bmwofstratham.com/bmw-sport-mode-vs-comfort-mode-stratham-nh

off state, and ON, representing an active state capable of executing actions like setting a specific speed. In the OFF state, the car can be, via dedicated methods (*i.e.,* turnOn), turned on, enabling access to functionalities such as setSpeed. Conversely, in the ON state, the car can be turned off. However, it is essential to note that, the turnOn method might encounter technical obstacles, potentially resulting in either a successful transition to the ON state or remaining in the OFF state, depending on the returned value. SUVs, as defined in the protocol outlined in Listing 5.11, undergo a distinct operational process. Upon successful activation through the turnOn method, they enter Comfort mode (COMF_ON), enabling access to specialised functions, like setEcoDrive. This mode can be altered via the switchMode action, whose outcome hinges on the current mode, either Comfort (in that subject to potential failures, *e.g.,* if the vehicle has not enough fuel) or Sport (SPORT_ON). Similarly, the Sport mode offers the flexibility of executing switchMode alongside mode-specific functionalities, like setFourWheels, potentially failing if, *e.g.,* the speed exceeds limits. This dual-mode capability ensures that SUVs can adapt their behaviour according to driving preferences and conditions. The setSpeed method is overridden within the *SUV* class. Specifically, when eco-drive is engaged, the speed must adhere to a predefined threshold, whereas without eco-drive, the speed can be set to any value. As detailed in Section 5.6, our verification process ensures correctness in overridden methods, based on typestate variance, thus guaranteeing the safe operation of dynamic dispatch.

By applying the subtyping algorithm outlined in [GH99, LY16, BBLZ21] to the initial typestates, *i.e.,* OFF in the Car protocol and OFF in the SUV one, we ascertain that the *SUV* protocol constitutes a subtype of the *Car* one. This classification underscores the hierarchical relation between the protocols, elucidating how the *SUV* protocol inherits and extends functionality from the parent *Car* one.

Key insight. Even in cases involving relatively straightforward classes and typestates, such as *Car* and *SUV*, imposing restrictions on casts solely at the beginning (*i.e.,* immediately after object creation) or conclusion of protocols (*i.e.,* after reaching the end state), significantly constrains the scope of programs the typestate-based analysis is capable of type checking. This limitation becomes evident when considering client code that may necessitate casts at various junctures in the proto-

Listing 5.12: *Upcast/downcast* limitation

```
public static void dispatch(@Requires("ON") Car c) { ... }
public static void providePoweredSUV(@Requires("OFF") SUV c) {
    if (c.turnOn()) dispatch(c); // Upcast rejected
}
```

col. The underlying assumption that casts are only required at protocol initiation (before any method calls) or termination, is frequently challenged by practical scenarios, as exemplified in Listing 5.12. For instance, within an automotive system, there might be a requirement to dispatch cars that are already powered on, *i.e.,* adhering to the typestate ON, regardless of whether they are SUVs or standard cars. Such operations, which appear reasonable and necessary in real-world contexts, underscore the inadequacy of rigidly enforcing casting restrictions solely at the protocol outset or conclusion.

Removing this limitation is challenging. The solution hinges on a pivotal consideration: to effectively address this issue, the subtyping algorithm must, not only consider the initial pair of typestates, but also evaluate all possible combinations of typestates across both protocols. This comprehensive approach enables the identification of all typestate pairs that exhibit a subtyping relation. To illustrate, let us consider the scenario depicted in Listing 5.13. In this example, the limitSpeed

Listing 5.13: Limitation of the subtyping algorithm application

```
void limitSpeed(@Requires("ON") Car c, int speed) {
    if (speed > 50) c.setSpeed(50);
    else c.setSpeed(speed);
}
```

method requires a parameter of type *Car* in the ON typestate. A client code that passes an object of type *SUV* in the SPORT_ON typestate to limitSpeed remains type-safe since, as we will demonstrate, SPORT_ON is indeed a subtype of ON. However, if we were to execute the subtyping algorithm solely with the initial pair of typestates, *i.e.,* (OFF, OFF), the resulting simulation relation [BBLZ21, GH05] would fail to encompass the pair (SPORT_ON, ON). This limitation is illustrated in the leftmost graph of Figure 5.1 (boxes represent input states, while diamonds output ones), where blue denotes typestates of the *SUV* protocol and red repre-

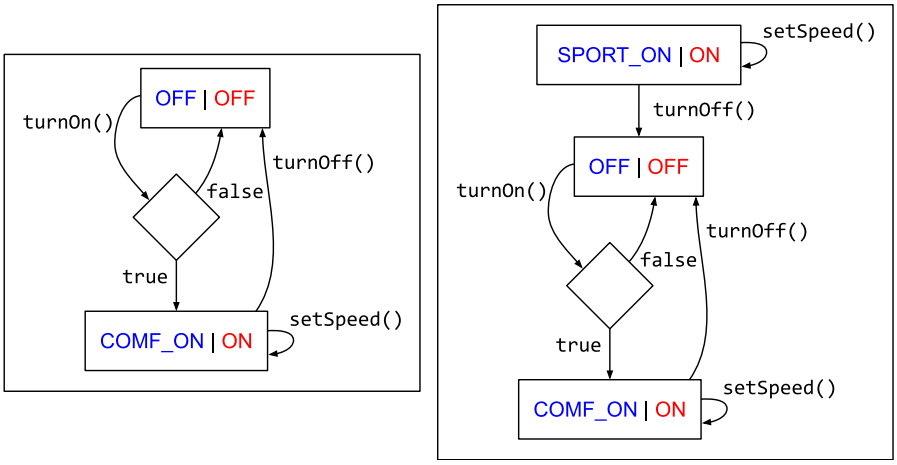


Figure 5.1: Subtyping simulations starting from different initial pair of states

sents those of the *Car* protocol. By contrast, when including the pair (SPORT_ON, ON) as input, it becomes clearly evident that this pair conforms to the typestate subtyping relation, as depicted in the rightmost graph.

A theory of typestate upcast and downcast. Building upon this crucial insight, we establish a mechanism for computing casting outcomes: when downcasting, we look for the typestates (within the protocol of the target class, *i.e.,* the class we are downcasting to) that are subtypes of the current one; when upcasting, instead, we look for the typestates (again, in the protocol of the target class) that are supertypes of the current one. Given that multiple typestates may fulfill these criteria, we require a structured notion of *types* to consolidate them effectively. When downcasting, we combine the subtypes in a *union type* [BDCd95, PC01] (modelling the fact that the actual type is unknown) so that a method call is allowed only if it is permitted by all elements of the union. Union types are also useful to allow branching code to be typed with different types, so the program continuation, *e.g.,* after an if statement, is correct no matter which branch the program takes at runtime. This is more flexible than some other approaches (*e.g.,* the session type one [Vas11]), which require both branches to have the same type. When upcasting, we combine the supertypes in an *intersection type* so that a method call is allowed if it is permitted by at least one element of the intersection.

However, the complexity of casting scenarios extends beyond the capabilities

of intersection and union types alone. To illustrate this challenge, let us consider a hypothetical scenario involving an Electric Car (*ECar*) class that also extends the base *Car* one, as depicted in Listing 5.14. After the execution of the if statement,

Listing 5.14: Typestate tree motivation

```
public class ClientCode {
  public static void example() {
    Car c;
    if (cond) c = new SUV();
    else c = new ECar();
    if (c.turnOn()) {
      if (c instanceof SUV) {
        SUV s = (SUV) c;
        s.setEcoDrive(true);
        c = s;
      }
      c.turnOff();
    }
  }
}
```

it becomes ambiguous whether the variable c is an instance of *SUV* or *ECar*. These uncertain situations necessitate a more comprehensive approach, *i.e.,* one that associates classes with distinct *types* and meticulously tracks all potential scenarios. To address this need, we introduce the concept of *typestate trees*, which closely mirror the class inheritance hierarchy. In this model, the typestate tree originates from a root node representing the static class of the variable to which a typestate tree is associated (*e.g., Car* in Listing 5.14), with child nodes branching out to encompass derived classes such as *SUV* and *ECar*. Each node within the typestate tree corresponds to a specific class and encapsulates the type of the object, acknowledging the possibility that the object is indeed an instance of that class. Consequently, when contemplating a future downcast to either *SUV* or *ECar* class, we can effortlessly focus on the corresponding subtree, with its root node corresponding to the child node representing the targeted class. The reader could wonder why we introduce typestate trees and if the complexity they introduce is actually needed. Observe that a similar program to the one presented in Listing 5.14 could be implemented using *ECar* instead of *SUV*. Thus, to seamlessly handle this variety of possible scenarios, trees represent a good choice and a simple structure that naturally captures the class hierarchy. As a matter of fact,

this hierarchical organisation allows for a precise tracking of object types, ensuring accurate handling of casting operations within our system.

The solution we devise is language agnostic, making it applicable across a spectrum of statically typed object-oriented programming languages. To validate its efficacy and expressiveness, we applied it to Java, extending our typestate-based type checker JaTyC to support casting operations at any points in protocols. This advancement contributes significantly to the field of related work (referenced in Section 5.9). Kouzapas *et al.* [KDPG16] mention in the future work Section that handling protocol inheritance between classes merely requires "a subtyping relation between their typestate specifications". While this approach suffices for extending class inheritance, it proves inadequate for addressing casting and polymorphism, commonplace features in programming scenarios. Our solution fills this crucial gap providing a comprehensive mechanism to handle castings within protocols, thereby enhancing the versatility and applicability of typestate-based systems.

Furthermore, we support, in our subtyping analysis, *droppable typestates* (see typestate OFF in Listing 5.11), typestates where one can either safely stop using the protocol or perform more actions (if there are any). A droppable typestate with no actions is similar to the end state in session types. To ensure comprehensive support for droppable typestates, we extend Gay and Hole session type subtyping definition. The formal proofs of theorems, lemmas and corollaries, presented in this Chapter, have been fully mechanised in Coq[7], ensuring the integrity and reliability of our approach.

5.5.1 Subtyping Over Droppable States

To include *droppable typestates* in our subtyping analysis, we extend Definition 4 as follows. Recall that, in our setting, end is considered as $\mathsf{drop}\{\}$, *i.e.,* a droppable typestate without input transitions.

Definition 6 (Extended state simulation)**.** *A relation $R \subseteq \mathcal{W} \times \mathcal{W}$ is a* state simulation*, if $({w_1}^{\widetilde{E_1}}, {w_2}^{\widetilde{E_2}}) \in R$ implies the following conditions:*

1. *If* $\mathsf{unf}({w_1}^{\widetilde{E_1}}) = {}_{d_1}\{\widetilde{m_1 : w_1}\}^{\widetilde{E_1}}$ *then* $\mathsf{unf}({w_2}^{\widetilde{E_2}}) = {}_{d_2}\{\widetilde{m_2 : w_2}\}^{\widetilde{E_2}}$ *and:*

[7]The Coq code containing these proofs is publicly accessible at `https://zenodo.org/record/7712822/files/behavioral-casting-coq.tar?download=1`.

   (a) *for each* $m{:}w_2'$ *in* $\widetilde{m_2 : w_2}$, *there is* w_1' *such that* $m{:}w_1'$ *in* $\widetilde{m_1 : w_1}$ *and* $({w_1'}^{\widetilde{E_1}}, {w_2'}^{\widetilde{E_2}}) \in R$.

   (b) *if* $d_2 = \mathsf{drop}$ *then* $d_1 = \mathsf{drop}$.

2. *If* $\mathsf{unf}({w_1}^{\widetilde{E_1}}) = \widetilde{\langle l_1 : u_1 \rangle}^{\widetilde{E_1}}$ *then* $\mathsf{unf}({w_2}^{\widetilde{E_2}}) = \widetilde{\langle l_2 : u_2 \rangle}^{\widetilde{E_2}}$ *and for each* $l{:}u_1$ *in* $\widetilde{l_1 : u_1}$, *there is* u_2 *such that* $l{:}u_2$ *in* $\widetilde{l_2 : u_2}$ *and* $({u_1}^{\widetilde{E_1}}, {u_2}^{\widetilde{E_2}}) \in R$.

Notice that, the common rule for session type subtyping of the end state (*i.e.,* $\mathsf{end} \leq_{\mathbf{S}} \mathsf{end}$) is derivable from the previous definitions by just picking the relation $R = \{(\mathsf{drop}\{\}, \mathsf{drop}\{\})\}$ and observing that it is a state simulation (Definition 6), thus $\mathsf{drop}\{\} \leq_{\mathbf{S}} \mathsf{drop}\{\}$ holds by Definition 5. In the same fashion, by picking the relation $R = \{(\mathsf{drop}\{\dots\}, \mathsf{drop}\{\})\}$ it holds that $\mathsf{drop}\{\dots\} \leq_{\mathbf{S}} \mathsf{drop}\{\}$.

As a sanity check, we show basic subtyping properties on states: reflexivity and transitivity.

Lemma 1 (Reflexivity)**.** *For all* $w^{\widetilde{E}}$, *then* $w^{\widetilde{E}} \leq_{\mathbf{S}} w^{\widetilde{E}}$.

Lemma 2 (Transitivity)**.** *For all* ${w_1}^{\widetilde{E_1}}$, ${w_2}^{\widetilde{E_2}}$, ${w_3}^{\widetilde{E_3}}$, *if* ${w_1}^{\widetilde{E_1}} \leq_{\mathbf{S}} {w_2}^{\widetilde{E_2}}$ *and* ${w_2}^{\widetilde{E_2}} \leq_{\mathbf{S}} {w_3}^{\widetilde{E_3}}$, *then also* ${w_1}^{\widetilde{E_1}} \leq_{\mathbf{S}} {w_3}^{\widetilde{E_3}}$.

Defining an algorithm to verify state subtyping is indispensable, as it, not only demonstrates the decidability of subtyping, but also serves as the foundation for implementing a type checking procedure (Definition 7). To ensure termination, we adopt a strategy that consistently applies the Assump rule whenever feasible. Initially, our algorithm aims to establish the judgment $\varnothing \vdash {w_1}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} {w_2}^{\widetilde{E_2}}$. This methodology mirrors the approach employed in the session type subtyping algorithm introduced by Gay and Hole [GH05]. Furthermore, we establish, in Theorems 1 and 2, that our subtyping algorithm maintains soundness and completeness concerning the coinductive definition $\leq_{\mathbf{S}}$ (Definition 5). This verification ensures the reliability and accuracy of our algorithm in determining subtyping relations between typestates.

Definition 7 (Algorithmic state subtyping)**.** *The following inference rules define the judgement* $\Sigma \vdash {w_1}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} {w_2}^{\widetilde{E_2}}$, *where* Σ *is a set of typestate pairs, containing assumed instances of the subtyping relation.*

$$\frac{(w_1{}^{\widetilde{E_1}}, w_2{}^{\widetilde{E_2}}) \in \Sigma}{\Sigma \vdash w_1{}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} w_2{}^{\widetilde{E_2}}} \textsc{Assump}$$

$$\frac{\begin{array}{c}\mathsf{unf}(w_1{}^{\widetilde{E_1}}) = {}_{d_1}\{\widetilde{m : w}\}_1{}^{\widetilde{E_1}} \qquad \mathsf{unf}(w_2{}^{\widetilde{E_2}}) = {}_{d_2}\{\widetilde{m : w}\}_2{}^{\widetilde{E_2}} \\ \forall\ m'{:}w'_2 \in\ {}_{d_2}\{\widetilde{m : w}\}_2{}^{\widetilde{E_2}}\ .\ \exists\ w'_1\ .\ m'{:}w'_1 \in {}_{d_1}\{\widetilde{m : w}\}_1{}^{\widetilde{E_1}}\ \wedge \\ \Sigma \cup (w_1{}^{\widetilde{E_1}}, w_2{}^{\widetilde{E_2}}) \vdash w'_1{}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} w'_2{}^{\widetilde{E_2}} \qquad d_2 = \mathsf{drop} \Rightarrow d_1 = \mathsf{drop}\end{array}}{\Sigma \vdash w_1{}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} w_2{}^{\widetilde{E_2}}} \textsc{Input}$$

$$\frac{\begin{array}{c}\mathsf{unf}(w_1{}^{\widetilde{E_1}}) = \langle\widetilde{l : u}\rangle_1{}^{\widetilde{E_1}} \\ \mathsf{unf}(w_2{}^{\widetilde{E_2}}) = \langle\widetilde{l : u}\rangle_2{}^{\widetilde{E_2}} \qquad \forall\ l'{:}u_1 \in \langle\widetilde{l : u}\rangle_1{}^{\widetilde{E_1}}\ .\ \exists\ u_2\ .\ l'{:}u_2 \in \langle\widetilde{l : u}\rangle_2{}^{\widetilde{E_2}}\ \wedge \\ \Sigma \cup (w_1{}^{\widetilde{E_1}}, w_2{}^{\widetilde{E_2}}) \vdash u_1{}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} u_2{}^{\widetilde{E_2}}\end{array}}{\Sigma \vdash w_1{}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} w_2{}^{\widetilde{E_2}}} \textsc{Output}$$

Theorem 1 (Algorithm completeness)**.** *If* $w_1{}^{\widetilde{E_1}} \leq_{\mathbf{S}} w_2{}^{\widetilde{E_2}}$ *then* $\varnothing \vdash w_1{}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} w_2{}^{\widetilde{E_2}}$.

Theorem 2 (Algorithm soundness)**.** *If* $\varnothing \vdash w_1{}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} w_2{}^{\widetilde{E_2}}$ *then* $w_1{}^{\widetilde{E_1}} \leq_{\mathbf{S}} w_2{}^{\widetilde{E_2}}$.

Corollary 1 (Algorithm soundness and completeness)**.** $\varnothing \vdash w_1{}^{\widetilde{E_1}} \leq_{\mathbf{S_{alg}}} w_2{}^{\widetilde{E_2}}$ *if and only if* $w_1{}^{\widetilde{E_1}} \leq_{\mathbf{S}} w_2{}^{\widetilde{E_2}}$.

5.5.2 Types and Subtyping

To statically track the possible typestates an object might be in, we combine them in union types. To describe combined behaviour from multiple typestates, we also combine them in intersection types. Their usefulness will be made clearer when we will see the result of upcasting a type. Our type hierarchy is a lattice, thus supporting $\top_{\mathbf{t}}$ and $\bot_{\mathbf{t}}$ types: $\top_{\mathbf{t}}$ signals erroneous scenarios, while $\bot_{\mathbf{t}}$ imppossible ones. Notice that, *types* do not include class information. *Typestate trees* will be used for that (Section 5.5.4).

Definition 8 (Type syntax)**.** *We call types, ranged over by meta-variable t, the terms generated by the following grammar. Recall that u refers to typestate terms (Definition 2).*

$$t ::= t \cup t \mid t \cap t \mid u^{\widetilde{E}} \mid \top_{\mathbf{t}} \mid \bot_{\mathbf{t}}$$

For example, the union type $\mathsf{COMF\_ON}^{\widetilde{E_{SUV}}} \cup \mathsf{SPORT\_ON}^{\widetilde{E_{SUV}}}$ describes an object that might be in typestate COMF_ON or SPORT_ON.

Let $\mathcal{T}$ be the set of types produced by rule t. Now we need to define a subtyping notion to apply to types. The setting is inspired in work by Muehlboeck and Tate [MT18], in particular, their definition of reductive subtyping.

Definition 9 (Subtyping on types)**.** *Let $\leq_{\mathbf{T}} \subseteq \mathcal{T} \times \mathcal{T}$ be the relation defined by the following inductive rules.*

$$\frac{}{t \leq_{\mathbf{T}} \top_{\mathbf{t}}}\ \textsc{TopT} \qquad \frac{}{\bot_{\mathbf{t}} \leq_{\mathbf{T}} t}\ \textsc{BotT} \qquad \frac{{u_1}^{\widetilde{E_1}} \leq_{\mathbf{S}} {u_2}^{\widetilde{E_2}}}{{u_1}^{\widetilde{E_1}} \leq_{\mathbf{T}} {u_2}^{\widetilde{E_2}}}\ \textsc{Typestates}$$

$$\frac{t \leq_{\mathbf{T}} t_i}{t \leq_{\mathbf{T}} t_1 \cup t_2}\ \textsc{Union\_R}\ (i \in \{1,2\}) \qquad \frac{t_i \leq_{\mathbf{T}} t}{t_1 \cap t_2 \leq_{\mathbf{T}} t}\ \textsc{Intersection\_L}\ (i \in \{1,2\})$$

$$\frac{t_1 \leq_{\mathbf{T}} t \qquad t_2 \leq_{\mathbf{T}} t}{t_1 \cup t_2 \leq_{\mathbf{T}} t}\ \textsc{Union\_L} \qquad \frac{t \leq_{\mathbf{T}} t_1 \qquad t \leq_{\mathbf{T}} t_2}{t \leq_{\mathbf{T}} t_1 \cap t_2}\ \textsc{Intersection\_R}$$

As a sanity check, we show basic subtyping properties on types: reflexivity and transitivity.

Lemma 3 (Reflexivity)**.** *For all t, then $t \leq_{\mathbf{T}} t$.*

Lemma 4 (Transitivity)**.** *For all t, t', t'', if $t \leq_{\mathbf{T}} t'$ and $t' \leq_{\mathbf{T}} t''$, then $t \leq_{\mathbf{T}} t''$.*

An algorithm to check that two types are in a subtyping relation (*i.e.*, $t \leq_{\mathbf{T}} t'$) can be implemented by proof search on the inference rules in Definition 9. For these, one can observe that the combined syntactic height of the two types being tested, always decreases [MT18]. Therefore, every recursive search path is guaranteed to always reach a point in which both types being compared are

typestates $u^{\widetilde{E}} \in \mathcal{U}$. Since the algorithm to test ${u_1}^{\widetilde{E_1}} \leq_{\mathbf{S}} {u_2}^{\widetilde{E_2}}$ terminates, the overall algorithm to check subtyping also terminates. For example, it is easy to check that $\mathsf{COMF\_ON}^{\widetilde{E_{SUV}}} \leq_{\mathbf{T}} \mathsf{COMF\_ON}^{\widetilde{E_{SUV}}} \cup \mathsf{SPORT\_ON}^{\widetilde{E_{SUV}}}$, using the UNION_R rule in Definition 9.

5.5.3 Basic Operations on Types

We begin outlining several foundational assumptions regarding the class hierarchy. Subsequently, we discuss the core operations performed during the type checking process: ucast, dcast, and evo. To illustrate these operations in action, we analyse the code snippet provided in Listing 5.15. This snippet instantiates an object of type *SUV*, invokes the turnOn method, switches the mode and eventually passes the object to the setSpeed method (lines $4-7$).

Listing 5.15: ClientCode class

```
1  public class ClientCode {
2    public static void example() {
3      SUV suv = new SUV();
4      while (!suv.turnOn()) { System.out.println("turning on..."); }
5      suv.switchMode();
6      setSpeed(suv);
7    }
8
9    private static void setSpeed(@jatyc.lib.Requires("ON") Car car) {
10     if (car instanceof SUV && ((SUV) car).switchMode() == Mode.SPORT) {
11         ((SUV) car).setFourWheels(true);
12     }
13     car.setSpeed(50);
14     car.turnOff();
15   }
16 }
```

The setSpeed method, as indicated by its signature and Requires annotation (line 9), receives a *Car* object in the ON typestate. Notice that, the functionalities associated with the ON typestate are also accessible in the COMF_ON and SPORT_ON typestates. Consequently, the method can be safely executed passing an object of class *Car* in the ON state as well as a *SUV* object in either the COMF_ON or SPORT_ON typestates, regardless of the specific mode. Upon receiving the car object, the method first checks whether it is of type *SUV* and attempts to transition it to the sport mode (line 10). If this transition is successful, the method proceeds

to engage the four-wheel drive feature (line 11). Subsequently, it sets the speed to a predefined value (line 13) and concludes the protocol powering off the car (line 14). This approach ensures that the setSpeed method remains adaptable to different vehicle types and modes, enabling seamless integration within a diverse range of scenarios.

Throughout this Dissertation, we denote the set of class names as $\mathcal{C}$, with C serving as a meta-variable ranging over its elements. Moreover, we operate under the assumption that all classes belong to a *single-inheritance hierarchy.*

Definition 10 (Super relation on classes)**.** super *is a partial function such that, given a class* C, $\mathsf{super}(C)$ *is the unique direct super class of* C*, if there is one.*

Definition 11 (Subtyping relation on classes)**.** *The relation* $\leq \subseteq \ \mathcal{C} \times \mathcal{C}$ *is the reflexive and transitive closure of the* super *relation.*

With classes and their super relation, we now need a notion of reachable states (see Definition 12).

Definition 12 (Reachable states)**.** *The* immediate *state reachability is a relation over* $\mathcal{W} \times \mathcal{W}$*, defined as follows:* $w'^{\widetilde{E}}$ *is immediately reachable from* $w^{\widetilde{E}}$*, if:*

1. $w^{\widetilde{E}} = {}_d\{\widetilde{m : w}\}^{\widetilde{E}}$ *and* $\exists\ m'\ .\ m'{:}w'$ *in* $\widetilde{m : w}$*;*
2. $w^{\widetilde{E}} = \langle\widetilde{l : u}\rangle^{\widetilde{E}}$ *and* $\exists\ l'\ .\ l'{:}w'$ *in* $\widetilde{l : u}$*;*
3. $w^{\widetilde{E}} = s^{\widetilde{E}}$ *and* $\widetilde{E}$ *includes the equation* $s = w'^{\widetilde{E}}$*.*

The state reachability relation *is the reflexive transitive closure of immediate state reachability.*

Recall, each class C has an associated protocol $s^{\widetilde{E}}$, where s is its initial typestate name. We enforce that for any classes C and C' such that $\mathsf{super}(C') = C$, their protocols are subtypes

Definition 13 (Protocol input states)**.** $\mathsf{protIn}(C)$ *is the set of all input states* $u^{\widetilde{E}}$ *that are reachable from protocol* $s^{\widetilde{E}}$ *of class* C*.*

By only considering reachable input states from the initial typestate name of the protocol, we perform an optimisation that avoids dealing with unnecessary typestates, *i.e.,* those unreachable.

To refer to the typestates occurring in a type, we introduce a dedicated auxiliary function, presented in Definition 14.

Definition 14 (Typestates in a type)**.** *Function* $\mathsf{typestates} : \mathcal{T} \to \mathcal{P}(\mathcal{U})$ *is such that, given a type* $t \in \mathcal{T}$*,* $\mathsf{typestates}(t)$ *yields the set of typestates occurring in* t*. Formally,*

$$\mathsf{typestates}(t) = \begin{cases} \mathsf{typestates}(t_1) \cup \mathsf{typestates}(t_2) & \text{if } t = t_1 \cap t_2 \\ \mathsf{typestates}(t_1) \cup \mathsf{typestates}(t_2) & \text{if } t = t_1 \cup t_2 \\ \{u^{\widetilde{E}}\} & \text{if } t = u^{\widetilde{E}} \\ \{\} & \text{if } t = \top_{\mathbf{t}} \vee t = \bot_{\mathbf{t}} \end{cases}$$

Upcast. In the process of upcasting a typestate from class C to class C', we gather all typestates from the protocol of C' that are supertypes of the original one. These supertypes are then combined into an intersection type, amalgamating behaviour from various types. If no supertypes are found, the resulting "empty intersection" yields $\top_{\mathbf{t}}$, signalling an error. In principle, upcast operations are always feasible, as they produce a supertype of the original type. However, the challenge lies in the fact that no operations are safely permissible on $\top_{\mathbf{t}}$. Consequently, while an error might not be immediately flagged during upcasting, attempting to use an object with a $\top_{\mathbf{t}}$ type will inevitably lead to an error in practice. By selecting supertypes during upcasting, we construct a new type that is guaranteed to be a supertype of the original one, as affirmed by Theorem 3. Moreover, by intersecting these supertypes, we create the most "precise" type possible, comprising typestates from C', as ensured by Theorem 4.

Definition 15 (Upcast on types)**.** *Function* $\mathsf{ucast} : \mathcal{T} \times \mathcal{C} \times \mathcal{C} \to \mathcal{T}$ *is such that, given a type* t*, a class* C *whose protocol the typestates in* t *belong to and a class* C' *we want to upcast to;* $\mathsf{ucast}(t, C, C')$ *yields the type obtained by taking the intersection of all supertypes (in the protocol of class* C'*) of typestates included in* t*. The domain of* ucast *only includes triples* (t, C, C') *such that* $\mathsf{typestates}(t) \subseteq$

$\mathsf{protIn}(C)$ *and* $C \leq C'$. *Formally,*

$$\mathsf{ucast}(t, C, C') = \begin{cases} \mathsf{ucast}(t_1, C, C') \cup \mathsf{ucast}(t_2, C, C') & \text{if } t = t_1 \cup t_2 \\ \mathsf{ucast}(t_1, C, C') \cap \mathsf{ucast}(t_2, C, C') & \text{if } t = t_1 \cap t_2 \\ \bigcap\{u'^{\widetilde{E}} \in \mathsf{protIn}(C') \mid t \leq_{\mathbf{T}} u'^{\widetilde{E}}\} & \text{if } t \in \mathcal{U} \\ t & \text{otherwise} \end{cases}$$

To illustrate how the ucast function works, let us examine the setSpeed call in Listing 5.15. After invoking switchMode, the type of suv becomes COMF_ON ∪ SPORT_ON, as we are uncertain about the actual typestate, due to ignoring the output value of switchMode. To determine the type of the object passed as a parameter, we employ the ucast function. We provide the following inputs: (*i*) COMF_ON ∪ SPORT_ON as the type to upcast; (*ii*) *SUV* as the starting class; and (*iii*) *Car* as the target class. Since the given type is a union type composed of two elements, the ucast function unfolds it and generates one intersection for each element (COMF_ON and SPORT_ON), containing all their supertypes. In this scenario, there is only one supertype for each: ON. Therefore,

$$\mathsf{ucast}(\mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}, \mathit{SUV}, \mathit{Car}) = \mathsf{ON}$$

As a sanity check, we show that ucast builds a type where the typestates composing it belong to the class we upcast to. Recall, Definition 15 has constraints $\mathsf{typestates}(t) \subseteq \mathsf{protIn}(C)$ and $C \leq C'$ (the following results assume them). To improve readability we omit stating the constraints explicitly and we simply universally quantify types and classes.

Lemma 5 (Upcast preserves protocol membership)**.** *For all t, C and C′, then* $\mathsf{typestates}(\mathsf{ucast}(t, C, C')) \subseteq \mathsf{protIn}(C')$.

To ensure ucast correctness, we show that the result: (*i*) is a supertype of the given type (Theorem 3); (*ii*) is the "closest" type to the original one with typestates in the protocol of the target class (Theorem 4); and (*iii*) preserves the subtyping relation (Theorem 5), *i.e.,* ucast on types in a subtyping relation produces types that are still in such relation.

Theorem 3 (Upcast consistency)**.** *For all t, C and C', we have* $t \leq_{\mathbf{T}} \mathsf{ucast}(t, C, C')$.

Theorem 4 (Upcast least upper bound)**.** *For all t, t', C and C', such that* $\mathsf{typestates}(t') \subseteq \mathsf{protIn}(C')$ *and* $t \leq_{\mathbf{T}} t'$*, we have* $\mathsf{ucast}(t, C, C') \leq_{\mathbf{T}} t'$.

Theorem 5 (Upcast preserves subtyping)**.** *For all t, t', C and C', such that* $t \leq_{\mathbf{T}} t'$*, we have* $\mathsf{ucast}(t, C, C') \leq_{\mathbf{T}} \mathsf{ucast}(t', C, C')$.

Downcast. To perform a downcast of a typestate from class C to C', we gather all typestates from the protocol of C' that are subtypes of the original one and combine them into a union type. We opt for a union type since we must accommodate all possible typestates that an object might inhabit. By selecting subtypes during downcasting, we construct a new type that is guaranteed to be a subtype of the original one, as formalised in Theorem 6. Furthermore, by forming the union of these subtypes, we assemble the most inclusive type feasible, incorporating typestates from C' that are "closest" to the original type, as ensured by Theorem 7.

Definition 16 (Downcast on types)**.** *Function* $\mathsf{dcast} : \mathcal{T} \times \mathcal{C} \times \mathcal{C} \to \mathcal{T}$ *is such that, given a type t, the class C whose protocol the typestates in t belong to and the class C' we want to downcast to;* $\mathsf{dcast}(t, C, C')$ *yields the type obtained by taking the union of all subtypes (in the protocol of class C') of typestates included in t. The domain of* dcast *only includes triples* (t, C, C') *such that* $\mathsf{typestates}(t) \subseteq \mathsf{protIn}(C)$ *and* $C' \leq C$*. Formally,*

$$\mathsf{dcast}(t, C, C') = \begin{cases} \mathsf{dcast}(t_1, C, C') \cup \mathsf{dcast}(t_2, C, C') & \text{if } t = t_1 \cup t_2 \\ \mathsf{dcast}(t_1, C, C') \cap \mathsf{dcast}(t_2, C, C') & \text{if } t = t_1 \cap t_2 \\ \bigcup \{ {u'}^{\widetilde{E}} \in \mathsf{protIn}(C') \mid {u'}^{\widetilde{E}} \leq_{\mathbf{T}} t \} & \text{if } t \in \mathcal{U} \\ t & \text{otherwise} \end{cases}$$

Notice that, dcast never fails (up to runtime downcasts not throwing exceptions). In no case sub-typestates in the protocol of C' are discovered, dcast returns an empty union that is equivalent to $\bot_{\mathbf{t}}$.

To see how dcast works, let us consider the downcast performed in Listing 5.15. To compute the type of (SUV) car, we use the dcast function presented in Definition 16, passing as parameter: (*i*) ON as the type to downcast (given the Requires annotation); (*ii*) *Car* as the starting class; and (*iii*) *SUV* as the target class. Since the type passed as parameter is a simple typestate, the dcast function just creates a union containing all the subtypes of ON. Concretely,

$$\mathsf{dcast}(\mathsf{ON}, \mathit{Car}, \mathit{SUV}) = \mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}.$$

As a sanity check, we show that dcast builds a type whose typestates belong to the class we downcast to. Recall, Definition 16 has constraints $\mathsf{typestates}(t) \subseteq \mathsf{protIn}(C)$ and $C' \leq C$ (the following results assume them). To improve readability, the constraints are implicit and we universally quantify types and classes.

Lemma 6 (Downcast preserves protocol membership)**.** *For all t, C and C', we have* $\mathsf{typestates}(\mathsf{dcast}(t, C, C')) \subseteq \mathsf{protIn}(C')$.

To ensure dcast correctness, we show that the result: (*i*) is a subtype of the given type (Theorem 6); (*ii*) is the "closest" type to the original with typestates in the protocol of the target class (Theorem 7); and (*iii*) preserves the subtyping relation *i.e.,* dcast on types in a subtyping relation produces types that are still in such relation (Theorem 8).

