NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

VLADYSLAV MIKYTIV

BSc in Computer Science and Engineering

ACTORCHESTRA: A TOOL FOR CATCHING FAULTY SYMPHONIES

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

September, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

ACTORCHESTRA: A TOOL FOR CATCHING FAULTY SYMPHONIES

VLADYSLAV MIKYTIV

BSc in Computer Science and Engineering

Adviser: Carla Ferreira
Full Professor, NOVA University Lisbon

Co-adviser: Bernardo Toninho
Associate Professor, IST

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2025

ACTORCHESTRA: a tool for catching faulty symphonies

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.3.9) [63].

To my mum and dad.

Acknowledgements

First, I would like to thank Carla Ferreira for being my adviser since my third year. The knowledge and guidance I have received from her are truly invaluable. I am grateful for all the tips, her patience when things did not go as planned, her constant availability, and for being the incredible mentor that she is. She is the most professional and inspiring teacher I have had the pleasure to learn from and work with, and for that, I am immensely thankful.

To Bernardo Toninho, thank you for all your guidance, encouragement, and the freedom you gave me to explore my ideas and discover new paths, while always being there to support me. The lessons I learned and the discoveries I made under your mentorship are priceless. Working with you and experiencing your excellence as a teacher is something I will always cherish.

Thank you both for being my advisers, the best mentors I could have asked for.

I would also like to thank everyone that has been part of this journey, either simply watching me or living it with me: to my mother, Svitlana, for her unconditional support and for always rooting for me, even when things did not go according to plan, without her I would not be able to achieve half of the things I achieved; to my father, Mykola, for his important lessons about life and for never doubting me, even when he did not understand what I was doing most of the time; to my sister, Mariana, for ~~annoying me~~ supporting me and listening to my complaints when things got south; to my dog Tracy, for all the support sitting on my lap on late night coding sessions. To all of my friends, that always offered me invaluable support, company, and friendship. Without them this would not have been as fun as it was.

I would also like to dedicate this work to the heroes of my homeland, Ukraine, for fighting for freedom and life. Slava Ukraini!

Thank you all so much! I am forever grateful.

”

“When the last tree has been cut down, the last fish caught, the last river poisoned, only then will we realize we cannot eat money.”

— Cree Indian Proverb

Abstract

Software verification is a key principle for building and deploying reliable and correct software. Multiple verification techniques are employed successfully nowadays, such as theorem proving, model checking, and testing, each with strengths and trade-offs. These methods help ensure that systems behave as expected and meet reliability standards.

Runtime Verification is a verification technique that has gained popularity for its lightweight and complementary approach to ensuring assurance over a system under scrutiny. Unlike the other traditional methods, runtime verification leverages the ability to perform checks over the system while it is running, enabling the dynamic monitoring of a system's behaviour. It adds the capability to verify and alert the system whenever it begins to behave in an unexpected or undesirable manner, enhancing the overall reliability and correctness.

In this thesis, we addressed the problem of runtime verification in actor-based systems, a programming paradigm employed by languages such as `Erlang`. We define a specification language, WALTZ, that enables the specification of properties that span multiple actors and perform causal-aware verification. After designing and implementing the compiler for WALTZ by generating `Erlang` monitor executables, we implemented the orchestration needed to maintain causal relations between systems that follow the OTP guidelines by surgically injecting code into the system. With these two components, we built `ACTORCHESTRA`, a tool that allows users to specify properties and automatically monitor them at runtime.

We evaluated two case studies to demonstrate the range of the specification language and the tool. We also analysed the overhead introduced by the tool compared to the baseline version of the code and explained the trade-offs of adding an extra layer of safety to our programs.

Keywords: runtime verification · monitoring · actor-based programming model · property monitoring · monitorable properties

Resumo

A verificação de software é um princípio fundamental para a construção e implementação de software fiável e correto. Atualmente, várias técnicas de verificação são empregues com sucesso: prova de teoremas, verificação de modelos (model checking) e testes, cada uma com os seus pontos fortes e compromissos. Estes métodos ajudam a garantir que os sistemas se comportam conforme o esperado e cumprem os padrões de fiabilidade.

A Verificação em Tempo de Execução (Runtime Verification) é uma técnica de verificação que se tornou popular e conhecida pela sua abordagem leve e complementar, oferecendo garantias adicionais sobre um sistema em análise. Ao contrário dos métodos tradicionais, a verificação em tempo de execução tira partido da capacidade de realizar verificações enquanto o sistema está em funcionamento, permitindo a monitorização dinâmica do comportamento do sistema. Isto acrescenta a possibilidade de verificar e alertar sempre que o sistema começa a comportar-se de forma inesperada ou indesejada, reforçando assim a fiabilidade e correção global.

Nesta tese, abordámos o problema da verificação em tempo de execução em sistemas baseados em atores, um paradigma de programação utilizado por linguagens como o `Erlang`. Definimos uma linguagem de especificação, WALTZ, que permite a definição de propriedades que abrangem múltiplos actores e realiza uma verificação sensível à causalidade. Após conceber e implementar o compilador para WALTZ, gerando executáveis de monitores em `Erlang`, implementámos a orquestração necessária para manter as relações de causalidade entre sistemas que seguem as diretrizes OTP, através da injeção cirúrgica de código no sistema. Com estes dois componentes, construímos o `ACTORCHESTRA`, a ferramenta que permite ao utilizador especificar propriedades e monitorizá-las automaticamente em tempo de execução.

Realizámos uma avaliação com base em dois estudos de caso, demonstrando o alcance da linguagem de especificação e da ferramenta, e analisámos a sobrecarga causada pela ferramenta em comparação com a versão original do código, explicando os compromissos associados a ter uma camada adicional de segurança nos nossos programas.

Palavras-chave: verificação em tempo de execução · monitorização de propriedades · propriedades monitorizávies · modelo de atores

Contents

List of Figures

Listings

List of Listings

Chapter 1

Introduction

Ensuring the production of correct, reliable and safe software is crucial for developing robust and safe software systems. This requirement has become even more critical as software increasingly impacts critical areas such as healthcare, transportation, and finance. To address the need for thorough verification of software behaviour and correctness, formal methods [32] have emerged.

Formally ensuring the correctness of a given piece of software can be performed by several techniques adopted, such as *theorem proving* [20], *model checking* [33], and *testing* [64], each with its strengths and trade-offs. Theorem proving is a static formal verification technique that uses mathematical logic to rigorously prove the correctness of software, ensuring it meets its specifications and is free from critical errors. One might use the classical pen-and-paper approach, but specialised software has been developed to assist and automate these proofs using computational proof assistants; one well-known example is the `Coq` [20] proof assistant.

Model checking is an alternative that attempts to automate the process of verifying software. It explores the system states for a given model to ensure they meet the specified properties, detecting errors early in the development process. It is also a static technique, providing the advantage of not adding overhead to the system it is analysing.

Testing, in contrast to the previously mentioned methods, is a dynamic approach to verifying software correctness. It typically involves selecting a finite set of input-output sequences, known as a test suite [37], to exercise the program. By comparing the outputs given by the system against the expected outcomes defined in the test suite, testing methods determine whether the software behaves as intended.

Theorem proving provides the highest safety assurance, albeit at the cost of being time-consuming, cumbersome, and requiring a high level of expertise. By contrast, model checking provides an automated way of exploring the state space, although it is often

constrained by the state explosion problem [58], limiting its applicability to large-scale systems. On the other hand, testing employs diverse methods to check that the program's outputs meet the expected results. However, it cannot definitively guarantee the absence of errors, as Dijkstra observed, "Testing shows the presence of bugs, not their absence".

1.1 Motivation

To address the limitations of the previously discussed methods, a new technique emerged called *runtime verification* [11, 17, 66, 58]. Runtime Verification (RV) studies methods that analyse a computation system's dynamic behaviour with several application scenarios. It can complement other techniques, providing an extra level of assurance for software, enabling us to verify and prevent failures by leveraging its runtime benefits [42]. RV can go beyond simple verification and might steer the system with corrections when there is unwanted behaviour. Additionally, it can be applied to collect information on a given running system for posterior analysis [41].

At its core, an RV approach begins with a running system and involves defining a set of properties that need to be verified. The novel aspect of this approach is that it automatically synthesises a monitor from these properties, typically expressed in a formal specification language like LTL (Linear Temporal Logic) [67]. This monitor operates alongside the system to verify whether the specified properties are met or not.

Instrumentation techniques manage the communication logic between the monitor and the system, enabling interaction with the system under scrutiny (SUS). By analysing the system's behaviour, the monitor determines whether it aligns with the defined correctness properties or deviates from them, ultimately delivering a verdict based on what the monitor observed from the execution trace produced by the SUS.

As a lightweight method that requires low computational resources, RV may serve as an alternative to the verification techniques previously discussed. Another advantage of RV is its ability to account for specific behaviours that occur only during runtime, which can often be unpredictable. Other methods may not adequately address this unpredictability.

Since RV examines the current run of a system, it can detect such unexpected behaviour in real-time and respond appropriately, either by alerting the system that something has gone wrong or applying a more refined corrective measure, like guiding the system onto the correct path.

The overall aim of RV is to ensure that a system's execution complies with its correctness properties as it unfolds through time. Achieving this in practice, however, is far from trivial. It requires several interdependent steps: systematically observing the system's behaviour, interpreting raw execution data, and evaluating it against the specified properties. When violations are detected, the monitor must raise timely alerts and support appropriate responses, such as triggering safeguards, initiating recovery, or adapting future behaviour. Each of these stages introduces its own technical and practical challenges, making RV both a powerful and complex verification approach.

1.2 Challenges

Numerous works have been developed in the field of RV, covering topics such as defining specification languages to express the properties to be monitored, developing instrumentation processes that determine how the monitor and the system operate together, designing algorithms to generate monitors from given specifications, investigating which properties can be monitored, and analysing the impact of RV on the system. A key drawback of RV is the potential overhead it may impose, which can degrade system performance.

One of the central challenges in RV lies in defining what exactly constitutes a trace [41]. A trace is the raw material from which properties are checked. However, its scope and granularity can vary greatly: should it capture low-level events, state transitions, or higher-level interactions? Most existing RV frameworks resolve this by relying on log files with a predefined syntax, which allows them to parse, analyse, and reason about system executions. While effective in controlled settings, this approach often imposes rigid constraints on how systems must expose their behaviour and can be brittle when dealing with highly concurrent or distributed environments.