Theorem 6 (Downcast consistency)**.** *For all t, C and C', we have* $\mathsf{dcast}(t, C, C') \leq_{\mathbf{T}} t$.

Theorem 7 (Downcast greatest lower bound)**.** *For all t, t', C and C', such that* $\mathsf{typestates}(t') \subseteq \mathsf{protIn}(C')$ *and* $t' \leq_{\mathbf{T}} t$*, we have* $t' \leq_{\mathbf{T}} \mathsf{dcast}(t, C, C')$.

Theorem 8 (Downcast preserves subtyping)**.** *For all t, t', C and C', such that* $t \leq_{\mathbf{T}} t'$*, we have* $\mathsf{dcast}(t, C, C') \leq_{\mathbf{T}} \mathsf{dcast}(t', C, C')$.

Additionally, we relate the result of upcasting and then downcasting with the original type as well as the result of downcasting and then upcasting. The first follows from Theorems 3 and 7, the second from Theorems 4 and 6. These corollaries are also important to ensure the soundness of our approach (see Theorem 16).

Corollary 2 (Downcast reverses upcast)**.** *For all t, C and C', we have* $t \leq_{\mathbf{T}} \mathsf{dcast}(\mathsf{ucast}(t, C, C'), C', C)$.

Corollary 3 (Upcast reverses downcast)**.** *For all t, C and C, we have* $\mathsf{ucast}(\mathsf{dcast}(t, C, C'), C', C) \leq_{\mathbf{T}} t$.

Evolve. Whenever we call a method call on an object with a specific type, we need to compute the resulting type, *i.e.,* the possible typestates the object could inhabit post-call. This computation, not only helps refining our understanding of the object state, but also aids in identifying any potential violations of protocol behaviour. To this aim, we introduce the evo function, which outputs $\top_{\mathbf{t}}$ when a method is not callable within the given type context.

Definition 17 (Evolve)**.** *Function* $\mathsf{evo} : \mathcal{T} \times \boldsymbol{MNames} \times \boldsymbol{LNames} \to \mathcal{T}$ *is such that, given a type t, a method m, and an output l,* $\mathsf{evo}(t, m, l)$ *yields the new type obtained by executing m on any object currently with type t, where l is an output value potentially returned by m. Its definition relies on the auxiliary function* $\mathsf{evoO} : \mathcal{W} \times \boldsymbol{LNames} \to \mathcal{U}$. *Formally,*

$$\mathsf{evo}(t, m, l) = \begin{cases} \mathsf{evo}(t_1, m, l) \cup \mathsf{evo}(t_2, m, l) & \text{if } t = t_1 \cup t_2 \\ \mathsf{evo}(t_1, m, l) \cap \mathsf{evo}(t_2, m, l) & \text{if } t = t_1 \cap t_2 \\ \mathsf{evoO}(w^{\widetilde{E}}, l) & \text{if } t = u^{\widetilde{E}} \wedge m\!:\!w \in \mathsf{unf}(u^{\widetilde{E}}) \\ t & \text{otherwise} \end{cases}$$

$$\mathsf{evoO}(w^{\widetilde{E}}, l) = \begin{cases} u^{\widetilde{E}} & w^{\widetilde{E}} = \langle l : u \ \widetilde{l : u} \rangle^{\widetilde{E}} \\ w^{\widetilde{E}} & w^{\widetilde{E}} \in \mathcal{U} \\ \bot_{\mathbf{t}} & \text{otherwise} \end{cases}$$

As evo operates deterministically, it is defined as a function rather than a labelled transition system.

To understand how evo works, let us examine the switchMode call in Listing 5.15. To compute the type of car, we employ evo as defined in Definition 17, with the following parameters: (i) the type COMF_ON∪SPORT_ON resulting from

the dcast function; (*ii*) the method switchMode to drive the evolution; and (*iii*) the expected output Mode.SPORT to trigger the if branch. Since the type passed as parameter is a union type consisting of two elements, the evo function is invoked recursively. Subsequently, the auxiliary function evoO is invoked for COMF_ON and SPORT_ON. Specifically,

$$\mathsf{evo}(\mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}, \mathsf{switchMode}, \mathsf{Mode.SPORT}) = \mathsf{SPORT\_ON}.$$

As a sanity check, we show that evo produces a type containing only typestates belonging to the initial class.

Lemma 7 (Evolve preserves protocol membership)**.** *For all t, m, l, C,* $\mathsf{typestates}(t) \subseteq \mathsf{protIn}(C)$ *implies* $\mathsf{typestates}(\mathsf{evo}(t, m, l)) \subseteq \mathsf{protIn}(C)$.

To ensure evo correctness, we show that evo on types in a subtyping relation produces types that still are in such a relation.

Theorem 9 (Evolve preserves subtyping)**.** *For all t and t' such that $t \leq_{\mathbf{T}} t'$, we have that* $\mathsf{evo}(t, m, l) \leq_{\mathbf{T}} \mathsf{evo}(t', m, l)$.

We further establish the relation between evo, ucast, and dcast by demonstrating that: (*i*) applying ucast after evo yields a subtype of the inverse sequence of operations (Theorem 10); and (*ii*) applying dcast after evo yields a supertype of the inverse sequence of operations (Theorem 11). These theorems play a pivotal role in ensuring the soundness (Theorem 16) of our approach. To enhance readability, we refrain from explicitly stating the constraints on the universally quantified variables necessary for using ucast and dcast.

Theorem 10 (Evolve and upcast)**.** *For all t, m, l, C and C', we have that* $\mathsf{ucast}(\mathsf{evo}(t, m, l), C, C') \leq_{\mathbf{T}} \mathsf{evo}(\mathsf{ucast}(t, C, C'), m, l)$.

Theorem 11 (Evolve and downcast)**.** *For all t, m, l, C and C', we have that* $\mathsf{evo}(\mathsf{dcast}(t, C, C'), m, l) \leq_{\mathbf{T}} \mathsf{dcast}(\mathsf{evo}(t, m, l), C, C')$.

5.5.4 Typestate Trees

Here, we describe *typestate trees*, the data structure we introduce to solve the problem of casting in the middle of a protocol. These trees associate class information with types containing only typestates in the protocol of the class they are associated to (*i.e.,* $\mathsf{typestates}(t) \subseteq \mathsf{protIn}(C)$). The tree root indicates the static type of a variable and the corresponding type t in $\mathcal{T}$ at a given program point. All other nodes describe what should be the type if we upcast/downcast to the corresponding class. The type in the root is always a sound approximation of the runtime execution. The types in other nodes are sound only if the object is an instance of the corresponding class. This implies that type safety is guaranteed up-to class downcasts being performed to a class of which an object is a subtype of. Hereafter we define well-formed typestate trees and auxiliary functions. The main operations on typestate trees are: $\mathsf{ucastTT}$, $\mathsf{dcastTT}$, evoTT, and mrgTT.

Definition 18 (Typestate trees)**.** *Recall that C ranges over classes (denoted by $\mathcal{C}$) and t ranges over types (denoted by $\mathcal{T}$). Let $\mathcal{TT}$ be the smallest set of triples satisfying the following rules:*

$$\frac{}{(C, t, \{\}) \in \mathcal{TT}} \qquad \frac{n \geq 1 \qquad \forall\, i,\, 1 \leq i \leq n \,.\, tt_i \in \mathcal{TT}}{(C, t, \{tt_i \mid 1 \leq i \leq n\}) \in \mathcal{TT}}$$

Notice that, triples in $\mathcal{TT}$ represent trees and are composed of: the class C, the type t of the root and a set of subtrees (again triples in $\mathcal{TT}$), one for each root child. Such a set is empty (case $n = 0$) if the tree root has no children (*i.e.,* the tree simply represents a leaf). Throughout this Chapter, tt ranges over elements of $\mathcal{TT}$ and tts ranges over sets of elements of $\mathcal{TT}$.

We need functions to destroy an element of $\mathcal{TT}$ (which is a triple). Let $\mathsf{cl}((C, t, tts)) = C$, $\mathsf{ty}((C, t, tts)) = t$, and $\mathsf{children}((C, t, tts)) = tts$.

Definition 19 (No duplicate classes)**.** *The predicate* nodup *asserts that, given a set $tts \in \mathcal{P}(\mathcal{TT})$, no two typestates trees in tts have the same associated class. Formally,* $\mathsf{nodup}(tts)$ *holds if:* $\forall\, tt, tt' \in tts \,.\, \mathsf{cl}(tt) = \mathsf{cl}(tt') \Rightarrow tt = tt'$.

Definition 20 (Well-formedness of typestate trees)**.** *The predicate $\vdash$ over*

$\mathcal{TT}$ asserts that, given a typestate tree (C, t, tts), it is well formed. Formally,

$$\frac{\mathsf{typestates}(t) \subseteq \mathsf{protIn}(C) \qquad \mathsf{nodup}(tts) \qquad \forall tt \in tts\,.\,\mathsf{super}(\mathsf{cl}(tt)) = C \wedge \mathsf{ucast}(\mathsf{ty}(tt), \mathsf{cl}(tt), C) \leq_{\mathbf{T}} t \wedge \vdash tt}{\vdash (C, t, tts)}$$

A typestate tree (C, t, tts) is well-formed under the following conditions: (i) all the typestates of type t belong to the protocol of class C; (ii) nodup holds; (iii) the classes associated with each child tree are direct subclasses of C; (iv) if we upcast a type of a child tree, we get a subtype of t; and (v) each child is also well-formed. Condition (iv) ensures that the type of a child tree is never less "precise" than the type of the parent. From now on, we only consider well-formed typestate trees.

To illustrate the concept, suppose that in Listing 5.15, instead of assigning the created object to a *SUV* variable, we assign it to a *Car* one. Since the static and actual type are different, we need a typestate tree to handle future casts. Given Definition 18, the resulting typestate tree is $(Car, \mathsf{OFF}, \{(SUV, \mathsf{OFF}, \{\})\})$.

Upcast. Upcasting a typestate tree to class C ensures that the resulting root class is C, by recursively following the super relation and building up new tree roots until the root class is C.

Definition 21 (Upcast on typestate trees)**.** *Function* $\mathsf{ucastTT} : \mathcal{TT} \times \mathcal{C} \to \mathcal{TT}$ *is such that* $\mathsf{ucastTT}((C, t, tts), C')$ *performs an upcast on typestate tree* (C, t, tts) *to class* C'*. The domain of* ucastTT *only includes pairs* $((C, t, tts), C')$ *such that* $C \leq C'$*. Formally,*

$\mathsf{ucastTT}((C, t, tts), C') =$

$$\begin{cases} (C, t, tts) & \text{if } C = C' \\ \mathsf{ucastTT}((\mathsf{super}(C), \mathsf{ucast}(t, C, \mathsf{super}(C)), \{(C, t, tts)\}), C') & \text{otherwise} \end{cases}$$

Notice that, under the assumption on the domain of the ucastTT, the function terminates since the distance between C and C' decreases with each recursive step.

Theorem 12 (Upcast preserves typestate trees well-formedness)**.** *For all* C'*,* tt*, such that* $\vdash tt$ *and* $\mathsf{cl}(tt) \leq C'$*, it holds that* $\vdash \mathsf{ucastTT}(tt, C')$*.*

To see how the ucastTT function works, let us consider the setSpeed call in Listing 5.15. After calling switchMode, the object suv has the following typestate tree $(SUV, \mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}, \{\})$. When passing suv to setSpeed, we need to upcast from SUV to Car. To do that, we use the ucastTT function passing as parameters: (i) $(SUV, \mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}, \{\})$ as the typestate tree to upcast; and (ii) Car as the target class. Thus,

$$\mathsf{ucastTT}((SUV, \mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}, \{\}), Car)$$

is equal to

$$(Car, \mathsf{ON}, \{(SUV, \mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}, \{\})\}).$$

To upcast a typestate tree, we must perform multiple upcasts, incrementally building up new tree roots, not only to preserve the well-formedness property, but also to ensure soundness. For readability sake, we show the problem with an abstract, but simple example. We take classes A, B and C, such that $\mathsf{super}(C) = B$, $\mathsf{super}(B) = A$ and the protocol equations associated with each class are listed below. Recall that $\mathsf{end} = {}_{\mathsf{drop}}\{\}$.

$$\begin{aligned}
\mathsf{A1} &= \{\ \mathsf{m1} : \mathsf{end}\ \} \\
\mathsf{B1} &= \{\ \mathsf{m1} : \mathsf{end},\ \mathsf{m2} : \mathsf{end}\ \} \\
\mathsf{C1} &= \{\ \mathsf{m1} : \mathsf{end},\ \mathsf{m2} : \mathsf{end},\ \mathsf{m3} : \mathsf{C2}\ \} \\
\mathsf{C2} &= \{\ \mathsf{m1} : \mathsf{end},\ \mathsf{m4} : \mathsf{end}\ \}
\end{aligned}$$

Given the protocols above and according to Definition 5 we have:

$$\mathsf{C1} \leq_{\mathbf{S}} \mathsf{B1} \leq_{\mathbf{S}} \mathsf{A1} \text{ and } \mathsf{C2} \leq_{\mathbf{S}} \mathsf{A1}, \text{ but } \mathsf{C2} \not\leq_{\mathbf{S}} \mathsf{B1}.$$

C2 not being a subtype of B1 is not a problem *per se*, but it may be when upcasting, if we define it to go directly to the root instead of going level-by-level, as downcasting after upcasting should lead to the original state [8]. To see that, consider the code in Listing 5.16, which contains an unsafe method call, but would be accepted. At first, we create an object c of class C and we call its method $m3$,

[8] Technically, downcasting after upcasting returns an over-approximation of the original state.

producing a new typestate, *i.e.,* C2. Then, we assign c to variable a performing an upcast from class C to A (and from typestate C2 to A1). We finally perform a sequence of downcasts on a leading the object to class C (and to typestate C1).

Listing 5.16: Direct upcast example

```
1 C c = new C(); // C1
2 c.m3(); // C2
3 A a = C; // A1: unsound upcast!
4 B b = (B) a; // B1: downcast level-by-level
5 C c = (C) b; // C1: incorrect! the state should be C2 (that of line 2)
6 c.m2(); // unsafe!
```

The result of upcasting C2 directly to class A (see line 3) is A1, since it is the only supertype of C2, *i.e.,* $\mathsf{C2} \leq_{\mathbf{S}} \mathsf{A1}$. To downcast A1 to typestates of class B, we check all the typestates in the protocol of B subtypes of A1, *i.e.,* B1 is the downcast result (see line 4). Similarly, only C1 is a subtype of B1, thus it is the result of downcasting from B1 to typestates of class C (see line 5). Notice how a direct upcast to class A, followed by a downcast to B and then to C, result in a different typestate with respect to the initial one. This is unsound: C1 and C2 are unrelated. The direct upcast to A makes us losing the information about C1 not having supertypes among typestates in B. Since we first upcast C2 to B, getting $\top_{\mathbf{t}}$ as result, we find out that C2 has no supertypes among typestates in B. Additionally, since we use typestate trees, downcasting to C leads back to C2.

Downcast. When downcasting a given typestate tree tt to class C, we ensure that the root class of the resulting tree is C. If we find a subtree in tt whose class is C, we pick it[9] as the result. Otherwise, we build a new tree downcasting from the most "precise" type information in tt. To this aim, we define the auxiliary function closestTT to look for the subtree whose class is hierarchically the "closest" to C. The definition is below, followed by examples.

Definition 22 (Closest subtree)**.** *The function* $\mathsf{closestTT} : \mathcal{TT} \times \mathcal{C} \to \mathcal{TT}$ *is such that* $\mathsf{closestTT}(tt, C)$ *yields the subtree associated with the closest superclass of* C *occurring in* tt*. The domain of* closestTT *only includes pairs* (tt, C) *such that*

[9]By well-formedness, it is unique.

$C \leq \mathsf{cl}(tt)$. *Formally,*

$$\mathsf{closestTT}(tt, C) = \begin{cases} \mathsf{closestTT}(tt', C) & \text{if } \exists\, tt' \in \mathsf{children}(tt)\,.\; C \leq_{\widetilde{D}} \mathsf{cl}(tt') \\ tt & \text{otherwise} \end{cases}$$

To illustrate the use of the closestTT function, consider classes A, B, and C, where $\mathsf{super}(B) = A$ and $\mathsf{super}(C) = B$. Let tt be $(A, t, \{(B, t', \{\})\})$. Then the following equalities hold: $\mathsf{closestTT}(tt, A) = tt$; $\mathsf{closestTT}(tt, B) = (B, t', \{\})$; and $\mathsf{closestTT}(tt, C) = (B, t', \{\})$. The first and second case are easy to understand: the function yields the subtree whose class is precisely the one we are looking for. In the third case, since there is no subtree in tt whose class is C, $\mathsf{closestTT}(tt, C)$ yields the subtree corresponding to B, which is the "closest" superclass of C occurring in tt, *i.e.,* $(B, t', \{\})$. Now, suppose instead that $\mathsf{super}(B) = A$ and $\mathsf{super}(C) = A$ (*i.e., B* and C are "siblings"). Then $\mathsf{closestTT}(tt, C)$ would yield the entire tree tt whose class is A, which is the "closest" superclass of C occurring in tt. Lemma 8 ensures the correctness of closestTT and is useful for the soundness proof (Theorem 16).

Lemma 8 (Closest correctness)**.** *For all tt and C, if $C \leq \mathsf{cl}(tt)$ then* $C \leq \mathsf{cl}(\mathsf{closestTT}(tt, C))$.

Definition 23 (Downcast on typestate trees)**.** *Function* $\mathsf{dcastTT} : \mathcal{TT} \times \mathcal{C} \to \mathcal{TT}$ *is such that* $\mathsf{dcastTT}(tt, C)$ *performs a downcast on typestate tree tt to class C. The domain of* dcastTT *only includes pairs (tt, C) such that $C \leq \mathsf{cl}(tt)$. Formally,* $\mathsf{dcastTT}(tt, C) =$

$$\begin{cases} tt' & \text{if } tt' = \mathsf{closestTT}(C, tt) \wedge C = \mathsf{cl}(tt') \\ (C, \mathsf{dcast}(\mathsf{ty}(tt'), \mathsf{cl}(tt'), C), \{\}) & \text{if } tt' = \mathsf{closestTT}(C, tt) \wedge C \neq \mathsf{cl}(tt') \end{cases}$$

Theorem 13 (Downcast preserves typestate trees well-formedness)**.** *For all C, tt, such that $\vdash tt$ and $C \leq \mathsf{cl}(tt)$, it holds that* $\vdash \mathsf{dcastTT}(tt, C)$.

To see how dcastTT works, let us suppose that in Listing 5.15, in the setSpeed method, we want to check the downcast (SUV) car. To compute its typestate tree, we use dcastTT passing as parameter: (i) (Car,ON,$\{\}$) as the typestate tree to

downcast; (*ii*) *SUV* as the target class. Notice that, in the case the root is also a leaf, we need to replace it with the result of dcastTT. Concretely,

$$\mathsf{dcastTT}((Car, \mathsf{ON}, \{\}), SUV)$$

is equal to

$$(SUV, \mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}, \{\}).$$

Evolve. To compute the typestate tree of an object after a method call, we define the evoTT function.

Definition 24 (Evolve on typestate trees)**.** *Function* $\mathsf{evoTT} : \mathcal{TT} \times \boldsymbol{MNames} \times \boldsymbol{LNames} \rightarrow \mathcal{TT}$ *is such that* $\mathsf{evoTT}(tt, m, l)$ *yields a new typestate tree resulting from applying* $\mathsf{evo}(t, m, l)$ *(Definition 17) to all the nodes of* tt*. Formally,*

$$\mathsf{evoTT}((C, t, tts), m, l) = (C, \mathsf{evo}(t, m, l), \bigcup_{tt \in tts} \mathsf{evoTT}(tt, m, l))$$

Notice that, whenever the set tts is empty, $\mathsf{evoTT}((C, t, tts), m, l)$ is equal to $(C, \mathsf{evo}(t, m, l), \{\})$

Theorem 14 (Evolve preserves typestate trees well-formedness)**.** *For all* tt*,* m*,* l*, such that* $\vdash tt$*, it holds that* $\vdash \mathsf{evoTT}(tt, m, l)$*.*

Listing 5.17: EvolveTT example

```
1 Car c = new SUV();
2 if (c.turnOn()) c.turnOff();
```

To see how evoTT works, consider the code presented in Listing 5.17. The typestate tree of c is $(Car, \mathsf{OFF}, \{(SUV, \mathsf{OFF}, \{\})\})$. When the turnOn call occurs, we need to "evolve" each node of the typestate tree. To compute the resulting tree, we use evoTT passing as parameter: (*i*) the typestate tree of c; (*ii*) turnOn as the method called; and (*iii*) true as the expected output to enter the if branch. Concretely,

$$\mathsf{evoTT}((Car, \mathsf{OFF}, \{(SUV, \mathsf{OFF}, \{\})\}), \mathsf{turnOn}, \mathsf{true})$$

is equal to

$$(Car, \mathsf{ON}, \{(SUV, \mathsf{ON}, \{\})\}).$$

Notice that, every node of the typestate tree is "evolved" using the evo function.

Merge. In the case of branching code, one has to merge type information coming from all different branches, so that subsequent code can be analysed considering all possibilities. To this end, we define the mrgTT function to merge two typestate trees and its auxiliary functions.

Definition 25. *Function* $\mathsf{height} : \mathcal{P}(\mathcal{TT}) \to \mathcal{N}$ *is such that* $\mathsf{height}(tt)$ *yields the greatest number of nodes traversed in* tt*, from the root to one of the leaves (both included).*

Definition 26. *Function* $\mathsf{clss} : \mathcal{P}(\mathcal{TT}) \to \mathcal{P}(\mathcal{C})$ *is such that* $\mathsf{clss}(tts)$ *yields the set of classes associated with the typestate trees in* tts*. Formally,* $\mathsf{clss}(tts) = \{\mathsf{cl}(tt) \mid tt \in tts\}$.

Definition 27. *Function* $\mathsf{find} : \mathcal{C} \times \mathcal{P}(\mathcal{TT}) \to \mathcal{TT}$ *is such that, given a class* C *and set of typestate trees* tts *with* $C \in \mathsf{clss}(tts)$ *and* $\mathsf{nodup}(tts)$*,* $\mathsf{find}(C, tts)$ *yields the unique typestate tree in set* tts *whose class is* C*.*

Definition 28 (Merge)**.** *Function* $\mathsf{mrgTT} : \mathcal{TT} \times \mathcal{TT} \to \mathcal{TT}$ *is such that, given typestate trees* tt *and* tt'*,* $\mathsf{mrgTT}(tt, tt')$ *yields the typestate tree obtained by merging* tt *and* tt'*. The domain of* mrgTT *only includes pairs* (tt, tt') *such that* $\mathsf{cl}(tt) = \mathsf{cl}(tt')$*. Formally,*
$\mathsf{mrgTT}((C, t, tts), (C, t', tts')) = (C, t \cup t', tts_1 \cup tts_2 \cup tts_3)$

$$\textit{where} \quad tts_1 = \bigcup_{C' \in \mathsf{clss}(tts) \cap \mathsf{clss}(tts')} \mathsf{mrgTT}(\mathsf{find}(C', tts), \mathsf{find}(C', tts'))$$

$$tts_2 = \bigcup_{C' \in \mathsf{clss}(tts) - \mathsf{clss}(tts')} \mathsf{mrgTT}(\mathsf{find}(C', tts), (C', \mathsf{dcast}(t', C, C'), \{\}))$$

$$tts_3 = \bigcup_{C' \in \mathsf{clss}(tts') - \mathsf{clss}(tts)} \mathsf{mrgTT}((C', \mathsf{dcast}(t, C, C'), \{\}), \mathsf{find}(C', tts'))$$

Notice that, mrgTT terminates as $\mathsf{height}(tt) + \mathsf{height}(tt')$ decreases at each recursive step. Moreover, it is symmetric, *i.e.,* $\mathsf{mrgTT}(tt, tt') = \mathsf{mrgTT}(tt', tt)$.

Theorem 15 (Merge preserves typestate tree well-formedness)**.** *For all* tt*,* tt'*, such that* $\mathsf{cl}(tt) = \mathsf{cl}(tt')$*,* $\vdash tt$*, and* $\vdash tt'$*, it holds that* $\vdash \mathsf{mrgTT}(tt, tt')$*.*

To see how mrgTT works, consider the if statement in Listing 5.15 (lines 12-14). Although the else branch is missing, to compute the typestate tree of car, we need to consider it to be there (to account for all possible outputs returned by switchMode). To compute such typestate tree, we use mrgTT passing as parameters: (i) $(SUV, \mathsf{SPORT\_ON}, \{\})$; and (ii) $(SUV, \mathsf{COMF\_ON}, \{\})$. Since no parameters have children nodes, it is enough to make the union of the root types. Concretely,

$$\mathsf{mrgTT}((SUV, \mathsf{SPORT\_ON}, \{\}), (SUV, \mathsf{COMF\_ON}, \{\}))$$

is equal to

$$(SUV, \mathsf{SPORT\_ON} \cup \mathsf{COMF\_ON}, \{\}).$$

Typestate trees and the functions to manage them are targeted at handling the problem of upcasting and downcasting from and to multiple typestates. In the following Section, we will explain how our approach is sound.

5.5.5 Typestate Trees Soundness

We now discuss why we consider type-safe a programming language equipped with the subtypestate mechanism outlined here. This classification hinges on a pivotal property we elucidate: when operating within a typestate tree that accurately mirrors the current runtime type of an object, subsequent operations yield new typestate trees that continue to accurately reflect the runtime type. This assertion is contingent upon class downcasts being conducted to a class of which the object is a subtype, *i.e.,* downcast not throwing exceptions at runtime. Consequently, we refrain from furnishing static assurances that class downcasts will be exempt from throwing such exceptions at runtime.

Definition 29 (Sequence of upcasts on types)**.** *Function* $\mathsf{ucast}^* : \mathcal{T} \times \mathcal{C} \times \mathcal{C} \to \mathcal{T}$ *is such that* $\mathsf{ucast}^*(t, C, C')$ *performs zero or more upcasts from* C *to* C' *step-by-step, following the class hierarchy. The domain of* ucast^* *only includes triples* (t, C, C')

such that $\mathsf{typestates}(t) \subseteq \mathsf{protIn}(C)$ *and* $C \leq C'$. *Formally,*

$$\mathsf{ucast}^*(t, C, C') = \begin{cases} t & \text{if } C = C' \\ \mathsf{ucast}^*(\mathsf{ucast}(t, C, \mathsf{super}(C)), \mathsf{super}(C), C') & \text{otherwise} \end{cases}$$

Since the distance between C and C' decreases with each recursive step, it is obvious that ucast^* terminates.

The subsequent relation delineates a typestate tree where type information is sound with respect to class C' and type t'. In essence, if C' and t' precisely represent the runtime type of a given object, a sound typestate tree aptly approximates this type. Notice that, the root must inherently be sound concerning runtime, while the remaining nodes necessitate soundness only if the initialising class of the object is a subclass of the one associated with that node. Thus, all non-root nodes describe the type of the object if indeed the object is an instance of the corresponding class. This implies that if we downcast, possibly turning a non-root node into the new root, we preserve soundness only if the runtime downcast succeeds.

Definition 30 (Soundness of typestate trees)**.** *Given a well-formed typestate tree* tt*, the predicate* $\vdash_{C,t}$*, with* $\mathsf{typestates}(t) \subseteq \mathsf{protIn}(C)$*, asserts that* tt *it is sound with respect to class* C *and type* t. *Formally,*

$$\frac{C \leq C' \qquad \mathsf{ucast}^*(t, C, C') \leq_{\mathbf{T}} t' \qquad \forall tt \in tts\,.\; C \leq \mathsf{cl}(tt) \;\Rightarrow \vdash_{C,t} tt}{\vdash_{C,t} (C', t', tts)}$$

The next theorem shows that soundness is preserved by typestate tree operations. Soundness after downcast is only preserved if it does not throw an exception (*i.e.,* the assumption $C \leq C' \leq \mathsf{cl}(tt)$ on the second item of Theorem 16).

Theorem 16 (Typestate tree soundness preservation)**.** *Soundness is preserved by:*

> ***upcast*** *– for all* C, t, C', tt, *such that* $\vdash_{C,t} tt$ *and* $\mathsf{cl}(tt) \leq C'$, *it holds that* $\vdash_{C,t} \mathsf{ucastTT}(tt, C')$*;*
>
> ***downcast*** *– for all* C, t, C', tt, *such that* $\vdash_{C,t} tt$ *and* $C \leq C' \leq \mathsf{cl}(tt)$, *it holds that* $\vdash_{C,t} \mathsf{dcastTT}(tt, C')$*;*

evolve – *for all* C, t, tt, m, l, *such that* $\vdash_{C,t} tt$, *it holds that* $\vdash_{C,evolve(t,m,l)} \mathsf{evoTT}(tt, m, l)$;

merge – *for all* C, t, tt_1, tt_2, *such that* $\vdash_{C,t} tt_1$ *or* $\vdash_{C,t} tt_2$, *and* $\mathsf{cl}(tt_1) = \mathsf{cl}(tt_2)$, *it holds that* $\vdash_{C,t} \mathsf{mrgTT}(tt_1, tt_2)$.

Having shown our approach to be sound, as we will see, Section 5.6 explains how the functions defined in Section 5.5.4 are implemented in JaTyC and used during the type checking process.

5.5.6 Typestate Trees Subtyping

While vertical subtyping (*i.e.,* among nodes of the same typestate tree) is ensured by the well-formedness property, we have no guarantee that the horizontal one (*i.e.,* among nodes with the same root class of different typestate trees) holds. Thus, *in the context of this Dissertation*, we establish a subtyping relation $\leq_{\mathbf{TC}} \subseteq \mathcal{TT} \times \mathcal{TT}$ among typestate trees (where TC stands for Type Checking, as such subtyping relation is used in the type checking procedure). The intuition is that to ensure that $tt \leq_{\mathbf{TC}} tt'$, we need to check such a relation for all existing nodes in tt and tt'. However, it may happen that some nodes included in tt do not appear in tt' and vice versa. Thus, we need to to build those nodes, during the subtyping check. We now define the notion of subtyping among typestate trees.

Definition 31 (Subtyping on typestate trees)**.** *Let* $\leq_{\mathbf{TC}} \subseteq \mathcal{TT} \times \mathcal{TT}$ *be the subtyping relation over typestate trees defined by the following inductive rules.*

$$\frac{C \leq C' \qquad (C, t, tts) \leq_{\mathbf{TC}} \mathsf{dcastTT}((C', t', tts'), C)}{(C, t, tts) \leq_{\mathbf{TC}} (C', t', tts')}\ \textsc{Tree}$$

$$\frac{\begin{array}{c} t \leq_{\mathbf{T}} t' \\ \forall C' \in \mathsf{clss}(tts) \cap \mathsf{clss}(tts') \,.\, \mathsf{find}(C', tts) \leq_{\mathbf{TC}} \mathsf{find}(C', tts') \\ \forall C' \in \mathsf{clss}(tts') - \mathsf{clss}(tts) \,.\, \mathsf{dcastTT}((C, t, tts), C') \leq_{\mathbf{TC}} \mathsf{find}(C', tts') \end{array}}{(C, t, tts) \leq_{\mathbf{TC}} (C, t', tts')}\ \textsc{Root}$$

The intuintion behind these rules is the following. The Tree rule removes nodes from the right-hand side typestate tree such that the root matches the left-hand

one, downcasting to the root class of the sub-typestate tree. Once the root classes match, via the Root rule, we check subtyping node by node, building via dcastTT the missing ones. If both typestate trees are leaves, the universal quantifiers are vacuosly true, thus if $t \leq_{\mathbf{T}} t'$ holds, then the subtyping relation holds.

Since the relation $\leq_{\mathbf{TC}}$ internally uses the $\leq_{\mathbf{T}}$ and $\leq$ relations, it is straightforward to show that basic subtyping properties on typestate trees such as reflexivity and transitivity hold.

Lemma 9 (Reflexivity)**.** *For all tt, then $tt \leq_{\mathbf{TC}} tt$.*

Lemma 10 (Transitivity)**.** *For all tt, tt', tt'', if $tt \leq_{\mathbf{TC}} tt'$ and $tt' \leq_{\mathbf{TC}} tt''$, then $tt' \leq_{\mathbf{TC}} tt''$.*

Notice that, the Root rule does not explicitly require that the root children whose class C appears *only* in the typestate tree on left-hand side of the relation to be subtypes of those built via dcastTT (using C as target class). Such a constraint is vacuosly true, as proved by Lemma 11.

Lemma 11 (Typestate tree subtyping preservation)**.** *Let tt and tt' be such that $\mathsf{cl}(tt) = \mathsf{cl}(tt')$ and $\mathsf{ty}(tt) \leq_{\mathbf{T}} \mathsf{ty}(tt')$. For all $C' \in \mathsf{clss}(\mathsf{children}(tt))$ such that $C' \notin \mathsf{clss}(\mathsf{children}(tt'))$ holds that $\mathsf{find}(C', \mathsf{children}(tt)) \leq_{\mathbf{TC}} \mathsf{dcastTT}(tt', C')$.*

5.6 Embedding Behavioural Casting in JaTyC

The theoretical machinery we presented so far is notably broad and adaptable to numerous statically typed object-oriented languages. In this Section, we embed our theoretical work into a real-world type checker, implementing the presented concepts within JaTyC. To this aim, we present the type system JaTyC implements to track object states and thoroughly describe how JaTyC type embeds typestate trees in the type checking procedure, using a common syntax as a reference point.

5.6.1 JaTyC Type System

Each variable or field declaration and each expression in the code is associated with a Java type, which is statically known. To be able to track the possible

type (those defined in Definition 8) of each object, we need a parallel type system with other types that encapsulates the information JaTyC needs to correctly type check programs. To this aim, in Definition 35, we introduce Type Checking (TC) types, as citizens of our type system. Notice that, to perform the type checking procedure, the types in Definition 8 are not expressive enough, as we need to account for, *e.g.,* shared objects and null. Thus, we now present the extend version of Definitions 8 and 9.