An alternative perspective emerges from the actor-based model [4], a foundational paradigm followed by languages such as `Erlang` [73] and `Elixir` [38]. In this model, computation is structured around independent actors, each encapsulating its own state, and communicating solely via asynchronous message passing. This communication structure provides a natural and semantically rich notion of a trace: the stream of messages exchanged between actors. Unlike static logs, these message flows directly reflect the causal structure of concurrent executions, making them particularly well-suited for RV. By observing interactions at this level, it becomes possible not only to reconstruct execution traces more faithfully but also to reason about correctness properties over the real execution trace of the system.

Verifying these concurrent systems for formal correctness with static verification techniques, such as model checking, can be challenging due to the state explosion problem. RV makes it possible to verify these systems on the fly by analysing their current execution.

Nevertheless, applying RV in such asynchronous settings introduces significant challenges. A primary challenge is the inherent delay in detecting violations, as the monitor operates externally to the system, making it susceptible to becoming "out of sync" with ongoing executions. This misalignment is particularly problematic when a violated property results in a critical failure, where timely intervention is crucial. A second challenge arises when scaling beyond a small number of actors. As the number of components grows, so too does the complexity of the traces, which must account for a vast space of possible interleavings and message orderings. Extracting meaningful information from such traces quickly becomes cumbersome, complicating the monitoring process. The key question, then, is how to instrument an RV framework in a way that minimises detection latency while being able to capture these complex interactions on the go.

1.3 Contributions

The contributions of this dissertation address fundamental challenges in applying RV to concurrent, actor-based systems. The first contribution is the design of WALTZ, a specification language that aligns directly with the semantics of the actor model. WALTZ enables users to define properties over chains of interactions between processes, capturing relevant messages and correlating them through variables that span entire causal chains. WALTZ allows for the specification of complex correctness properties that track how data flows and transforms across multiple interactions between actors.

A distinguishing feature of WALTZ is its context-aware semantics, where properties are evaluated within the causal context of each message rather than a shared global context. This design choice prevents the unintended mixing of messages from different causal chains, ensuring that verification occurs in the correct context, a critical requirement for concurrent systems. By embedding causality directly into the language semantics, WALTZ enables both localised behavioural checks and system-wide invariants to be expressed naturally, without the user needing to disentangle concurrent execution paths manually.

Building on this foundation, we developed a compiler that translates WALTZ specifications into executable `Erlang` monitors. The compilation process leverages the close alignment between WALTZ's semantics and the behaviour of the underlying actor system. The generated monitors are scalable, capable of tracking distinct message chains independently and evaluating properties within their correct causal context. This design ensures that each chain is verified without interference from concurrent events, with monitors producing verdicts as soon as they detect either a violation or satisfaction of the specified property. By leveraging WALTZ's causal semantics and variable correlation mechanisms, these monitors can track complex interaction patterns across multiple actors while preserving the contextual information required for accurate verification.

A further contribution is the design and implementation of the `conductor`, a causality-tracking entity that enriches system traces with contextual information before the monitors process them. Given that many actor-based systems in practice rely on the Open Telecom Platform (OTP) [8, 29, 50] framework for structuring concurrent applications, our entity correlates messages belonging to the same causal chain for client-server system architectures following OTP conventions. This entity enables seamless property verification by the monitor, eliminating the need to consider all possible interleavings that can occur in concurrent and asynchronous `Erlang` systems.

Together, WALTZ and `ACTORCHESTRA` enhance runtime verification by enabling context-aware, multi-actor verification, advancing the state of the art in actor-based system verification. The complete framework is publicly available online at `ACTORCHESTRA`. Finally, the significance of this work has been acknowledged by the research community, as evidenced by the acceptance of the WALTZ design for publication at the INFOrum 2025 conference.

1.4 Document Outline

This document begins by introducing the background knowledge of runtime verification in Chapter 2, which will be used as a pedagogical chapter to explain our perspective on runtime verification, as there are many different interpretations in the literature. We present the global concept of runtime verification, along with some of its key steps and challenges that must be addressed in general, as well as in our specific case. In the background, we also present how `Erlang` systems behave and the explanation of OTP behaviour. The novelty of our work is presented in Chapter 3 and Chapter 4, where we demonstrate our specification language, the compilation process, and the distributed management of causality that we have developed. In Chapter 5, we demonstrate the impact of our implementation and approach through case studies developed to test the tool's usage. Then, in Chapter 6, we compare our specification language and tool with other existing work in the literature, showcasing the advancements made in the state of the art. Finally, in Chapter 7, we present our conclusions and discuss potential paths of further research and improvement.

Chapter 2

Background

This chapter will introduce the foundational knowledge necessary to understand runtime verification. We will start by defining classical terminology, including specification languages and monitors. We will clarify what it means for a process to be monitorable, followed by discussing various deployment and instrumentation techniques that can be utilized. Additionally, we will explore how the actor-based model can be leveraged within runtime verification and why it is a good abstraction for runtime verification.

2.1 Runtime Verification

Runtime verification (RV) is a field in formal methods that deals with the study, orchestration, and application of techniques that grant one the ability to verify if a given run of a system under scrutiny (SUS) satisfies or violates a given behaviour [58].

RV operates on the execution trace of a running system, verifying formal properties with the traces of events incrementally generated by the system; this makes it distinguishable from other approaches [42].

The essence of RV lies in generating a monitor from a high-level specification language that outlines the properties to be monitored during runtime; essentially, the properties that the SUS must adhere to [11]. This process automatically synthesises a monitor based on a correctness property, denoted as ϕ, described in a specification language. The generated monitor is then employed to observe either the ongoing execution of the system or a recorded history of its execution to check whether it adheres to property ϕ.

In more formal terms, if $\mathcal{L}(\phi)$ represents the set of valid executions defined by the property ϕ, then runtime verification can be boiled down to verifying whether the current execution, denoted as w, is part of that set, $w \in \mathcal{L}(\phi)$ [17]; this is known as the word problem.

Runtime verification revolves around three core questions: how to formally define desired and undesired behaviours, how to synthesise executable monitors from high-level specifications, and how to instrument these monitors with the system under scrutiny efficiently. Addressing these questions is essential for building any effective RV framework.

2.1.1 Terminology

In RV, terminology is often used broadly, and definitions can vary significantly depending on the context. The terminology for concepts like monitorability varies widely, with differing interpretations despite attempts at standardization [11, 58, 57, 43]. To ensure clarity and simplicity in this part of the document, we will introduce some basic notions of RV in straightforward terms, as presented in [41]. A more detailed discussion and precise definitions will follow in subsequent sections, depending on the level of detail needed in those contexts. In Figure 2.1 (adapted from [11]), we can see an overview of these concepts and the main structure of RV frameworks.

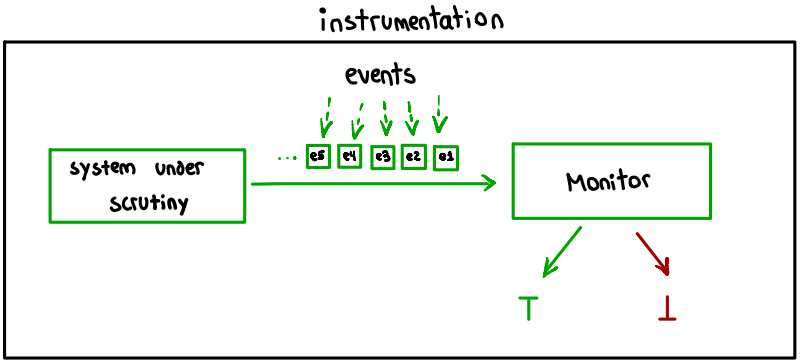


Figure 2.1: A basic RV setup

An event refers to a produced action of a system that can be observed. A trace is a sequence of events generated by the system. A trace is an abstraction of a single system run, represented as a finite sequence of events. Although a system could run indefinitely and produce an infinite trace, RV works with finite traces. Therefore, a trace in this context will be a prefix of the infinite trace the system is producing.

A property is intuitively something that one expects the SUS to adhere to. These properties can be formally defined using a specification language, which provides a concrete property description. More rigorously, a property can be seen as a set of traces, representing a partition of the set of all possible traces of the system, as described by Falcone et al. [41]. A specification explicitly describes a property that is defined using a specification language. This language defines the expected behaviour of the system. A specification language is a formalism that describes a property, such as regular expressions or automata, each providing a different approach to specifying the same underlying behaviour.

A monitor is a runtime entity automatically derived from a formal property specification. Executing in parallel with the system, it observes relevant events and determines

whether the property is satisfied or violated during execution. It reads the system's trace incrementally and produces a verdict, alerting the SUS or, in some cases, even steering it when a property violation occurs. A verdict is the monitor's result, usually expressed as a value from a given truth domain. The simplest form of a verdict is a binary value, such as true or false. However, as explored further in the document, truth domains may allow for additional verdict values beyond these two.

Instrumentation is the computational link connecting the system's execution with the monitor's analysis. It involves extracting traces from the system and feeding them into the monitor. It also addresses how the system and monitor are deployed and run in parallel [11].

Example 1. This chapter introduces a simple chat room system as the running example. The system can be instrumented to capture both the events it produces and the messages exchanged within the chat. These events need not be limited to primitive actions but may also carry contextual information, such as the specific chat room in which a user posts a message. On this basis, we can define several correctness properties to govern the system's behaviour. For instance, a user must `login` before entering a chat room; once inside, the user may `post` messages; and before logging out, the user must first `leave` the room. Taken together, these rules characterise the intended session behaviour of the system. Such behaviour can be formally expressed in a specification language, for example, through automata, as illustrated in Figure 2.2.

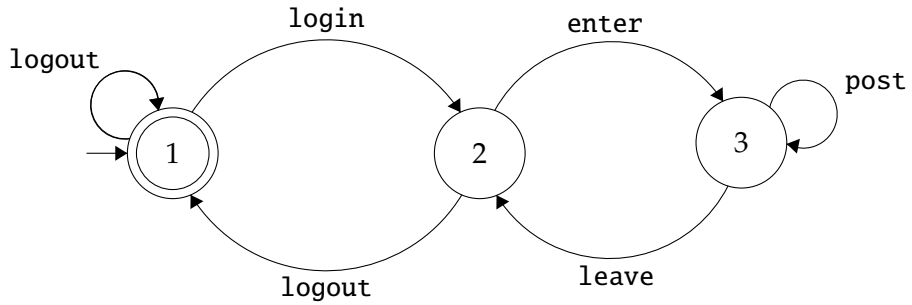


Figure 2.2: Chat room automata

2.2 Specification of a System's Behaviour

A system's behaviour refers to how it operates and responds to specific external requests or stimuli. These triggers cause the system to process the input, resulting in changes to its internal state over time, and produce outcomes or effects which may also influence how the system behaves in future interactions.

The behaviour of a system is something that one must take into account when working with it or, in this case, verifying it. Besides that, it is important to know how that behaviour is represented to interpret what the given system is doing [11].

There are several methods to describe the behaviour of a given system, including automata, regular expressions, grammars, and various types of logic. Even a hand-drawn diagram can effectively represent a system.

The different possibilities to describe a system lead to important questions: What is the correct way to describe a system? What properties do we want to verify at runtime, and can they be expressed in a given specification language?

This section explores the system's behaviour at varying levels of detail, providing deeper insight and reinforcing key definitions introduced in Section 2.1.1.

In this document, we adopt the definition of a run given by Leucker [58], namely, as a possibly infinite sequence of system events. The sequence may be infinite if the system executes indefinitely; however, when execution traces are recorded into a log file and analysed offline by a monitor, the resulting trace is finite.

Here, an execution of a system is understood as a finite prefix of a run, and is therefore referred to as a finite trace. RV primarily operates over such finite traces, analyzing executions derived from an ongoing run. While the run itself is expected to satisfy the specified property in its entirety, it is these finite executions that form the practical subject of analysis for monitors.

Event. An observable action performed by a system [11]. If we consider Σ the alphabet of the system then each element of Σ is an event. For example, if $\Sigma = \{ev_1, ev_2, ev_3\}$ then a single event is ev_1.

Events can be described at different levels of abstraction, be about internal or external behaviour; the information they carry may have information about the whole system or one specific part. The observable events to the exterior might even be a subset of the alphabet of a given system because we might be only interested in monitoring a specific part of the system [11]. The choice of how we represent these events and the associated information is all part of the specification process and will depend on the goal in mind.

For instance, in Figure 2.2 an event could be `enter` but we could refine the information present on that event by using a parametric approach and have information attached to the payload on the event stating what is the name of the chat room being entered, such as `enter(channel_one)`. As presented in Section 2.7, the actor-based model has an excellent abstraction for events. An event is a message exchanged between two actors and can carry metadata inside it for further analysis from the monitor.

Trace. A trace is a sequence of events produced by the system. In RV, traces instrumented to the monitor are always finite sequences. Therefore, we can think of a trace as a prefix of the run of a given system since that run might be infinite and produce an infinite growing trace. The trace the monitor will analyse is a prefix of that infinitely growing trace of the system. For instance, if $\Sigma = \{a, b, c, d\}$ is our alphabet of observable events, then $\sigma = aabcd$ is a trace produced by the system.

Upon these traces, our monitor will dictate the verdicts about given properties. The notion of trace is rather tricky in RV, as presented in [41]; most of the tools in the RV community do not formally express the notion of a trace in a practical scenario and hard-code the traces for simplicity, creating log files with a manually defined trace.

What is interesting is having a running system and extracting the events directly from it rather than having a hard-coded trace. However, this is a challenging task since there is no standard for systems to represent their output as events. Some efforts were made towards this problem, such as trying to mine the properties out of the execution log of a system, as it was performed in [55]. However, due to this challenging task, there are many limitations on what can be extracted from the log file.

The reason to leverage the actor-based model is because of its paradigm; it is intuitive to have a glimpse of what a trace is. It is the history of messages exchanged between the different actors. With this paradigm, we can have a running system and examine its execution at runtime with a standard notion of events and traces. Of course, instrumentation will always be required, but it is much more tangible than working with an utterly non-standard notion of traces.

Now, having the notion of what the system produces, it is possible to define how to specify things about our system, more precisely, what specifications and properties are. A property, as stated in Section 2.1.1, is something that we want the system to satisfy, and specifications are used to describe properties in a well-formed formalism.

There is a difference between these two terminologies because it is possible to describe the same properties in multiple specification languages. For instance, following our running example, we could define the following property "a user has to enter a chat room before posting messages in a chat room" in plain English, or we could specify the same thing using automata (like the one used in Figure 2.2). Not all specification languages have the same expressive power; some are more expressive than others, like English. However, due to this possibility of expressiveness, one might add ambiguity to the language, which is something to avoid. The specification of a given property has to be clear, like the one present in the automata or other specification languages, such as Linear Temporal Logic (LTL) [67]. For example, in Figure 2.3, we could define the property mentioned above as an LTL formula and a regular expression:

$$\square(\texttt{post} \rightarrow (\texttt{enter}\ S\ \texttt{post}))\quad (\texttt{enter}(\texttt{post}^*))^*$$

Figure 2.3: Specification of a property in LTL and in a regular expression

Usually, in the majority of the literature [11], the concepts of property and specification are entangled because the specification itself is the only existing entity that describes the given properties.

2.2.1 Some Specification Language Features

As defined in [41], there are two levels of defining a property. The implicit properties are the ones that we do not define manually and are properties that we might want to check in runtime. Such properties include, among others, deadlock freedom, no null pointer dereferences, and the absence of data races. These properties can be considered a collection of pre-set properties worth monitoring. No specification is written for these; instead, ad-hoc algorithms are written to detect a violation of these properties [11].

On the other hand, explicit properties are the properties that the user can express in a given specification language. Most of the work is focused on explicit properties since it gives more freedom on what one might want to monitor.

There are two main classes of explicit specification paradigms that we might use in order to define a property. On one hand, in a specification language like an automata, the specification is directly executable; this is labelled as an operational language and explicitly describes a given system's behaviour. On the other hand, a language can be declarative (e.g., a temporal logic formula) where instead of explicitly stating what is the behaviour of the system, it simply declares what should happen, not how to check it [41], and that specification is then used to generate an operational model to check the system at runtime.

Operational specifications often have simpler monitoring algorithms but are generally more low-level, making it challenging to express high-level properties and combine multiple behaviours effectively [11]. In contrast, declarative specification languages enable seamless composition of properties. For instance, if ϕ represents one behaviour and ψ another, they can be easily combined into a new formula, $\delta = \phi \rightarrow \psi$. Achieving the same result with an automata would be less straightforward and could reduce readability. However, the increased expressiveness of declarative approaches comes at the cost of a more complex algorithm for generating a monitor.

It is also possible to incorporate time into our specifications, and it is possible to work with logical and physical time. In the case of logical time, the specification only describes a relative ordering between events [41]. In the case of physical time, the specification describes the desired physical time that must be observed. For example, if we want a user to `post` a message within the first 5 minutes after entering a chat, we can add that notion to our specification.

Certain specification languages are more suited to specify properties over infinite traces (such as LTL), while others work with finite traces (such as automata). As the observations at runtime are finite, this often leads to creating new semantics or mapping existing ones over infinite traces to work with finite traces [17].

It is also possible to employ parametric specifications, which incorporate data quantification directly into the specification. This approach enables the expression of properties that depend on dynamic data values observed at runtime [41, 11].

2.3 Specification Languages

Properties can be specified in several ways; however, this section focuses on the most widely used approach within the RV community: temporal logic, with linear temporal logic (LTL) [67] being the most fundamental and commonly applied variant.

2.3.1 Linear Temporal Logic

Linear Temporal Logic (LTL) is the most widely used temporal logic in the field. As its name suggests, it is concerned with the ordering of events rather than their absolute timing, allowing the expression of temporal properties using logical quantifiers over sequences of events. LTL also serves as the foundation for many other runtime specification languages, such as TLTL [17], which extends LTL to systems where quantitative timing constraints are important. Additional specification languages will be discussed in more detail in Chapter 6.

Definition 2.3.1 (LTL formula [67])**.** The set of LTL formulas is inductively defined by the following grammar:

$$\varphi ::= \mathit{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \,\mathcal{U}\, \varphi \mid X\varphi$$

where p is an atomic proposition (i.e. an event).

The base semantics for LTL are propositional variables, the logical operators $\neg$ and $\vee$, and the temporal operators until (U) and next (X). $\varphi \; U \; \gamma$ means that φ is true until γ is true, and $X\varphi$ means that φ is true at the next point of a trace.

The additional logical operators can be defined as follows:

$$\begin{aligned}
\texttt{false} &\triangleq \neg\texttt{true} \\
\varphi \wedge \psi &\triangleq \neg(\neg\varphi \vee \neg\psi) \\
\Box\varphi &\triangleq \varphi \,\mathcal{U}\, \texttt{false} \\
\Diamond\varphi &\triangleq \neg\Box\neg\varphi \\
\varphi \rightarrow \psi &\triangleq \neg\varphi \vee \psi
\end{aligned}$$

The most frequently used operators in this type of logic are those defined at the extent of the base U and X operators.

The **always** operator, which intuitively states that something holds forever, is defined as:

$$\Box\varphi = \varphi \,\mathcal{U}\, \texttt{false}$$

and, for simplification and understanding purposes, the simplified version will always be used. But, intuitively this is true because φ is waiting for `false` to become `true`, but it will never be `true` therefore φ holds forever.

There is also the **eventually** operator, stating something eventually will be `true`. Formally, it is defined as:

$$\Diamond\varphi = \neg\Box\neg\varphi$$

These LTL operators are often called future-time LTL [11] due to their nature to look at the future of the trace. There are also past-time LTL, which have symmetric operators for each base operator of future-time LTLs. These are the $\bullet$, previous operator, and S, as the since operator, stating that some property is always true since another one was last true.

2.3.2 Satisfiability of LTL

A system $\mathcal{S}$ has an alphabet Σ containing all its observable events. Given an alphabet Σ, a trace $\sigma = \sigma_0\sigma_1\ldots$, is a sequence of events containing elements of Σ. σ_i is the i-th element of σ , σ^i is the suffix of σ starting from i (i.e., $\sigma_i\sigma_{i+1}\ldots$), Σ^* is the set of all possible finite traces over Σ, and Σ^ω is the set of all possible infinite traces over Σ.