Definition 32 (Extended type syntax)**.** *We extend the types grammar (see Definition 8) with* Shared *and* Null *types.*

$$t \;::=\; \ldots \mid \mathsf{Shared} \mid \mathsf{Null}$$

Whenever JaTyC assigns the Shared type to a typestate tree, it signals that such variable points to a non-linear object, *e.g.,* an aliased object or an object without protocol. A client code can use these variables via anytime method calls (see Section 5.4), because they do not own any protocol. This draws inspiration from the ownership concept of the Rust language: if something takes ownership of some data, such data are considered to be "moved" and the previous reference cannot be used[10]. The ownership concept also avoids the need to nullify variables after a value is read, as also proposed by Boyland [Boy01]. This ensures that objects are used linearly. Whenever JaTyC assigns the Null type, it signals that such object stores the null value. Having a specific type t encoding the null value, different from $\top_{\mathbf{t}}$, is crucial to distinguish among errors (signalled by $\top_{\mathbf{t}}$) and expected scenarios, *e.g.,* objects storing null, making our type checking procedure more flexible and enlarging the set of correct programs we are able to type check. During type checking, if we encounter these assignments, we need to relate the right-hand side expression, *i.e.,* the null value, with a dedicated typestate tree. To this aim, we introduce Definition 33, a specific class for the null value (as we already have a specific type, *i.e.,* Null).

Definition 33 (Null class)**.** *The class $\bot_{\mathbf{C}}$, used as dummy class for the value* null *during type checking, is such that $\bot_{\mathbf{C}} \in \mathcal{C}$ and for all $C \in \mathcal{C}$, it holds that $\bot_{\mathbf{C}} \leq C$.*

[10]https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

Notice that, we use C to range over both class names and $\bot_{\mathbf{C}}$. Thus, during type checking, whenever JaTyC encounter the null value, such value is typed with the typestate tree $(\bot_{\mathbf{C}}, \mathsf{Null}, \{\})$.

As shown in Definition 2, end denotes $_{\mathsf{drop}}\{\}^{\widetilde{E}}$, *i.e.,* a typestate $u^{\widetilde{E}}$ representing termination. Recall, end is a supertype of a droppable state.

Definition 34. *We extend the $\leq_{\mathbf{T}} \subseteq \mathcal{T} \times \mathcal{T}$ relation (see Definition 9) with the following rules.*

$$\frac{}{\mathsf{Shared} \leq_{\mathbf{T}} \mathsf{Shared}}\ \textsc{Shared} \qquad \frac{t \leq_{\mathbf{T}} \mathsf{end}}{t \leq_{\mathbf{T}} \mathsf{Shared}}\ \textsc{Linear} \qquad \frac{}{\mathsf{Null} \leq_{\mathbf{T}} \mathsf{Null}}\ \textsc{Null\_1}$$

It is easy to show that reflexivity (Lemma 3) and transitivity (Lemma 4) still hold. Concerning reflexity, the newly introduced types are subtypes of themselves; concerning transitivity, since the newly introduced types only appears to the right-hand side of the subtyping relation (besides being subtypes of themselves), they do not invalidate transitivity, thus transitivity still holds.

The grammar in Definition 35 defines the types in the JaTyC type system.

Definition 35 (TC type syntax)**.** *TC (Type Checking) types, ranged over by meta-variable tc, are terms generated by the following grammar.*

$$tc \ ::= \ \bot \mid tt \mid b \mid \mathsf{void} \mid L$$

In Definition 35, b is a Java primitive type, *e.g.,* boolean, integer, float, L is the Java type associated to an enum L, void is used to type methods with such declared return type, $\bot$ unreachable code, *i.e.,* code after return statement and tt is a typestate tree. Notice that, in the type checking informal description, we disregard primitive and enum types as they are not relevant in the type checking process. Thus, in Definition 36, we omit the subtyping rule for primitive types.

Let **TCTypes** be the set of TC types produced by rule *tc*.

Definition 36 (Subtyping on TC types)**.** *We extend the relation $\leq_{\mathbf{TC}} \subseteq \mathcal{TT} \times \mathcal{TT}$ to $\boldsymbol{TCTypes}$. Thus, let $\leq_{\mathbf{TC}} \subseteq \boldsymbol{TCTypes} \times \boldsymbol{TCTypes}$ be the subtyping relation*

over TC types defined by the following inductive rules.

$$\frac{}{\bot \leq_{\mathbf{TC}} tc}\ \textsc{BotTC} \qquad\qquad \frac{}{\mathsf{void} \leq_{\mathbf{TC}} \mathsf{void}}\ \textsc{Void}$$

Notice that, the newly introduced rules do not invalidate reflexivity and transitivity properties.

5.6.2 Application to Type Checking

We now present the type checking process JaTyC performs, using a common syntax as a reference point. We initiate our exploration taking into account the analysis of declarations, proceeding to dissect expressions and culminating with a breakdown of statements. Notice that, the triple (C, t, tts) indicates a typestate tree. Throughout our discourse, we utilise the Kleene star to indicate sequences that may be empty.

Class declarations and overriding. First, to ensure that the overall approach is sound, it is crucial to guarantee that protocols are well-formed and the relation with the corresponding class makes sense. For instance, one has to ensure that all methods mentioned in the protocol are declared in the class. Similarly, one has to ensure that all mentioned outputs are return values of the corresponding method signatures. Additionally, we check typestate input contravariance and output covariance in overridden methods since these may include Requires and Ensures annotations in parameters and return types, respectively, which limit the typestates received or returned. Notice that, to perform such an additional check on the expected parameter/return TC type of the overridden method, we check that the TC type returned by such method is a subtype of the one in the super class (according to Definitions 31 and 36). Concerning the TC types of the parameters in the overridden method, if they are linear objects, we require the Java class to be the same as the one specified in the superclass and the type specified in the Requires annotation to be a supertype of the corresponding one in the superclass. In case such annotations are absent, it implies that the method expects/returns an aliased reference. Finally, we need to make sure that the subclass protocol is a

subtype of the superclass protocol (*i.e.,* the initial state of the former is a subtype of the initial state of the latter according to Definition 5). Thanks to these checks, dynamic dispatch works in a transparent manner: we are sure that if a method is callable on a supertype, it is also callable on the subtype.

To type check a class, we start from the initial typestate of the protocol using the initial values of the fields as starting type information. Then, to analyse the other typestates, we follow the approach of Bravetti *et al.* [BFG+20]. In particular, we first retrieve the field type information stemming from the analyses of the stateful methods that (according to the protocol) are encountered along the path to reach the typestate containing the method under examination. Then, we use such information as the starting point to type check such method declaration. Technically, type information is stored in a map from locations, *e.g.,* fields of the this object, to typestate trees (Definition 18). Notice that, we store the typestate tree of expressions in this map since these may evaluate to typestated-objects, which must be tracked. Moreover, type information associated with the class final states are checked to ensure all fields either correspond to a terminated protocol or are aliased (explained later), to ensure *protocol completion* of references in fields.

Method declarations: `@Ensures(s) C m((@Requires(s) C' x)*) {st}`. To examinine a method, we construct a control flow graph [All70] using the Checker Framework [PAJ+08], outlining its execution pathway. Subsequently, we navigate this graph, visiting each expression or statement, while propagating type information. For every expression encountered, we utilise the type information derived from the analysis of the preceding expression, generating updated information that characterises the state of locations after the evaluation of that expression. Details regarding expression and statement analyses will be provided later.

The initial type information, serving as the starting point for graph traversal, consists of the types specified by the parameters, denoted with the Requires annotation. This information is augmented with details regarding fields, which are gleaned from preceding method analyses, as previously discussed. In cases where the Requires annotation is absent, it implies the expectation of a typestate tree subtype of $(C', \mathsf{Shared}, \{\})$.

Return statements undergo analysis akin to assignments, with a critical focus

on ensuring that the type of the returned expression is a subtype of the one declared via the Ensures annotation (checking classes and types subtyping). In methods where no annotation is provided, we return an aliased reference. Upon reaching the end of a method body, we enforce a condition where all variables and parameters must either be aliased or in a final state, thus ensuring *protocol completion*.

Variable declarations with or without initialisation: `C x [= exp].` To handle variable (or field) declarations, we assign to the left-hand side a single node typestate tree whose TC type is $(C, \mathsf{Null}, \{\})$. Whenever variable (or field) initialisation occurs, we distinguish among the following cases. If exp is evaluated to $(\bot_{\mathbf{C}}, \mathsf{Null}, \{\})$ (*i.e.,* the typestate tree assigned to null), we do not change the typestate tree assigned to variable x. If C is a class without protocol and exp is evaluated to (C', t, tts): (i) we ensure that $C' \leq C$; (ii) we check that t is a subtype of Shared; and (iii) we assign $(C, \mathsf{Shared}, \{\})$ to variable x. Finally, if C is a class with protocol and exp is evaluated to (C', t, tts): (i) we call ucastTT on (C', t, tts), using C as target class; and (ii) we assign the result of ucastTT to x. In all other cases, we report an error. Assignments may produce aliasing among variables. Since an object state could be modified via multiple aliases, we restrict aliasing to allow us to statically track object states. We enforce a linear discipline: only one variable is "active", while the others are marked as aliased (and cannot call protocol methods). We also mark the right-hand side expression as aliased when checking a variable declaration or assignment, *i.e.,* we assign to exp a root typestate tree with Shared as type.

Assignments: `x = exp.` To check an assignment, we first scrutinise the typestate tree of the left-hand side: if the TC type assigned to the root is (C, t, tts) (with C being the Java type of x), then it must hold $t \leq_{\mathbf{T}} \mathsf{Shared}$, otherwise we report an error. Then we proceed as described for variable intialisation. Again, to enfornce a linear disicpline, we mark the right-hand side expression as aliased.

Method call expressions: `exp.m(exp*).` When examining a method call, our first step is to verify that the receiver expression is not null. This check is possible because of the Null type assigned to objects storing the null value. Subsequently, we

scrutinise each parameter assignment, applying the same rules elucidated earlier. This ensures that calls such as obj.m(x,x) do not inadvertently create unintended aliases. Furthermore, we ascertain that the root of the typestate trees associated with the parameter expressions exp are subtypes of the expected ones in the method signature (specified in the Requires annotation), via Definitions 31 and 36. Recall, if no Requires annotations are provided for parameters, it implies the expectation of an aliased reference and, consequently, we just check that the typestate tree of exp is a subtype of $(C, \mathsf{Shared}, \{\})$, where C is the Java class specified in the method signature for a given parameter. Once these checks are completed, we proceed to validate the call itself. We ensure that the receiver expression is a non-aliased reference and utilise the evoTT function to compute the typestate tree associated with the receiver after the call. This computation involves passing the current typestate tree, the method name and any potentially returned output (applicable if the method call appears within an if or switch statement). Notice that, evoTT may be invoked multiple times to consider all possible outputs. If the result of evoTT yields a typestate tree with $\top_{\mathbf{t}}$ as root type, meaning that the method we are calling is unavailable in the current type t, we promptly report an error. If the method signature presents the Ensures annotation, the method call returns an object associated to a typestate tree composed by a single node, *i.e.,* the one defined in the annotation. If no annotation is provided, the typestate tree is $(C', \mathsf{Shared}, \{\})$ (wich C' being the Java type specified in the method signature for the returned value).

Cast expressions: `(C) exp`**.** When checking a cast, we know that the inner expression was already checked, similarly to what happens to other expressions. To check cast expressions, we must use either ucastTT or dcastTT, passing the inner expression typestate tree and the target class. We test if we are upcasting or downcasting comparing the inner expression static class with the target one. The result is associated with the cast expression and the inner one is marked as aliased.

One key detail about cast expressions is that if a cast expression is the receiver object of a method call, after checking the call, the new type of the receiver object is associated with the innermost expression, *i.e.,* the type resulting from evoTT,

not with the cast expression itself. For example, if the receiver is (A) ((B) x), the new type information is associated with x directly, not with (A) ((B) x), so that x can be used again later (instead of being aliased). This will require an upcast to the class of x, but no information is lost, thanks to typestate trees.

New expression: `new C(exp*).` The initialisation of a new object is analysed similarly to a method call (since we are calling the constructor), except that it returns a new object. So, we associate the expression with a typestate tree with only a root: the class is the object type we are constructing, and the type is the initial typestate of the protocol or $(C, \mathsf{Shared}, \{\})$, if the object class has no protocol. The parameters are analysed as explained in the method call analysis.

If statements: `if ( exp ) { st } else { st' }.` For simplicity, up until now we omitted an implementation detail crucial to type check if and switch statements: during the control flow graph traversal, we do not simply propagate a map from locations to typestate trees, but we also keep track of type information depending on the values of other expressions. For instance, to analyse the condition of an if statement, if it is a method call, we track the type information holding when the call evaluates to true, separately from the one that holds when it evaluates to false. Given this, to check an if statement, we just need to propagate the former to the first branch, and the latter to the second branch. We also make sure to invalidate such "conditional" type information once it is no longer relevant. Finally, the typestate trees associated with each location after the if statement are the result of merging type information from both branches, using mrgTT (Definition 28).

Switch statements: `switch ( exp ) { (case val : st)* }.` We approach the analysis of a switch statement much like we do with an if statement. However, a method call within the expression of a switch statement yields varying type information for each case. Instead of boolean values, we consider enum values that may be returned. Therefore, when examining a switch statement, our focus is on propagating the relevant information corresponding to each matched case to its associated branch. Similar to our handling of if statements, we promptly invalidate this "conditional" type information when it ceases to be applicable.

While statements: `while ( exp ) { st }`. Concerning the analysis of the while statement, we follow a similar approach to analysing if one, albeit with a different flow graph structure: upon executing the body, control returns to the condition. Consequently, it is possible to traverse the same expression or statement in the graph multiple times. In such cases, we check that the type information at the end of the loop body is a subtype (according to Definition 36) of the one before the while statement. Outside the loop body, we keep analysing the code using as type information the one stemming from the analysis of the condition, considering only the type information related to the false case.

5.7 Extending JaTyC Language: Linear Arrays

In the context of this Dissertation, we enlarge the supported JaTyC language with a new syntactical construct, providing a preliminary support for linear arrays, *i.e.,* arrays of objects with protocol attached. Since arrays are covariant, *i.e.,* an array of a subclass can be passed where an array of a superclass is expected, runtime exceptions can occur. Thus, we prohibit passing arrays as parameters.

The support for arrays of linear objects entails first the extension of the JaTyC type system with a proper type storing a sequence of typestate trees (*i.e.,* those related to the objects stored inside the array):

$$\langle tt_0, tt_1, \ldots, tt_{n-1} \rangle.$$

In this context, $\langle tt_0, tt_1, \ldots, tt_{n-1} \rangle$ represents an ordered sequence that can range over indexes $0, 1, \ldots, n-1$. Notice that, such sequence can also be emtpy, *i.e.,* in case of array declaration without initialisation.

We now present the type checking process JaTyC performs accounting for the newly introduced syntactical construct, *i.e.,* arrays of linear objects.

Array declarations with or without initialisation: `C[] x [= exp]`. To handle array declarations, we assign to the left-hand side the TC type $\langle\ \epsilon\ \rangle$ (*i.e.,* the empty array). Whenever array initialisation occurs, we check that the type of exp is $\langle tt_0, tt_1, \ldots, tt_{n-1} \rangle$. Notice that, the type of exp can also be an empty array. For

each tt_i, we call the ucastTT function on tt_i, using C as the target class, obtaining tt'_i. Finally, we compose each tt'_i into $\langle tt'_0, tt'_1, \ldots, tt'_{n-1} \rangle$ and assign it to x. To enforce linearity, we mark all the elements of the right hand side linear arrays as aliased.

New array with dimension expressions: `new C[n].` The initialisation of a new array is analysed similarly to a method call, except that it returns a new array with the specified length. Therefore, we associate the expression with the TC type $\langle tt_0, tt_1, \ldots, tt_{n-1} \rangle$, where each tt_i is $(C, \mathsf{Null}, \{\})$.

New array with values expressions: `{exp*}`. To handle the creation of a new array with values, we take the typestate tree tt_i assigned to each $\mathsf{exp_i}$ and compose them in $\langle tt_0, tt_1, \ldots, tt_{n-1} \rangle$, *i.e.,* the type assigned to the new array with value expression. Again, to enforce linearity, we mark all expressions $\mathsf{exp_i}$ as aliased.

Array element access expressions: `x[i].` The access to an array element is trivially handled: in the type checking process we just ensure that we can actually access the element, *i.e.,* we do not go over array boundaries. Subsequently, we distinguish among three cases: (*i*) call a method on an array object; (*ii*) assign an expression to the array element; and (*iii*) assign an array element. In the first case, we proceed as described in method call expressions; in the second and third case we proceed as outlined in variable declarations/assignments. In all cases, as final step, we propagate the type information of the element we are accessing to the array storing such element.

5.8 Use Cases

To demonstrate the practicality and versatility of our methodology, we commence this Section providing a comprehensive breakdown of the type checking process applied to the code snippet in Listing 5.15. Subsequently, we offer a diverse array of examples featuring polymorphic code, accessible via GitHub[11], drawing inspiration from cyber-physical system world. Through these examples, we illustrate two

[11] `https://github.com/jdmota/java-typestate-checker/tree/master/examples`

key points: (*i*) our tool is adept at identifying errors that evade detection by the standard Java type checker; (*ii*) the adaptability and expressiveness of our framework enable the modelling of complex and realistic scenarios.

Type checking Listing 5.15. To type check the ClientCode class (which has no protocol), we check both static methods, example and setSpeed, independently. This is so because static methods are never part of a class protocol. The list of steps taken by the type checker to check the example method is the following:

- check the expression new SUV(), associating it with a leaf typestate tree with class SUV and type OFF (*i.e.,* $(SUV, \mathsf{OFF}, \{\})$);
- check the assignment, associating the previous typestate tree with the variable suv and marking the expression on the right as aliased;
- check the call suv.turnOn(), allowed in type OFF, generating "conditional" type information: if true, suv has typestate tree $(SUV, \mathsf{COMF\_ON}, \{\})$, otherwise it has $(SUV, \mathsf{OFF}, \{\})$;
- check the negating expression which "inverts" the conditional information;
- inside the body of the while statement suv is associated with $(SUV, \mathsf{OFF}, \{\})$ and after exiting the while, suv is associated with $(SUV, \mathsf{COMF\_ON}, \{\})$;
- check the call suv.switchMode(), which is allowed in type COMF_ON, generating "conditional" type information: the variable suv has the typestate tree $(SUV, \mathsf{SPORT\_ON}, \{\})$ if the call returns Mode.SPORT and it returns Mode.COMFORT, suv has $(SUV, \mathsf{COMF\_ON}, \{\})$. Since the returned value is not checked in a switch statement, we combine both typestate trees into $(SUV, \mathsf{SPORT\_ON} \cup \mathsf{COMF\_ON}, \{\})$;
- check the *parameter assignment* of suv by upcasting from SUV to Car, generating the typestate tree $(Car, \mathsf{ON}, \{(SUV, \mathsf{SPORT\_ON} \cup \mathsf{COMF\_ON}, \{\})\})$. Since the root type ON is a subtype of the one in the Requires annotation, the *parameter assignment* is allowed. Additionally, variable suv is marked as aliased: the setSpeed method is now the one responsible to complete the protocol of the given instance;

- no further checks are necessary for the call expression on setSpeed since it is a static method and methods are checked in a modular way;
- type checking the example method finishes by checking protocol completion. Since all locations are marked as aliased at the end, no error about completion is reported.

To finish the type checking process, we analyse setSpeed. The list of steps taken follows:

- associate car with typestate tree $(Car, \mathsf{ON}, \{\})$, according to the Requires annotation;
- downcast from class *Car* to *SUV*, which results in the following typestate tree, $(SUV, \mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}, \{\})$;
- check the method call ((SUV) car).switchMode(), which is allowed in type $\mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}$, generating "conditional" type information: (SUV) car has the typestate tree $(SUV, \mathsf{SPORT\_ON}, \{\})$, if the method call returns Mode.SPORT and if it returns Mode.COMFORT, the expression (SUV) car has $(SUV, \mathsf{COMF\_ON}, \{\})$;
- to make car usable again, upcast it to *Car*, associating the typestate tree $(Car, \mathsf{ON}, (SUV, \mathsf{SPORT\_ON}, \{\}))$ if the call returned Mode.SPORT; otherwise $(Car, \mathsf{ON}, (SUV, \mathsf{COMF\_ON}, \{\}))$ if the call returned Mode.COMFORT;
- check the if statement propagating the type information corresponding to each branch;
- in the body of the if statement, downcast (again) from *Car* to *SUV*, resulting in the typestate tree $(SUV, \mathsf{SPORT\_ON}, \{\})$;
- check the method call ((SUV) car).setFourWheels(true), which is allowed in type SPORT_ON, in a similar fashion as before, associating car with $(Car, \mathsf{ON}, (SUV, \mathsf{SPORT\_ON}, \{\}))$;
- merge type information from both branches, resulting in car being associated with typestate tree $(Car, \mathsf{ON}, (SUV, \mathsf{SPORT\_ON} \cup \mathsf{COMF\_ON}, \{\}))$;

- check the call car.setSpeed(50), which is allowed in type ON, leading to ON;
- check the call car.turnOff(), which is allowed in type ON, leading to OFF;
- finish by checking protocol completion. Since all locations are marked as aliased or are in a final state (car is in the droppable typestate OFF), no completion error is reported.

Type checking Listing 5.14. The list of steps taken by JaTyC to check the example method is the following:

- check the if statement by propagating the type information corresponding to each branch;
- in the if branch, perform an upcast and assign the type SUV to variable c, associating it with the typestate tree $(Car, \mathsf{OFF}, \{(SUV, \mathsf{OFF}, \{\})\})$;
- in the else branch, perform an upcast assigning the type $Ecar$ to variable c, associating it with the typestate tree $(Car, \mathsf{OFF}, \{(ECar, \mathsf{OFF}, \{\})\})$ (supposing that the protocol of class $ECar$ has type OFF as the initial one);
- merge type information from both branches, resulting in the following typestate tree, $(Car, \mathsf{OFF}, \{(SUV, \mathsf{OFF}, \{\}), (ECar, \mathsf{OFF}, \{\})\})$;
- check the call c.turnOn(), allowed in the type of each node of the associated typestate tree, generating "conditional" type information: if it returns true, c has typestate tree $(Car, \mathsf{ON}, \{(SUV, \mathsf{COMF\_ON}, \{\}), (ECar, \mathsf{E\_ON}, \{\})\})$ (assuming that E_ON is the resulting type, in the protocol of $ECar$, in case the turnOn method call successfully ends); otherwise the variable c is associated to $(Car, \mathsf{OFF}, \{(SUV, \mathsf{OFF}, \{\}), (ECar, \mathsf{OFF}, \{\})\})$;
- in the if branch, downcast from Car to SUV, resulting in the typestate tree $(SUV, \mathsf{SPORT\_ON}, \{\})$;
- check the call s.setEcoDrive(true), which is allowed in type COMF_ON, leading to COMF_ON;

- assign s of type SUV to c, performing an upcast, resulting in the following typestate tree $(Car, \mathsf{ON}, \{(SUV, \mathsf{COMF\_ON}, \{\})\})$;
- check that the call car.turnOff() is allowed in the type of each nodes; such call leads to the typestate tree $(Car, \mathsf{OFF}, \{(SUV, \mathsf{OFF}, \{\})\})$, if c is an instance of SUV; $(Car, \mathsf{OFF}, \{\})$ otherwise;
- type checking the example method finishes by checking protocol completion. Since all locations are marked as aliased at the end, no error about completion is reported.

By now, the reader that has thoroughly understood the concept of typestate trees might realise that even if a typestate tree forgets all nodes except the root, the type system would still be sound. Thus, the reader could now be puzzled about the importance and the need for typestate trees, wondering if it is worth such an heavy machinery to perform castings in the middle of a protocol. To clarify that, let us consider the example in Listing 5.14 type checked above. Notice that, if we just consider the root node the above program, even if, as shown, is correct, would not compile. The reason is the following. If we just consider the root node, after the if statement, the type of c is OFF (according to the protocol in Listing 5.11). If the turnOn call successfully ends, the type becomes ON. After downcasting to SUV the new type of c is the union type $\mathsf{COMF\_ON} \cup \mathsf{SPORT\_ON}$ (even if we know it actually is COMF_ON). In such type, the method calls allowed are only those that are common to both typestates, *i.e.,* setSpeed. Thus, even if the above program is correct, it does not type check, since setEcoDrive is only allowed in the typestate COMF_ON. In conclusion, just considering the root node of a typestate tree is indeed a sound approach, but it significantly reduces the set of correct programs we are able to type check.

Type checking Listing 5.18: linear arrays. To type check the ClientCode class, we check both static methods, example and setSpeed, independently. Notice that, since the setSpeed is the same as in Listing 5.15, we avoid repeating its static checking procedure. The list of steps taken by the type checker to analyse the example method is the following:

Listing 5.18: Type checking a linear array

```
1  public static void example() {
2    int x = 5;
3    SUV[] suvs = new SUV[x];
4    for (int i = 0; i < x; i++) {
5      suvs[i] = new SUV();
6      while (!suvs[i].turnOn()) { System.out.println("turning on..."); }
7      suvs[i].switchMode();
8      setSpeed(suvs[i]);
9    }
10 }
```

- check the expression x = 5, associating it with an integer type, holding its numerical value;
- check the expression new SUV[x], associating it with a leaf typestate tree with class SUV[] and type $\langle \mathtt{tt}_0, \mathtt{tt}_1, \ldots, \mathtt{tt}_4 \rangle$, where each tt_i is $(SUV, \mathsf{Null}, \{\})$;
- check the assignment, associating the previous typestate tree with the variable suvs, and marking the expression on the right as aliased;
- check the expression i = 0, within the for loop, associating it with an integer type, holding its numerical value;
- check the expression i < 5, within the for loop: if it holds we traverse the loop body;
- check the loop body as done in the Listing 5.15. Since we are dealing with arrays, here we additionally check that the array element access is legal, *i.e.,* the type of the variable used as index holds a value greater than or equal to 0 and lower than the array length.
- check the expression i++, which is implicitly converted in i = i + 1, associating it with an integer type, holding the updated numerical value;
- check the expression i < 5 against the newly updated numerical value of i.

Examples suite. The most significant examples within our suite are succinctly outlined in Table 5.1: the **Directory** column denotes the sub-directory housing each example; **Features** encapsulates the primary aspects emphasised by each

example; **Type checks** signals whether our tool accepts or rejects the example; and lastly, the **Runtime** column delineates any runtime errors manifested by the example when executed. Notice that, all the examples are correct with respect to the Java standard type checker, *i.e.,* they are complied without errors.

Name	**Directory**	**Features**	**Type checks**	**Runtime**
Removable Iterator (1)	removable-iterator	Polymorphic safe code	Y	Ok
Removable Iterator (2)	removable-iterator2	Wrong method call order	N	Index Out Of Bounds
Alarms	alarm-example	Polymorphic safe code	Y	Ok
Cars (1)	car-example	Polymorphic safe code	Y	Ok
Cars (2)	car-example2	Wrong method call order	N	Null pointer exception
Drones (1)	drone-example	Typestated data structure Simple objects interaction	Y	Ok
Drones (2)	drone-example2	Typestated data structure Complex objects interaction	Y	Ok
Drones (3)	drone-example3	Typestated data structure Complex objects interaction Incorrect test for null value	N	Null pointer exception
Robots (1)	robot-example	Typestated data structure Simple objects interaction	Y	Ok
Robots (2)	robot-example2	Wrong typestate upcast	N	Null pointer exception

Table 5.1: Summary of examples

In *Removable Iterator (1)*, *Alarms*, and *Cars (1)*, the examples serve to assess the behaviour of our approach with polymorphic code. As anticipated, the code compiles seamlessly, without encountering any errors.

In *Drones (1)* and *Robots (1)*, the examples advance in complexity introducing a typestate-endowed data structure, enhancing the degree of flexibility. Here, the quantity of linear objects, namely Drones and Robots, is arbitrary. The primary feature highlighted is the interaction among these linear objects: upon utilisation, each object must be extracted from the data structure and reinserted once its task is completed. In *Drones (2)*, the interaction between the data structure and the objects becomes more intricate. We do not wait for an object to finish its task, but, instead, we immediately place it back in the data structure and

proceed to the next one, simulating parallel task execution. In *Drones (3)*, the example is a modification of the previous one, with the key difference lying in an incorrectly negated test for null within the return expression of an instance method. This oversight leads to a null pointer error in subsequent calls. Our tool adeptly propagates type information regarding the order in which methods must be called, thereby statically identifying and capturing this issue. In *Removable Iterator (2)* and *Cars (2)*, we show two different problematic scenarios: index out of bounds and null pointer exceptions, respectively. The former is caused by repeatedly retrieving the next element, without checking whether there are remaining elements or not. The latter is caused due to a field usage before initialisation. We are able to statically catch both cases, since they are caused by an illegal order of method calls. Finally, in *Robots (2)*, we show another null pointer exception. Here, the exception is caused by a field being assigned to null in the subclass and used in the superclass, after performing an upcast. Thanks to our machinery, we are able to detect that, after assigning the field to null, the object is in a typestate with no supertypes, thus we raise an error preventing such runtime problems.

5.9 Related work

This Chapter presents two main contributes: the extension of the language supported by JaTyC and the formalisation of its type system and the development of a novel theory to safely perform behavioural up/down casts at any points in protocols together with its implementation in our typestate-based checker for Java. While the language extension follows mainstream approaches, our novel theory on behavioural casts at any points in protocols significantly advances the state of the art on typestate-based analyses. Thus, in this Section, we review the literature in the context of typestate-based programming and session types subtyping.

Fugue [DF04] provides a method for verifying typestates, conceptualised as predicates over fields, annotating methods with pre- and post-conditions and then checking invariants. It adeptly handles casting and subtyping, allowing subclasses to introduce additional typestates relative to their superclasses. Similar to our approach, if an object ends up in a state unknown to its supertype, Fugue prohibits upcasting. To manage inheritance, Fugue introduces the concept of *frame types-*

tates, where each frame represents a set of fields declared in a specific class, and an *object typestate* is comprised of these frames. In contrast, our approach defines protocols globally with automata (such as those depicted in Listing 5.11), rather than relying on method contracts, which we consider more intuitive. Moreover, instead of employing frames, we treat each class as a whole. This simplification is viable because we interpret typestates as sequences of method calls, rather than predicates or invariants over fields. This streamlined perspective facilitates handling overriding and dynamic dispatch seamlessly.

Plural [BA07] performs static checks to ensure that clients adhere to usage protocols based on typestates. It builds upon earlier work [BA05] that tackles the challenge of subtype substitutability while ensuring *behavioural subtyping* in an object-oriented language. Plural supports subtyping requiring the programmer to explicitly specify which typestates "refine" (*i.e.,* are substates of) others in the superclass. In contrast, our approach eliminates the need for explicitly defining subtyping relations. We define protocols in terms of state machines and automatically identify all subtyping pairs, making the process easier for developers.

Obsidian [COE+20] is a programming language for smart contracts that provides strong compile-time features to prevent bugs. It is based on a type system that uses typestate to statically ensure that objects are manipulated correctly according to their current states. While Obsidian supports parametric polymorphism, it forbids casting to maintain robust static guarantees.

Kellogg *et al.* [KSSE22] define a subset of typestates termed *accumulation typestate specifications*, which can be verified without requiring aliasing information. Within this subset, enabled methods are incapable of being disabled. It is worth noting that the formal language introduced in their work does not incorporate modelling of inheritance.

The Shelley framework [dFCM23] implements typestates within a programming language, specifically Python, with a focus on model checking pertaining to call ordering constraints. Notably, their approach does not consider aspects such as inheritance, subtyping or polymorphism.

The exploration of session types subtyping for synchronous communication was initially pioneered by Gay and Hole [GH05]. Building upon this foundation, Lange and Yoshida [LY16] devised two algorithms to determine if a session type serves as

a subtype of another, leveraging the work of Gay and Hole [GH05] and Kozen *et al.* [KPS93]. Bacchiani *et al.* [BBLZ21] further contributed by developing a tool to generate simulation graphs for synchronous session subtyping, drawing inspiration from the advancements made by Lange and Yoshida [LY16].

Gay *et al.* [GVR+10], build upon prior research on session types for object-oriented languages, introduce a protocol, represented as a session type, attached to a class definition. Their work unifies communication channels with their session types, distributed object-oriented programming and introduces a form of typestates supporting non-uniform objects. While their formal language incorporates a subtyping relation on session types [GH05], it excludes class inheritance, as subtyping is primarily applied to channel communication. This approach is manifested in two implementations: Papaya [JRD21] and Mungo [KDPG16]. Papaya, inspired by the protocols introduced by Gay *et al.* [GVR+10], employs Scala as the target language. However, like its predecessor, it does not handle inheritance. On the other hand, Mungo, also influenced by the protocols outlined by Gay *et al.* [GVR+10], uses Java (as we do) as the target object-oriented language. Similarly, inheritance is not supported in Mungo, except for classes lacking protocols, thus no static guarantees are provided in polymorphic code.

The work done by Bravetti *et al.* [BFG+20] introduces a type system specifically tailored for a Java-like language, wherein objects are annotated with usages and typestate-like specifications delineating the permissible sequences of method calls. Through type-based analysis, their framework ensures adherence to protocols, completeness, and memory safety, by preventing null pointer dereferencing. Notably, their approach does not support subtyping, hence precluding protocol inheritance and behaivoural casting, as well as their standard Java counterparts.

Bouma *et al.* [BdGJ23] introduce a tool designed to streamline the process of generating Java classes that model APIs from a global type, representing the behaviour of processes within a multiparty session typing framework [HYC16]. In their approach, the state is represented by a `state` field and transitions are encoded using methods annotated with preconditions and post-conditions. Verification of client APIs is facilitated by the programmer, who annotates Java code with logical formulas, all of which are statically checked by VerCors [BH14]. In contrast, our approach differs significantly. Rather than dispersing annotations throughout

the codebase to specify or use protocols, we simply associate them directly with classes. The type system then ensures memory safety, protocol compliance, and completion—properties that developers would typically need to specify for each program individually. This streamlined approach reduces the burden on developers and promotes cleaner, more maintainable code.

5.10 Discussion

In this Chapter, we presented an extension of JaTyC that enables behavioural casting at any juncture within protocols, accompanied by its thorough theoretical foundation. Our approach, as shown in the study of Mastrangelo *et al.* [MHN19], addresses a significant challenge hindering the widespread adoption of typestates in the static analyses of object-oriented programs: the restricted flexibility in performing cast operations at any point of a protocol. By introducing a novel theory based on typestate trees, we overcome this obstacle. Our theory is equipped with a comprehensive set of functions designed to manage the abstraction of typestate trees. We establish the soundness of typestate trees through formal mechanisation in the Coq proof system. We assert that typestate trees possess broad applicability across various program analysers for object-oriented languages with inheritance. This language-agnostic nature opens doors for the acceptance of programs and features that were previously deemed unfeasible in such contexts. To support this assertion, we developed a type checker for Java and evaluate the expressiveness of our approach. The significance of our theory and its practical applications is demonstrated through the typestate-based type checking of realistic Java code within an automotive system equipped with driving dynamics control. This system facilitates the customisation of drive modes for SUVs, showcasing the versatility of our approach in real-world scenarios. To the best of our knowledge, existing research has not addressed the issue of castings in the middle of protocols. Additionally, the notion of droppable states, which indicate points in the protocol where it can safely terminate, introduces another crucial concept. Therefore, our work represents a significant advancement in the state of the art, offering thorough support for inheritance and casting in object-oriented languages. This advancement is achieved leveraging the principles of behavioural types [ABB$^+$16, HLV$^+$16],

enhancing the expressiveness and practicality of our approach.