Definition 2.3.2 (Semantics of LTL)**.** Let us state that $\sigma = \sigma_1\sigma_2... \in \Sigma^\omega$ be an infinite sequence of events. Then we can define the semantics of LTL formula inductively as follows:

$$\sigma \models \texttt{true}$$

$$\sigma \models \neg\varphi \text{ iff } \sigma \not\models \varphi$$

$$\sigma \models p \text{ iff } p \in \sigma_0 \ (i.e\ \sigma_0 \models p)$$

$$\sigma \models \varphi \vee \psi \text{ iff } \sigma \models \varphi \text{ or } \sigma \models \psi$$

$$\sigma \models \varphi \wedge \psi \text{ iff } \sigma \models \varphi \text{ and } \sigma \models \psi$$

$$\sigma \models \mathcal{X}\varphi \text{ iff } \sigma^1 \models \varphi$$

$$\sigma \models \varphi\ \mathcal{U}\ \psi \text{ iff } \exists_{i>=0} : \sigma^i \models \psi \text{ and } \forall_{0<=j<i} : \sigma^j \models \varphi$$

Intuitively, given a trace σ, we can state the following: σ satisfies an atomic proposition p if it is the initial event of the trace. σ satisfies the negation of the LTL property φ, if σ does not satisfy φ. The trace satisfies the conjunction of two LTL properties if σ satisfies both properties and the disjunction of two LTL properties if it satisfies at least one. For the more complicated ones, σ satisfies next-time φ, if the suffix of σ starting in the next step (σ^1) satisfies φ. Finally, a trace σ satisfies the until operand $\varphi\ \mathcal{U}\ \psi$, if there exists a suffix of σ such that ψ is satisfied, and for all suffixes before that one, φ holds.

These semantics can be extended by introducing the temporal operators eventually ($\Diamond$) and always ($\Box$), as shown below:

$$\sigma \models \Diamond\varphi \text{ iff } \exists i \geqslant 0 \text{ such that } \sigma^i \models \varphi$$

$$\sigma \models \Box\varphi \text{ iff } \forall i \geqslant 0 \text{ it is true that } \sigma^i \models \varphi$$

$\models$ is called the satisfaction relation, it is a relation between traces and formulas. It can be read as: the trace σ satisfies φ, or in other words, φ is true in σ.

Therefore, for a given property φ, $[\![\varphi]\!]$ is the language of the property, i.e, the set of all traces that satisfy φ, and it can be represented as $[\![\varphi]\!] = \{\sigma \mid \sigma \models \varphi\}$. This is a regular set of infinite traces which is accepted by a corresponding Büchi Automata [72, 57, 17].

In Figure 2.4 there is a visual representation of the satisfaction of some LTL formulas.

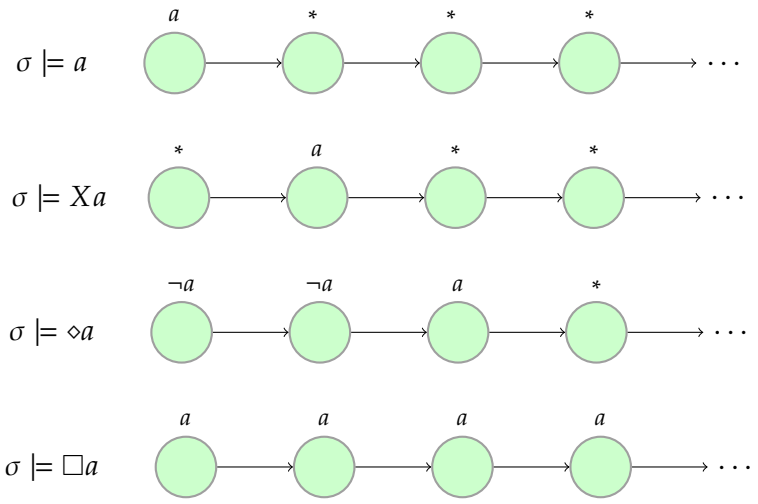


Figure 2.4: Illustration of several property satisfactions

2.3.3 Automata

Automata are often used to specify properties and one of their key advantages is that they do not require a synthesis algorithm, as they are directly executable. While the definition of automata is well established in computer science, it is important to highlight a specific type known as Büchi Automata [68]. These automata are commonly used in RV and are typically generated from an algorithm that converts a LTL formula into a Büchi Automata.

Büchi Automata are incorporated into algorithms that synthesize a monitoring entity that operates alongside the system [17, 70]. The key distinction between standard automata and Büchi Automata is that the latter is designed to handle infinite inputs, making it suitable for reactive systems, often the focus of RV. Büchi Automata can recognise properties over infinite behaviours, emphasising their importance in this field of RV.

2.3.4 Regular Languages

As regular languages share similarities with automata, since we can convert one to an automata; they also caught the eye of the RV community. Regular expressions are popular in various fields of computer science, ranging from theory of computation to runtime verification. Although not commonly used in RV frameworks, like temporal logics, they are occasionally used alongside other specification languages. One example is suffix-matching for violations [11], which, as the name suggests, tries to match the traces with

a given suffix, and if they match, there is a violation. Whilst regular expressions have been extended with a quantitative notion of time [71] and to handle data [59], these do not receive much interest in RV [11].

2.4 Monitors

Monitors are active entities that analyse a SUS and produce a verdict based on the system's execution trace. It is essential to understand how a monitor evaluates a given trace and determines whether it satisfies or violates a specified behaviour. This section presents the general algorithm for generating a monitor.

2.4.1 Linear Temporal Logic for Runtime Verification

As stated, RV focuses on finite traces because they are observable at runtime. A problem comes with this: LTL semantics are specified over infinite traces; we cannot use them in a runtime setting. Pnueli's LTL [67] is a well-accepted linear temporal logic used for specifying properties over infinite traces, and we usually want to check the same properties in runtime. However, to be able to use these specification languages, we must interpret and map the semantics over infinite traces to finite prefixes of the system's trace.

Bauer et al. [17] propose an extension of the LTL language, using it as a basis, presenting LTL_3, a three-valued semantics to evaluate standard LTL formula on finite words. The main idea that we have to follow here is that the trace currently being examined, in a runtime context, is a prefix of a so-far unknown infinite trace.

Thus, rather than evaluating a formula simply to true ($\top$) or false ($\bot$), LTL_3 distinguishes three possible outcomes when analyzing a finite trace. First, the observed finite word w may be sufficient to conclude that the monitored property is satisfied, regardless of any future behaviour. Second, w may already demonstrate that the property cannot hold in any possible continuation. Finally, if neither of these conditions is met, the trace is inconclusive, and monitoring must continue as additional events are observed.

In LTL_3, the syntax remains identical to that of standard LTL; however, the semantics are adapted to account for the finite nature of observed traces. The formal definition is given as follows:

Definition 2.4.1 (Semantics over LTL_3 [17])**.** Let $u \in \Sigma^*$ a finite word. The truth value of a LTL_3 formula φ with respect to u, denoted by $[u \models \varphi]$, is an element of $\mathbb{B}_3 = \{\bot, \top, ?\}$ defined as follows:

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega,\ u \bullet \sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega,\ u \bullet \sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$

where $\bullet$ is the standard trace concatenation operator.

Notice that now, instead of having only $\top$ and $\bot$, there is a third value in this semantics; the inconclusive value denoted by "?" which means that the so far observed trace itself is insufficient to determine how a given property evaluates in any possible future continuation of the currently observed trace.

For instance, the property $\Diamond p$, over the alphabet $\Sigma = \{p, q\}$, and given the current trace as $\sigma = qqq$ would be evaluated as `false` using the base LTL semantics defined by Pnueli. On the other hand, in these new LTL_3 semantics, it would be evaluated as "?" because there is no way up until the current visualized trace to determine with guarantees that the property is either violated or satisfied.

Therefore, if every infinite trace extending the prefix u evaluates to the same truth value, either $\bot$ or $\top$, then $[u \models \varphi]$ is assigned that value. Otherwise, if different continuations of u lead to different truth values, the evaluation is inconclusive, denoted by "?".

As studied in [52], a semantic for both LTL on finite and infinite words were given. Nevertheless, runtime verification's objective and primary goal is to check the properties of infinite traces considering their finite prefixes. That is why base LTL is not appropriate in these RV scenarios, which is why LTL_3 is one of the key languages used in the context of runtime programs. It also serves as a foundation for other specification languages presented in Chapter 6.

Contrary to the LTL logic presented, LTL_3 is not defined in an inductive manner; the semantics of a given formula are not based on the semantics of its subformulas [17, 15]. For example, consider the trace $\sigma = \epsilon$ and the alphabet $\Sigma = \{a, b, c\}$. The property represented by $\varphi = \Diamond a \vee \Diamond \neg a$ evaluates to $\top$ when considered as a whole. However, if we evaluate the sub-formulas separately, we get $? \vee ?$, which does not align with our expectations. Similarly, both components of the formula $\psi = \Diamond b \vee \Diamond \neg c$ yield ? for the given trace. Unlike φ, the overall evaluation of this formula also results in ? [6] . Thus, it has been argued that LTL_3 is not defined inductively.

Later, Amjad et al. [6] prove that the claim from Bauer et al. [17] stating that it is impossible to use inductive semantics over LTL_3, is false. It is possible if one associates sets of traces with each formula.

2.4.2 Defining a Monitor

As stated previously, a run of a given system is understood as a potentially infinite sequence of the system's alphabet, denoted as Σ. In contrast, an execution refers to a finite prefix of this infinite trace produced by the system. This execution is what provides information to monitors in RV. Monitors are an essential components, as the primary purpose of RV is to automatically generate them and deploy them to operate alongside the SUS.

Checking whether an execution meets a correctness property is typically performed using these monitors. Simply, a monitor determines if the current execution trace fulfils a specified correctness property by producing a truth value. Therefore, a monitor is a

device that reads a finite trace and yields a certain verdict [58]. The concept of what a monitor can monitor at runtime will be discussed further.

Formally, when denoting the set of valid executions by $[\![\varphi]\!]$, the verification process boils down to checking whether the current execution belongs to this set. As mentioned earlier, this process is referred to as the word problem, and it is a less complex operation than the model checking approach, which involves the subset problem [58].

Definition 2.4.2 (Monitor [43])**.** Let S be a system with alphabet Σ, and ϕ be an LTL property. Then, a monitor for ϕ is a function M that maps a trace to a given value of a truth domain:

$$M_{\varphi,\Sigma}(u) = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^{\omega},\ u \bullet \sigma \in [\![\varphi]\!] \\ \bot & \text{if } \forall \sigma \in \Sigma^{\omega},\ u \bullet \sigma \notin [\![\varphi]\!] \\ ? & \text{otherwise} \end{cases}$$

where $\bullet$ is the standard trace concatenation operator.

The definition of a monitor and the semantics of LTL_3 are closely aligned, since a monitor is precisely the entity that evaluates a finite prefix of a trace and produces a verdict corresponding to one of the three possible semantic values.

Intuitively, a monitor returns $\top$ if all possible continuations of the currently observed trace satisfy the property, and $\bot$ if all continuations violate it. In the context of runtime verification, a monitor can also return an inconclusive verdict ("?"), indicating that, based on the information observed so far, it is impossible to determine whether the property is satisfied or violated. This inconclusiveness reflects the fact that RV operates on a system that is still executing, which may yet produce events leading to a desirable outcome.

RV relies heavily on monitors since they are the pillars of the RV architecture. They are the entities that will be deployed and are responsible for detecting violations or satisfactions. Usually, we monitor for violations, but a monitor can also detect satisfactions [11].

2.4.3 Monitor Synthesis Algorithm

Defining the semantics of how a monitor verifies if a prefix of a trace satisfies a property is an essential step to understanding monitor functionality. Having the specification written in an LTL_3 formula, how can we create an executable monitor that will monitor for that property? This question will be answered in this section. Note that this is only one of the many monitor synthesis algorithms; depending on the specification language chosen, the algorithm might vary.

The algorithm to synthesise a monitor from an LTL_3 formula was defined in Bauer et.al [17]. The synthesis procedure will follow an automata-based approach by starting from a LTL_3 formula and producing a Moore Machine that will act as our monitor for a given correctness property.

A Moore Machine [16, 17], is a finite state machine where the output value of a state is only determined by that given state. It can be defined as a tuple $\langle Q, q_0, \Sigma, O, \delta, \gamma \rangle$, where Q is a finite set of states, q_0 is the initial state, Σ is the input alphabet, O is the output alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function mapping a state and an event to the next state, and $\gamma : Q \to O$ is the function mapping state to the output alphabet.

The work of Bauer et al. is detailed in their article [17] and illustrated in Figure 2.5, adapted from the same article. Briefly, the process involves the following steps: given a property φ, transformations are applied to the formula and its negation 2). These transformations allow the construction of Büchi Automata for both φ and its negation, generating one automata for each 3). This step leverages the Gerth et al. algorithm [48], and the resulting automata recognize the sets of infinite traces that satisfy φ and violate φ, respectively.

After that, each step of the resulting automata is evaluated, and if the states that are selected as initial states do not generate the empty language, they are added to the following step of the algorithm 4). With those, a non-deterministic automata is generated, which recognizes the finite traces (our prefixes) with at least one infinite continuation satisfying φ and violating it, respectively 5). After that, by applying the famously used Rabin-Scott algorithm, the deterministic version of this automata is built 6).

Finally, by having these two deterministic automata, one that recognizes finite traces that satisfy φ, and the other recognizes finite traces that violate φ, we can built the Moore Machine as a standard automata product between the two generated automatons 7).

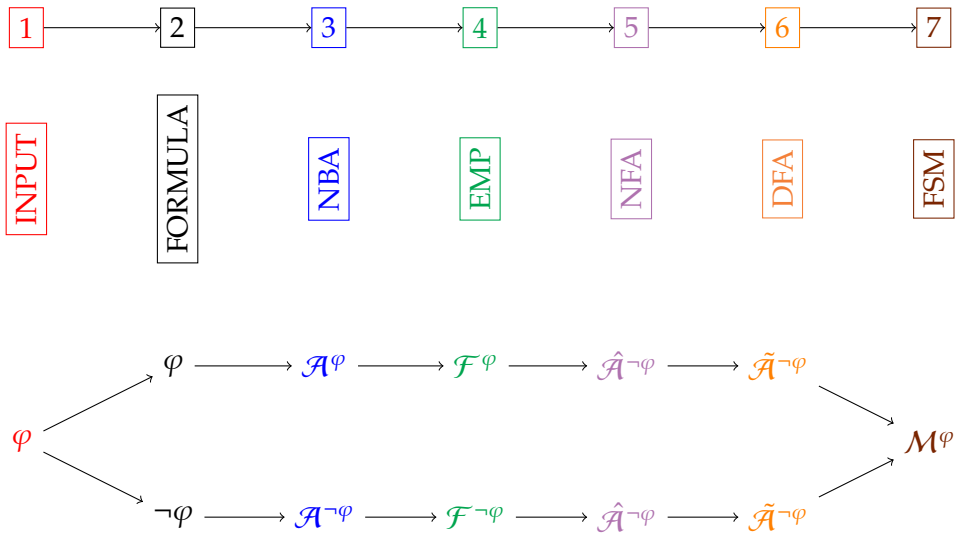


Figure 2.5: Diagram illustrating the algorithm of generating a monitor

This algorithm can generate a Moore Machine to monitor for a given property φ defined in LTL_3. The authors of the algorithm implemented it in an on-the-fly fashion to avoid some steps to be fully executed in order to obtain the final monitor, avoiding exponential blow-ups, like the one presented in d'Amorim and Rosu [36].

Example 2. Let us consider our chat room from Figure 2.2, and $\varphi = \neg\texttt{post}\mathcal{U}\texttt{enter}$ the LTL property to verify, and the alphabet will be reduced to $\Sigma = \{\texttt{enter}, \texttt{post}\}$. Intuitively, we can read φ as "a user must first `enter` a chat in order to perform a `post`", and the violation of this property would be read as "if there is a `post` before an `enter` then the monitor reports a violation". The Moore Machine that is generated from this property using the algorithm depicted in Figure 2.5 can be seen in Figure 2.6.

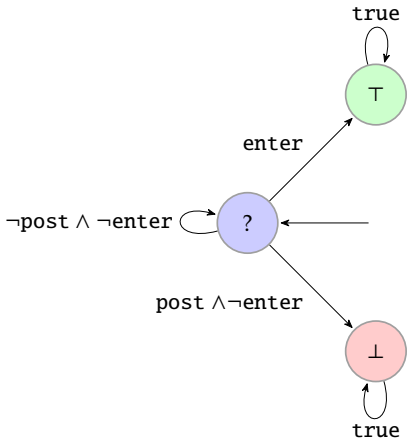


Figure 2.6: The resulting Moore Machine for $\varphi = \neg\texttt{post}\mathcal{U}\texttt{enter}$

Observing the resulting Moore Machine, we can reason about its transitions and check how a property might be satisfied, violated, or state inconclusiveness. Suppose we detect that a `post` occurred and no `enter` was observed. In that case, no possible continuation can satisfy the property from that point onwards because it was violated. All the possible continuations of that trace will violate the property. The same goes for whenever we detect an `enter`; from that point onwards, we know that all the possible continuation will not violate the property (let us assume that a `leave` will never occur in this simple example). If the monitor has not yet observed events to state that there was a violation or satisfaction, it will continue to output an inconclusive verdict.

2.4.4 The Two Maxims

To ensure a monitor functions correctly in a runtime context, it must adhere to two fundamental principles, or maxims, defined in [58, 18]. The first is impartiality, which requires that a monitor never assigns a verdict of $\bot$ or $\top$ to a finite trace if there exists any infinite continuation that could lead to a different outcome. In other words, once a monitor produces a definitive verdict, it must remain fixed and cannot be reversed. The verdict must be irrevocable. Complementing this, the second maxim is anticipation. Anticipation dictates that as soon as every possible infinite continuation of a finite trace leads to the same verdict, the monitor should immediately assign that verdict to the trace. This ensures that violations or satisfactions of properties are detected as early as possible during runtime. Together, impartiality and anticipation provide a formal foundation for

designing reliable monitors. A more detailed discussion and formal definitions of these maxims can be found in [18]. Depending on the specification language that we choose, these maxims might not be respected; for instance, LTL_3 respects both the maxims [18].

Example 3. Let us consider $\varphi = \Diamond a$, over the alphabet $\Sigma = \{a, b\}$, and evaluate this property over the LTL semantics and LTL_3, respectively. Having a growing trace that begins with, $\sigma = b$, then $\sigma = bb$ and finally $\sigma = bba$. Evaluating this growing trace using the LTL semantics, we get the following verdicts, $\bot\bot\top$, which violates the impartiality maxim. Contrarily, evaluating the growing trace over the LTL_3 semantics, one gets $? ? \top$, and now every single possible continuation of this trace evaluates to $\top$, therefore respecting the impartiality maxim.

2.5 Monitorability

Unfortunately, some formal properties cannot be monitored at runtime. Intuitively, a formal property is considered monitorable if a monitor can be synthesized to verify it; otherwise, it is deemed non-monitorable.

Example 4. Consider the following properties: $\varphi = \Diamond p$ and $\psi = \Box(p \rightarrow \Diamond q)$, stating that p will eventually hold and that whenever p occurs, eventually q will hold. The first property is said to be monitorable because, given that a monitor is currently examining the trace, it can state that the property is satisfied whenever it observes the value p in the trace. It is impossible to change the verdict from that point onwards; the monitor can easily detect this behaviour. For the second property, a monitor cannot determine whether the property will be violated or satisfied no matter what trace it has at its disposal; even if the monitor observers that a q happened after a p, it will never be able to state for sure that the property was violated or not because it will always need more information. Therefore, we might already note that this property is impossible to verify at runtime since no helpful information can be gained by the monitor, leading the monitor to produce inconclusive verdicts infinitely.

More formal definitions were created to define this property, but there is no consensus among the community on the definition of monitorability [43, 2]. However, the most common requirement is that a property should always allow a monitor to conclude either a violation or satisfaction of the property, i.e., to reach a verdict. The first ever mention of this definition appeared in Viswanathan and Kim [74], where the authors state that a monitorable property is a property that can be recursively enumerable, and they defined the computational constraints of monitorability.