In addition, we extended the language JaTyC support, by introducing a new syntactical construct to support linear arrays, *i.e.,* arrays of objects with protocol. Such an extension requires to carefully reason on the impact it has on the existing ecosystem. Thus, we first formalise the JaTyC type system and the subtyping relations among the existing types JaTyC defines. Then, we introduce the new type to handle linear arrays in the type system and we extend the type checking process, in order to account for linear arrays.

In future endeavors, we intend to formally establish the runtime soundness of typestate trees. This involves devising a core object calculus incorporating inheritance, static typestate semantics and dynamic operational semantics. We aim to mechanise a type safety result, ensuring that well-typed programs at runtime adhere to object protocols in terms of method call order and completion, while also avoiding null pointer exceptions. Additionally, we plan to explore how these concepts can be adapted to accommodate settings with multi-inheritance and generics, further expanding the scope of our approach.

Chapter 6

A Formal Specification of the Java Type Checker

In Chapter 5, we introduced a theoretical framework that advances the current state of the art in typestate-based analyses, fully supporting behavioural casting at any point of the protocol, hence polymorphism. Polymorphism, being a key feature in modern programming, enables more flexible and reusable code. By integrating it into typestate-based type checking analyses, we aim to enhance their precision and applicability. To evaluate the effectiveness of our approach, we implemented this theoretical foundation within JaTyC, a typestate-based type checker for Java. However, the type checking procedure outlined in Section 5.6 has only been described informally, lacking the rigor necessary for a comprehensive understanding of its functioning. Therefore, in this Chapter, we provide a detailed and precise perspective of the type checking process. To this aim, we first design a subset of the Java language, drawing inspiration from the work of Bravetti *et al.* [BFG+20]: we include key features, *e.g.,* primitive types, classes, inheritance hierarchies, instance variables. *In the context of this Dissertation*, we further extend the language of Bravetti *et al.* [BFG+20] with additional elements including, *e.g.,* variable declarations. Moreover, in addition to these standard Java elements, our approach extends the work of Bravetti *et al.* [BFG+20] incorporating aliasing, method signatures in protocols, droppable states (as defined in Definition 2), protocol inheritance, polymorphism and typestate tree, enhancing both the ex-

pressiveness and applicability of our type system. The integration of these new elements in our extended language significantly enlarges the scope of programs a user can define within the context of our core language.

Upon our core language, we formally define the type checking procedure enacted by JaTyC. In particular, for each syntactical construct (*e.g.,* statements, expressions), we define the corresponding set of rules required to properly handle its type checking process. These rules serve as a framework to ensure that each component of the language adheres to the expected type constraints. By grounding the type checking procedure within a rigorous formal specification, we provide a comprehensive and unambiguous representation of the internal operations of JaTyC. This formalisation, not only clarifies the steps involved in type checking, but also enhances the transparency and reliability of the tool, offering insights into its behaviour and functionality at a deeper level.

Wrapping up, the Chapter is structured as follows. In Section 6.1, we formally define the syntax of the core language adopted in this Chapter; in Section 6.2, we introduce the type system as well as the type checking rules supporting and clarifying the procedure described in Section 5.6, while in Section 6.3 we conclude the Chapter.

6.1 Core Language Syntax

The grammar presented in Definition 39, which draws inspiration from the work of Bravetti *et al.* [BFG+20], defines the user syntax for the subset of the Java language explored in this Chapter. This syntax serves as the formal foundation upon which the structure of programs in our language is constructed. By meticulously outlining the syntactic rules, we provide a precise specification of how various language constructs, *e.g.,* expressions and statements, can be combined to form valid programs. These rules, not only ensure clarity and consistency, but also facilitate a deeper understanding of the language and its underlying principles.

In this Chapter, to model typestates effectively, we adhere to the formal framework introduced in Section 5.1. However, we extend this syntax incorporating the entire method Java signature (only considering Java types) into the protocol, rather than relying solely on the method name. This extension is significant as it

enables the inclusion of method overloading in our static analyses. By distinguishing methods, not only by their names, but also by their parameter Java types within the protocols, we enhance the precision and flexibility of our analytical tools. This advancement allows for a more nuanced and accurate representation of program behaviour, particularly in scenarios where method overloading plays a critical role. Consequently, this enriched syntax provides a robust basis for further exploration and analysis of typestates in the context of Java-like languages.

Definition 37 (Typestate syntax)**.**

$$
\begin{array}{rrcl}
\textit{(defining equation)} & E & ::= & s = {}_d\{\widetilde{jms : w}\} \\
\textit{(input state terms)} & u & ::= & {}_d\{\widetilde{jms : w}\} \mid s \\
\textit{(state terms)} & w & ::= & u \mid \widetilde{\langle l : u \rangle} \\
\textit{(droppable)} & d & ::= & \varepsilon \mid \mathsf{drop} \\
\textit{(Java method signature)} & jms & ::= & jt\, m(\widetilde{jt})
\end{array}
$$

where in ${}_d\{\widetilde{jms : w}\}$ *either* $d = \mathsf{drop}$ *or the sequence* $\widetilde{jms : w}$ *is not empty and, if* $w = \widetilde{\langle l : u \rangle}$*, the type of* l *must be compatible with the returned one declared in* jms*.*

Recall, end is an alias for ${}_{\mathsf{drop}}\{\}$, *i.e.,* a droppable typestate $u^{\widetilde{E}}$ without input actions whose set is omitted as it is irrelevant in a terminated state.

We model annotation types by closely adhering to the syntax outlined in Section 5.5.2. As a reminder, union types permit a method call only if it is allowed by every element within the union. This ensures that the method adheres to the constraints imposed by all constituent types, thereby maintaining type safety and consistency. In the user syntax, we intentionally restrict the range of types that can be explicitly defined, currently disregarding intersections and $\top_{\mathbf{t}}$. This limitation is deliberate, as these elements are introduced exclusively during the type-checking phase and cannot be directly specified by the user within the program. Their exclusion from the user syntax simplifies the language for the programmer while still enabling the type system to handle more complex type relationships internally. Notice that, annotation types *at* encapsulate the information of Requires

and Ensures annotations (see Section 5.2). These annotations play a crucial role in specifying pre-conditions and post-conditions for methods.

Definition 38 (Annotation type syntax)**.**

$$\textit{(annotation types)} \qquad at \ ::= \ at \cup at \ \mid \ u^{\widetilde{E}}$$

In our setting, a program $\widetilde{D}$ is a set of *enum*, *i.e.,* enum L $\{\widetilde{id}\}$ (where id is a generic identifier), and *linear class declarations*, *i.e.,* classes with protocols, of the forms class $C\,\{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}$ or class $C\,\{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}$ extends C' in case of class inheritance, where $u^{\widetilde{E}}$ is the class initial typestate, $\widetilde{F}$, $\widetilde{B}$ and $\widetilde{M}$ are the sequence of declared fields, class constructors and defined methods, respectively. We assume programs $\widetilde{D}$ to always include a special class for the null value, denoted by $\perp_{\mathbf{C}}$.

A class constructor B is declared as $C(\widetilde{pt\ id})\{cbst\}$ (possibly including super$(\widetilde{e})$), where C denotes the class owning the constructor and $\widetilde{pt\ id}$ denotes the sequence of its arguments. It is important to note that, $\widetilde{pt\ id}$ is treated as an ordered sequence of elements, with the order being determined by the method signature. This order is crucial for ensuring that method invocations are correctly matched with their parameters during the type checking process. Notice that, a parameter type (and the return one for methods) of the form $C[at]$ models the Requires and Ensures annotations, presented in Chapter 5. These annotations provide a formal means to express the required type, *i.e.,* a typesatete $u^{\widetilde{E}}$ or a union type, defining the behaviour the object must adhere to in the method body and the one a possibly returned object provides after the method completes. This mechanism is central to our analysis, allowing the type checker to enforce correct usage patterns of objects and prevent runtime errors, *e.g.,* invoking a method on an uninitialised object.

A method M is declared as $rt\,m(\widetilde{pt\ id})\{\widetilde{bst}\}$, where $\widetilde{pt\ id}$ the sequence of arguments. Differently from constructors, methods additionally have a return type rt that specifies the data type the method returns, which could be a primitive type (such as int or bool), void or a class with its current type (*i.e.,* $C[at]$).

The body of a constructor/method is defined as a sequence of statements $\widetilde{bst}$ that can include *e.g.,* while (e) st, if (e) st else st, switch (e) $\{\widetilde{cbl}\}$, return and break, variable declarations $jt\ id = e$, expressions e used as statements and compound statements $\{\widetilde{bst}\}$ (*i.e.,* a group of statements treated as a single unit).

Expressions e include a variety of operations such as method calls of the form $eid.m(\widetilde{e}), m(\widetilde{e}), \mathsf{this}.m(\widetilde{e})$ and $\mathsf{super}.m(\widetilde{e})$, where a method m is invoked on a reference id of the current class or its parent, on the this (which can be omitted) or super objects. The sequence of expressions $\widetilde{e}$ represent the arguments passed to the method call. Other forms of expressions include accesses and assignments of/to fields of the current class, fields inherited from superclasses and fields that belong to the objects referenced by the current class fields, *i.e.,* $eid, eid.id, eid = e$ and $eid.id = e$. The syntax also includes object creation expressions $\mathsf{new}\ C(\widetilde{e})$, which instantiate new objects of class C, allowing for dynamic memory allocation and the creation of new instances at runtime. Explicit cast operations $(jt)\ e$ are used to convert an expression e to a different Java type jt, enabling polymorphism and type hierarchies to be navigated. Additionally, we include logical operators (*e.g.,* logical negation, equality and inequality operators) to provide more flexibility: expressions can be also used in composition with the logical negation operator $!\,e$, with $e == e'$ and $e\,!= e'$, useful, *e.g.,* in the condition of if or while statements. Expressions can also assume the form of values v, allowing methods to also operate on primitive values directly. Primitive values in the language include boolean values of type bool, integer values of type int and double values of type double (represented by intLit and doubleLit), which are the basic data types supported by the system. Although only these primitive types are considered, extending the system to handle others is straightforward, as the underlying theoretical framework is general enough to accommodate additional types without significant modifications.

In Java, identifiers and values are not considered as statements. As a matter of fact, following the Java philosophy of adding constraints to improve readability, maintainability and avoid common programming errors, we disregard this aspect.

Definition 39 (User Syntax)**.**

$$
\begin{array}{rrcl}
\textit{(class declaration)} & D & ::= & \mathsf{enum}\ L\ \{\widetilde{id}\} \mid \mathsf{class}\ C\ \{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\} \mid \\
 & & & \mathsf{class}\ C\ \{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}\ \mathsf{extends}\ C' \\
\textit{(field declaration)} & F & ::= & jt\ id \\
\textit{(method declaration)} & M & ::= & rt\, m(\widetilde{pt\ id})\{\widetilde{bst}\} \\
\textit{(class constructor)} & B & ::= & C(\widetilde{pt\ id})\{cbst\}
\end{array}
$$

$$
\begin{array}{rrcl}
\textit{(user value)} & v & ::= & \mathsf{null} \mid \mathsf{intLit} \mid \mathsf{doubleLit} \mid l \\
\textit{(label)} & l & ::= & \mathsf{true} \mid \mathsf{false} \mid L.id \\
\textit{(basic type)} & b & ::= & \mathsf{bool} \mid \mathsf{int} \mid \mathsf{double} \\
\textit{(java type)} & jt & ::= & b \mid C \mid L \\
\textit{(parameter type)} & pt & ::= & b \mid C[at] \mid L \\
\textit{(return type)} & rt & ::= & \mathsf{void} \mid pt \\
\textit{(extended id)} & eid & ::= & id \mid \mathsf{this}.id \mid \mathsf{super}.id \\
\textit{(extended method call)} & emc & ::= & m(\widetilde{e}) \mid \mathsf{this}.m(\widetilde{e}) \mid \mathsf{super}.m(\widetilde{e}) \\
\textit{(expression)} & e & ::= & v \mid emc \mid eid.m(\widetilde{e}) \mid eid \mid eid.id \mid \\
 & & & eid = e \mid eid.id = e \mid \mathsf{new}\ C(\widetilde{e}) \mid \\
 & & & (jt)\ e \mid \mathsf{!}\, e \mid e == e \mid e \mathsf{!=} e \\
\textit{(statement)} & st & ::= & \mathsf{if}\ (e)\ st\ \mathsf{else}\ st \mid \{\widetilde{bst}\} \mid \mathsf{switch}\ (e)\ \{\widetilde{cbl}\} \mid \\
 & & & \mathsf{break} \mid \mathsf{return}\ e \mid \mathsf{while}\,(e)\ st \mid e \\
\textit{(block statement)} & bst & ::= & jt\ id = e \mid st \\
\textit{(case block)} & cbl & ::= & \mathsf{case}\ v : \widetilde{bst} \mid \mathsf{default} : \widetilde{bst} \\
\textit{(constructor body)} & cbst & ::= & \widetilde{bst} \mid \mathsf{super}(\widetilde{e})\ \widetilde{bst}
\end{array}
$$

During type checking, we need to ensure that all the classes considered are defined in the context of a program $\widetilde{D}$. Thus, for convenience, in Definition 40, we define the set of classes and enums defined in a program $\widetilde{D}$.

Definition 40 (Classes and Enums in a program)**.** *We define $\mathcal{C}_{\widetilde{D}}$ and $\mathcal{L}_{\widetilde{D}}$ as the sets of classes and enums, respectively, declared in a program $\widetilde{D}$. Formally,*

$$
\begin{array}{l}
\mathcal{C}_{\widetilde{D}} \stackrel{\text{def}}{=} \{C \mid \mathsf{class}\ C\,\{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\} \in \widetilde{D} \vee \\
\qquad\qquad \mathsf{class}\ C\,\{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}\ \mathsf{extends}\ C' \in \widetilde{D} \vee C = \bot_{\mathbf{C}}\} \\
\mathcal{L}_{\widetilde{D}} \stackrel{\text{def}}{=} \{L \mid \mathsf{enum}\ L\ \{\widetilde{id}\} \in \widetilde{D}\}
\end{array}
$$

According to the previous Chapter (see Section 5.5.3), $\bot_{\mathbf{t}}$ can only be produced after a downcasting a super-typestate with no subtypes in the protocol of the

subclass. Notice that, this scenario can only occur when, in a given state, the supertypestate has more choices than its subtypes (*i.e.,* output covariance). However, given that in Java there is no inheritance among enums, outputs are invariant and $\bot_{\mathbf{t}}$ can never occur. Consequently, for simplicity, in the next Sections, we devise all our functions based on this assumption.

6.2 Type System

In this Section, we provide a detailed exposition of the type system that forms the foundation of our core language, accompanied by the formal rules that govern type assignments within programs. The type system is a critical component of the language, serving as a rigorous framework to ensure that programs are well-structured and comply with the specified semantic constraints. By defining how types are assigned and validated, the type system plays a pivotal role in maintaining the correctness and reliability of programs written in our language. We describe the mechanics of the type system, elaborating on the systematic process through which various language constructs are assigned their corresponding types. This includes expressions, statements, variables and other syntactic elements, each of which must adhere to the type rules to ensure consistency and coherence across the program. The formal rules we present are designed to be both precise and expressive. The defined type checking rules constitute a rigorous and precise representation of JaTyC functioning informally described in Section 5.6. By examining them, we can gain insights into how JaTyC ensures that data types are correctly assigned and verified throughout programs, thereby enhancing its trustworthiness.

In the type checking rules, we adopt the following semantics for subscripts: normal font represents variable elements (*e.g.,* parameters of a relation), while bold font is used to assign different meanings to the same symbol.

Class/enum information. During type checking, we may need to use static information about classes and enums, *e.g.,* protocol, declared fields and methods: in Definitions 41 and 42 we define the functions to retrieve such information.

Definition 41 (Class functions)**.** *Let class* C *be either* $\mathsf{class}\ C\,\{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}$ *or*

$\mathsf{class}\ C\ \{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}\ \mathsf{extends}\ C'$, *we define the following functions.*

$$C.\mathsf{prot}_{\widetilde{D}} \stackrel{\text{def}}{=} u^{\widetilde{E}} \qquad C.\mathsf{fields}_{\widetilde{D}} \stackrel{\text{def}}{=} \widetilde{F}$$
$$C.\mathsf{cons}_{\widetilde{D}} \stackrel{\text{def}}{=} \widetilde{B} \qquad C.\mathsf{meths}_{\widetilde{D}} \stackrel{\text{def}}{=} \widetilde{M}$$
$$C.\mathsf{sup}_{\widetilde{D}} \stackrel{\text{def}}{=} \begin{cases} C' & \textit{if } \mathsf{class}\ C\ \{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}\ \mathsf{extends}\ C' \\ \varepsilon & \textit{otherwise} \end{cases}$$

Definition 42 (Enum values)**.** *Let* L *be* $\mathsf{enum}\ L\ \{\widetilde{id}\}$*, we define the following function.*

$$L.\mathsf{vals}_{\widetilde{D}} \stackrel{\text{def}}{=} \widetilde{id}$$

Class inheritance enables seamless transfer of fields and methods from superclasses and requires careful reasoning: (*i*) fields can be inherited and shadowed (*i.e.,* a subclass defines a field with the same identifier of a field defined in one of its superclasses); and (*ii*) methods can be inherited and overridden (*i.e.,* a subclass re-defines a method defined in one of its superclasses). During type checking, whenever we encounter a class with some superclasses, we need to filter out all the shadowed fields and overridden methods (see Definitions 45 and 46). While for fields shadowing it is enough to compare their identifiers, to detect overridden methods we need to compare Java signatures (without considering return types). To this aim, we introduce in Definitions 43 and 44 the required functions.

Let **RTTypes** be the set of return types ranged over by meta-variable rt; let $\mathcal{JT}$ be the set of Java types ranged over by meta-variable jt; recall, **TCTypes** is the set of TC types ranged over by meta-variable tc.

Definition 43 (Extract Java type)**.** *The function* $\mathsf{toJT} : \boldsymbol{TCTypes} \cup \boldsymbol{RTTypes} \to \mathcal{JT}$ *is defined as follows.*

$$\mathsf{toJT}(tcrt) = \begin{cases} \mathsf{cl}(tt) & \textit{if } tcrt = tt \\ C & \textit{if } tcrt = C[at] \\ tcrt & \textit{otherwise} \end{cases}$$

Whenever we use toJT *on a sequence, we apply it to all its element.*

Let $\mathcal{M}$ be the set of method definitions ranged over by meta-variable M; let $\mathcal{MS}$ be the set of method signatures ranged over by meta-variable ms.

Definition 44 (Method signature)**.** *Given a method definition M, the function* $\mathsf{sig} : \mathcal{M} \to \mathcal{MS}$ *is defined as follows.*

$$\mathsf{sig}(rt\, m(\widetilde{pt\ id})\{\widetilde{bst}\}) = rt\, m(\widetilde{pt})$$

Let **IdNames** be the set of identifiers ranged over by meta-variable id.

Definition 45 (All fields)**.** *Given a program $\widetilde{D}$ and a class $C \in \mathcal{C}_{\widetilde{D}}$, the partial function $C.\mathsf{allF}_{\widetilde{D}} : \boldsymbol{IdNames} \rightharpoonup \mathcal{C}_{\widetilde{D}}$ is defined as follows.*
$C.\mathsf{allF}_{\widetilde{D}} =$

$$\begin{cases} \{id \mapsto C \mid jt\ id \in C.\mathsf{fields}_{\widetilde{D}}\} \cup \mathsf{fs} & \textit{if } \mathsf{class}\ C\ \{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}\ \mathsf{extends}\ C' \in \widetilde{D} \\ \{id \mapsto C \mid jt\ id \in C.\mathsf{fields}_{\widetilde{D}}\} & \textit{otherwise} \end{cases}$$

where

$$\mathsf{fs} = \{id \mapsto C'' \in C'.\mathsf{allF}_{\widetilde{D}} \mid \nexists\, jt\,.\ jt\ id \in C.\mathsf{fields}_{\widetilde{D}}\}$$

Definition 46 (All methods)**.** *Given a program $\widetilde{D}$ and a class $C \in \mathcal{C}_{\widetilde{D}}$, the partial function $C.\mathsf{allM}_{\widetilde{D}} : \mathcal{MS} \rightharpoonup \mathcal{C}_{\widetilde{D}}$ is defined as follows.*
$C.\mathsf{allM}_{\widetilde{D}} =$

$$\begin{cases} \{\mathsf{sig}(M) \mapsto C \mid M \in C.\mathsf{meths}_{\widetilde{D}}\} \cup \mathsf{ms} & \textit{if } \mathsf{class}\ C\ \{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}\ \mathsf{extends}\ C' \in \widetilde{D} \\ \{\mathsf{sig}(M) \mapsto C \mid M \in C.\mathsf{meths}_{\widetilde{D}}\} & \textit{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathsf{ms} = \{&rt\, m(\widetilde{pt}) \mapsto C'' \in C'.\mathsf{allM}_{\widetilde{D}} \mid \\ &\nexists\, rt'\, m(\widetilde{pt'\ id'})\{\widetilde{bst}\} \in C.\mathsf{meths}_{\widetilde{D}}\,.\ \mathsf{toJT}(\widetilde{pt}) = \mathsf{toJT}(\widetilde{pt'})\} \end{aligned}$$

Subtyping relations. During type checking, we crucially use the subtyping relations among Java types and TC types in Definitions 36, 47 and 48. Notice

that, these relations are bound to a specific program $\widetilde{D}$, since some of them may hold in $\widetilde{D}$ and may not in $\widetilde{D'}$.

Let $\mathcal{JT}$ be the set of Java types ranged over by meta-variable jt.

Definition 47 (Subtyping on Java types)**.** *The reflexive and transitive relation* $\leq_{\widetilde{D}} \subseteq \mathcal{JT} \times \mathcal{JT}$ *is inductively defined as follows.*

$$\textsc{SubB}\ \frac{}{\mathsf{int} \leq_{\widetilde{D}} \mathsf{double}} \qquad \textsc{SubC}\ \frac{\mathsf{class}\ C\ \{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}\ \mathsf{extends}\ C' \in \widetilde{D}}{C \leq_{\widetilde{D}} C'}$$

$$\textsc{BotC}\ \frac{C \in \mathcal{C}_{\widetilde{\mathcal{D}}}}{\bot_{\mathbf{C}} \leq_{\widetilde{D}} C} \qquad \textsc{RefCL}\ \frac{jt \in \mathcal{C}_{\widetilde{\mathcal{D}}} \cup \mathcal{L}_{\widetilde{\mathcal{D}}}}{jt \leq_{\widetilde{D}} jt}$$

$$\textsc{RefB}\ \frac{}{b \leq_{\widetilde{D}} b} \qquad \textsc{Trans}\ \frac{jt \leq_{\widetilde{D}} jt' \qquad jt' \leq_{\widetilde{D}} jt''}{jt \leq_{\widetilde{D}} jt''}$$

During type checking, there are scenarios where having a notion of subtyping among sequences of Java types is advantageous, *e.g.,* in case of method overloading. Thus, we extend the subtyping relation presented in Definition 48 to sequences.

Definition 48 (Subtyping on sequences of Java types)**.** *The reflexive and transitive relation* $\leq_{\widetilde{D}} \subseteq \mathcal{JT}^n \times \mathcal{JT}^m$ *is defined as follows.*

$$\textsc{SeqSub}\ \frac{|\widetilde{jt'}| = |\widetilde{jt}| \qquad \forall\, i\,,\, 1 \leq i \leq |\widetilde{jt}|\,.\, jt_i \leq_{\widetilde{D}} jt'_i}{\widetilde{jt} \leq_{\widetilde{D}} \widetilde{jt'}}$$

Upon subtyping among Java types, we build the notion of subtyping for TC types. In the following, we use $\leq_{\mathbf{TC},\widetilde{D}}$ to indicate the subtyping relation defined in Definitions 31 and 36 adopting $\leq_{\widetilde{D}}$ as subtyping relation among Java types. Differently from the previous Chapter where we disregarded basic types and enums as they were irrelevant for the type checking, we now consider them. Consequently, we need to extend the subtyping relation presented in Definitions 31 and 36.

Definition 49 (Subtyping on basic TC types)**.** *We extend the* $\leq_{\mathbf{TC},\widetilde{D}}$ *subtyping*

relation, to account for basic types and enums, as follows.

$$\frac{b \leq_{\widetilde{D}} b'}{b \leq_{\mathbf{TC},\widetilde{D}} b'}\ \textsc{Base} \qquad\qquad \frac{L \in \mathcal{L}_{\widetilde{D}}}{L \leq_{\mathbf{TC},\widetilde{D}} L}\ \textsc{Enum}$$

Reflexivity and transitivity still holds, as new relations are reflexive (*i.e.,* Enum rule) and rely on $\leq_{\widetilde{D}}$, which is reflexive and transitive (*i.e.,* Base rule).

Terminability. In typestate-based analyses, it is crucial to ensure, at the end of a program, protocol completion, *i.e.,* all objects reaching the end state, to avoid dangerous scenarios, *e.g.,* resource leaks. To this aim, we define, in Definitions 50 and 51, a notion of terminable type and TC type.

Definition 50 (Terminable type)**.** *We define the predicate* term *on a type t as the predicate to determine if t is terminable. Formally,*

$$\mathsf{term}(t) \stackrel{\text{def}}{=} t \leq_{\mathbf{T}} \mathsf{Shared} \cup \mathsf{Null}$$

Requiring $t \leq_{\mathbf{T}} \mathsf{Shared} \cup \mathsf{Null}$ implies that either $t \leq_{\mathbf{T}} \mathsf{Shared}$ or $t \leq_{\mathbf{T}} \mathsf{Null}$ (see Union_R in Definition 9). Given that, droppable states are subtypes of Shared, $t \leq_{\mathbf{T}} \mathsf{Shared}$ captures t being a droppable state, Shared or a union/intersection of droppable states and/or Shared; $t \leq_{\mathbf{T}} \mathsf{Null}$ captures t being either Null or a union/intersection of Null only.

Since the building blocks of our type system are TC types, it is convenient to establish a notion of terminable TC type (see Definition 51).

Definition 51 (Terminable TC type)**.** *A type tc is said to be terminable if it is a typestate tree storing a terminable type. Formally,*

$$\mathsf{term}(tc) \stackrel{\text{def}}{=} \exists\, tt \in \mathcal{TT}\ .\ (tc = tt) \wedge \mathsf{term}(\mathsf{ty}(tt))$$

Unresolved types/typestate trees. During type checking, it may happen that a type t can temporarily be in an output state, *e.g.,* after a method call. However, the types presented in Definitions 8 and 32 are not flexible enough and do not

account for such scenarios. Therefore, we need to introduce the notion of *unresolved type* (Definition 52) and *unresolved typestate tree* (Definition 53).

Definition 52 (Unresolved type syntax)**.** *An unresolved type t_{unr} is a type (see Definition 8) possibly containing states of the form $\widetilde{\langle l : t \rangle}$. Formally,*

$$t_{unr} \ ::= \ t_{unr} \cup t_{unr} \ \mid \ t_{unr} \cap t_{unr} \ \mid \ u^{\widetilde{E}} \ \mid \ \widetilde{\langle l : t \rangle} \ \mid \ \top_{\mathbf{t}}$$

Definition 53 (Unresolved typestate tree)**.** *An unresolved typestate tree tt_{unr} is a typestate tree (see Definition 18) storing unresolved types t_{unr}.*

Unresolved typestate trees are crucial to keep track of conditional information stemming from, *e.g.,* a method call. Let us consider the following scenario: in the context of an if statement, whose condition is a method call, we want to keep track of the conditional type information (*i.e.,* type information depending on the value returned by the condition) to correctly propagate it to the corresponding branch. In our example, the unresolved typestate tree, by containing states of the form $\langle \mathsf{true} : t' \ \ \mathsf{false} : t'' \rangle$, makes it possible to use as starting information of the if branch the one labelled with true and the one labelled with false for the else one.

Overloading: static binding. During type checking, we may encounter classes defining multiple constructors and methods with the same name (but different parameter types/number). Thus, we need to deal with the static binding process: whenever a method or constructor call matches multiple candidates, the static binder selects the one whose parameters are subtypes of the others. If no suitable candidate is found, the type checking halts. Concretely, we first look for all candidates that can possibly work with the supplied sequence of parameters (see Definitions 54 and 55) and then we retrieve the method/constructor whose parameters are subtype of all the others (see Definitions 56 and 57).

Let $\mathcal{BS}$ be the set of constructor signatures $C(\widetilde{pt})$, ranged over by meta-variable bs; recall, $\mathcal{MS}$ is the set of method signatures ranged over by meta-variable ms.

Definition 54 (Compatible methods)**.** *Given a program $\widetilde{D}$, a class C, a method name m and a sequence of Java types $\widetilde{jt}$, the function $\mathsf{cands}_{\widetilde{\mathsf{D}}} : \mathcal{C}_{\widetilde{D}} \times \boldsymbol{MNames} \times$*

$\boldsymbol{JTypes}^n \to \mathcal{P}(\mathcal{MS})$ *returns the set of methods compatible with* $\widetilde{jt}$.

$$\mathsf{cands}_{\widetilde{\mathsf{D}}}(C, m, \widetilde{jt}) = \{rt\, m(\widetilde{pt}) \mid rt\, m(\widetilde{pt}) \in \mathrm{dom}(C.\mathsf{allM}_{\widetilde{D}}) \wedge \widetilde{jt} \leq_{\widetilde{D}} \mathsf{toJT}(\widetilde{pt})\}$$

Definition 55 (Compatible constructors)**.** *Given a program* $\widetilde{D}$*, a class* C *and a sequence of Java types* $\widetilde{jt}$*, the function* $\mathsf{cands}_{\widetilde{\mathsf{D}}} : \mathcal{C}_{\widetilde{D}} \times \boldsymbol{JTypes}^n \to \mathcal{P}(\mathcal{BS})$ *returns the set of constructors compatible with* $\widetilde{jt}$.

$$\mathsf{cands}_{\widetilde{\mathsf{D}}}(C, \widetilde{jt}) = \{\, C(\widetilde{pt}) \,|\, C(\widetilde{pt\, id})\{\widetilde{bst}\} \in C.\mathsf{cons}_{\widetilde{D}} \;\wedge\; \widetilde{jt} \leq_{\widetilde{D}} \mathsf{toJT}(\widetilde{pt})\}$$

Definition 56 (Minimum method)**.** *The partial function* $\mathsf{min}_{\widetilde{\mathsf{D}}} : \mathcal{P}(\mathcal{MS}) \rightharpoonup \mathcal{MS}$ *is such that, given program* $\widetilde{D}$ *and a set of method signatures, it returns the minimum one with respect to the subtyping relation among parameter sequences of Java types* $\widetilde{jt}$*, in case it exists. Formally,*
$\mathrm{dom}(\mathsf{min}_{\widetilde{\mathsf{D}}}) =$

$$\{MS \in \mathcal{P}(\mathcal{MS}) \,|\, \exists\, rt\, m(\widetilde{pt}) \in MS\,.\, \forall\, rt'\, m(\widetilde{pt'}) \in MS\,.\, \mathsf{toJT}(\widetilde{pt}) \leq_{\widetilde{D}} \mathsf{toJT}(\widetilde{pt'})\},$$

and for $MS \in \mathrm{dom}(\mathsf{min}_{\widetilde{\mathsf{D}}})$*,* $\mathsf{min}_{\widetilde{\mathsf{D}}}(MS)$ *is the only* $rt\, m(\widetilde{pt}) \in MS$ *such that*

$$\forall\, rt'\, m(\widetilde{pt'}) \in MS\,.\, \mathsf{toJT}(\widetilde{pt}) \leq_{\widetilde{D}} \mathsf{toJT}(\widetilde{pt'})$$

Definition 57 (Minimum constructor)**.** *The partial function* $\mathsf{min}_{\widetilde{\mathsf{D}}} : \mathcal{P}(\mathcal{BS}) \rightharpoonup \mathcal{BS}$ *is defined exactly as the one for methods in Definition 56 with* $\mathcal{BS}/BS$ *replacing* $\mathcal{MS}/MS$*,* C *replacing* m *and omitting the return types* rt/rt'.

6.2.1 Type Environment: Definition and Operators

Definition. The notion of type environment (see Definition 58) is pivotal in our type checking procedure, since we base all our reasoning on the information it stores. Intuitively, a type environment is a mapping from identifiers to TC types. In our type checking rules, we use several type environments specifically crafted for different purposes, *e.g.,* keeping track of field types. As already anticipated, during type checking, unresolved typestate trees may occur. Thus, in the definition of the type environment, we need to account for that, *i.e.,* including the possibility of

having unresolved typestate trees in the co-domain of the type environments.

Let $\mathcal{T}_{unr}$ be the set of unresolved types ranged over by meta-variable t_{unr}; let $\mathcal{TT}_{unr}$ be the set of unresolved typestate trees ranged over by meta-variable tt_{unr}; let **CIdNames** be the set of complex identifiers ranged over by meta-variable *cid*, where

$$cid \ ::= \ C.id \ \mid \ id$$

Notice that, **IdNames** $\subseteq$ **CIdNames**, thus all identifiers of the form *id* are included in **CIdNames**.

Definition 58 (Type environment)**.** *The type environment, used to store the type information, is a partial function*

$$T : \boldsymbol{CIdNames} \rightharpoonup \boldsymbol{TCTypes} \cup \mathcal{TT}_{unr}$$

Complex identifiers are assigned to fields to handle *shadowing*: the class information is used to access the field at the correct level of the class hierarchy.

We now define the notion of terminability (in Definition 59) for a type environment: it is terminable if all elements are mapped to terminable TC types.

Definition 59 (Terminable type environment)**.** *A type environment T is said to be terminable if all the stored elements are terminable. Formally,*

$$\mathsf{term}(T) \stackrel{\text{def}}{=} \forall\, cid \in \mathrm{dom}(T)\,.\, \mathsf{term}(T(cid))$$

Operators. To facilitate the type checking procedure, we define a set of operators on type environments, targeted at dealing with specific scenarios occurring during type checking. Operators play a vital role in our type checking as they ensure the correct management of type information throughout our static analysis.

The first operator we introduce is the *resolution* one, required to deal with unnecessary conditional type information. More specifically, a method call might generate an unresolved typestate tree, which is useful in the context of, *e.g.,* `if` statements. However, if conditional information is not needed, it prevents correct program from compiling. To better understand its importance, we now consider the example in Listing 6.1. Let us suppose that, after the method call m, the

Listing 6.1: Resolution operator example

```
1 class Main {
2     void main() {
3         A a = new A();
4         a.m();
5         a.m1();
6     }
7 }
```

object a is in a state of the form $\widetilde{\langle l : t \rangle}$. The subsequent call $m1$ always fails (even if it turns out to be actually safe), as the state $\widetilde{\langle l : t \rangle}$ does not allow any operation. Moreover, if the program terminates and an object is left in $\widetilde{\langle l : t \rangle}$, even if such object is actually terminable, the program does not compile.

To this aim, in Definition 60, we formalise the resolution operator to discard conditional type information.

Let **TypeEnv** the set of type environments ranged over by meta-variable T.

Definition 60 (Resolve type environment)**.** *Given a type environment T, the operator $\widehat{T}$: **TypeEnv** $\rightarrow$ **TypeEnv** is defined as follows.*

$$\begin{aligned}\widehat{T} \stackrel{\text{def}}{=} &\{cid \mapsto \mathsf{resolveTT}(T(cid)) \mid \exists\, tt_{unr} \in \mathcal{TT}_{unr} \,.\, T(cid) = tt_{unr}\} \cup \\ &\{cid \mapsto T(cid) \mid \nexists\, tt_{unr} \in \mathcal{TT}_{unr} \,.\, T(cid) = tt_{unr}\},\end{aligned}$$

where $cid \in \mathrm{dom}(T)$.

The resolution operator relies on resolveTT (see Definition 61) and resolve (see Definition 62). In particular, the first iterates over all the nodes of the inputed unresolved typestate tree, while the second actually performs the operation.

Definition 61 (Resolve typestate tree)**.** *Given an unresolved typestate tree tt_{unr}, the function* resolveTT : $\mathcal{TT}_{unr} \rightarrow \mathcal{TT}$ *is defined as follows.*

$$\mathsf{resolveTT}((C, t_{unr}, tts_{unr})) = (C, \mathsf{resolve}(t_{unr}), \bigcup_{tt_{unr} \in tts_{unr}} \mathsf{resolveTT}(tt_{unr}))$$

Definition 62 (Resolve type)**.** *Given an unresolved type t_{unr}, the function* resolve :

$\mathcal{T}_{unr} \to \mathcal{T}$ *is defined as follows.*

$$\mathsf{resolve}(t_{unr}) = \begin{cases} \mathsf{resolve}(t'_{unr}) \cup \mathsf{resolve}(t''_{unr}) & \textit{if } t_{unr} = t'_{unr} \cup t''_{unr} \\ \mathsf{resolve}(t'_{unr}) \cap \mathsf{resolve}(t''_{unr}) & \textit{if } t_{unr} = t'_{unr} \cap t''_{unr} \\ \bigcup\{t \mid l : t \in \widetilde{l : t}\} & \textit{if } t_{unr} = \langle\widetilde{l : t}\rangle \\ t_{unr} & \textit{otherwise} \end{cases}$$

Conversely to scenarios where conditional type information must be discarded, we might encounter language constructs that crucially depends on such information, *i.e.,* if, while and switch statements. Let us consider the example presented in Listing 6.2. Suppose that, after calling the method m, the receiver object a is in

Listing 6.2: Evolve operator

```
1  class Main {
2      A a;
3      void main() {
4          a = new A();
5          if (a.m()) {
6              a.m2();
7          }
8          else {
9              a.m3();
10         }
11     }
12 }
```

the unresolved type $\langle \mathsf{true} : t' \ \ \mathsf{false} : t'' \rangle$. If we discard conditional type information and apply the resolution operator, the unresolved type of a is resolved into $t' \cup t''$ and such information is then propagated to the branches. Despite this approach being sound, it is too restrictive since union states only allow method calls that are permitted in all elements of the union. Thus, if $m2$ and $m3$ are not allowed in t' and t'', the program does not compile. However, at runtime, the program executes only one of the branches, thus it is enough that $m2$ is allowed in t' (the type information labelled with true) and $m3$ in t'' (the type information labelled with false). To this aim, we formalise in Definition 63 the *evolution* operator to compute the type information to use with branches.

Let **LNames** the set of labels ranged over by meta-variable l.

Definition 63 (Evolve type environment)**.** *Given a type environment T and a*

label l, the operator $\overset{l}{\widehat{T}} : \boldsymbol{TypeEnv} \times \boldsymbol{LNames} \rightarrow \boldsymbol{TypeEnv}$ *is defined as follows.*

$$\begin{aligned}\overset{l}{\widehat{T}} \overset{\text{def}}{=} &\{cid \mapsto \mathsf{evoTTO_{unr}}(T(cid), l) \mid \exists\, tt_{unr} \in \mathcal{TT}_{unr} \,.\, T(cid) = tt_{unr}\} \cup \\ &\{cid \mapsto T(cid) \mid \nexists\, tt_{unr} \in \mathcal{TT}_{unr} \,.\, T(cid) = tt_{unr}\},\end{aligned}$$

where $cid \in \mathrm{dom}(T)$.

The evolution operator crucially relies on $\mathsf{evoTTO_{unr}}$ (see Definition 64) and $\mathsf{evoO_{unr}}$ (see Definition 65), to correctly work: the former iterates over all the nodes of the received unresolved tree, while the latter performs the evolution.

Definition 64 (Evolve unresolved typestate tree)**.** *Given an unresolved typestate tree tt_{unr} and a label l, the function* $\mathsf{evoTTO_{unr}} : \mathcal{TT}_{unr} \times \boldsymbol{LNames} \rightarrow \mathcal{TT}$ *is defined as follows.*

$$\mathsf{evoTTO_{unr}}((C, t_{unr}, tts_{unr}), l) = (C, \mathsf{evoO_{unr}}(t_{unr}, l), \bigcup_{tt_{unr} \in tts_{unr}} \mathsf{evoTTO_{unr}}(tt_{unr}, l))$$

Definition 65 (Evolve unresolved type)**.** *Given an unresolved type t_{unr} and a label l, the function* $\mathsf{evoO_{unr}} : \mathcal{T}_{unr} \times \boldsymbol{LNames} \rightarrow \mathcal{T}$ *is defined as follows.*

$$\mathsf{evoO_{unr}}(t_{unr}, l) = \begin{cases} \mathsf{evoO_{unr}}(t'_{unr}, l) \cup \mathsf{evoO_{unr}}(t''_{unr}, l) & \text{if } t_{unr} = t'_{unr} \cup t''_{unr} \\ \mathsf{evoO_{unr}}(t'_{unr}, l) \cap \mathsf{evoO_{unr}}(t''_{unr}, l) & \text{if } t_{unr} = t'_{unr} \cap t''_{unr} \\ t & \text{if } t_{unr} = \langle l : t\ \widetilde{l : t} \rangle \\ t_{unr} & \text{otherwise} \end{cases}$$

As we dealt with code entering branches, we now need to devise a strategy to compute the resulting type information for code exiting branches. Let us consider the example presented in Listing 6.2; after the if statement, we need to calculate the resulting type information to use for the remainder of the program. Intuitively, since we cannot statically know which branch the code will embark at runtime, we must ensure soundness regardless the executed branch. Thus, we need to merge the information stemming from the branches. To this aim, we implement in Definition 66 the *merge* operator.

Definition 66 (Merge type environments)**.** *Given type environments T, T', the partial operator $T \Cup T' : \textbf{\textit{TypeEnv}} \times \textbf{\textit{TypeEnv}} \rightharpoonup \textbf{\textit{TypeEnv}}$ is defined as follows.*

$$T \Cup T' \stackrel{\text{def}}{=} \{cid \mapsto \mathsf{mrgTC}(T(cid), T'(cid))\},$$

where $cid \in \mathrm{dom}(T)$ and $\mathrm{dom}(T) = \mathrm{dom}(T')$.

As can be seen in Definition 66, the merge operator relies on mrgTC to perform the merge process. Intuitively, mrgTC behaves as follows. It takes as parameter two TC types and, if they are related by the subtyping relation, it keeps the most general one (*i.e.,* the supertype). Notice that, we cannot have basic types that do not fit such subtyping relation in the context of mrgTC. By absurd, let us suppose that such scenario occurs, it would mean that in one of the branches, we assigned a value to a field, whose type is unrelated to the statically declared one (which, of course, is not possible). Thus, whenever the subtyping relation does not hold, it signals that the TC types to merge are typestate trees. The same logic for basic types applies here. Given two typestate trees related to specific field/parameter/-variable resulting from the analysis of code branches, it is guaranteed that the root classes match. As a matter of fact, these typestate trees refer to the same field/parameter/variable and, consequently, to the same statically declared class, *i.e.,* the trees root class. Therefore, we can safely apply the mrgTT function (which assumes to receive as input typestate trees with the same root classes).

Definition 67 (Merge TC types)**.** *Given TC types tc and tc', we define the function $\mathsf{mrgTC} : \textbf{\textit{TCTypes}} \times \textbf{\textit{TCTypes}} \rightharpoonup \textbf{\textit{TCTypes}}$ to merge tc and tc' as follows.*

$$\mathsf{mrgTC}(tc, tc') = \begin{cases} tc' & \text{if } tc \leq_{\mathbf{TC},\widetilde{D}} tc' \\ tc & \text{if } tc' \leq_{\mathbf{TC},\widetilde{D}} tc \\ \mathsf{mrgTT}(tt, tt') & \text{if } tc = tt \wedge tc' = tt' \end{cases}$$

The last operator we introduce is required to perform an optimisation during type checking. Let us suppose we have to type check the methods a class C that extends C' and, in particular, we are in the verge of analysing the method m, which has been inherited from C'. As we will see later, methods are type checked accounting for the current state of fields (stored in a dedicate type environment).

Being that the method m belongs to class C', it is obvious that it cannot access fields of class C. Thus, it is useless to include these fields in the type environment used to type check m. To this aim, we formalise in Definition 68 an operator to filter out useless elements.

Definition 68 (Restrict type environment)**.** *Given a program $\widetilde{D}$, a class C and a type environment T, the operator $T\restriction_{C,\widetilde{D}}$: **TypeEnv** $\rightarrow$ **TypeEnv** is defined as follows.*

$$T\restriction_{C,\widetilde{D}} \stackrel{\text{def}}{=} \{C'.id \mapsto T(C'.id) \mid C \leq_{\widetilde{D}} C'\}$$

where $C'.id \in \text{dom}(T)$.

6.2.2 Typing Program and Class Definitions

The first element to type check is the program itself. In our setting, a program is represented as a collection of class and enum definitions, denoted by $\widetilde{D}$. Type checking the program ensures that the code adheres to the rules and structures defined by the type system, thereby statically preventing type-related errors. The type checking process requires the definition of the initial environment containing type information of uninitialised fields, not only belonging to the class under analysis, but also to its superclasses (if there are any). Thus, in Definition 69, we define the $\mathsf{initTC}_{\widetilde{D}}$ function that, given a class C, builds its field type environment.

Definition 69 (Initial types)**.** *The function $\mathsf{initTC}_{\widetilde{D}} : \mathcal{C}_{\widetilde{D}} \rightarrow$ **TypeEnv** is such that given a program $\widetilde{D}$ and a class C, it creates the corresponding type environment mapping fields to their initial TC Types. Formally,*

$$\mathsf{initTC}_{\widetilde{D}}(C) = \bigcup_{C' \in \mathcal{C}_{\widetilde{D}}, C \leq_{\widetilde{D}} C'} \{C'.id \mapsto \mathsf{inittype}(jt) \mid jt\ id \in C'.\mathsf{fields}_{\widetilde{D}}\},$$

where,

$$\mathsf{inittype}(\mathsf{bool}) \stackrel{\text{def}}{=} \mathsf{bool} \qquad \mathsf{inittype}(\mathsf{int}) \stackrel{\text{def}}{=} \mathsf{int} \qquad \mathsf{inittype}(L) \stackrel{\text{def}}{=} L$$
$$\mathsf{inittype}(\mathsf{double}) \stackrel{\text{def}}{=} \mathsf{double} \qquad \mathsf{inittype}(C) \stackrel{\text{def}}{=} (C, \mathsf{Null}, \{\})$$

Recall, for fields we do not directly store their identifiers, but we extend them with the class information (*i.e.,* the class they belong to) to correctly deal with field shadowing.

While type checking a program, we need to ensure that it has no classes and enums with duplicate identifiers. To this aim, we formalise the dedicated predicates in Definitions 70 and 71.

Definition 70 (No duplicate class/enum)**.** *Given a program $\widetilde{D}$, the predicate* $\mathsf{nodup}_{\mathbf{D}}(\widetilde{D})$ *holds if all classes and enums in $\widetilde{D}$ have unique identifiers.*

Definition 71 (No duplicate enum values)**.** *Given a program $\widetilde{D}$ and an enum L, the predicate* $\mathsf{nodup}_{\mathbf{L},\widetilde{D}}(L)$ *holds if all values in $L.\mathsf{vals}_{\widetilde{D}}$ have unique identifiers.*

Since we have a notion of subtyping among TC types that already includes behavioural subtyping, it is convenient to convert parameter and return types into TC ones. To this aim, we introduce in Definition 72 a function for such a conversion. Notice that, the set of return types includes parameter types.

Let **RTTypes** be the set of return types ranged over by meta-variable rt.

Definition 72 (Convert RT type to TC type)**.** *Given a RT type rt, the function* $\mathsf{toTC} : \boldsymbol{RTTypes} \to \boldsymbol{TCTypes}$ *is defined as follows.*

$$\mathsf{toTC}(rt) \stackrel{\text{def}}{=} \begin{cases} (C, at, \{\}) & \text{if } rt = C[at] \\ rt & \text{otherwise} \end{cases}$$

During the type checking of classes, it is crucial to assess some property related to Java classes, enums, and protocols. To this aim, in Definitions 73 to 75, we formalise dedicated predicates.

Definition 73 (Check class)**.** *Given a program $\widetilde{D}$ and a class $C \in \mathcal{C}_{\widetilde{D}}$, the predicate* $\mathsf{chkClss}_{\widetilde{\mathsf{D}}}(C)$ *holds if:*

- *all the Java classes and enums referenced in $C.\mathsf{fields}_{\widetilde{D}}$, $C.\mathsf{cons}_{\widetilde{D}}$ and $C.\mathsf{meths}_{\widetilde{D}}$ are defined in $\widetilde{D}$;*
- *all fields have unique identifiers and no two methods or constructors have the same partial signature,* i.e., *no two methods or constructors have the same name and parameter sequence $\widetilde{jt}$ of Java types;*

- *all annotations types are well formed,* i.e., *they only include typestates belonging to the referenced Java class.*

Definition 74 (Check protocol)**.** *Given a program* $\widetilde{D}$ *and a class* $C \in \mathcal{C}_{\widetilde{D}}$*, the predicate* $\mathsf{chkProt}_{\widetilde{\mathsf{D}}}(C)$ *holds if in* $C.\mathsf{prot}_{\widetilde{D}}$*:*

- *no input states contain duplicate method signatures,* i.e., *no two methods have identical names and parameter types;*

- *all output states can only be reached after calling methods whose signatures declare* bool *or enum types as the returned one;*

- *all output states must present as many choices as the possible values of the returned type,* i.e., *in case of* bool *the values* true *and* false*; in case of an enum* L *all values in* $L.\mathsf{vals}_{\widetilde{D}}$*.*

Definition 75 (Check overriding)**.** *Given a program* $\widetilde{D}$ *and classes* $C, C' \in \mathcal{C}_{\widetilde{D}}$*, the predicate* $\mathsf{chkOvr}_{\widetilde{\mathsf{D}}}$ *is defined as follows.*

$$\mathsf{chkOvr}_{\widetilde{\mathsf{D}}}(C, C') \stackrel{\text{def}}{=} \forall\, rt\, m(\widetilde{pt\ id})\{\widetilde{bst}\} \in C.\mathsf{meths}_{\widetilde{D}},\ rt'\, m(\widetilde{pt'}) \in \mathrm{dom}(C'.\mathsf{allM}_{\widetilde{D}})\ .$$
$$\mathsf{toJT}(\widetilde{pt}) = \mathsf{toJT}(\widetilde{pt'}) \Rightarrow \Big(\mathsf{toTC}(rt) \leq_{\mathbf{TC},\widetilde{D}} \mathsf{toTC}(rt') \wedge \forall\, i,\ 1 \leq i \leq |\widetilde{pt}|\ .$$
$$\mathsf{toTC}(pt'_i) \leq_{\mathbf{TC},\widetilde{D}} \mathsf{toTC}(pt_i)\Big)$$

For each method definition M in the subclass C such that there is a method definition in a superclass C' that expects the same sequence of Java types as parameters, the predicate $\mathsf{chkOvr}_{\widetilde{\mathsf{D}}}$ ensures that: (i) if the parameters have an annotation type at, such annotation type is a supertype of the corresponding one in the superclass; and (ii) its return type is a subtype of the one in the superclass, in terms of both Java type and annotation type (the latter if present). Since we are devising a type checking procedure for a subset of the Java language, we have to strictly adhere to its interpretation of method overriding, *i.e.,* the Java types of the parameter in methods of the subclass must be the same in the corresponding methods in the superclass. However, we could have relaxed the equality constraint on parameter Java types, asking for a supertype in the

overridden methods (following the approach for annotation types) and the type checking would be still sound, as done in, *e.g.,* the Python programming language.

The judgement for programs is of the form $\vdash \widetilde{D}$, while the one to type check classes and enums is of the form $\vdash_{\widetilde{D}} D$. The rules, defining how these judgements work in the context of our type checking, are presented in Definition 76.

Definition 76 (Typing rules for program and class definitions)**.**

$$\textsc{TProg}\ \frac{\mathsf{nodup}_{\mathbf{D}}(\widetilde{D}) \qquad \forall D \in \widetilde{D}\,.\,\vdash_{\widetilde{D}} D}{\vdash \widetilde{D}} \qquad \textsc{TEnum}\ \frac{\mathsf{nodup}_{\mathbf{L},\widetilde{D}}(L)}{\vdash_{\widetilde{D}} \mathsf{enum}\ L\ \{\widetilde{id}\}}$$

$$\textsc{TClass1}\ \frac{\begin{array}{c}|\widetilde{B}| > 0 \qquad \mathsf{chkClss}_{\widetilde{\mathsf{D}}}(C) \qquad \mathsf{chkProt}_{\widetilde{\mathsf{D}}}(C) \\ \forall i,\, 1 \leq i \leq |\widetilde{B}|\,.\,\mathsf{initTC}_{\widetilde{D}}(C) \vdash_{\widetilde{D}} B_i \rhd T_{\mathbf{f},i} \wedge \\ \emptyset; T_{\mathbf{f},i} \vdash_{\widetilde{D}} C[u^{\widetilde{E}}] \rhd T_{\mathbf{f},i} \ \wedge\ \mathsf{term}(\widehat{T_{\mathbf{f},i}})\end{array}}{\vdash_{\widetilde{D}} \mathsf{class}\ C\,\{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}}$$

$$\textsc{TClass2}\ \frac{\vdash_{\widetilde{D}} \mathsf{class}\ C\,\{u^{\widetilde{E}}; \widetilde{F}; C(\,)\{\,\}; \widetilde{M}\}}{\vdash_{\widetilde{D}} \mathsf{class}\ C\,\{u^{\widetilde{E}}; \widetilde{F}; \varepsilon; \widetilde{M}\}}$$

$$\textsc{TExt}\ \frac{\begin{array}{c}C' \in \mathcal{C}_{\widetilde{D}} \qquad u^{\widetilde{E}} \leq_{\mathbf{S_{alg}}} C'.\mathsf{prot}_{\widetilde{D}} \\ \mathsf{chkOvr}_{\widetilde{\mathsf{D}}}(C, C') \qquad \vdash_{\widetilde{D}} \mathsf{class}\ C\,\{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}\end{array}}{\vdash_{\widetilde{D}} \mathsf{class}\ C\,\{u^{\widetilde{E}}; \widetilde{F}; \widetilde{B}; \widetilde{M}\}\ \mathsf{extends}\ C'}$$

By applying the TProg rule, we are able to type check all the classes and enums defined in $\widetilde{D}$. For simplicity, enums contain only constant values. In both TProg and TEnum rules we ensure that $\mathsf{nodup}_{\mathbf{D}}$ and $\mathsf{nodup}_{\mathbf{L},\widetilde{D}}$ hold.

We type check class definitions ensuring that the predicates $\mathsf{chkClss}_{\widetilde{\mathsf{D}}}$, $\mathsf{chkProt}_{\widetilde{\mathsf{D}}}$ and protocol completion hold. If constructors are available (see TClass1 rule), we analyse each of them individually and use the field type environment stemming from such analysis to type check the initial typestate of the protocol. Otherwise (see TClass2 rule), we synthesize the default constructor and resort to TClass1.

If a class extends another, we apply the TExt rule, additionally checking that: (i) the superclass is defined in the current program; (ii) the protocol of the subclass

is a subtype of the one attached to the superclass[1]; and (*iii*) the $\mathsf{chkOvr}_{\widetilde{\mathsf{D}}}$ predicate holds. Finally, we resort to the rules to type check classes without supertypes.

To analyse constructors, we use a dedicated judgement of the form

$$T_{\mathbf{f}} \vdash_{\widetilde{D}} B \rhd T'_{\mathbf{f}}.$$

Intuitively, $T_{\mathbf{f}}$ and $T'_{\mathbf{f}}$ are the initial and final field type environments and B is the constructor to analyse. The rules for constructors are presented in Definition 77. Notice that, $\mathrm{dom}(T)_{\perp}$ signals an environment whose elements are mapped to $\perp$.

Definition 77 (Typing rules for constructor definitions)**.**

$$\textsc{TCns}\ \frac{\begin{array}{c} C.\mathsf{sup}_{\widetilde{D}} = \varepsilon \qquad Tb_{\mathbf{f}} = Tr_{\mathbf{f}} = \mathrm{dom}(T_{\mathbf{f}})_{\perp} \\ T_{\mathbf{s}} = \{id \mapsto \mathsf{toTC}(pt) \mid pt\ id \in \widetilde{pt\ id}\} \qquad Tb_{\mathbf{s}} = \mathrm{dom}(T_{\mathbf{s}})_{\perp} \\ T_{\mathbf{f}}, T_{\mathbf{s}}; Tb_{\mathbf{f}}, Tb_{\mathbf{s}}; Tr_{\mathbf{f}} \vdash_{C,\mathsf{void},\widetilde{D}} \widetilde{bst} \rhd T'_{\mathbf{f}}, T'_{\mathbf{s}}; Tb_{\mathbf{f}}, Tb_{\mathbf{s}}; Tr_{\mathbf{f}} \qquad \mathsf{term}(T'_{\mathbf{s}}) \end{array}}{T_{\mathbf{f}} \vdash_{\widetilde{D}} C(\widetilde{pt\ id})\{\widetilde{bst}\} \rhd T'_{\mathbf{f}}}$$

$$\textsc{TCnsExt1}\ \frac{\begin{array}{c} C.\mathsf{sup}_{\widetilde{D}} = C' \\ T_{\mathbf{f}}, \{id \mapsto \mathsf{toTC}(pt) \mid pt\ id \in \widetilde{pt\ id}\} \vdash_{C',\widetilde{D}} \mathsf{super}(\widetilde{e}) \rhd T'_{\mathbf{f}}, T'_{\mathbf{s}} \\ Tb_{\mathbf{f}} = Tr_{\mathbf{f}} = \mathrm{dom}(T'_{\mathbf{f}})_{\perp} \qquad Tb_{\mathbf{s}} = \mathrm{dom}(T'_{\mathbf{s}})_{\perp} \\ T'_{\mathbf{f}}, T'_{\mathbf{s}}; Tb_{\mathbf{f}}, Tb_{\mathbf{s}}; Tr_{\mathbf{f}} \vdash_{C,\mathsf{void},\widetilde{D}} \widetilde{bst} \rhd T''_{\mathbf{f}}, T''_{\mathbf{s}}; Tb_{\mathbf{f}}, Tb_{\mathbf{s}}; Tr_{\mathbf{f}} \qquad \mathsf{term}(T''_{\mathbf{s}}) \end{array}}{T_{\mathbf{f}} \vdash_{\widetilde{D}} C(\widetilde{pt\ id})\{\mathsf{super}(\widetilde{e})\ \widetilde{bst}\} \rhd T''_{\mathbf{f}}}$$

$$\textsc{TCnsExt2}\ \frac{C.\mathsf{sup}_{\widetilde{D}} = C' \qquad T_{\mathbf{f}} \vdash_{\widetilde{D}} C(\widetilde{pt\ id})\{\mathsf{super}(\,)\ \widetilde{bst}\} \rhd T'_{\mathbf{f}}}{T_{\mathbf{f}} \vdash_{\widetilde{D}} C(\widetilde{pt\ id})\{\widetilde{bst}\} \rhd T'_{\mathbf{f}}}$$

The rules introduced in Definition 77 make use of several type environments:

- $T_{\mathbf{s}}$ is used to store type information of variables and parameters;
- $Tb_{\mathbf{f}}$ and $Tb_{\mathbf{s}}$, initially mapping their elements to $\perp$, store type information of fields (the former), variables and parameters (the latter) at the moment a break statement is encountered;

[1] We use the algorithm in Definition 7, but checking partial method signatures, *i.e.,* signatures without return types, equality

- $Tr_{\mathbf{f}}$, initially mapping its elements to $\perp$, stores type information of fields at the moment a return statement is encountered.

Notice that, $Tb_{\mathbf{f}}, Tb_{\mathbf{s}}$ and $Tr_{\mathbf{f}}$ are only modified in case the code presents break and return statements, respectively.

Parameters and variables type environment have their domain composed by simple identifiers, id. Since parameters/variables cannot be shadowed: including class information within identifiers is not required.

We ensure that $Tb_{\mathbf{f}}, Tb_{\mathbf{s}}$ and $Tr_{\mathbf{f}}$ remain consistent with their initial states. Deviations indicate the presence of a break outside the scope of a while/switch or return in the constructor body (here, constructors cannot have return).

The TCns rule type checks the constructors of a class that does not have supertypes. We first create the necessary type environments: (i) $T_{\mathbf{s}}$ is initially built from the parameters declared in the signature of the constructor under analysis ($i.e.$, $\widetilde{pt\ id}$); and (ii) $Tb_{\mathbf{f}}, Tb_{\mathbf{s}}$ and $Tr_{\mathbf{f}}$ are built mapping the elements of the corresponding domain to $\perp$. Then, we analyse the constructor body $\widetilde{bst}$, represented as a sequence of statements, with the corresponding judgement. As a result, we get an environment $T'_{\mathbf{f}}$ containing type information of fields at the end of the constructor body and $T'_{\mathbf{s}}$ containing type information of parameters and variables at the end of the body. Once we finished analysing the constructor, we ensure termination of parameters and variables, additionally checking the termination of $T'_{\mathbf{s}}$.

The TCnsExt1 rule type checks the constructors of a class extending another with an explicit super call. We first analyse the super call with a dedicated judgement, getting as result $T'_{\mathbf{f}}$ and $T'_{\mathbf{s}}$. Then, we proceed as in TCns1: we build the necessary environments, analyse the constructor body using as initial type environments $T'_{\mathbf{f}}$ and $T'_{\mathbf{s}}$ and ensure that parameters and variables are terminated.

The TCnsExt2 rule type checks the constructors of a class extending another without an explicit super constructor call: we add an explicit parameterless super call to the superclass constructor and then use TCnsExt1 to analyse the constructor.

To type check super calls, we use a dedicated judgement of the form

$$T_{\mathbf{f}} \vdash_{C,\widetilde{D}} \mathsf{super}(\widetilde{e}) \rhd T'_{\mathbf{f}},$$

where C is the class owning the constructor we are invoking with the super call.

The rules to type check the super constructor calls are presented in Definition 78.

Definition 78 (Typing rule for super constructor call)**.**

$$\text{TSup1}\ \frac{\begin{array}{c}|C.\mathsf{cons}_{\widetilde{D}}| > 0 \qquad T_{\mathbf{f}}, T_{\mathbf{s}} \vdash_{\widetilde{D}} \widetilde{e : tc} \rhd T_{\mathbf{f}}, T'_{\mathbf{s}} \\ C(\widetilde{pt\ id}) = \mathsf{min}_{\widetilde{\mathsf{D}}}(\mathsf{cands}_{\widetilde{\mathsf{D}}}(C, \widetilde{tc})) \\ \forall i, 1 \leq i \leq |\widetilde{tc}|\ .\ tc_i \leq_{\mathbf{TC},\widetilde{D}} \mathsf{toTC}(pt_i) \\ C(\widetilde{pt\ id})\{cbst\} \in C.\mathsf{cons}_{\widetilde{D}} \qquad T_{\mathbf{f}}{\restriction}_{C,\widetilde{D}} \vdash_{\widetilde{D}} C(\widetilde{pt\ id})\{cbst\} \rhd T'_{\mathbf{f}}\end{array}}{T_{\mathbf{f}}, T_{\mathbf{s}} \vdash_{C,\widetilde{D}} \mathsf{super}(\widetilde{e}) \rhd (T_{\mathbf{f}} - T_{\mathbf{f}}{\restriction}_{C,\widetilde{D}}) \cup T'_{\mathbf{f}}, T'_{\mathbf{s}}}$$

$$\text{TSup2}\ \frac{|C.\mathsf{cons}_{\widetilde{D}}| = 0 \qquad T_{\mathbf{f}}{\restriction}_{C,\widetilde{D}} \vdash_{\widetilde{D}} C(\,)\{\,\} \rhd T'_{\mathbf{f}}}{T_{\mathbf{f}}, T_{\mathbf{s}} \vdash_{C,\widetilde{D}} \mathsf{super}(\,) \rhd (T_{\mathbf{f}} - T_{\mathbf{f}}{\restriction}_{C,\widetilde{D}}) \cup T'_{\mathbf{f}}, T_{\mathbf{s}}}$$

The TSup1 rule analyses super calls to classes with constructors. We first analyse the sequence of expressions $\widetilde{e}$ passed as parameters (whose judgement will be introduced later). Then, we statically bind the correct constructor and check that the super calls receives correct parameters. Finally, we type check the constructor we bound before, using the rules in Definition 77. Notice that, the starting field type environment used to type check the super call is restricted to filter out fields of the subclass that performed such super call (we cannot access those fields from the superclass). Thus, we perform the union of the type environment containing subclass fields only with $T'_{\mathbf{f}}$ (the restricted environment).

The TSup2 rule analyses super calls to classes without constructors: we synthesize the default constructor and analyse it using the rules in Definition 77.

6.2.3 Typing Class Typestate Definitions

The rules in Definition 76 make use of the judgement to type check the initial typestate of a class C, which ensures the correct usage of the class in the program. The judgement, used to type check typestate definitions of a class C, is

$$\Theta, T_{\mathbf{f}} \vdash_{\widetilde{D}} C[w^{\widetilde{E}}] \rhd T'_{\mathbf{f}},$$

where $T'_{\mathbf{f}}$ is the resulting field type environment after the typing computation and Θ is an environment to deal with recursive behaviour of typestates.

Let **EVars** be the set of defining equations ranged over by meta-variable E.

Definition 79 (Typestate variable environment)**.** *A typestate variable environment, mapping defining equations to type environments, is a partial function*

$$\Theta : \boldsymbol{EVars} \rightharpoonup \boldsymbol{TypeEnv}$$

The typestate variable environment stores field type information to deal with recursive behaviour: if we encounter a typestate name s, which already is in $\mathrm{dom}(\Theta)$, we can use the stored type information to check s.

During the analysis of typestates $\{\widetilde{jms : w}\}^{\widetilde{E}}$, we need to match each jms with a corresponding method in class C, checking method signature equality. Since in jms we only include Java types, while methods in C can possibly have annotation types, in Definition 80 we present a function to convert a signature with annotation types to one with only Java types. Notice that, we overload the function toJT.

Let $\mathcal{JMS}$ be the set of Java method signatures ranged over by meta-variable jms; $\mathcal{MS}$ is the set of method signatures ranged over by meta-variable ms;

Definition 80 (Extract Java method signature)**.** *Given a method signature ms, the function* $\mathsf{toJT} : \mathcal{MS} \to \mathcal{JMS}$ *is defined as follows.*

$$\mathsf{toJT}(rt\, m(\widetilde{pt})) = \mathsf{toJT}(rt)\, m(\mathsf{toJT}(\widetilde{pt}))$$

In Definition 81, we present the type checking rules for typestate definitions.