Following the initial definition, Pnueli and Zaks provided another definition of monitorability [66]. This definition serves as the basis for many subsequent definitions, expanding upon the interpretation offered by Viswanathan and Kim. The definition by Pnueli and Zaks will be utilized in this document to explain what monitorability is and

how one can determine whether a given property is monitorable. As defined in Pnueli and Zaks, a property φ is positively determined by trace σ if it is the case that $\sigma \bullet u \models \varphi$ for all finite or infinite completions u. The same goes for negatively determined properties, but instead of satisfaction of the property, it is a violation. With this comes the following definition:

Definition 2.5.1 (σ-monitorable)**.** A property φ is σ-monitorable, with $\sigma \in \Sigma^*$, if there is some $u \in \Sigma^*$ such that φ is positively or negatively determined by $\sigma \bullet u$

Definition 2.5.1 states that a property is said to be σ-monitorable with respect to a given finite trace of events σ, if we can find at least one continuation u, such that φ is satisfied or violated. Note here, that the continuation has to be finite ($u \in \Sigma^*$). Therefore, intuitively a property is said to be σ-monitorable if there is at least one possible continuation that allows the monitor to yield a satisfactory or violation verdict.

Following this definition, several notions of monitorability have been proposed, which vary depending on how restrictive we wish to be. Definition 2.5.2 presents the least restrictive form, known as existential monitorability. A property is existentially monitorable if there exists at least one finite trace of events $\sigma \in \Sigma^*$ for which a continuation can be found that either satisfies or violates the property. This relaxed notion, often referred to as weak monitorability [31], allows a monitor to remain inconclusive on certain traces. While this provides greater flexibility within the monitorability spectrum, it comes at the potential cost of the monitor being indefinitely unable to reach a definitive verdict.

Definition 2.5.2 ($\exists_{PZ}$-monitorable)**.** A property is $\exists_{PZ}$-monitorable (existentially Pnueli-Zaks) if it is σ-monitorable for some finite trace $\sigma \in \Sigma^*$.

Example 5. Let us consider the property $\varphi = (a \wedge \Diamond b) \vee (c \wedge \Box\Diamond d)$, and the alphabet $\Sigma = \{a, b, c, d\}$ [43]. This is a $\exists_{PZ}$-monitorable property since it is possible to find some trace σ for which φ is σ-monitorable. For instance, any trace that begins with the symbol a can eventually satisfy the left branch of the disjunction by observing b. However, every trace starting with c is not σ-monitorable for any possible trace since there is no continuation $u \in \Sigma^*$ that positively or negatively determines the property. Figure 2.7 represents the monitor generated by the algorithm presented in Section 2.4.3. It becomes easier to detect what is monitorable by looking at the Moore Machine because whenever we have a state stuck forever in an inconclusive state, it can never be monitorable because there is a possibility of never reaching a verdict.

Definition 2.5.2 is a more relaxed take on monitorability. Usually, in literature most researches consider the true notion of monitorability as the one defined in Definition 2.5.3.

Definition 2.5.3 ($\forall_{PZ}$-monitorable)**.** A property is $\forall_{PZ}$-monitorable (universally Pnueli-Zaks) if it is σ-monitorable for all finite trace $\sigma \in \Sigma^*$.

Intuitively, a property is considered monitorable if, for every possible trace observed at runtime, that is, for every progressively growing trace examined by a monitor, there exists

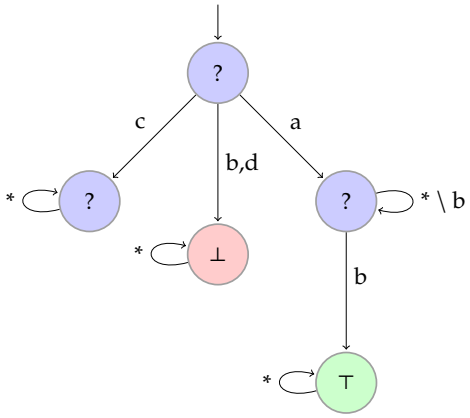


Figure 2.7: The resulting Moore Machine for $\varphi = (a \wedge \Diamond b) \vee (c \wedge \Box\Diamond d)$

a continuation that eventually leads to either a violation or satisfaction. This represents the strongest notion of monitorability. Figure 2.8 illustrates a simple example of a property that is $\forall_{PZ}$-monitorable.

Example 6. Let us consider the property $\varphi = \Diamond a$ and the alphabet $\Sigma = \{a, b, c\}$. This property is $\forall_{PZ}$-monitorable. For any continuation $u \in \Sigma^*$, it is always possible to either satisfy or violate the property; in this case, in particular, we will always be able to find a violation of the property. Furthermore, the property is σ-monitorable for every $\sigma \in \Sigma^*$. Figure 2.8 illustrates the monitor generated by the algorithm presented in Section 2.4.3. It is clear that the Moore Machine will never enter an inconclusive loop.

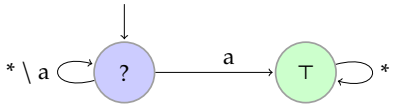


Figure 2.8: The resulting Moore Machine for $\varphi = \Diamond a$

Monitorable properties are usually defined according to Definition 2.5.3 in most literature [17, 11, 58, 74]. There is a reason for that since this is the strictest definition for monitorability because this definition guarantees that if a property is said to be monitorable, a monitor that was synthesised to monitor for it will never be stuck with inconclusive verdicts and will eventually be able to state that a given property was violated or satisfied. Definition 2.5.3 comes at the price of having a higher strictness, reducing the properties that can be expressed and stamped as monitorable, compared to its softer definition presented in Definition 2.5.2.

When a less restrictive notion of monitorability is adopted, allowing for traces that may yield inconclusive verdicts indefinitely, Bauer et al. [17], building on the concepts of good and bad prefixes introduced by Kupferman and Vardi [54], introduce the notion of an ugly prefix. An ugly prefix is a trace for which, regardless of how it is continued, no definitive verdict can ever be reached. To avoid this limitation, stronger definitions

of monitorability, such as that in Definition 2.5.3, are employed to determine whether a property can be reliably monitored. Usually, all non-monitorable properties have one thing in common: the notion of something happening infinitely often, as stated in Pnueli and Zaks [66]. Consequently, a property of the form $\Box\Diamond\varphi$, which requires that φ occurs infinitely often, is not monitorable. For a comprehensive classification of monitorable properties, the reader is referred to Bauer et al. [17].

2.6 Instrumentation and Deployment

As shown in Figure 2.1, the instrumentation will govern the relevant information from the running system and record them as events. These events can include various factors, such as the written and read variables or even the method calls and returns. In the context of the actor-based model, which will be explained further ahead, instrumentation can focus specifically on the messages exchanged between processes, forwarding this information to the monitor for analysis.

These events are delivered to the monitor in a way that aims to preserve the correct execution order of the trace. However, in concurrent or distributed systems with multiple processes, preserving a total order is not always possible. In such cases, a partial ordering can be used, providing a reliable sequence for the monitor to observe [11].

Instrumentation determines how a monitor interacts with the system during execution. Often referred to as monitor deployment, as mentioned in the taxonomy paper by Falcone et al. [41], instrumentation addresses various challenges. For instance, the system may need to pause its execution while the monitor processes incoming events. Alternatively, it is possible to design an interleaving approach that allows the monitor and the system to operate concurrently, avoiding execution stalls. In concurrent environments, the nature of the instrumentation will influence how closely the monitor and the system are entangled.

Though the structure depicted in Figure 2.1 is the one that is commonly used in RV tools, it is not at all the only way that we can monitor a system. Novel approaches that embed the notions of RV in the code-writing paradigm have been studied. In Monitor-Oriented-Programming (MOP) [51, 30], the notion of monitors is entangled with the creation of the system and is followed as a design principle. The system's code is organized and coded in such a way that we do not have to worry about the instrumentation process of the monitor because the monitor is embedded in the system, which is extremely invasive. In MOP, the monitors are compelling tools that do more than alert the system whenever a violation occurs; these monitors might react and inject adaptations to the system to avoid critical scenarios from happening [51, 23, 21].

As shown, instrumentation determines how the monitor infrastructure interacts with the SUS. It specifies what parts of the system are visible to the monitor for analysis, how we organize the monitor and the system to work as a whole, and how the monitor operates alongside the system. This section will focus on these two aspects: the instrumentation itself and monitor deployment [11, 28, 41].

2.6.1 Online and Offline Monitoring

There are two possible approaches to instrument our monitoring framework: online and offline monitoring [11, 41]. The most basic form of "runtime" monitoring is offline monitoring. This method occurs not during execution but after the system has finished running. In offline monitoring, the primary strategy is to analyse log files that record the system's execution traces or other relevant data that can be used for further analysis.

During execution, the system is not directly monitored; pertinent events are logged in a storage medium (such as a log file) and later examined by the offline monitor. Since the offline monitor operates independently from the system, it does not add any overhead, making it less intrusive than online monitoring. It only interacts with the system through the event recording mechanism. The typical setup for offline monitoring is illustrated in Figure 2.9.

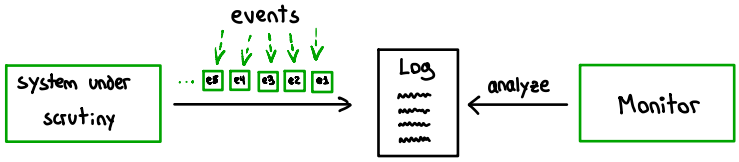


Figure 2.9: Offline monitoring

In this case, we always work with a finite trace because the information has been logged into a file. Therefore, there is no need to worry about concepts such as non-monitorability explored in Section 2.5, since everything is monitorable in such molds.

Offline monitoring introduces no overhead to the examined system, as it only analyses the system's historical trace after execution. This method is beneficial for verifying properties that require a global view of the system, allowing insight into the complete execution of a system. However, it cannot provide real-time alerts for property violations during runtime nor adjust the system to prevent software faults that might lead to failures [58]. Hence, offline monitoring is usually used to double-check a system with high correctness and confidence.

In contrast, online monitoring operates alongside the system, incrementally and efficiently analyzing execution traces as they are produced. This approach is the most widely adopted in RV, as it embodies its core principle: observing and reasoning about a system while it is actively executing.

Online monitoring focuses on targeting the dynamic execution of a system, checking for property violations or satisfactions throughout the entire program execution. This incremental approach requires the monitor to receive notifications or events from the system about relevant actions as they happen and to make decisions based on the data collected so far, this interaction can be visualized in Figure 2.1.