Definition 81 (Typing rules for typestate definitions)**.**

$$\textsc{TBr}\ \frac{\begin{array}{c}\forall i,\, 1 \le i \le |\widetilde{jms : w}|\ .\ \exists\, T'_{\mathbf{f}}, ms, C'\ .\ ms \mapsto C' \in C.\mathsf{allM}_{\widetilde{D}} \wedge \\ jms_i = \mathsf{toJT}(ms) \wedge \exists\, M \in C'.\mathsf{meths}_{\widetilde{D}}\ .\ ms = \mathsf{sig}(M) \wedge \\ \widehat{T}_{\mathbf{f}}\restriction_{C',\widetilde{D}} \vdash_{C',\widetilde{D}} M \rhd T'_{\mathbf{f}} \wedge \Theta; (\widehat{T}_{\mathbf{f}} - \widehat{T}_{\mathbf{f}}\restriction_{C',\widetilde{D}}) \cup T'_{\mathbf{f}} \vdash_{\widetilde{D}} C[w_i^{\widetilde{E}}] \rhd T''_{\mathbf{f},i}\end{array}}{\Theta; T_{\mathbf{f}} \vdash_{\widetilde{D}} C[\{\widetilde{jms : w}\}^{\widetilde{E}}] \rhd \bigcup_{1 \le i \le |\widetilde{m:w}|} T''_{\mathbf{f},i}}$$

$$\textsc{TBrDrop}\ \frac{\mathsf{term}(T_{\mathbf{f}}) \qquad |\widetilde{jms : w}| > 0 \qquad \Theta; T_{\mathbf{f}} \vdash_{\widetilde{D}} C[\{\widetilde{jms : w}\}^{\widetilde{E}}] \rhd T'_{\mathbf{f}}}{\Theta; T_{\mathbf{f}} \vdash_{\widetilde{D}} C[_{drop}\{\widetilde{jms : w}\}^{\widetilde{E}}] \rhd T'_{\mathbf{f}}}$$

$$\textsc{TCh}\ \frac{\forall i,\, 1 \leq i \leq |\widetilde{l : u}|\,.\ \Theta; \overset{l_i}{\widehat{T}}_{\mathbf{f}} \vdash_{\widetilde{D}} C[u_i^{\widetilde{E}}] \rhd T'_{\mathbf{f},i}}{\Theta; T_{\mathbf{f}} \vdash_{\widetilde{D}} C[\widetilde{\langle l : u \rangle}^{\widetilde{E}}] \rhd \bigcup\limits_{1 \leq i \leq |\widetilde{l:u}|} T'_{\mathbf{f},i}}$$

$$\textsc{TEnd}\ \frac{}{\Theta; T_{\mathbf{f}} \vdash_{\widetilde{D}} C[\mathsf{end}^{\widetilde{E}}] \rhd T_{\mathbf{f}}} \qquad \textsc{TRec}\ \frac{\Theta{\cdot}\{s \mapsto T_{\mathbf{f}}\}; T_{\mathbf{f}} \vdash_{\widetilde{D}} C[u^{\widetilde{E}}] \rhd T'_{\mathbf{f}}}{\Theta; T_{\mathbf{f}} \vdash_{\widetilde{D}} C[s^{\widetilde{E} \uplus \{s=u\}}] \rhd T'_{\mathbf{f}}}$$

$$\textsc{TVar}\ \frac{\forall cid \in \mathrm{dom}(T'_{\mathbf{f}})\,.\ T'_{\mathbf{f}}(cid) \leq_{\mathbf{TC},\widetilde{D}} T_{\mathbf{f}}(cid)}{\Theta{\cdot}\{s \mapsto T_{\mathbf{f}}\}; T'_{\mathbf{f}} \vdash_{\widetilde{D}} C[s^{\widetilde{E}}] \rhd \mathrm{dom}(T_{\mathbf{f}})_{\perp}}$$

For each Java method signature jms appearing in the current typestate, the TBr rule looks for a method definition M in the current class or its superclasses whose Java signature is equal to jms. Once we identified M, we type check it using as starting field type environment the one stemming from previous analyses (if we are type checking the initial state of the protocol, the field type environment comes from the TClass1 rule). The type environment resulting from the analysis of method M is then used as the starting one for the typestate reached after calling M. Since a typestate can allow several method calls, we collect all the field type environments stemming from methods analyses and merge them (and in the TClass1 rule we ensure termination of the fields). During type checking, it may happen that an unresolved typestate tree is propagated from one typestate to another, *i.e.,* if the expression in the return statement generates an unresolved tree. The conditional information carried by such tree becomes necessary if the continuation, *i.e.,* the typestate reached after calling a given method, is of the form $\widetilde{\langle l : u \rangle}^{\widetilde{E}}$: the type information to analyse each choice is computed by evolving the unresolved tree with the proper label. However, since the TBr rule analyses typestate of the form $u^{\widetilde{E}}$, we need to resolve the initial field type environment beforehand: we cannot determine the label to use to resolve the unresolved typestate tree, thus, we must ensure that the code compiles with respect to all the possible choices in the unresolved tree. As for the type checking of the super constructor call, we apply the restriction operator to avoid including fields that cannot be accessed.

The TBrDrop rule deals with droppable typestates. If the sequence $\widetilde{jms : w}$ is not empty, we apply the TBr rule; otherwise, we use the TEnd rule as end is an

alias for $\mathsf{drop}\{\}$. Since an object in a droppable state can be stopped from further use, we need to ensure that all the fields of such object are terminable.

Whenever we reach a state of the form $\langle \widetilde{l : u} \rangle^{\widetilde{E}}$, the TCh rule comes into play: it ensures that such typestate is well typed if all choices are well typed. Differently from the TBr rule, type information is crucial: the typestate reached after a state of the form $\langle \widetilde{l : u} \rangle^{\widetilde{E}}$ depends on the value returned by the method call that led to the current state. Thus, the unresolved tree generated by such method call is evolved beforehand using as labels those appearing in $\langle \widetilde{l : u} \rangle^{\widetilde{E}}$. This operation ensures that subsequent typestates are analysed with correct and precise type information.

The TEnd rule is straightforward: the type checking of the end state does not change the field type environment.

The rules TVar and TRec handle recursive behaviour of typestates. TVar states that a variable s is well typed if the initial field type environment is a subtype of the one stored in Θ, ensuring that all the operations doable when we first met s are still available; TRec maps a variable s to the current field type environment and type checks the typestate on the right-hand side, *e.g.,* $u^{\widetilde{E}}$ if $s = u^{\widetilde{E}}$. In the TVar rule, we never need the resulting field type environment, as the recursion brings us back to the typestate on the right-hand side of the equation. Thus, to avoid non-determinism, we map the fields of the resulting type environment to $\bot$.

The TBr rule exploits a dedicated judgement for method definitions of the form

$$T_{\mathbf{f}} \vdash_{C,\widetilde{D}} rt\, m(\widetilde{pt\ id})\{\widetilde{bst}\} \rhd T'_{\mathbf{f}},$$

where $T_{\mathbf{f}}$ and $T'_{\mathbf{f}}$ are the type environments containing field information before and after the analysis of the method and C is the class implementing the method under analysis. For clarity sake, in Definition 82, we devise separate rules for methods returning void (*i.e.,* TMeth1) and those returning a different type (*i.e.,* TMeth2).

Definition 82 (Typing rules for method definitions)**.**

$$\text{TMeth1}\ \frac{\begin{array}{c} T_{\mathbf{s}} = \{id \mapsto \mathsf{toTC}(pt) \mid pt\ id \in \widetilde{pt\ id}\} \\ Tb_{\mathbf{f}} = Tr_{\mathbf{f}} = \mathrm{dom}(T_{\mathbf{f}})_{\bot} \qquad Tb_{\mathbf{s}} = \mathrm{dom}(T_{\mathbf{s}})_{\bot} \\ T_{\mathbf{f}}, T_{\mathbf{s}}; Tb_{\mathbf{f}}, T_{\mathbf{s}}; Tr_{\mathbf{f}} \vdash_{C,\mathsf{void},\widetilde{D}} \widetilde{bst} \rhd T'_{\mathbf{f}}, T'_{\mathbf{s}}; Tb_{\mathbf{f}}, Tb_{\mathbf{s}}; Tr_{\mathbf{f}} \qquad \mathsf{term}(T'_{\mathbf{s}}) \end{array}}{T_{\mathbf{f}} \vdash_{C,\widetilde{D}} \mathsf{void}\, m(\widetilde{pt\ id})\{\widetilde{bst}\} \rhd T'_{\mathbf{f}}}$$

$$\textsc{TMeth2}\ \frac{\begin{array}{c} rt \neq \mathsf{void} \\ T_{\mathbf{s}} = \{id \mapsto \mathsf{toTC}(pt) \mid pt\ id \in \widetilde{pt\ id}\} \\ Tb_{\mathbf{f}} = Tr_{\mathbf{f}} = \mathrm{dom}(T_{\mathbf{f}})_{\perp} \qquad Tb_{\mathbf{s}} = \mathrm{dom}(T_{\mathbf{s}})_{\perp} \\ T_{\mathbf{f}}, T_{\mathbf{s}}; Tb_{\mathbf{f}}, T_{\mathbf{s}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \widetilde{bst} \rhd \mathrm{dom}(T_{\mathbf{f}})_{\perp}, \mathrm{dom}(T_{\mathbf{s}})_{\perp}; Tb_{\mathbf{f}}, Tb_{\mathbf{s}}; Tr'_{\mathbf{f}} \end{array}}{T_{\mathbf{f}} \vdash_{C,\widetilde{D}} rt\ m(\widetilde{pt\ id})\{\widetilde{bst}\} \rhd Tr'_{\mathbf{f}}}$$

The rules presented in Definition 82 work as those for constructors presented in Definition 77: we first build the necessary type environments (*i.e.,* $T_{\mathbf{s}}, Tb_{\mathbf{f}}, Tb_{\mathbf{s}}$ and $Tr_{\mathbf{f}}$) and then type check the method body. The important things to notice here are the constraints we impose on some of the final environments. In particular, we impose that $Tb_{\mathbf{f}}, Tb_{\mathbf{s}}$ remain unchanged: modifications to these environments signal the presence of a break statement outisde the scope of while/switch statement.

In the TMeth1 rule, where we analyse methods returning void type, we impose that $Tr_{\mathbf{f}}$ remains unchanged after the analysis of the method body. Changes in $Tr_{\mathbf{f}}$ imply the presence of a return statement somewhere in the method body, which is, again, an error (in our setting methods returning void cannot have return).

In the TMeth2 rule, where we analyse methods returning a *pt* type, we impose that the resulting type environments of fields, parameters and variables map all their domain elements to $\perp$ after the analysis of the method body. In our type checking procedure, whenever we encounter a return statement we update $Tr_{\mathbf{f}}$ with the current types of the fields and map all elements of $T_{\mathbf{f}}$ and $T_{\mathbf{s}}$ to $\perp$. The latter operation is crucial to check that all possible execution paths (*i.e.,* branches in the code) terminate with return. Recall, if the code presents branches, we use the operator in Definition 66 to merge the type environments stemming from the analyses of such branches. This operator is such that if we merge two TC types related by the subtyping relation, it keeps the most general, *i.e.,* the supertype. Consequently, the result of merging two type environments T and T' is an environment mapping all its elements to $\perp$ only if T and T' already map their elements to $\perp$ (since $\perp$ is subtype of everything, the merge operator discards it unless it is merged with itself). Thus, either all branches terminates with a return statement and maps type environments to $\perp$ or the resulting type environment after such branches violates our constraint. Notice that, in the TMeth2 rule, we do not need to ensure parameter termination as it is checked every time we encounter a return.

6.2.4 Typing Statements

In this Section, we thoroughly present the type checking rules required to deal with all statements. Throughout these rules, we use Δ and $\Delta_{\mathbf{b}}$ as shorthands for pair of type environments: Δ represents $T_{\mathbf{f}}, T_{\mathbf{s}}$, *i.e.,* the field and parameter/variable type environments and $\Delta_{\mathbf{b}}$ denotes $Tb_{\mathbf{f}}, Tb_{\mathbf{s}}$, *i.e.,* the field and parameter/variable type environments used to deal with the break statement. Whenever we apply an operator to a pair of type environments, it operates on both elements of the pair, *e.g.,* $\widehat{\Delta} = \widehat{T}, \widehat{T'}$. The merge operator, with pairs of type environments, works as follows: being $\Delta = T, T'$ and $\Delta' = T'', T'''$, we have $\Delta \Cup \Delta' = T \Cup T'', T' \Cup T'''$.

First, we need to devise a mechanism to analyse the sequence of statements $\widetilde{bst}$. To analyse sequence of statements, we use a judgment of the form

$$\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \widetilde{bst} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}},$$

where $\Delta, \Delta_{\mathbf{b}}$ and $Tr_{\mathbf{f}}$ are the type environments before the analysis and $\Delta', \Delta'_{\mathbf{b}}$ and $Tr'_{\mathbf{f}}$ are the resulting ones. In the judgement, we also keep track of the return type rt of the method we are analysing and C the class implementing such method. We now present, in Definition 83 the rules to type check a sequence of statements.

Definition 83 (Typing rules for sequence of statements)**.**

$$\textsc{TEmpty}\ \frac{}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \epsilon \rhd \Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}}}$$

$$\textsc{TSeqSt}\ \frac{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} bst \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}} \quad \Delta' \neq \mathrm{dom}(\Delta)_{\perp} \quad \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \widetilde{bst} \rhd \Delta''; \Delta''_{\mathbf{b}}; Tr''_{\mathbf{f}}}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} bst\ \widetilde{bst} \rhd \Delta''; \Delta''_{\mathbf{b}}; Tr''_{\mathbf{f}}}$$

The TEmpty rule is straightforward: if the sequence is empty the type environments do not change.

The TSeqSt rule analyses the first element of the sequence and uses the resulting environments to type check the remainder. This rule also ensures that the type environments in Δ' do not map their elements to $\perp$. This is crucial to reject unreachable code, *i.e.,* code immediately after break or return. Since type environments in Δ' are set to $\perp$ only if we encouter a break or return statements,

if we detect that, we halt the type checking process as the subsequent code is unreachable.

Definition 84 (No duplicate cases)**.** *Given a sequence of case blocks* $\widetilde{cbl}$*, the predicate* $\mathsf{nodup}_{\mathbf{S}}(\widetilde{cbl})$ *holds if all cases have unique labels.*

Definition 85 (Unused labels)**.** *Given a program* $\widetilde{D}$*, an enum* $L \in \mathcal{L}_{\widetilde{D}}$ *and a sequence of case blocks* $\widetilde{cbl}$*, the function* $\mathsf{noCaseVals}_{\widetilde{D}}(\widetilde{cbl})$ *returns the difference between* $L.\mathsf{vals}_{\widetilde{D}}$ *and* $\widetilde{cbl}$*,* i.e.*, the labels not appearing in* $\widetilde{cbl}$*.*

Definition 86 (Typing rules for statements)**.**

$$\textsc{TExp}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : tc \rhd \Delta' \qquad \mathsf{term}(tc)}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} e \rhd \widehat{\Delta'}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}}}$$

$$\textsc{TBlock}\ \frac{\begin{array}{c} T_{\mathbf{f}}, T_{\mathbf{s}}; Tb_{\mathbf{f}}, Tb_{\mathbf{s}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \widetilde{bst} \rhd T'_{\mathbf{f}}, T'_{\mathbf{s}}; Tb'_{\mathbf{f}}, Tb'_{\mathbf{s}}; Tr'_{\mathbf{f}} \\ \forall cid \in \mathrm{dom}(T'_{\mathbf{s}}) - \mathrm{dom}(T_{\mathbf{s}})\,.\ \mathsf{term}(T'_{\mathbf{s}}(cid)) \\ \forall cid \in \mathrm{dom}(Tb'_{\mathbf{s}}) - \mathrm{dom}(T_{\mathbf{s}})\,.\ \mathsf{term}(Tb'_{\mathbf{s}}(cid)) \\ T''_{\mathbf{s}} = \{cid \mapsto T'_{\mathbf{s}}(cid) \mid cid \in \mathrm{dom}(T_{\mathbf{s}})\} \\ Tb''_{\mathbf{s}} = \{cid \mapsto Tb'_{\mathbf{s}}(cid) \mid cid \in \mathrm{dom}(T_{\mathbf{s}})\} \end{array}}{T_{\mathbf{f}}; T_{\mathbf{s}}; Tb_{\mathbf{f}}, Tb_{\mathbf{s}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \{\widetilde{bst}\} \rhd T'_{\mathbf{f}}, T''_{\mathbf{s}}; Tb'_{\mathbf{f}}, Tb''_{\mathbf{s}}; Tr_{\mathbf{f}}}$$

$$\textsc{TIf}\ \frac{\begin{array}{c} \Delta \vdash_{C,\widetilde{D}} e : \mathsf{bool} \rhd \Delta' \\ \overset{\mathsf{true}}{\widehat{\Delta'}}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} st' \rhd \Delta^{\mathsf{T}}; \Delta^{\mathsf{T}}_{\mathbf{b}}; Tr^{\mathsf{T}}_{\mathbf{f}} \\ \overset{\mathsf{false}}{\widehat{\Delta'}}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} st'' \rhd \Delta^{\mathsf{F}}; \Delta^{\mathsf{F}}_{\mathbf{b}}; Tr^{\mathsf{F}}_{\mathbf{f}} \end{array}}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \mathsf{if}\ (e)\ st'\ \mathsf{else}\ st'' \rhd \Delta^{\mathsf{T}} \Cup \Delta^{\mathsf{F}}; \Delta^{\mathsf{T}}_{\mathbf{b}} \Cup \Delta^{\mathsf{F}}_{\mathbf{b}}; Tr^{\mathsf{T}}_{\mathbf{f}} \Cup Tr^{\mathsf{F}}_{\mathbf{f}}}$$

$$\textsc{TWhl}\ \frac{\begin{array}{c} T_{\mathbf{f}}, T_{\mathbf{s}} \vdash_{\widetilde{D}} e : \mathsf{bool} \rhd T''_{\mathbf{f}}, T''_{\mathbf{s}} \\ \overset{\mathsf{true}}{\widehat{T''_{\mathbf{f}}}}, \overset{\mathsf{true}}{\widehat{T''_{\mathbf{s}}}}; \mathrm{dom}(\Delta)_{\perp}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} st \rhd T'_{\mathbf{f}}, T'_{\mathbf{s}}; Tb'_{\mathbf{f}}, Tb'_{\mathbf{s}}; Tr'_{\mathbf{f}} \\ \forall cid \in \mathrm{dom}(T_{\mathbf{s}})\,.\ T'_{\mathbf{s}}(cid) \leq_{\mathbf{TC},\widetilde{D}} T_{\mathbf{s}}(cid) \\ \forall cid \in \mathrm{dom}(T_{\mathbf{f}})\,.\ T'_{\mathbf{f}}(cid) \leq_{\mathbf{TC},\widetilde{D}} T_{\mathbf{f}}(cid) \end{array}}{T_{\mathbf{f}}, T_{\mathbf{s}}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \mathsf{while}\,(e)\ st \rhd \overset{\mathsf{false}}{\widehat{T''_{\mathbf{f}}}} \Cup Tb'_{\mathbf{f}}, \overset{\mathsf{false}}{\widehat{T''_{\mathbf{s}}}} \Cup Tb'_{\mathbf{s}}; \Delta_{\mathbf{b}}; Tr'_{\mathbf{f}}}$$

$$\textsc{TRet}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : tc \rhd T'_{\mathbf{f}}, T'_{\mathbf{s}} \qquad tc \leq_{\mathbf{TC},\widetilde{D}} \mathsf{toTC}(rt) \qquad \mathsf{term}(\widehat{T'_{\mathbf{s}}})}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \mathsf{return}\, e \rhd \mathrm{dom}(\Delta)_{\perp}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \Cup T'_{\mathbf{f}}}$$

$$\textsc{TVDecl}\ \frac{\begin{array}{c} jt \neq C' \qquad id \notin \mathrm{dom}(T_{\mathbf{s}}) \qquad T_{\mathbf{f}}, T_{\mathbf{s}} \vdash_{\widetilde{D}} e : tc \rhd T'_{\mathbf{f}}, T'_{\mathbf{s}} \\ \mathsf{toJT}(tc) \leq_{\widetilde{D}} jt \qquad T''_{\mathbf{s}} = T'_{\mathbf{s}} \cup \{id \mapsto \mathsf{inittype}(jt)\} \end{array}}{T_{\mathbf{f}}, T_{\mathbf{s}}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} jt\ id = e \rhd \widehat{T'_{\mathbf{f}}}, \widehat{T''_{\mathbf{s}}}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}}}$$

$$\textsc{TVDeclO}\ \frac{\begin{array}{c} C' \in \mathcal{C}_{\widetilde{D}} \qquad id \notin \mathrm{dom}(T_{\mathbf{s}}) \qquad T_{\mathbf{f}}, T_{\mathbf{s}} \vdash_{\widetilde{D}} e : tt \rhd T'_{\mathbf{f}}, T'_{\mathbf{s}} \\ \mathsf{cl}(tt) \leq_{\widetilde{D}} C' \qquad tt' = \mathsf{ucastTT}(tt, C') \qquad T''_{\mathbf{s}} = T'_{\mathbf{s}} \cup \{id \mapsto tt'\} \end{array}}{T_{\mathbf{f}}, T_{\mathbf{s}}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} C'\ id = e \rhd T'_{\mathbf{f}}, T''_{\mathbf{s}}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}}}$$

$$\textsc{TBreak}\ \frac{}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \mathsf{break} \rhd \mathrm{dom}(\Delta)_{\perp}; \Delta_{\mathbf{b}} \Cup \Delta; Tr_{\mathbf{f}}}$$

$$\textsc{TSwitchL}\ \frac{\begin{array}{c} \Delta \vdash_{C,\widetilde{D}} e : L \rhd \Delta'' \qquad L \in \mathcal{L}_{\widetilde{D}} \qquad \mathsf{noDup}(\widetilde{cbl}) \\ ncvs = \mathsf{noCaseVals}_{\widetilde{D}}(L, \widetilde{cbl}) \\ \Delta''; \mathrm{dom}(\Delta)_{\perp}; Tr_{\mathbf{f}} \vdash_{C,rt,L,ncvs,\widetilde{D}} \widetilde{cbl} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}} \end{array}}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \mathsf{switch}\ (e)\ \{\widetilde{cbl}\} \rhd \Delta' \Cup \Delta'_{\mathbf{b}}; \Delta_{\mathbf{b}}; Tr'_{\mathbf{f}}}$$

$$\textsc{TSwitchI}\ \frac{\begin{array}{c} \Delta \vdash_{C,\widetilde{D}} e : \mathsf{int} \rhd \Delta'' \qquad \mathsf{noDup}(\widetilde{cbl}) \\ \Delta''; \mathrm{dom}(\Delta)_{\perp}; Tr_{\mathbf{f}} \vdash_{C,rt,\mathsf{int},\emptyset,\widetilde{D}} \widetilde{cbl} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}} \end{array}}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \mathsf{switch}\ (e)\ \{\widetilde{cbl}\} \rhd \Delta' \Cup \Delta'_{\mathbf{b}}; \Delta_{\mathbf{b}}; Tr'_{\mathbf{f}}}$$

The TExp rule applies whenever an expression e is used as a statement. An expression is well typed if it type checks with the judgment for expressions and the resulting TC type is terminable. Notice that, using an expression as a statement implies that all possible objects created/returned by expression e are left behind and cannot complete their protocol. Therefore, ensuring that expression e is typed with a terminable TC type is crucial to guarantee protocol completion. Whenever we detect an expression used as a statement, we can safely discard conditional information (*i.e.,* transforming the unresolved typestate tree into a standard one), as we are certainly not in the context of if, while or switch conditions (where conditional information is, instead, required). Given that unresolved typestate tree only stems from method calls (which are considered as expressions e) and we

discard conditional type information generated by expressions used as statements, our type checking procedure has the following invariant property: at most one unresolved typestate tree can occur in type environments at any time.

The TBlock rule is used to analyse compound statements, *i.e.,* sequence of statements enclosed within curly brackets. Besides analysing the sequence, the rule also ensures that all the variables created within the scope of the compound statement are terminable and remove them from type environments. As a matter of fact, once we exited the scope of the compound statement, all the variables it defines are out of scope, *i.e.,* no longer accessible. Thus, ensuring termination is crucial for protocol completion while removing them from type environments allows the definition of new variables with the same identifier.

The TIf rule deals with if statements. It first analyses the expression e used as condition, then it evolves the type environment stemming from the analysis of e, *i.e.,* Δ', with labels true and false. The evolution of the type environments with the proper label is crucial to convert the unresolved typestate tree possibly generated by the analysis of e into a typestate tree suitable for type checking. The resulting type environments after the analysis of the if statement are computed merging those stemming from the analysis of the branches, using the dedicated operator. Notice that, after the analysis of the if statement, the unresolved typestate tree possibly generated by e is transformed into a standard one. Thus, the invariant property of our type checking procedure, *i.e.,* at most one unresolved typestate tree can occur in type environments at any time, still holds.

The analysis of the condition and the construction of the type environments to be used inside and outside the while loop body are the same as those described for if statements. In particular, the unresolved typestate tree possibly generated by the analysis of e is evolved with the proper label into a standard one. Thus, again, the invariant property of our type checking procedure.

The type checking of while statements requires careful reasoning. Since such syntactical construct has the ability to jump back to the beggining of the loop, we need to ensure that the type information at the end of the loop body is a subtype of the one before evaluating the condition. This is crucial to ensure soundness of loops. To understand why, let us consider the example in Listing 6.3. We consider classes A, B and C, with $B \leq_{\widetilde{D}} A$, with the following protocols:

Listing 6.3: Soundness of *while* loops

```
1  class Main {
2      A a;
3      B b;
4      C c;
5      void main() {
6          a = new B();
7          c = new C();
8          while(cond) {
9              b = (B) a;
10             b.m1();
11             a = c.play(b);
12         }
13     }
14 }
```

- $\mathsf{A1}^{E_A=\{\mathsf{A1}=_{\mathsf{drop}}\{m:\mathsf{A1}\}\}}$, for class A;
- $\mathsf{B1}^{E_B=\{\mathsf{B1}=\{_{\mathsf{drop}}\{m:\mathsf{B1},m1:\mathsf{B2}\},\mathsf{B2}=_{\mathsf{drop}}\{m:\mathsf{B1},m2:\mathsf{B2}\}\}\}}$, for class B. Notice that, according to Definition 7, we have $\mathsf{B1}^{E_B} \leq_{\mathbf{T}} \mathsf{A1}^{E_A}$ and $\mathsf{B2}^{E_B} \leq_{\mathbf{T}} \mathsf{A1}^{E_A}$;
- $\mathsf{C1}^{E_C=\{\mathsf{C1}=_{\mathsf{drop}}\{\mathsf{play}:\mathsf{C1}\}\}}$, for class C.

For brevity sake, we do not present the implementation of these classes. The only important implementation detail to know is that the method play of class C takes as parameter an object of class B in state $\mathsf{B1}^{E_B}$ and returns an object of class B in type $\mathsf{B1}^{E_B} \cup \mathsf{B2}^{E_B}$. During the type checking, after object creation (line 6), we associate to field a the typestate tree resulting from ucastTT, *i.e.,* $(A, \mathsf{A1}^{E_A}, \{(B, \mathsf{B1}^{E_B}, \{\})\})$. Thus, the field type environment at the beginning of the loop body is:

$$[\mathsf{a} \mapsto (A, \mathsf{A1}^{E_A}, \{(B, \mathsf{B1}^{E_B}, \{\})\}), \mathsf{b} \mapsto (B, \mathsf{Null}, \{\})].$$

Inside loop body, we associate to field b the typestate tree $(B, \mathsf{B1}^{E_B}, \{\})$, applying dcastTT (line 9). After calling $m1$ (line 10), the field b is associated to the typestate tree $(B, \mathsf{B2}^{E_B}, \{\})$. Since the method play takes as input an object of class B in state B2, the method call is legal. As a result, the field a is associated to $(A, \mathsf{A1}^{E_A}, \{(B, \mathsf{B1}^{E_B} \cup \mathsf{B2}^{E_B}, \{\})\})$ (obtained via ucastTT to the typestate tree returned by play). Thus, the type environment at the end of the loop body is:

$$[\mathsf{a} \mapsto (A, \mathsf{A1}^{E_A}, \{(B, \mathsf{B1}^{E_B} \cup \mathsf{B2}^{E_B}, \{\})\}), \mathsf{b} \mapsto (B, \mathsf{Null}, \{\})].$$

The subtyping among the initial and final type environments comes into play. If we do not ensure such a relation, the loop body is not sound: given the final type environment, if we were to execute the body again, the m1 method call would not be safe, as it is not allowed by the typestate tree of b, which is $(B, \mathsf{B1}^{E_B} \cup \mathsf{B2}^{E_B}, \{\})$ (in a union type a method call is allowed only if it is permitted by *all* elements). Instead, ensuring the subtyping among the initial and final type environments prevents unsound programs as the one in Listing 6.3 from compiling, reducing the likelihood of potential errors. Once we type checked the loop body, we need to carefully compute the resulting type environments. Those for fields, parameters and variables are evolved with the label false and merged with the type environments used to deal with the break. The first operation is required as the evaluation of the loop condition may produce an unresolved typestate tree (transformed into a typestate tree using false, *i.e.,* the value that causes the control flow to exit the loop). The second operation is required because the loop body can have break statements, which cause the control flow to immediately exit the loop. Merging the type environments of fields, parameters and variables with those to deal with break statements ensures that the code outside the loop is safe regardless of whether the loop prematurely terminates or reaches the end of its body. Notice that, the initial and final type environments $\Delta_{\mathbf{b}}$ remain unchanged after the analysis. This guarantees that break statements are confined within the scope of the while statement to which they belong and enables the detection of their misuse. Finally, the loop body can encounter a return statement: to account for the possibility of prematurely exiting the method and ensure a safe type checking process, we merge the field type environment for return, *i.e.,* $Tr'_{\mathbf{f}}$, with the initial one, *i.e.,* $Tr_{\mathbf{f}}$.

The TRet rule first ensures that the TC type assigned to expression e is a subtype of the statically declared one, *i.e.,* the rt appearing in the judgement, and ensures that all parameters are terminable. To perform such check, we need to explicitly apply the resolution operator on the parameters/variables type environment, since it may happen that expression e is a method call (not type checkable

with the TExp rule) and generates an unresolved typestate tree, which is not recognised as terminable. Notice that, if such scenario occurs, we cannot discard conditional type information: if the protocol continuation is a state $\widetilde{\langle l : u \rangle}^{\widetilde{E}}$, such conditional information is needed. Instead, if the protocol continuation is a typestate $u^{\widetilde{E}}$, we can safely discard the conditional type information (see TBr rule).

The TRet rule merges the current field type environment, *i.e.,* $T'_{\mathbf{f}}$ with $Tr_{\mathbf{f}}$ and assigns $\bot$ to the resulting field and parameter type environments (see $\mathrm{dom}(\Delta)_{\bot}$). Updating $Tr_{\mathbf{f}}$ and mapping fields and parameters to $\bot$ is crucial: it allows us to statically detect and catch a variety of errors in the TBr rule. If the method currently being analysed defines void as return type and includes a return statement in its body, the $Tr_{\mathbf{f}}$ is updated and does not map fields to $\bot$ anymore. If the method has a return type different from void and one or more execution path does not end with the return statement, the field type environment at the end of its body does not map all its elements to $\bot$.

The TVDecl and TVDeclO rules handle variable declarion for basic types and objects, respectively. The rules check that the variable we are declaring has not been defined already and the Java type of the expression e on the right-hand side is a subtype of the one assigned to the variable we are declaring. In case of basic types, since the analysis of expression e might generate an unresolved typestate tree, we have to resolve the type environments of fields, parameters and variables, since conditional type information is not needed. In case of objects, we first upcast the typestate tree stemming from the analysis of e. Notice that, we do not need to resolve type environments since unresolved typestate trees are generated only after method calls that return an enum or a bool type. Finally, in both rules, we add the newly declared variable to both the type environment of parameters and variables (mapped to its current TC type) and the type environment of parameters and variables dealing with break statements (mapped to $\bot$).

The TBreak rule is straightforward: whenever we encounter a break statement, we update the type environments of fields, parameters and variables with their current TC types and set all other environments but the one dealing with return to $\bot$. This mechanism prevents unreachable code from compiling and ensures that the code after exiting the scope of the break statement is safe.

To analyse switch statements, we use TSwitchL and TSwitchI. Intuitively, the

former is used to analyse switch statements whose condition is asssigned to an enum L, while the latter statements whose condition is typed with int. In the TSwitchL rule, we ensure that the enum L assigned to the condition is defined in the program $\widetilde{D}$, we check that cases have no duplicate constant values and build the set of unsed labels via $\mathsf{noCaseVals}_{\widetilde{D}}$ (see Definition 85). Finally, we analyse the sequence of case blocks with the dedicated judgment. Following the approach of while statements, we compute the resulting type environments merging the one stemming from the analysis of case blocks with the ones dealing with break and impose that $\Delta_{\mathbf{b}}$ remains unchanged (following the reasoning for while statements).

In the TSwitchI rule, we check that cases have no duplicate constant values and analyse the sequence of case blocks. As before, the resulting type environments is computed following the reasoning for while statements.

To analyse the sequence of case blocks, we need to define dedicated judgements. The first one deals with the sequence and it is of the form

$$\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} \widetilde{cbl} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}},$$

where tc is the type assigned to the expression in the condition of the switch statement and $ncvs$ is the set of unsused labels (necessary to type check the default case). The second one explicitly analyses single case blocks and it is of the form

$$\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} cbl \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}}.$$

We present in Definition 87 the rules to analyse sequences of case blocks.

Definition 87 (Typing rules for sequence of case blocks)**.**

$$\textsc{TEmptyCB}\ \frac{}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} \epsilon \rhd \Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}}}$$

$$\textsc{TSeqCB1}\ \frac{\begin{array}{c}\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} cbl \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}} \qquad \Delta' \neq \mathrm{dom}(\Delta)_{\perp} \\ \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} \widetilde{cbl} \rhd \Delta''; \Delta''_{\mathbf{b}}; Tr''_{\mathbf{f}} \\ \Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} \widetilde{cbl} \rhd \Delta'''; \Delta'''_{\mathbf{b}}; Tr'''_{\mathbf{f}}\end{array}}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} cbl\ \widetilde{cbl} \rhd \Delta'' \Cup \Delta'''; \Delta''_{\mathbf{b}} \Cup \Delta'''_{\mathbf{b}}; Tr''_{\mathbf{f}} \Cup Tr'''_{\mathbf{f}}}$$

$$\textsc{TSeqCB2}\ \frac{\Delta;\Delta_{\mathbf{b}};Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} cbl \rhd \mathrm{dom}(\Delta)_{\perp};\Delta'_{\mathbf{b}};Tr'_{\mathbf{f}} \quad \Delta;\Delta_{\mathbf{b}};Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} \widetilde{cbl} \rhd \Delta'';\Delta''_{\mathbf{b}};Tr''_{\mathbf{f}}}{\Delta;\Delta_{\mathbf{b}};Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} cbl\ \widetilde{cbl} \rhd \Delta'';\Delta'_{\mathbf{b}} \Cup \Delta''_{\mathbf{b}};Tr'_{\mathbf{f}} \Cup Tr''_{\mathbf{f}}}$$

The TEmptyCB rule is straightforward: it leaves type environments unchanged.

Whenever we analyse a switch statement, we cannot determine statically which case will execute, as the entry point depends on the runtime value of the condition. Therefore, in both TSeqCB1 and TSeqCB2 rules, we must analyse each element of the sequence using the type environments obtained after evaluating the condition.

The TSeqCB1 rule analyses sequences whose first element does not have all excution paths terminating with break or return, (checked with $\Delta' \neq \mathrm{dom}(\Delta)_{\perp}$). Consequently, we also have to analyse each case block using as initial type environment the one stemming from the analysis of the previous element, as we have sequential execution of cases. Finally, the resulting type environments is computed merging those from previous analyses.

The TSeqCB2 rule analyses sequences whose first element has all excution paths terminating with break or return. Here, it is enough to proceed as described above: we analyse each element of the sequence using the type environments obtained after evaluating the expression used as the condition in the switch statement. The resulting type environments dealing with break and return statements are computed merging those stemming from the analyses of the elements of the sequence.

To better understand how the type checking of the sequence of case blocks works, let us consider the example in Listing 6.4.

In the example, we assume that the method calls do not violate the protocol and the return type of m is a value in $L.\mathsf{vals}_{\widetilde{D}}$. To type check the sequence of case blocks, we apply the TSeqCB1 rule as in the first element of the sequence break does not occur. The rule first analyses the sequence propagating the resulting type environments from one element to another (as the flow sequentially moves from the first to the second case block). Next, the rule checks the remainder of the sequence using the type environments we had after evaluating the condition of the switch. This is necessary, since we cannot statically know the actual value returned by m and, consequently, the entrypoint in the case block sequence. The remainder of the sequence, *i.e.,* the second and third case blocks, is type checked with TSeqCB2 as

Listing 6.4: Typing sequence of case blocks

```
class Main {
    A a;
    void main() {
        a = new A();
        switch(a.m()) {
            case L.l1:
                a.m1();
            case L.l2:
                a.m2();
                break;
            case L.l3:
                a.m3();
                break;
        }
    }
}
```

the first element terminates with break. Differently from the previous case block, here, we do not need to analyse the remainder of the sequence, *i.e.,* the last case block, with the type environments stemming from the analysis of the first element, as the break makes us exiting the switch scope.

In Definition 88, we present the rules to type check single case blocks.

Definition 88 (Typing rules for case blocks)**.**

$$\textsc{TCaseL}\ \frac{\Delta \vdash_{C,\widetilde{D}} l : L \rhd \Delta \qquad \overset{l}{\widehat{\Delta}}; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \{\widetilde{bst}\} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}}}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,L,ncvs,\widetilde{D}} \mathsf{case}\ l :\ \widetilde{bst} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}}}$$

$$\textsc{TCaseI}\ \frac{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \{\widetilde{bst}\} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}}}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,tc,ncvs,\widetilde{D}} \mathsf{case\ intLit} :\ \widetilde{bst} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}}}$$

$$\textsc{TDefL}\ \frac{ncvs \neq \emptyset \qquad (\bigcup_{l \in ncvs} \overset{l}{\widehat{\Delta}}); \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \{\widetilde{bst}\} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}}}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,L,ncvs,\widetilde{D}} \mathsf{default} : \widetilde{bst} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}}}$$

$$\textsc{TDefI}\ \frac{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\widetilde{D}} \{\widetilde{bst}\} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}}}{\Delta; \Delta_{\mathbf{b}}; Tr_{\mathbf{f}} \vdash_{C,rt,\mathsf{int},\emptyset,\widetilde{D}} \mathsf{default} : \widetilde{bst} \rhd \Delta'; \Delta'_{\mathbf{b}}; Tr'_{\mathbf{f}}}$$

The TCaseL rule analyses case blocks whose label is typed with an enum type

and check that such enum is the same as the one in the switch condition (we keep track of such information within the judgement). We evolve type environments (as the switch condition may be a method call possibly generating an unresolved typestate tree) and we analyse the case block body as a compound statement.

The TCaseI rule simply analyses case blocks, whose label is typed with int, as a compound statement.

The TDefL rule type checks default case blocks belonging to a switch statement whose condition is typed with an enum. Since this case block is executed whenever no other cases match the value of the condition, the type environment to type check this case block is computed as follows (recall, in the type environments for fields, parameters and variables we may have an unresolved typestate tree): for each label in *ncvs*, *i.e.,* the set of unused labels in the other case blocks, we evolve the unresolved tree using such label; then, we merge all the resulting type environments with the corresponding operator. If *ncvs* is empty, the type checking process halts. Finally, we analyse the body of the case block as a compound statement.

The TDefI rule simply analyses default case blocks, belonging to a switch statement whose condition is typed with int, as a compound statement.

6.2.5 Typing Expressions

In this Section, we formalise and thoroughly present the type checking rules required to deal with all the possible expressions allowed in our language. Throughout these rules, we use Δ as shorthand for pair of type environments. Recall, Δ denotes $T_{\mathbf{f}}, T_{\mathbf{s}}$. Notice that, operators applied to pair of type environments behave as explained in the previous Section. As shown in Definition 39, the parameters of a method call are represented as an *ordered* sequence of expressions e. Thus, we first need to devise a mechanism to type check such sequences. Similarly to the rules dealing with other sequences of syntactical elements, we formalise, in Definition 89, dedicated rules that use of the following judgement

$$\Delta \vdash_{C,\widetilde{D}} e \rhd \Delta'.$$

Definition 89 (Typing rules for sequences of expressions)**.**

$$\textsc{TEmptyExp}\ \frac{}{\Delta \vdash_{C,\widetilde{D}} \epsilon \rhd \Delta}$$

$$\textsc{TSeqExp}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : tc \rhd \Delta' \qquad \widehat{\Delta'} \vdash_{C,\widetilde{D}} e' : tc' \rhd \Delta''}{\Delta \vdash_{C,\widetilde{D}} e : tc \ \ \widetilde{e' : tc'} \rhd \Delta''}$$

The TEmptyExp rule is straightforward: type environments remain unchanged.

The TSeqExp rule type checks the first element of the sequence and, separately, the remainder. Before analysing the remainder we resolve the type environments, as it may happen that the first element generated an unresolved typestate tree.

As shown in the user syntax (see Definition 39), expressions can be method calls, which modify the typestate tree of the receiver object. Thus, we define in Definition 90 the function to modify all the nodes in a typestate tree and in Definition 91 the function to modify the typestate based on the method called.

Recall, $\mathcal{JMS}$ is the set of Java method signatures ranged over by meta-variable jms.

Definition 90 (Evolve typestate tree)**.** *Given a typestate tree tt and a Java method signature jms, the function* $\mathsf{evoTTI} : \mathcal{TT} \times \mathcal{JMS} \to \mathcal{TT}_{unr}$ *is defined as follows.*

$$\mathsf{evoTTI}((C, t, tts), jms) = (C, \mathsf{evol}(t, jms), \bigcup_{tt \in tts} \mathsf{evoTTI}(tt, jms))$$

Definition 91 (Evolve type)**.** *Given a type t and a Java method signature jms, the function* $\mathsf{evol} : \mathcal{T} \times \mathcal{JMS} \to \mathcal{T}_{unr}$ *is defined as follows.*

$$\mathsf{evol}(t, jms) = \begin{cases} \mathsf{evol}(t_1, jms) \cup \mathsf{evol}(t_2, jms) & \text{if } t = t_1 \cup t_2 \\ \mathsf{evol}(t_1, jms) \cap \mathsf{evol}(t_2, jms) & \text{if } t = t_1 \cap t_2 \\ w^{\widetilde{E}} & \text{if } t = u^{\widetilde{E}} \wedge \exists w \,.\, jms : w \in \mathsf{unf}(u^{\widetilde{E}}) \\ \top_{\mathbf{t}} & \text{otherwise} \end{cases}$$

As methods can be marked as *anytime*, *i.e.,* methods callable regardless of the typestate of the receiver, we define in Definition 92 a predicate to recognise them.

Definition 92 (Anytime methods predicate)**.** *Given a program $\widetilde{D}$, a class C and a Java method signature jms, the predicate $\mathsf{anytime}_{\widetilde{\mathsf{D}}}(C, jms)$ holds if:*

- *exists a method M in C or its superclasses such that its Java signature is jms;*
- *M does not syntactically appear in the protocol of C in program $\widetilde{D}$;*
- *M can only perform read operations;*
- *M can only invoke anytime methods.*

Our language incudes the logical negation, *i.e.,* $!\,e$, whose analysis intuitively requires the inversion of the choices in states of the form $\langle \mathsf{true} : t' \;\; \mathsf{false} : t'' \rangle$. Thus, for convenience, in Definition 93 we define a specific environment operator to perform such inversion and its auxiliary functions (see Definitions 94 and 95).

Definition 93 (Invert type environment)**.** *Given a type environment T, the operator $!\,T : \boldsymbol{TypeEnv} \to \boldsymbol{TypeEnv}$ is defined as follows.*

$$\begin{aligned} !\,T \stackrel{\text{def}}{=} &\{cid \mapsto \mathsf{invertTT}(T(cid)) \mid \exists\, tt_{unr} \in \mathcal{TT}_{unr} \,.\, T(cid) = tt_{unr}\} \cup \\ &\{cid \mapsto T(cid) \mid \;\; \nexists\, tt_{unr} \in \mathcal{TT}_{unr} \,.\, T(cid) = tt_{unr}\}, \end{aligned}$$

where $cid \in \mathrm{dom}(T)$.

Definition 94 (Invert unresolved typestate tree)**.** *Given an unresolved typestate tree tt_{unr}, the function $\mathsf{invertTT} : \mathcal{TT}_{unr} \to \mathcal{TT}_{unr}$ is defined as follows.*

$$\mathsf{invertTT}((C, t_{unr}, tts_{unr})) = (C, \mathsf{invert}(t_{unr}), \bigcup_{tt_{unr} \in tts_{unr}} \mathsf{invertTT}(tt_{unr}))$$

Definition 95 (Invert unresolved type)**.** *Given an unresolved type t_{unr} the function $\mathsf{invert} : \mathcal{T}_{unr} \to \mathcal{T}_{unr}$ is defined as follows.*

$$\mathsf{invert}(t_{unr}) = \begin{cases} \mathsf{invert}(t'_{unr}) \cup \mathsf{invert}(t''_{unr}) & \text{if } t_{unr} = t'_{unr} \cup t''_{unr} \\ \mathsf{invert}(t'_{unr}) \cap \mathsf{invert}(t''_{unr}) & \text{if } t_{unr} = t'_{unr} \cap t''_{unr} \\ \langle \mathsf{true} : t'' \;\; \mathsf{false} : t' \rangle & \text{if } t_{unr} = \langle \mathsf{true} : t' \;\; \mathsf{false} : t'' \rangle \\ t_{unr} & \text{otherwise} \end{cases}$$

The language makes it possible to check equality among expressions, *i.e.*, $e == e'$. This construct may have e or e' being a method call and leave the receiver object in an unresolved type state tree, whose stored types are $\langle\widetilde{l : t}\rangle$. Since the equality check is typed to bool and may be used as condition of if or while statements, the types $\langle\widetilde{l : t}\rangle$ might need to be mapped to $\langle \mathsf{true} : t \;\; \mathsf{false} : t'\rangle$. We now consider the example in Listing 6.5 to better understand how the analysis of such construct works. In our example, we assume method m to leave object a in $\langle\widetilde{l : t}\rangle$ and e to be a generic expression.

Listing 6.5: Equality check example

```
1  class Main {
2      void main() {
3          A a = new A();
4          if (a.m() == e) {
5              a.m1();
6          }
7          else {
8              a.m2();
9          }
10     }
11 }
```

To compute the type information for the if statement, we proceed as follows. If e is not a label, we use the resolution operator (see Definition 60) to compute the type information for both branches. Otherwise, we map true to the evolution of $\langle\widetilde{l : t}\rangle$ using the label on the right-hand side of the equality check and false to the type resulting, merging those corresponding to the other labels. To manage the second scenario, we devise a dedicated environment operator in Definition 96 and its helper functions in Definitions 97 and 98.

Recall, **LNames** is the set of labels ranged over by meta-variable l.

Definition 96 (Pairify type environment)**.** *Given a type environment T and a label l, the operator $\overset{l}{\longmapsto} T$: $\boldsymbol{TypeEnv} \times \boldsymbol{LNames} \to \boldsymbol{TypeEnv}$ is defined as follows.*

$$\overset{l}{\longmapsto} T \overset{\text{def}}{=} \{cid \mapsto \mathsf{toPairTT}(T(cid), l) \mid \exists\, tt_{unr} \in \mathcal{TT}_{unr} \,.\, T(cid) = tt_{unr}\} \cup \{cid \mapsto T(cid) \mid \nexists\, tt_{unr} \in \mathcal{TT}_{unr} \,.\, T(cid) = tt_{unr}\},$$

where $cid \in \mathrm{dom}(T)$.

Definition 97 (Pairify unresolved typestate tree)**.** *Given an unresolved typestate tree tt_{unr} and a label l, the function* $\mathsf{toPairTT} : \mathcal{TT}_{unr} \times \boldsymbol{LNames} \to \mathcal{TT}_{unr}$ *is defined as follows.*

$$\mathsf{toPairTT}((C, t_{unr}, tts_{unr}), l) = (C, \mathsf{toPair}(t_{unr}, l), \bigcup_{tt_{unr} \in tts_{unr}} \mathsf{toPairTT}(tt_{unr}, l)).$$

Definition 98 (Pairify unresolved type)**.** *Given an unresolved type t_{unr} and a label l, the function* $\mathsf{toPair} : \mathcal{T}_{unr} \times \boldsymbol{LNames} \to \mathcal{T}_{unr}$ *is defined as follows.*

$$\mathsf{toPair}(t_{unr}, l) = \begin{cases} \mathsf{toPair}(t'_{unr}, l) \cup \mathsf{toPair}(t''_{unr}, l) & \text{if } t_{unr} = t'_{unr} \cup t''_{unr} \\ \mathsf{toPair}(t'_{unr}, l) \cap \mathsf{toPair}(t''_{unr}, l) & \text{if } t_{unr} = t'_{unr} \cap t''_{unr} \\ \langle \mathsf{true} : t \ \ \mathsf{false} : \bigcup\{t' \mid l' : t' \in \widetilde{l : t}\}\rangle & \text{if } t_{unr} = \langle l : t \ \ \widetilde{l : t}\rangle \\ t_{unr} & \text{otherwise} \end{cases}$$

Expressions can seamlessly occur on fields, parameters and variables. Therefore, we define functions to read and write TC types mapped to a given identifier, regardless of the nature of such identifier. Write operations are performed only for objects, as their types can evolve. To this aim, we define in Definitions 99 and 100 functions to read and write TC types from type environments.

Recall, **IdNames** is the set of identifiers ranged over by meta-variable *id*.

Definition 99 (Read TC type)**.** *Given a field flag F (f or ε), a program $\widetilde{D}$, class C, an identifier id and a pair of type environments T, T', the function* $\mathsf{lookup}_{F,\widetilde{D}} : \mathcal{C}_{\widetilde{D}} \times \boldsymbol{IdNames} \times (\boldsymbol{TypeEnv} \times \boldsymbol{TypeEnv}) \to \boldsymbol{TCType}$ *is defined as follows.*
$\mathsf{lookup}_{F,\widetilde{D}}(C, id, (T, T')) =$

$$\begin{cases} T(C.\mathsf{allF}_{\widetilde{D}}(id).id) & \text{if } id \in \mathrm{dom}(C.\mathsf{allF}_{\widetilde{D}}) \wedge F = f \\ T(C.\mathsf{allF}_{\widetilde{D}}(id).id) & \text{if } id \in \mathrm{dom}(C.\mathsf{allF}_{\widetilde{D}}) \wedge id \notin \mathrm{dom}(T') \wedge f = \varepsilon \\ T'(id) & \text{if } id \in \mathrm{dom}(T') \wedge F = \varepsilon \\ \bot & \text{otherwise} \end{cases}$$

In the $\mathsf{upd}_{F,\widetilde{D}}$ function, given a type environment $T = T' \cup \{cid \mapsto tc\}$, the shorthand notation $T\{tc'/cid\}$ stands for the type environment $T' \cup \{cid \mapsto tc'\}$.

Definition 100 (Write TC type)**.** *Given a field flag F (f or ε), a program $\widetilde{D}$,*

a class C, an identifier id, a typestate tree tt and a pair of type environments T, T', the function $\mathsf{upd}_{F,\widetilde{D}} : \mathcal{C}_{\widetilde{D}} \times \boldsymbol{IdNames} \times \mathcal{TT}_{unr} \times (\boldsymbol{TypeEnv} \times \boldsymbol{TypeEnv}) \rightarrow (\boldsymbol{TypeEnv} \times \boldsymbol{TypeEnv})$ *is defined as follows.*
$\mathsf{upd}_{F,\widetilde{D}}(C, id, tt, (T, T')) =$

$$\begin{cases} (T\{tt/C.\mathsf{allF}_{\widetilde{D}}(id).id\}, T') & \textit{if } id \in \mathrm{dom}(C.\mathsf{allF}_{\widetilde{D}}) \wedge F = f \\ (T\{tt/C.\mathsf{allF}_{\widetilde{D}}(id).id\}, T') & \textit{if } id \in \mathrm{dom}(C.\mathsf{allF}_{\widetilde{D}}) \wedge id \notin \mathrm{dom}(T') \wedge F = \varepsilon \\ (T, T'\{tt/id\}) & \textit{if } id \in \mathrm{dom}(T') \wedge F = \varepsilon \\ (\mathrm{dom}(T)_{\perp}, \mathrm{dom}(T')_{\perp}) & \textit{otherwise} \end{cases}$$

If the inputed identifier corresponds to both a field and a variable/parameter, we ignore the field version. Otherwise, we build its complex identifier, *i.e., $C.id$*, retrieving the class information from $\mathsf{allF}_{\widetilde{D}}$.

The field flag F is necessary to deal with accesses of fields via this and super keywords. Specifically, it ensures that the inputed id must be a field, forcing the $\mathsf{lookup}_{F,\widetilde{D}}$ and $\mathsf{upd}_{F,\widetilde{D}}$ functions to read/write from/to the field type environment.

Finally, we have the alias function to mark a reference as Shared (*i.e.,* it no longer owns the protocol) and ensuring a linear discipline.

Definition 101 (Aliasing linear type)**.** *Given a TC type tc, we define the function* $\mathsf{alias} : \boldsymbol{TCTypes} \rightarrow \boldsymbol{TCTypes}$ *is defined as follows.*

$$\mathsf{alias}(tc) = \begin{cases} \mathsf{alias}TT(tt) & \textit{if } tc = tt \\ tc & \textit{otherwise} \end{cases}$$

Definition 102 (Aliasing typestate tree)**.** *Given a typestate tree tt, the function* $\mathsf{alias}TT : \mathcal{TT} \rightarrow \mathcal{TT}$ *is defined as follows.*

$$\mathsf{alias}TT((C, t, tts)) = \begin{cases} (C, \mathsf{Shared}, \{\}) & \textit{if } t \leq_{\mathbf{T}} \mathsf{Shared} \\ (C, t, \{\}) & \textit{if } t \leq_{\mathbf{T}} \mathsf{Null} \\ (C, \top_{\mathbf{t}}, \{\}) & \textit{otherwise} \end{cases}$$

To type check expressions, we use a judgement of the form

$$\Delta \vdash^{F}_{C,\widetilde{D}} e : tc \rhd \Delta',$$

where F is a field flag (either f or ε) and tc is the type assigned to e. The absence of such flag from the judgement is a shorthand for $F = \varepsilon$. Notice that, in the rules presented in Definition 89 we omitted F, as its value is always ε.

The type checking rules for expressions are presented in Definition 103.

Definition 103 (Typing rules for expressions)**.**

$$\textsc{TNew}\ \frac{C' \in \mathcal{C}_{\widetilde{D}} \qquad \Delta \vdash_{C,\widetilde{D}} \widetilde{e : tc} \rhd \Delta' \qquad C'(\widetilde{pt}) = \mathsf{min}_{\widetilde{\mathsf{D}}}(\mathsf{cands}_{\widetilde{\mathsf{D}}}(C', \widetilde{tc})) \quad \forall i, 1 \leq i \leq |\widetilde{tc}| \,.\, tc_i \leq_{\mathbf{TC},\widetilde{D}} \mathsf{toTC}(pt_i)}{\Delta \vdash_{C,\widetilde{D}} \mathsf{new}\ C'(\widetilde{e}) : (C', C'.\mathsf{prot}_{\widetilde{D}}, \{\}) \rhd \Delta'}$$

$$\textsc{TUpdB}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : tc \rhd \Delta' \qquad tc \neq tt \qquad tc \leq_{\mathbf{TC},\widetilde{D}} \mathsf{lookup}_{F,\widetilde{D}}(C, id, \Delta')}{\Delta \vdash^{F}_{C,\widetilde{D}} id = e : tc \rhd \Delta'}$$

$$\textsc{TUpdO}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : tt' \rhd \Delta' \qquad tt = \mathsf{lookup}_{F,\widetilde{D}}(C, id, \Delta') \qquad \mathsf{term}(tt) \quad \mathsf{cl}(tt') \leq_{\widetilde{D}} \mathsf{cl}(tt) \qquad tt'' = \mathsf{ucastTT}(tt', \mathsf{cl}(tt))}{\Delta \vdash^{F}_{C,\widetilde{D}} id = e : \mathsf{alias}(tt') \rhd \mathsf{upd}_{F,\widetilde{D}}(C, id, tt'', \Delta')}$$

$$\textsc{TUpdT}\ \frac{\Delta \vdash^{f}_{C,\widetilde{D}} id = e : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{this}.id = e : tc \rhd \Delta'}$$

$$\textsc{TUpdS}\ \frac{C.\mathsf{sup}_{\widetilde{D}} = C' \qquad \Delta \vdash_{C',\widetilde{D}} \mathsf{this}.id = e : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{super}.id = e : tc \rhd \Delta'}$$

$$\textsc{TUpdExt}\ \frac{\Delta \vdash^{F}_{C,\widetilde{D}} id.id' : tc \rhd \Delta \qquad \Delta \vdash_{C,\widetilde{D}} e : tc' \rhd \Delta' \qquad tc' \leq_{\mathbf{TC},\widetilde{D}} tc}{\Delta \vdash^{F}_{C,\widetilde{D}} id.id' = e : tc' \rhd \Delta'}$$

$$\textsc{TUpdExtT}\ \frac{\Delta \vdash^{f}_{C,\widetilde{D}} id.id' = e : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{this}.id.id' = e : tc \rhd \Delta'}$$

$$\textsc{TUpdExtS}\ \frac{C.\mathsf{sup}_{\widetilde{D}} = C' \qquad \Delta \vdash_{C',\widetilde{D}} \mathsf{this}.id.id' = e : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{super}.id.id' = e : tc \rhd \Delta'}$$

$$\textsc{TCastB}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : b' \rhd \Delta' \qquad b \leq_{\mathbf{TC},\widetilde{D}} b' \vee b' \leq_{\mathbf{TC},\widetilde{D}} b}{\Delta \vdash_{C,\widetilde{D}} (b)\ e : b \rhd \Delta'}$$

$$\textsc{TUcastO}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : tt \rhd \Delta' \qquad \mathsf{cl}(tt) \leq_{\widetilde{D}} C}{\Delta \vdash_{C,\widetilde{D}} (C)\ e : \mathsf{ucastTT}(tt, C) \rhd \Delta'}$$

$$\textsc{TDcastO}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : tt \rhd \Delta' \qquad C \leq_{\widetilde{D}} \mathsf{cl}(tt)}{\Delta \vdash_{C,\widetilde{D}} (C)\ e : \mathsf{dcastTT}(tt, C) \rhd \Delta'}$$

$$\textsc{TCall}\ \frac{\begin{array}{c}\Delta \vdash_{C,\widetilde{D}} \widetilde{e : tc} \rhd \Delta' \qquad tt = \mathsf{lookup}_{F,\widetilde{D}}(C, id, \Delta) \\ rt\, m(\widetilde{pt}) = \mathsf{min}_{\widetilde{\mathsf{D}}}(\mathsf{cands}_{\widetilde{\mathsf{D}}}(\mathsf{cl}(tt), m, \widetilde{tc})) \\ \forall i, 1 \leq i \leq |\widetilde{tc}|\ .\ tc_i \leq_{\mathbf{TC},\widetilde{D}} \mathsf{toTC}(pt_i) \\ tt_{unr} = \mathsf{evoTTI}(tt, \mathsf{toJT}(rt\, m(\widetilde{pt}))) \qquad \top_{\mathbf{t}} \neq \mathsf{ty}(tt_{unr})\end{array}}{\Delta \vdash_{F,C,\widetilde{D}} id.m(\widetilde{e}) : \mathsf{toTC}(rt) \rhd \mathsf{upd}_{F,\widetilde{D}}(C, id, tt_{unr}, \Delta')}$$

$$\textsc{TAnyt}\ \frac{\begin{array}{c}\Delta \vdash_{C,\widetilde{D}} \widetilde{e : tc} \rhd \Delta' \qquad tt = \mathsf{lookup}_{F,\widetilde{D}}(C, id, \Delta) \\ rt\, m(\widetilde{pt}) = \mathsf{min}_{\widetilde{\mathsf{D}}}(\mathsf{cands}_{\widetilde{\mathsf{D}}}(\mathsf{cl}(tt), m, \widetilde{tc})) \\ \forall i, 1 \leq i \leq |\widetilde{tc}|\ .\ tc_i \leq_{\mathbf{TC},\widetilde{D}} \mathsf{toTC}(pt_i) \\ \mathsf{anytime}_{\widetilde{\mathsf{D}}}(\mathsf{cl}(tt), \mathsf{toJT}(rt\, m(\widetilde{pt}))) \qquad \mathsf{ty}(tt) \neq \mathsf{Null}\end{array}}{\Delta \vdash^{F}_{C,\widetilde{D}} id.m(\widetilde{e}) : \mathsf{toTC}(rt) \rhd \Delta'}$$

$$\textsc{TCallT}\ \frac{\Delta \vdash^{f}_{C,\widetilde{D}} id.m(\widetilde{e}) : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{this}.id.m(\widetilde{e}) : tc \rhd \Delta'}$$

$$\textsc{TCallS}\ \frac{C.\mathsf{sup}_{\widetilde{D}} = C' \qquad \Delta \vdash_{C',\widetilde{D}} \mathsf{this}.id.m(\widetilde{e}) : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{super}.id.m(\widetilde{e}) : tc \rhd \Delta'}$$

$$\textsc{TAnytM}\ \frac{\begin{array}{c}\Delta \vdash_{C,\widetilde{D}} \widetilde{e : tc} \rhd \Delta' \qquad rt\, m(\widetilde{pt}) = \mathsf{min}_{\widetilde{\mathsf{D}}}(\mathsf{cands}_{\widetilde{\mathsf{D}}}(C, m, \widetilde{tc})) \\ \forall i, 1 \leq i \leq |\widetilde{tc}|\ .\ tc_i \leq_{\mathbf{TC},\widetilde{D}} \mathsf{toTC}(pt_i) \qquad \mathsf{anytime}_{\widetilde{\mathsf{D}}}(C, \mathsf{toJT}(rt\, m(\widetilde{pt})))\end{array}}{\Delta \vdash_{C,\widetilde{D}} m(\widetilde{e}) : \mathsf{toTC}(rt) \rhd \Delta'}$$

$$\textsc{TAnytMT}\ \frac{\Delta \vdash_{C,\widetilde{D}} m(\widetilde{e}) : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{this}.m(\widetilde{e}) : tc \rhd \Delta'}$$

$$\textsc{TAnytMS}\ \frac{C.\mathsf{sup}_{\widetilde{D}} = C' \qquad \Delta \vdash_{C',\widetilde{D}} \mathsf{this}.m(\widetilde{e}) : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{super}.m(\widetilde{e}) : tc \rhd \Delta'}$$

$$\textsc{TNot}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : \mathsf{bool} \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} !\, e : \mathsf{bool} \rhd !\, \Delta'}$$

$$\textsc{TEq}\ \frac{e \neq l \qquad e' \neq l \qquad \Delta \vdash_{C,\widetilde{D}} e : tc \rhd \Delta' \qquad \widehat{\Delta'} \vdash_{C,\widetilde{D}} e' : tc' \rhd \Delta'' \qquad \mathsf{toJT}(tc) \leq_{\widetilde{D}} \mathsf{toJT}(tc') \vee \mathsf{toJT}(tc') \leq_{\widetilde{D}} \mathsf{toJT}(tc)}{\Delta \vdash_{C,\widetilde{D}} e == e' : \mathsf{bool} \rhd \widehat{\Delta''}}$$

$$\textsc{TEqL1}\ \frac{\Delta \vdash_{C,\widetilde{D}} e : tc \rhd \Delta' \qquad \Delta' \vdash_{C,\widetilde{D}} l : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} e == l : \mathsf{bool} \rhd \overset{l}{\longmapsto} \Delta'} \qquad \textsc{TEqL2}\ \frac{\Delta \vdash_{C,\widetilde{D}} l : tc \rhd \Delta \qquad \Delta \vdash_{C,\widetilde{D}} e : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} l == e : \mathsf{bool} \rhd \overset{l}{\longmapsto} \Delta'}$$

The TNew rule checks that the class we are instantiating is defined in program $\widetilde{D}$ and analyses the expressions passed as parameter to the class constructor. Then, given the types of the expressions passed as parameter, the rule looks for the correct constructor to invoke and checks that the passed parameters are subtypes of the expected ones. Finally, the rule is typed with a root typestate tree whose class is the one we are instantiating and the type is the initial typestate of its protocol.

The TUpdB and TUpdO rules (the former for basic types, the latter for objects) type check assignments. Both rules exploit the field flag to identify fields referenced via this and super keywords. The TUpdB rule is straightforward: we just check that the right-hand side is a subtype of the left-hand one. We type the assignment of basic types with the TC type of the right-hand side. The TUpdO rule: (*i*) analyses the expression on the right-hand side; (*ii*) retrieves the typestate tree associated to the *id* on the left-hand side and ensures its terminability (otherwise we would violate protocol completion); and (*iii*) checks that the Java class of the right-hand side expression is a subtype of the left-hand side. Finally, to compute the type associated to the assignment expression, we use the alias function.

The TUpdT and TUpdS rule handle assignments of fields referenced via this and super, respectively. The TUpdT removes the this keyword, sets the field flag and inductively resorts to either TUpdB or TUpdO. The TUpdS retrieves the superclass

information and resorts to the TUpdT rule.

The TUpdExt manages assignments of fields from external contexts, *i.e.,* accessed with f.f′. The rule type checks the access to the external field we are accessing (via dedicated rules presented later), the expression we are assigning and ensures the subtyping between the right-hand side type and the left-hand one.

The TUpdExtT and TUpdExtS rules manage assignments of fields from external contexts via this and super keywords, respectively. The TUpdExtT rule sets the field flag and resorts to the TUpdExt one; the TUpdExtS rule retrieves the superclass information and resorts to the TUpdExtT one, passing such class information.

The TCastB, TUCastO and TDCastO rules manage upcasts and downcasts. In case of basic types (TCastB), we just check the subtyping relation (in both directions) among the type of the expression we are casting and the target one. If the subtyping relation holds, the cast is legal and we assign to the expression the basic type specified as the cast target. In case of objects, we check the subtyping relation among the cast target and the Java class of the object we are casting. To type check an upcast (TUCastO), we type the expression with the result of ucastTT; otherwise, we type the expression with the result of dcastTT (TDCastO). Notice that, if the cast target corresponds to the Java class of the typestate tree root assigned to expression e, we can non deterministically apply either TUCastO or TDCastO. However, the resulting TC type does change.

The TCall rule manages method calls. We type check the expressions passed as parameter and retrieve the typestate tree associated to the receiver object. We look for the correct method to call and ensure that: (i) the type of the expressions passed as parameters are subtypes of the expected ones; and (ii) the current type stored in the typestate tree allows for such method call. Finally, we type the method call with the return type of the method we invoked.

The rule to type check anytime method calls, *i.e.,* the TAnyt rule, behaves exactly as the TCall one. However, instead of checking that the method call is allowed in the the current type stored in the typestate tree of the receiver object, it ensures that the method we are calling is anytime and that the receiver object is not null.

The TCallT and TCallS rules manage method call whose receiver is referenced via this and super keywords, respectively. The TCallT rule sets the field flag and

resorts to either TCall or TAnyt; the TCallS rule retrieves the superclass information and resorts to the TCallT one.

The TAnytM rule type checks method calls of the form $m(\widetilde{e})$. This rule performs the same checks done by the TAnyt one, except for the one to ensure that the receiver object is not null (as here the receiver object is implicitly this).

The TAnytMT and TAnytMS rules handle method calls of the form $\mathsf{this}.m(\widetilde{e})$ and $\mathsf{super}.m(\widetilde{e})$, respectively. The first resorts to the TAnytM rule; the second retrieves the superclass information and resorts to TAnytMT, passing such information.

The type checking of the logical negation operator is straightforward. In the TNot rule, we ensure that expression e is typed with the bool type and we apply the inversion operator.

The rules TEq, TEqL1 and TEqL2 analyse equality checks. The TEq rule analyses checks where neither the left-hand side nor the right-hand one are labels. This rule resolves the type environments stemming from the analysis of both expressions (as they can be method calls possibly generating unresolved typestate trees) and ensures that the types associated to the expressions are related by the corresponding subtyping relation. The TEqL1 and TEqL2 rules analyse check where the left-hand side or the right-hand side expression is a label l. These rule ensure that the tc types assigned to both sides of the equality check are equal and apply the operator defined in Definition 96 to the resulting type environments.

As formalised in our syntax, expressions e can be identifiers, thus in Definition 104 we present the rules to type check their accesses.

Definition 104 (Typing rule for identifiers)**.**

$$\textsc{TId}\ \frac{tc = \mathsf{lookup}_{F,\widetilde{D}}(C, id, \Delta)}{\Delta \vdash^{F}_{C,\widetilde{D}} id : tc \rhd \mathsf{upd}_{F,\widetilde{D}}(C, id, \Delta, \mathsf{alias}(tc))} \qquad \textsc{TIdT}\ \frac{\Delta \vdash^{f}_{C,\widetilde{D}} id : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{this}.id : tc \rhd \Delta'}$$

$$\textsc{TIdS}\ \frac{C.\mathsf{sup}_{\widetilde{D}} = C' \qquad \Delta \vdash_{C',\widetilde{D}} \mathsf{this}.id : tc \rhd \Delta'}{\Delta \vdash_{C,\widetilde{D}} \mathsf{super}.id : tc \rhd \Delta'}$$

$$\textsc{TIdExt}\ \frac{tt = \mathsf{lookup}_{F,\widetilde{D}}(C, id, \Delta) \qquad \exists jt\ id' \in \mathsf{cl}(tt).\mathsf{allF}_{\widetilde{D}}(id').\mathsf{fields}_{\widetilde{D}}\ .\ jt \neq C}{\Delta \vdash^{F}_{C,\widetilde{D}} id.id' : \mathsf{inittype}(jt) \rhd \Delta}$$

$$\textsc{TIdExtT}\ \frac{\Delta \vdash^{f}_{C,\widetilde{D}} id.id' : tc \rhd \Delta}{\Delta \vdash_{C,\widetilde{D}} \mathsf{this}.id.id' : tc \rhd \Delta}$$

$$\textsc{TIdExtS}\ \frac{C.\mathsf{sup}_{\widetilde{D}} = C' \qquad \Delta \vdash_{C',\widetilde{D}} id.id' : tc \rhd \Delta}{\Delta \vdash_{C,\widetilde{D}} \mathsf{super}.id.id' : tc \rhd \Delta}$$

The TId rule deals with accesses of identifiers, *i.e.,* read operations. The rule retrieves the type of the identifier we are accessing via $\mathsf{lookup}_{F,\widetilde{D}}$ and ensures that such identifier is typed with the result of alias. The use of alias is crucial to ensure a linear discipline, as it avoids that multiple references have the ability of modifying the protocol of a given object.