The main advantage of online monitoring is its ability to address the late detection issue associated with offline monitoring, where property violations are only identified

after the system has stopped executing, losing the opportunity to alert or mitigate the violation [42].

Due to the nature of these monitors, online monitors can detect issues early and notify the system, which is essential when dealing with critical properties that the system must adhere to. In such cases, immediate corrective actions may be required, or, in some instances, the system may need to be halted until appropriate measures are taken to prevent software failure.

Unlike the offline approach, online monitoring works with a partial system execution that continues to grow over time. However, the orchestration required to run the monitor alongside the system, introduces additional overhead to the overall system.

Although overhead is introduced during the execution of the monitor, the complexity of building it can be overlooked since it is generated beforehand. In order to reduce the overhead of online monitoring, the focus should be on the instrumentation of both the monitor and the system to minimize the monitor's impact on the SUS.

The main drawback is that the overhead from the online monitor is unavoidable. While it allows for dynamic system checks, exchanging messages, analyzing traces, and determining violations of correctness properties is cumbersome and introduces significant overhead [27, 49, 11].

Addressing the overhead introduced by runtime monitoring requires substantial effort [24, 46, 26]. Many approaches focus on optimizing the monitoring framework to reduce this cost. Nevertheless, ensuring the security and reliability of a system at runtime inherently incurs some overhead, which is a trade-off that must be accepted: it is preferable to maintain a certain level of assurance than none at all. The actual overhead of an online monitor depends on factors such as the deployment context and the specific actions taken when a property violation is detected.

2.6.2 Synchronous and Asynchronous Monitoring

As previously stated, the deployment part dictates how the monitor executes in relation to the monitored system, and this can be further explored in the case where we are dealing with online monitoring. In the online monitoring scenario, we can choose whether the monitor will execute synchronously with the system or asynchronously.

Depending on the decision made by a runtime tool, the system's execution can either pause, waiting for the monitor to allow it to continue, which represents synchrony, or proceed asynchronously, with both components operating independently and detached from one another.

In the synchronous scenario, the system produces events and waits for the monitor to process them before proceeding with its execution. With this, we allow the monitor to catch a violation and alert the system immediately when it is detected, but at the cost of adding extra overhead to the monitored system since it has to wait every single time before executing the next event and has to send that information to the monitor.

Conversely, asynchronous monitoring reduces the high entanglement between the monitor and the system. This approach is less intrusive than synchronized monitoring, leading to lower overheads [24], but does so at the cost of potential delays in detection.

Since the monitoring process runs independently, it may take some time to identify a property violation, the system might continue executing additional actions beyond the one that caused the violation, potentially leading to a software failure even if the monitor detects the violation.

It is also possible to adopt a partially synchronous approach, in which the monitor synchronizes with the system for certain observations or at specific intervals. Hybrid strategies that combine synchronous and asynchronous monitoring have also been explored, as discussed in [28].

2.6.3 Inline and Outline Monitoring

Another choice that we can make in deployment is whether the monitor will be executed inline or outline of the monitored system. Usually, inline monitoring means that the monitor and the system are tightly coupled, and the monitor is executed in the same address space as the monitored system.

The system and the monitor run alongside each other, allowing us to achieve lower overheads. In contrast to its counterpart, this approach is typically more finely tuned because it has complete access to the system code [39]. However, this makes it impractical when access to the system code is unavailable.

On the other hand, outline monitoring treats the monitor as a separate unit from the system; in this scenario, there is no entanglement between the monitor and the SUS. There comes the need for an instrumentation, some interface, where these two entities communicate and exchange information. Outline monitoring offers the ability to deploy these monitors to other machines and the advantage of minimally altering the code of the monitored system.

In the actor-based model, it is common to use outline monitoring where the monitor is treated as a separate entity that communicates with the system, intercepting the messages between the processes [9]. In other cases, such as JavaMOP [51], the inline approach is applied because of the whole paradigm for monitor-oriented programming where we have the system code completely entangled with the monitor code.

2.6.4 Centralised and Decentralised Monitoring

A monitor can be deployed separately, in a decentralised manner, or as a monolithic component [41, 11]. The most common RV tool approach is generating a single monitor from a high-level specification [17].

Alternatives have been investigated to enable multiple monitors to communicate with the SUS, allowing for parallelism since the system is not confined to a single monolithic object [19].

Having this in mind, when working with decentralised monitoring, we can implement and dictate how the monitors will behave and, more importantly, how the monitors will work in a distributed manner.

Usually, there are two strategies to tackle the coordination needed in a distributed scenario. Orchestration relies on a single coordinating entity that gathers the information that is available and orchestrates the whole message exchanging between the parties involved (the monitors and the system), whereas in choreography, every component involved disseminates the task across each other [45].

Orchestration is the option that is usually taken in a RV tool, since it resembles the monolithic approach and is simple to translate from that architecture. However, as a side effect, it adds more network traffic to the system. It is less error-prone compared to the choreography approach since it was designed specifically for distributed scenarios.

2.6.5 Passive and Active Monitoring

A monitor can work as a simple entity that only examines the SUS and does not interact with it in any way; this is called passive reaction and means that the monitor does not influence the execution of the system, usually simply observing and collecting information from it.

These monitors can still issue a verdict, but that is their sole purpose: providing a verdict on a specific property violation or satisfaction without taking further action or alerting the system. There might also be an explanation attached to the verdict (e.g., the trace currently observed that led to that given verdict) or even statistics about the observations of the monitor, for example, the number of violated properties.

In contrast, the monitor may take an active role when examining the system's execution. It can intervene whenever a violation occurs, steering the system to prevent further violations of the detected property by communicating necessary adjustments.

The concept of actively steering a system at runtime falls within the area of runtime enforcement [42, 60, 3, 40]. This field is being explored as a more proactive alternative to traditional runtime verification. By operating alongside the system, enforcement mechanisms can adapt its behaviour in real time to prevent property violations before they occur. Such approaches involve not only enforcing specific behaviours but also dynamically responding to ongoing actions [25] and mitigating the potential impact of undesired executions.

2.6.6 Instrumentation Interference

As pointed out previously, the instrumentation is the connection wire between the monitor and the system being monitored that extracts the events and other information that might be useful to the monitor during the execution of the system to analyse and evaluate its behaviour [11].

The instrumentation can happen at two different levels: at the hardware level and the software level. The focus will be on the latter version of instrumentation, which creates software that monitors other software applications. In this case, the level of interference will depend on the instrumentation techniques that we choose to use when building a RV tool, ranging from the overhead caused to the system to the amount of reliability we want on the monitored system by sacrificing some performance.

For example, offline monitoring is less intrusive because the system is only analysed after execution. In contrast, in online monitoring, we have to be more intrusive to a certain degree depending on how the monitor will be deployed and the level of reaction chosen.

Hardware instrumentation involves integrating monitoring mechanisms directly into physical hardware, where signals are transmitted to the monitor through analog systems and physical connections [11]. This approach is highly invasive due to its tight coupling with the underlying hardware.

2.7 Actor-Based Model

The actor-based model [4] is a concurrency paradigm in which independent computation units, called actors, encapsulate both state and behaviour. Each actor operates autonomously, communicating exclusively through asynchronous message passing. This design naturally supports modularity and fault tolerance: components can fail or recover without affecting the rest of the system.

Programming languages such as `Erlang` and `Elixir` [38, 8] are built on the actor model, providing robust frameworks for creating and managing concurrent processes. Actors can be dynamically spawned, enabling multiple concurrent entities to perform tasks simultaneously. Each actor has a unique identifier, allowing precise tracking of the sender, recipient, and content of exchanged messages. Communication occurs solely through these messages, which are stored in individual actor mailboxes and delivered in first-in, first-out (FIFO) order. Actors can selectively process messages from their mailbox, further enhancing flexibility. Figure 2.10 illustrates this communication network.

We can take advantage of this paradigm since one of the biggest problems in runtime verification is extracting and understanding a trace from the SUS [41]. Most tools hard-code these traces in files and do not work with a system emitting the events at runtime. Within this model, a system's trace can be naturally understood as the sequence of messages exchanged between actors. These messages have well-defined signatures and can be monitored at runtime using `Erlang`'s tracing libraries [73]. Moreover, these tracing mechanisms can be customized or instrumented to capture the specific events relevant to runtime verification.

One of the standout features of `Erlang`'s actor model is its robust support for fault tolerance, which is rooted in the "let it crash" philosophy [8]. Instead of attempting to anticipate and handle every possible failure scenario within an actor, the system relies on a hierarchical structure of supervisors. These special actors are tasked with monitoring

their child actors and swiftly restarting them if they encounter errors. This design decision significantly simplifies error handling, allowing systems to recover autonomously from failures and return to a stable, known good state without requiring manual intervention. It is important to note that the "let it crash" philosophy does not advocate for the total collapse of the application; instead, it emphasizes the importance of transparency in failure reporting, ensuring that issues are highlighted.

In an actor-based system, each actor operates as an independent unit of computation, identified by a unique process ID, and its behaviour is defined by a set of fundamental operations. Actors communicate exclusively by sending messages to one another using the `!` operator, which delivers messages asynchronously. When an actor sends a message, it does not wait for the recipient to process it, allowing both actors to continue executing independently. Incoming messages are stored in a mailbox, and actors use a `receive` operation to examine and process messages selectively, often employing pattern matching to distinguish different types of messages. Actors can also dynamically create new actors through a `spawn` operation, enabling systems to scale their computations by generating multiple concurrent actors, each with its own state, behaviour, and mailbox.

Because actors encapsulate their state privately and do not share memory, many of the concurrency issues that occur in shared-memory systems, such as race conditions, are largely avoided. These mechanisms, including message passing, selective reception, dynamic spawning, and isolated state, define the operation, interaction, and evolution of actors. They provide a foundation for systems that are modular, fault-tolerant, and highly concurrent.

Figure 2.10: Actor-based model

2.7.1 `Erlang` Concurrency

To illustrate the actor model in practice, consider a simple `Erlang` program that manages a counter. The code below defines a process that loops, handling two types of messages: a synchronous `get` request and an asynchronous `increment` command.