The TIdT and TIdS rules deals with accesses of fields via this and super keywords. The former sets the field flag and resorts to TId; the latter retrieves the superclass information and resorts to TIdT passing such information.

The TIdExt rule type checks accesses to externale fields. This rule first ensures that the *id* from which we access the external field is an object and then we retrieve the Java type of the accessed external field id' checking that such Java type is a basic one. The latter check is crucial: if we allowed id' to be an object, we could not guarantee that its usage follows the protocol attached to the class of id', since we do not have access to the field type environment of the class *id*, thus, we can neither know the current type assigned to id' nor modify it.

The TIdExtT and TIdExtS rules handle accesses of external fields via this and super. The former sets the field flag and resorts to TIdExt; the latter retrieves the superclass information and resorts to TIdExtT passing such information.

The last syntactical element to analyse are values v (recall, labels are values as well). Thus, in Definition 105, we introduce the typing rules for values.

Definition 105 (Typing rules for values)**.**

$$\textsc{TInt}\ \frac{v = \mathsf{intLit}}{\Delta \vdash_{C,\widetilde{D}} v : \mathsf{int} \rhd \Delta} \qquad \textsc{TDouble}\ \frac{v = \mathsf{doubleLit}}{\Delta \vdash_{C,\widetilde{D}} v : \mathsf{double} \rhd \Delta}$$

$$\textsc{TBool}\ \frac{v \in \{\mathsf{true}, \mathsf{false}\}}{\Delta \vdash_{C,\widetilde{D}} v : \mathsf{bool} \rhd \Delta} \qquad \textsc{TNull}\ \frac{}{\Delta \vdash_{C,\widetilde{D}} \mathsf{null} : (\bot_{\mathbf{C}}, \mathsf{Null}, \{\}) \rhd \Delta}$$

$$\textsc{TEnumVal}\ \frac{v = L.id \qquad L \in \mathcal{L}_{\widetilde{D}} \qquad id \in L.\mathsf{vals}_{\widetilde{D}}}{\Delta \vdash_{C,\widetilde{D}} v : L \rhd \Delta}$$

6.3 Discussion

This Chapter provided a formal and detailed exploration of the type checking procedure JaTyC implements, grounded in the theoretical framework we discussed so far. We defined our core language extending the work done by Bravetti *et al.* [BFG+20] with critical features such as variable declarations, aliasing, polymorphism, protocol inheritance and our typestate trees. These enhancements significantly broaden the range of programs expressible in our language and improve the flexibility and precision of our approach. Building upon this foundation, we systematically defined the rules governing the type checking for each syntactical construct, ensuring adherence to the expected type constraints. For every syntactic construct of our language, we provided a corresponding type checking rule. Each rule is specifically designed to statically ensure type correctness, thus reducing the likelihood of runtime errors. Moreover, our analysis, enhanced by typestates and typestate trees, is capable, not only of preventing null pointer exceptions at compile time, even in programs using polymorphism, but also of ensuring that a progras adhere to a prescribed behaviour. This is achieved by carefully tracking the flow of object typestates and their transitions throughout programs, enabling the type checker to detect potential null dereferences and misbehaviours, before they actually occur.

The objective of this Chapter is twofold. First, it seeks to precisely support the informal description provided earlier in Section 5.6, with formal type checking rules. This explanation goes beyond a superficial overview and goes into the internal workings of the checker, revealing the intricate process that occurs behind the scenes. Second, this Chapter aims to establish a solid foundation that underpins our implementation. By grounding our work in formal definitions and type checking rules, we, not only enhance the rigor of our approach, but also provide a framework that can be extended and applied to other programming scenarios involving typestate and protocol management.

Chapter 7

Conclusion

In conclusion, this Dissertation has tackled some of the inherent complexities of modern large-scale software systems, which are distributed, collaborative, and communication-centric by nature. As these systems form the backbone of critical infrastructure, addressing the challenges identified in the context of this Dissertation is paramount to maintaining the integrity of our digital world. The stakes are high: even minor deviations or misbehaviours in system components can lead to cascading failures, resulting in severe consequences *e.g.,* security breaches, reputational damage. To address these challenges, this Dissertation aimed at improving modern software engineering practices, such as Continuous Integration and Continuous Deployment/Delivery (CICD) and automatic adaptation. More precisely, this Dissertation has sought to address these gaps focusing on three key challenges:

- challenge C1 concerns the evaluation of system behaviour as a whole at early stages of software development, *i.e.,* at modelling level, fostering a development approach where DevOps teams can analyse the consequences of their choices early on, *e.g.,* during the system design phase;

- challenge C2 aims at introducing architectural reconfiguration approaches, leveraging *correct-by construction* orchestrations. Such orchestrations are built upon declarative specifications of, *e.g.,* component requirements, deployment constraints. Moreover, the knowledge of components behaviours is crucial to overcome the drawbacks of the state of the art service-level adaptation, *i.e.,* the approach of the Kubernetes horizontal Pod Autoscaler (HPA),

avoiding the "domino effect" of uncoordinated scaling;

- challenge C3 focuses on endowing CICD practices with the ability of catching misbehaviours due to wrong internal service interactions, *e.g.,* dereferencing null pointers or using objects wrongly.

We addressed C1 in Chapter 3, presenting a timed integrated modelling/execution language capable of precisely simulating system behaviour early at design phase, without the need for an actual implementation. The integration is fully realised thanks to our Timed SmartDeployer that, starting from declarative annotations of microservice features (*e.g.,* required resources, strong/weak ports), virtual machine properties (*e.g.,* provided resources) and deployment constraints (*e.g.,* microservice instances to deploy), it automatically synthesizes correct timed deployment orchestrations. In particular, these deployment orchestrations explicitly manage startup time and speed dynamically setting their values (the previous version of SmartDeployer statically assigns them to each DC instance, producing orchestrations with unexpected behaviours). To test the expressive power of our timed integrated modelling/execution language, we modelled a real-world microservice system, *i.e.,* the AcmeAir system. In particular, we picked the workload from [IPT23] and tested that the simulated system behaves as we expect. As we showed, our timed integrated modelling/execution language is capable of precisely reproducing the behaviour of the real-world system, enabling DevOps teams to analyse deployment strategies and their implications early in the software development process, ensuring informed decision-making before starting the actual implementation. Thus, the work done in Chapter 3 addresses challenge C1, enabling early-stage evaluation of system behaviour at the modelling level.

We dealt with C2 in Chapter 4 introducing two approaches to architectural reconfiguration based on deployment orchestrations. Our approaches leverage *correct-by construction* deployment orchestrations automatically synthesized starting from declarative specifications of, *e.g.,* component characteristics, deployment constraints. The first approach focuses on service autoscaling. In particular, we proposed an innovative proactive-reactive scaling algorithm working in two phases: (i) combines the signals coming from a predictive module (proactiveness) and the system monitor (reactiveness) to compute the target workload the system has to

handle; and (*ii*) replicates the microservice architecture under examination as a whole. Our algorithm has been tested through a series of benchmarks, conducted in both simulated and realistic environments. The first set of benchmarks showed that our reactive global scaling algorithm (without proactive capabilities), not only surpasses the performance of the reactive local one (the mainstream approach), but it also outperforms an enhanced version of the local scaling equipped with an oracle, *i.e.,* a perfect predictor. In the final set of benchmarks, we assessed the effectiveness of our proactive-reactive scaling algorithm in handling workloads that are particularly challenging to predict. The results testified the effectiveness of our approach, showing that it can maintain optimal performance even under unpredictable conditions. The second approach focuses on edge-cloud continuum service migration. In particular, we presented a microservice architecture designed specifically for deployment across the edge-cloud continuum, targeted at reducing communication latency. To do that, our architecture includes an orchestrator to migrate services according to user-defined policies. We empirically showed, via simulated and real-world execution, that our orchestrator is capable of significantly reducing communication latency exploiting the data locality principle, *i.e.,* moving code towards data is cheaper than vice versa. Our orchestration-based architectural reconfiguration approaches addresses challenge C2: they instill a target system behaviour, *i.e.,* architectural reconfiguration, exploiting deployment orchestrations built via declarative descriptions of component behaviour (*e.g.,* resource requirements, strong/weak dependencies). These correct-by construction orchestrations allow developers to focus only on the logic of the reconfiguration technique, without worrying about dependency management. Thus, the development of more complex approaches to architectural reconfiguration, possibly mixing service autoscaling and migration, is easier and less error-prone (due to wrong dependency management). Moreover, we showed that such knowledge, not only is useful for orchestration synthesis, but it is also crucial to build scaling approaches capable of overcoming the drawbacks of the state of the art service-level adaptation, *i.e.,* the "domino effect" of uncoordinated scaling.

Finally, we tackled C3 in Chapter 5, extending and enhancing our Java Typestate Checker (JaTyC). Specifically, our efforts are directed in two main areas: (*i*) we extended the state of the art of the typestate-based analysis to safely support

polymorphism; (*ii*) we enlarged JaTyC supported language with a new syntactical construct, *i.e.,* arrays of linear objects. To fully support polymorphism, we introduced the typestate tree data structure and a comprehensive set of functions designed to manage it (see Section 5.5), *i.e.,* ucastTT, dcastTT, evoTT and mrgTT. The theoretical work we devised is language agnostic, making it applicable across a spectrum of object-oriented programming languages. To validate its efficacy and expressiveness, we implemented it in Java, extending JaTyC to support casting operations at any points in protocols. In addition to the integration of polymorphism in typestate-based analyses, we extended the language JaTyC supports with arrays of linear (*i.e.,* typestate-endowed) objects. To realise such integration, we first extended its type system with a type specifically tailored for such arrays, then we completed the integration enhancing the type checking process, to include the new syntactical construct. In addition to the work done in Chapter 5, in Chapter 6 we complemented the JaTyC type checking process with formally defined type checking rules. To this aim, we defined a subset of the Java programming language serving as the syntactic basis for our analysis. This subset, inspired by Bravetti *et al.* [BFG$^+$20], extends it with critical features such as variable declarations, aliasing, polymorphism, protocol inheritance and our new data structure for typestates, *i.e.,* typestate trees. These enhancements significantly expand the range of programs expressible in our language with respect to Bravetti *et al.* [BFG$^+$20], offering greater flexibility and precision in addressing diverse programming challenges. By introducing these features, we created a foundation that allows developers to express complex constructs with clarity, while statically guaranteeing absence of null pointer exceptions. Building upon this solid foundation, we systematically defined a comprehensive set of rules governing the type checking process for every syntactical construct in the language. Each rule is meticulously designed to ensure that type constraints are adhered to, enabling the static verification of type correctness. This approach, not only reinforces the integrity of the code, but also minimises the potential for runtime errors, contributing to the reliability of applications built using our framework. For each syntactical construct within our language, we provided a corresponding type checking rule tailored to its specific requirements. These rules are instrumental in enforcing correctness at compile time, ensuring that violations are detected and resolved early in the devel-

opment process. The static nature of these checks significantly reduces the risk of runtime errors, creating a more predictable and secure development environment. Moreover, our analysis extends beyond traditional type systems by incorporating advanced concepts like typestates and typestate trees. These mechanisms enable the detection and prevention of null pointer exceptions at compile time, even in scenarios involving polymorphism. Developers are empowered to write code that is both expressive and secure, benefiting from a language that actively supports the prevention of common yet critical errors. The objective of formally describing the type checking process performed by JaTyC is to offer a more thorough and detailed understanding of what JaTyC does during the type checking procedure. This explanation goes beyond a superficial overview and thoroughly presents the internal workings of the checker, revealing the intricate process that occurs behind the scenes. Second, this Chapter aims to establish a solid theoretical foundation that underpins our implementation. By grounding our work in formal definitions and type checking rules, we, not only enhanced the rigor of our approach, but also provided a framework that can be easily extended and applied to other programming scenarios involving typestate and protocol management. We proved that our theoretical machinery, applicable to any statically typed object oriented language, can be implemented in JaTyC capable of analysing an expressive subset of Java. Notice that, our tool can be easily integrated within CICD pipelines, thus these Chapters addressed challenge C3.

Without any claim to be complete in any of the discussed matters, we believe that elevating flat component behavioural descriptions to a more structured abstraction can foster the development of more reliable, efficient and robust systems.

Bibliography

[ABB+16] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016.

[ÁCJ+16] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. Zephyrus2: On the fly deployment optimization using SMT and CP technologies. In Martin Fränzle, Deepak Kapur, and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings*, volume 9984 of *Lecture Notes in Computer Science*, pages 229–245, 2016.

[AGS+20] Cristiano Aguzzi, Lorenzo Gigli, Luca Sciullo, Angelo Trotta, and Marco Di Felice. From cloud to edge: Seamless software migration at the era of the web of things. *IEEE Access*, 8:228118–228135, 2020.

[AIB+22] Muhammad Abdullah, Waheed Iqbal, Josep Lluis Berral, Jorda Polo, and David Carrera. Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing*, 15(3):1448–1460, 2022.

[AKR19] Abeer Abdel Khaleq and Ilkyeun Ra. Agnostic approach for microservices autoscaling in cloud applications. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1411–1415, 2019.

[All70] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[Amaa] Amazon. AWS auto scaling. `https://aws.amazon.com/autoscaling/`.

[Amab] Amazon. Aws cloudwatch. `https://aws.amazon.com/it/cloudwatch/`.

[AMS+22] Mahin K. Atiq, Raheeb Muzaffar, Óscar Seijo, Iñaki Val, and Hans-Peter Bernhard. When ieee 802.11 and 5g meet time-sensitive networking. *IEEE Open Journal of the Industrial Electronics Society*, 3:14–36, 2022.

[Apa] Apache. Apache mesos. `https://mesos.apache.org`.

[ATWS24] Hussain Ahmad, Christoph Treude, Markus Wagner, and Claudia Szabo. Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures. *CoRR*, abs/2403.07909, 2024.

[BA05] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005*, pages 217–226, Portugal, 2005. ACM.

[BA07] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, pages 301–320, Canada, 2007. ACM.

[Bac24a] Antonioni Giovanni Bacchiani, Lorenzo. Global scaler. `https://github.com/giovaz94/global-scaler`, 2024.

[Bac24b] Antonioni Giovanni Bacchiani, Lorenzo. Global scaling platform. `https://github.com/giovaz94/custom-pipeline`, 2024.

[Bac24c] Lorenzo Bacchiani. ABS AcmeAir - Simulation code for the AcmeAir microservices architecture. `https://github.com/LBacchiani/acmeair_abs.git`, 2024.

[Bac24d] Lorenzo Bacchiani. ABS Migration Simulation - Simulation code fo the SEAWALL platform. `https://github.com/LBacchiani/migration-simulation.git`, 2024.

[Bac24e] Lorenzo Bacchiani. Global scaling platform executable model. `https://github.com/LBacchiani/global-scaling`, 2024.

[Bac24f] Lorenzo Bacchiani. Kubernetes smart deployer. `https://github.com/LBacchiani/k8s_smart_deployer`, 2024.

[Bar18] Jeff Barr. AWS auto scaling. https://aws.amazon.com/autoscaling/, 2018.

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[BBG+22a] Lorenzo Bacchiani, Mario Bravetti, Maurizio Gabbrielli, Saverio Giallorenzo, Gianluigi Zavattaro, and Stefano Pio Zingaro. Proactive-reactive global scaling, with analytics. In Javier Troya, Brahim Medjahed, Mario Piattini, Lina Yao, Pablo Fernández, and Antonio Ruiz-Cortés, editors, *Service-Oriented Computing - 20th International Conference, ICSOC 2022, Seville, Spain, November 29 - December 2, 2022, Proceedings*, volume 13740 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2022.

[BBG+22b] Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara. A java typestate checker supporting inheritance. *Sci. Comput. Program.*, 221:102844, 2022.

[BBG+24a] Lorenzo Bacchiani, Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro. Timed Smart-Deployer source code. `https://github.com/jacopoMauro/abs_deployer`, 2024.

[BBG+24b] Lorenzo Bacchiani, Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. Integrated timed architectural modeling/execution language. In Frank S. de Boer, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Eduard Kamburjan, editors, *Active Object Languages: Current Research Trends*, volume 14360 of *Lecture Notes in Computer Science*, pages 169–198. Springer, 2024.

[BBG+24c] Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara. Behavioural Up/down Casting For Statically Typed Languages. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:28, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BBG+25] Lorenzo Bacchiani, Mario Bravetti, Saverio Giallorenzo, Maurizio Gabbrielli, Gianluigi Zavattaro, and Stefano Pio Zingaro. Proactive-reactive microservice architecture global scaling. *J. Syst. Softw.*, 220:112262, 2025.

[BBLZ21] Lorenzo Bacchiani, Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. A session subtyping tool. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed*

Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings, volume 12717 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2021.

[BBP+22] Nicolò Bartelucci, Paolo Bellavista, Thomas W. Pusztai, Andrea Morichetta, and Schahram Dustdar. High-level metrics for service level objective-aware autoscaling in polaris: a performance evaluation. *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, pages 73–77, 2022.

[BdBdG17] Nikolaos Bezirgiannis, Frank S. de Boer, and Stijn de Gouw. Human-in-the-Loop Simulation of Cloud Services. In *ESOCC*, volume 10465 of *LNCS*, pages 143–158. Springer, 2017.

[BDCd95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119:202–230, 1995.

[BDD+20] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Patricia Lago, Manuel Mazzara, Victor Rivera, and Andrey Sadovykh, editors. *Microservices, Science and Engineering*. Springer, 2020.

[BdGJ23] Jelle Bouma, Stijn de Gouw, and Sung-Shik Jongmans. Multiparty session typing in java, deductively. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 19–27, France, 2023. Springer.

[BDMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[BFG+20] Mario Bravetti, Adrian Francalanza, Iaroslav Golovanov, Hans Hüttel, Mathias Jakobsen, Mikkel Kettunen, and António Ravara. Behavioural Types for Memory and Method Safety in a Core Object-Oriented Language. In *Asian Symposium on Programming Languages and Systems*, volume 12470 of *Lecture Notes in Computer Science*, pages 105–124, Japan, 2020. Springer.

[BGM+19] Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro. Optimal and automated deployment for microservices. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11424 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2019.

[BGM+20] Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro. A formal approach to microservice architecture deployment. In *Microservices, Science and Engineering*, pages 183–208. Springer, 2020.

[BH14] Stefan Blom and Marieke Huisman. The vercors tool for verification of concurrent programs. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 127–131, Singapore, 2014. Springer.

[BK21] Ataollah Fatahi Baarzi and George Kesidis. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 427–441, New York, NY, USA, 2021. Association for Computing Machinery.

[BKA11] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference,*

Lancaster, UK, July 25-29, 2011 Proceedings, volume 6813 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2011.

[BLV+19] André Bauer, Veronika Lesch, Laurens Versluis, Alexey Ilyushkin, Nikolas Herbst, and Samuel Kounev. Chamulteon: Coordinated auto-scaling of micro-services. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 2015–2025. IEEE, 2019.

[BNOW93] Andrew Birrell, Greg Nelson, Susan S. Owicki, and Edward Wobber. Network objects. In Andrew P. Black and Barbara Liskov, editors, *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, pages 217–230. ACM, 1993.

[BOP22] Grzegorz J. Blinowski, Anna Ojdowska, and Adam Przybylek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.

[Boy01] John Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exp.*, 31(6):533–553, 2001.

[BPS+22a] Lorenzo Bacchiani, Giuseppe De Palma, Luca Sciullo, Mario Bravetti, Marco Di Felice, Maurizio Gabbrielli, Gianluigi Zavattaro, and Roberto Della Penna. Low-latency anomaly detection on the edge-cloud continuum for industry 4.0 applications: the SEAWALL case study. *IEEE Internet Things Mag.*, 5(3):32–37, 2022.

[BPS+22b] Lorenzo Bacchiani, Giuseppe De Palma, Luca Sciullo, Mario Bravetti, Marco Di Felice, Maurizio Gabbrielli, Gianluigi Zavattaro, Roberto Della Penna, Corrado Iorizzo, Andrea Livaldi, Luca Magnotta, and Mirko Orsini. SEAWALL: seamless low latency cloud platforms for the industry 4.0. In *5th Conference on Cloud and Internet of Things, CIoT 2022, Marrakech, Morocco, March 28-30, 2022*, pages 90–91. IEEE, 2022.

[BZ07] M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Proc. of 6th Int. Symposium Software Composition, SC'07*, volume 4829 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2007.

[BZ09] Mario Bravetti and Gianluigi Zavattaro. On the expressive power of process interruption and compensation. *Mathematical Structures in Computer Science*, 19(3):565–599, 2009.

[Car96] Luca Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, 1996.

[CDSY17] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Logical Methods in Computer Science*, 13(2), 2017.

[CHLR23] Héléne Coullon, Ludovic Henrio, Frédéric Loulergue, and Simon Robillard. Component-based distributed software reconfiguration:a verification-oriented survey. *ACM Comput. Surv.*, 56(1), aug 2023.

[COE+20] Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *ACM Trans. Program. Lang. Syst.*, 42(3):14:1–14:82, 2020.

[Con23] Checker Framework Contributors. The Checker Framework, 2023.

[CR20] Nora Jones Casey Rosenthal. *Chaos Engineering*. O'Reilly Media, Inc., 1 edition, 2020.

[DCZZ12] Roberto Di Cosmo, Stefano Zacchiroli, and Gianluigi Zavattaro. Towards a Formal Component Model for the Cloud. In *SEFM 2012*, volume 7504 of *LNCS*, pages 156–171. Springer, 2012.

[Dd09] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: An overview. In Cosimo Laneve and Jianwen Su, editors, *Web Services and Formal Methods, 6th International Workshop,*

WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers, volume 6194 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2009.

[DF04] Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[dFCM23] Carlos Mão de Ferro, Tiago Cogumbreiro, and Francisco Martins. Shelley: A Framework for Model Checking Call Ordering on Hierarchical Systems. In *Proceedings of the 18th International Federated Conference on Distributed Computing Techniques (DisCoTec 2023), COORDINATION 2023*, volume 13908 of *Lecture Notes in Computer Science*, pages 93–114, Portugal, 2023. Springer.

[DGL+17a] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[DGL+17b] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In *PAUSE*, pages 195–216. Springer, 2017.

[DGS17] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017.

[DGVV12] Sourav Dutta, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. Smartscale: Automatic application scaling in enterprise clouds. In Rong Chang, editor, *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 221–228. IEEE Computer Society, 2012.

[Dij72] Edsger W. Dijkstra. The humble programmer, 1972. ACM Turing Award acceptance speech.

[dMZ19] Stijn de Gouw, Jacopo Mauro, and Gianluigi Zavattaro. On the modeling of optimal and automatized cloud application deployment. *Journal of Logical and Algebraic Methods in Programming*, 107:108 – 135, 2019.

[DMZZ14] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: A component model for the cloud. *Inf. Comput.*, 239:100–121, 2014.

[Doc] Docker. Docker swarm. `https://docs.docker.com/engine/swarm/`.

[FSZS18] Amina Fellan, Christian Schellenberger, Marc Zimmermann, and Hans D. Schotten. Enabling communication technologies for automated unmanned vehicles in industry 4.0. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 171–176, 2018.

[Fur14] Mark Furman. *OpenVZ essentials*. Packt Publishing Ltd, 2014.

[GCW19] Alim Ul Gias, Giuliano Casale, and C. Murray Woodside. ATOM: model-driven autoscaling for microservices. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 1994–2004. IEEE, 2019.

[GH99] Simon J. Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In *Proc. of Programming Languages and Systems (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999.

[GH05] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.

[Goo] Google. Scaling based on predictions. `https://cloud.google.com/compute/docs/autoscaler/predictive-autoscaling?hl=it`.

[Gro02] W. Grosso. *Java RMI*. O'Reilly Media, 2002.

[GTWA14] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, 2014.

[GVR+10] Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 299–312, Spain, 2010. ACM.

[Has] HashiCorp. Automate infrastructure on any cloud with Terraform. `https://www.terraform.io`.

[HBB17] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 1st edition, 2017.

[HF10a] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[HF10b] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[HFG+19] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[HHK02] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theoretical computer science*, 274(1-2):43–87, 2002.

[HLV+16] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

[Hoa09] Tony Hoare. Null References: The Billion Dollar Mistake, 2009. Presentation at QCon London.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[HYC16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 14–26. ACM, 2001.

[IPT23] Emilio Incerto, Roberto Pizziol, and Mirco Tribastone. μopt: An efficient optimal autoscaler for microservice applications. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2023, Toronto, ON, Canada, September 25-29, 2023*, pages 67–76. IEEE, 2023.

[ISH21] Md Hasan Ibrahim, Mohammed Sayagh, and Ahmed E Hassan. A study of how docker compose is used to compose multi-component systems. *Empirical Software Engineering*, 26:1–27, 2021.

[Iva17] Konstantin Ivanov. *Containerization with LXC*. Packt Publishing Ltd, 2017.

[JHS+10] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.

[JHS+12] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012.

[JI17] Dmitry Jemerov and Svetlana Isakova. *Kotlin in action*. Manning Publications Company, 2017.

[JRD21] Mathias Jakobsen, Alice Ravier, and Ornela Dardha. Papaya: Global Typestate Analysis of Aliased Objects. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP'21)*, pages 19:1–19:13, Estonia, 2021. ACM.

[JSP+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume

6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[JSTT15] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *J. Log. Algebraic Methods Program.*, 84(1):67–91, 2015.

[KDPG16] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J Gay. Typechecking protocols with Mungo and StMungo. In *Proc. of Principles and Practice of Declarative Programming (PPDP)*, pages 146–159. ACM, 2016.

[KDPG18] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Science of Computer Programming*, 155:52 – 75, 2018. Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2016.

[KHA19] Amir Karamoozian, Abdelhakim Hafid, and El Mostapha Aboulhamid. On the fog-cloud cooperation: How fog computing can address latency concerns of iot applications. In *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 166–172, 2019.

[KMKP21] Dragi Kimovski, Narges Mehran, Christopher Emanuel Kerth, and Radu Prodan. Mobility-aware iot applications placement in the cloud edge continuum. *IEEE Transactions on Services Computing*, pages 1–1, 2021.

[KML+20] Matthias Kovatsch, Ryuichi Matsukura, Michael Lagally, Toru Kawaguchi, Kunihiko Toumura, and Kazuo Kajimoto. Web of Things (WoT) Architecture. W3C recommendation, April 2020. https://www.w3.org/TR/wot-architecture/.

[KMND20] John D Kelleher, Brian Mac Namee, and Aoife D'arcy. *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. MIT press, 2020.

[Kob98] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.

[KPS93] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient Recursive Subtyping. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, USA, January 1993*, pages 419–428. ACM Press, 1993.

[KSSE22] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Accumulation analysis. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022*, volume 222 of *LIPIcs*, pages 10:1–10:30, Germany, 2022. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[KY04] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *Machine Learning: ECML 2004, 15th European Conference on Machine Learning, Pisa, Italy, September 20-24, 2004, Proc*, volume 3201 of *Lecture Notes in Computer Science*, pages 217–226. Springer, 2004.

[Lau19] Lorenzo De Lauretis. From monolithic architecture to microservices architecture. In Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ramezani Ivaki, and Nuno Laranjeiro, editors, *IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019*, pages 93–96. IEEE, 2019.

[LBT18] Bingfeng Liu, Rajkumar Buyya, and Adel Nadjaran Toosi. A fuzzy-based auto-scaler for web applications in cloud computing environments. In Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu, editors, *Service-Oriented Computing - 16th International Conference,*

ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings, volume 11236 of *Lecture Notes in Computer Science*, pages 797–811. Springer, 2018.

[LMTY21] Torgeir Lebesbye, Jacopo Mauro, Gianluca Turin, and Ingrid Chieh Yu. Boreas - A service scheduler for optimal kubernetes deployment. In Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik, editors, *Service-Oriented Computing - 19th International Conference, ICSOC 2021, Virtual Event, November 22-25, 2021, Proceedings*, volume 13121 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2021.

[LP07] Cosimo Laneve and Luca Padovani. The *Must* preorder revisited. In *Proc. of 18th Int. Conference Concurrency Theory, CONCUR'07*, volume 4703 of *Lecture Notes in Computer Science*, pages 212–225. Springer, 2007.

[LY16] Julien Lange and Nobuko Yoshida. Characteristic formulae for session types. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 833–850. Springer, 2016.

[MA20] Nicolas Marie-Magdelaine and Toufik Ahmed. Proactive autoscaling for cloud-native applications using machine learning. In *IEEE Global Communications Conference, GLOBECOM 2020, Virtual Event, Taiwan, December 7-11, 2020*, pages 1–7. IEEE, 2020.

[Mau15] Tony Mauro. Adopting microservices at netflix: Lessons for team and process design. https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/, 2015.

[MB21] Gabriele Proietti Mattia and Roberto Beraldi. Leveraging reinforcement learning for online scheduling of real-time tasks in the edge/fog-to-cloud computing continuum. In *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, pages 1–9, 2021.

[Mer14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), mar 2014.

[MGR21] João Mota, Marco Giunti, and António Ravara. Java typestate checker. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2021.

[MHN19] Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. Casting about in the dark: an empirical study of cast operations in java programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):158:1–158:31, 2019.

[Mic] Microsoft. Overview of autoscale in Azure. `https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview`.

[Mil93] Robin Milner. The polyadic π-calculus: a tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[MT18] Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proc. ACM Program. Lang.*, 2(OOPSLA):112:1–112:29, 2018.

[New15] Sam Newman. *Building microservices - designing fine-grained systems, 1st Edition*. O'Reilly, 2015.

[NNSN24] Joao Paulo Karol Santos Nunes, Shiva Nejati, Mehrdad Sabetzadeh, and Elisa Yumi Nakagawa. Self-adaptive, requirements-driven autoscaling of microservices. *2024 IEEE/ACM 19th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 168–174, 2024.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[NZDP21] Zeinab Nezami, Kamran Zamanifar, Karim Djemame, and Evangelos Pournaras. Decentralized edge-to-cloud load balancing: Service placement for the internet of things. *IEEE Access*, 9:64983–65000, 2021.

[OAS] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html`. Accessed on May, 2020.

[OMG11] OMG. Business Process Model and Notation (BPMN), Version 2.0. `http://www.omg.org/spec/BPMN/2.0`, January 2011.

[OPR95] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA (Common Object Request Broker Architecture)*. Prentice-Hall, Inc., USA, 1995.

[PAJ+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the*

ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, pages 201–212. ACM, 2008.

[PC01] Jens Palsberg and Pavlopoulou Chirstina. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, 2001.

[PCLH21a] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. GRAF: a graph neural network based proactive resource allocation framework for slo-oriented microservices. In Georg Carle and Jörg Ott, editors, *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021*, pages 154–167. ACM, 2021.

[PCLH21b] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. Graf: a graph neural network based proactive resource allocation framework for slo-oriented microservices. *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, 2021.

[PCLH24] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. Graph neural network-based slo-aware proactive resource autoscaling framework for microservices. *IEEE/ACM Transactions on Networking*, 2024.

[PVM20] Carlo Puliafito, Antonio Virdis, and Enzo Mingozzi. Migration of multi-container services in the fog to support things mobility. In *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 259–261, 2020.

[QCB18] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Comput. Surv.*, 51(4):73:1–73:33, 2018.

[QCZ+20] Tie Qiu, Jiancheng Chi, Xiaobo Zhou, Zhaolong Ning, Mohammed Atiquzzaman, and Dapeng Oliver Wu. Edge computing in industrial internet of things: Architecture, advances and challenges. *IEEE Communications Surveys Tutorials*, 22(4):2462–2488, 2020.

[Raw] Amir Rawdat. Testing the performance of nginx and nginx plus web servers. `http://tinyurl.com/a9n2n8wv`.

[Red24] Redis. Redis: In-memory data structure store. `https://redis.io`, 2024. Accessed: 2024-08-22.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[RV15] Björn Rabenstein and Julius Volz. Prometheus: A Next-Generation monitoring system (talk). Dublin, May 2015. USENIX Association.

[SK16] Fabrizio Soppelsa and Chanwit Kaewkasi. *Native docker clustering with swarm*. Packt Publishing Ltd, 2016.

[SPF+07] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy C. Bavier, and Larry L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pages 275–287. ACM, 2007.

[SWVDT21] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions. *IEEE Communications Surveys Tutorials*, 23(4):2557–2589, 2021.

[SY86] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[SZL+22] Jie Sun, Yi Zhang, Feng Liu, Huandong Wang, Xiaojian Xu, and Yong Li. A survey on the placement of virtual network functions. *Journal of Network and Computer Applications*, 202:103361, 2022.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994.

[TIRB22] Shreshth Tuli, Shashikant Ilager, Kotagiri Ramamohanarao, and Rajkumar Buyya. Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks. *IEEE Transactions on Mobile Computing*, 21:940–954, 2022.

[TS21] Doug. Tollefson and Andrew Spyker. Acme air sample and benchmark. `https://github.com/acmeair/acmeair`, 2021.

[USC+08] Bhuvan Urgaonkar, Prashant J. Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1:1–1:39, 2008.

[Vas11] Vasco T. Vasconcelos. Sessions, from types to programming languages. *Bull. EATCS*, 103:53–73, 2011.

[VK22] Juan Cruz Viotti and Mital Kinderkhedia. A benchmark of json-compatible binary serialization specifications. *CoRR*, abs/2201.03051, 2022.

[Wet20] Nicole Wetsman. Contact tracing app for england and wales failed to flag people exposed to covid-19. *The Verge*, 2020.

[Wil12] B. Wilder. *Cloud Architecture Patterns: Using Microsoft Azure*. O'Reilly Media, 2012.

[XLL+20] Dianlei Xu, Tong Li, Yong Li, Xiang Su, Sasu Tarkoma, Tao Jiang, Jon Crowcroft, and Pan Hui. Edge intelligence: Architectures, challenges, and applications. *arXiv: Networking and Internet Architecture*, 2020.

[XWL+24] Shuaiyu Xie, Jian Wang, Bing Li, Zekun Zhang, Duantengchuan Li, and Patrick C. K. Hung. Pbscaler: A bottleneck-aware autoscaling framework for microservice-based applications. *IEEE Transactions on Services Computing*, 17(2):604–616, 2024.

[XYGG18] Hansong Xu, Wei Yu, David Griffith, and Nada Golmie. A survey on industrial internet of things: A cyber-physical systems perspective. *IEEE Access*, 6:78238–78259, 2018.

[YCZ22] Guangba Yu, Pengfei Chen, and Zibin Zheng. Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. *IEEE Trans. Cloud Comput.*, 10(2):1100–1116, 2022.

[YLH+18] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the internet of things. *IEEE Access*, 6:6900–6919, 2018.