In this manual implementation, presented in Listing 2.1, we explicitly spawn a new process `spawn(fun() -> loop(0) end)` and register it under a name. The loop function implements a simple receive loop that handles messages. A call to `inc` sends an `{increment}` message (fire-and-forget), while a call to `get` sends a `{self(), get}` message and waits for a reply. The code must explicitly pattern-match on `{From, get}` and send a reply tuple back. Although straightforward, this approach requires boilerplate

```erlang
-module(simple_counter).
-export([start/0, get/0, inc/0]).

start() ->
  % Spawn a new process initialized with count value zero
  register(simple_counter, spawn(fun() -> loop(0) end)).

inc() ->
  % Send an asynchronous increment message
  simple_counter ! increment.

get() ->
  % Send a synchronous get message and wait for the reply
  simple_counter ! {self(), get},
  receive
    {simple_counter, Count} -> Count
  end.

loop(Count) ->
  receive
    {From, get} ->
      % On get, reply with current count
      From ! {simple_counter, Count},
      loop(Count);
    increment ->
      % On increment, update state
      loop(Count + 1)
end.
```

Listing 2.1: `Erlang` counter system

since we manually maintain state in the loop, handle mailbox messages, and deal with process restart logic if something goes wrong.

Notice how the counter process is an isolated actor: its state (`Count`) is private and can only be accessed by the loop itself. Other processes cannot read `Count` except by sending messages to it. This aligns with the actor model principle that "data local to a process can only be accessed by the process itself". Such programs demonstrate how `Erlang` uses asynchronous message passing to build concurrent services.

2.8 The Open Telecom Platform (OTP)

The Open Telecom Platform (OTP) provides a set of libraries and design principles to simplify the construction of concurrent, fault-tolerant systems [8, 29, 50]. One of its core abstractions is the `gen_server` behaviour, a generic server template for client-server processes. By using `gen_server`, developers need only implement the specific logic via callbacks, while OTP handles the low-level details, such as spawning the process, linking it into a supervision tree, and serializing access to the state. Importantly, the server maintains a single shared state for the entire process instance, ensuring that all callbacks operate over

a consistent and centralized state. This approach significantly reduces boilerplate and removes the burden of process management from the developer. Consider the counter example presented previously, now rewritten using `gen_server`. The same counter system shown in Listing 2.1 is adapted to follow OTP principles in Listing 2.2.

```
-module(counter).
-behaviour(gen_server).

-export([start_link/0, get_count/0, increment/0]).
-export([init/1, handle_call/3, handle_cast/2, terminate/2]).

start_link() ->
  gen_server:start_link({local, ?MODULE}, ?MODULE, 0, []).

get_count() ->
  gen_server:call(?MODULE, get_count).

increment() ->
  gen_server:cast(?MODULE, increment).

init(InitialCount) ->
  {ok, InitialCount}.

handle_call(get_count, _From, State) ->
  {reply, State, State};

handle_cast(increment, State) ->
  {noreply, State + 1}.
```

Listing 2.2: OTP counter system

In this example, the `start_link` function initializes the server process and links it to the calling process, establishing a supervision relationship. The `get_count` function demonstrates a synchronous interaction using `gen_server:call`, which blocks until a reply is received from the server. Conversely, `increment` employs `gen_server:cast` to send an asynchronous message that does not require a response. Internally, `handle_call` handles synchronous requests, while `handle_cast` handles asynchronous messages, updating the server's state accordingly. The `init` callback establishes the initial state of the server upon startup.

By leveraging `gen_server`, `Erlang` developers can build robust server processes with minimal boilerplate, focusing on the core logic rather than the intricacies of process management, message handling, and fault recovery.

2.8.1 Synchronous Calls

The function `gen_server:call(Name, Request)` sends a synchronous request to the process identified by `Name` and blocks the caller until a reply is returned. Internally, each call generates a unique reference that associates the request with its corresponding response, ensuring that replies are correctly matched to the originating request even in highly concurrent systems. In response to `gen_server:call`, the receiving process invokes the callback `handle_call(Request, From, State) -> ReplyResult`. To reply to the caller, `handle_call` should return a tuple of the form `{reply, ReplyValue, NewState}`. The `ReplyValue` is sent back to the caller as the result of `gen_server:call`. For example, in our counter above, `handle_call(get_count, _From, State)` returns `{reply, State, State}`, meaning it replies with the current count and leaves the state unchanged. Under the hood, `gen_server` delivers the reply automatically, removing the need for the developer to send any explicit `From ! Reply` message, meaning that `Reply` is sent to `From`.

`call` establishes synchronous behaviour, as the process performing the request blocks until receiving a response. Even with this synchrony, we can achieve asynchronous behaviours by using the `spawn` directive, replying back once the result is ready with the `gen_server:reply` directive. This approach also benefits from automatic reference management, which ensures that each response is correctly matched to its request. It is also worth noting that if the server crashes or does not respond within a timeout, the call will return an error.

2.8.2 Asynchronous Casts

The `gen_server:cast(Name, Request)` operation sends an asynchronous message to the server and returns immediately with `ok`, without generating any internal reference or providing a response to the caller. The calling process continues execution without waiting, making `cast` a true fire-and-forget mechanism. On the server side, the message is handled via `handle_cast(Request, State) -> Result`, which typically returns `noreply, NewState` to update the server state or `stop, Reason, NewState` to terminate the server. Unlike `call`, `cast` does not send any direct reply back to the sender, placing the responsibility for subsequent actions entirely on the server or the system's design.

For example, `handle_cast(increment, State)` returning `{noreply, State+1}` updates the state without replying to anyone. This is usually used in systems where we want to "fire and forget", due to the nature of the operation. There is no reference management in `cast`, so if we want some sort of correlation between message we would have to do that manually.

Using `gen_server:call` vs `gen_server:cast` allows a clear separation between requests (which expect replies) and commands (fire-and-forget updates). This pattern aligns naturally with the client-server model: synchronous calls for requests, asynchronous casts for updates. The callbacks `handle_call` and `handle_cast` ensure the server's internal state is updated correctly in each case. While it is possible to introduce asynchrony

within a process performing `call` requests, by using the `spawn` operator to handle delayed responses, the spawned process itself will still be blocked until the response arrives, preserving the synchronous semantics of `call`.

By relying on `gen_server`, `Erlang` developers can implement concurrent services with well-defined interfaces and built-in fault recovery. The framework manages the event loop, state passing, and integration with supervisors, allowing programmers to focus on concise callback code that implements the specific application logic. As a result, OTP and `gen_server` are widely regarded as best practices for building robust, concurrent servers in `Erlang` [29, 8].

CHAPTER 3

WALTZ

In this chapter, we introduce WALTZ ♭♩, the specification language that allows us to specify the relations between actor interactions, the messages that they exchange, and relate the contents of the messages exchanged in the system. Notably, the language semantics allow for context-aware verification of properties, which is one of its main strengths in highly concurrent distributed client-server systems.

WALTZ is designed around three fundamental abstractions that directly correspond to the concepts of actor-based systems: actors are autonomous computational entities, message patterns are the communication signatures between them, and state bindings that will correlate information across chained interactions of messages. These abstractions work together to provide a natural and intuitive way to express properties that span multiple actors. Broadly, the language consists of two key components. One dictates the control flow of the messages, defining how messages are exchanged, chained, and related across different actors. The other focuses on verifying logical properties within the content of these messages, ensuring correctness within the established flow.

3.1 Motivation

The necessity of defining a specification language for actor-based systems arise from the paradigm these systems follow and the limitations of other formalisms and tools in addressing such systems. The differences are explained in more detail in Chapter 6.

LTL, as we have seen in Section 2.3.1, is one form of input to RV frameworks, but the natural abstraction level of actor-based systems differs from what general temporal logics provide. It is not simply a matter of convenience; it is more of a semantic alignment between the specification language and the computational model. Another problem of LTL is that it assumes a global state that evolves linearly over time. However, actor-based systems are

built on the principle of local state and asynchronous interactions. Actor-based systems often need properties that span multiple locality levels, local actor state, actor-to-actor relationships, and system-wide properties. These typically force us to choose a global or local level, making mixed properties cumbersome.

Runtime verification needs efficient monitors. Translating actor-specific properties through general formalisms often creates monitors that track unnecessary global state, making them inefficient for distributed runtime checking. Our specification language is designed to align with natural instrumentation points in actor systems (message sends, receives, state changes), while general formalisms might require extensive additional instrumentation. Even if we used these other formalisms, the specifications themselves can become cumbersome, which may lead to incorrect property specifications. The closer the formalism is to the way the system works, the less error-prone that formalism will be. Research has shown that specification languages closer to the domain reduce cognitive load and error rates [5]. Additionally, a domain-specific language can offer more effective error messages, enhanced debugging support, and seamless integration with actor-based development tools.

3.2 The Language

We begin by examining the formal grammar of WALTZ and the semantics to evaluate WALTZ formulas. Particular attention is given to the role of a monitor in the runtime evaluation of such formulas, where the presence of uncertainty must be addressed. In this context, the monitor does not yield a binary outcome alone; instead, it produces one of three possible verdicts: satisfaction, violation, or inconclusiveness.

In an actor-based system, various operations are available, as discussed in Section 2.8. For the purposes of WALTZ, however, our focus lies on the `send` operation, as it provides a sufficient basis for defining the properties we aim to verify. Although other operations, such as `receive`, could be incorporated into the language in future extensions, our current emphasis remains on message sending as the central abstraction.

The primary goal of WALTZ is twofold: first, to allow the definition of relationships between message payloads across multiple interactions, and second, to evaluate these interactions within the same causal context. Together, these capabilities enable the specification of rich properties that can span entire system interactions or focus on a smaller subset, depending on the user's requirements. The level of detail in property definitions, as well as the scope of runtime monitoring, is fully determined by the user, providing flexibility to target both broad system behaviour and specific interaction patterns.

3.2.1 Definition

WALTZ addresses two complementary aspects of actor-based systems: the relationships between messages and the logical reasoning over variables and data contained within those

messages. Its logical fragment resembles programmable code, allowing users to define variables, perform binary operations, and essentially use it as a small, domain-specific programming language. All of these specifications are then compiled into executable `Erlang` code, enabling seamless integration with the monitored system.

Definition 3.2.1 (WALTZ Formula)**.** The set of WALTZ formulas is defined by the following grammar:

$$
\begin{aligned}
\varphi &::= \Omega(\varphi) \mid \Theta(\varphi) \mid \varphi \,;\, \varphi \mid \alpha : \delta \\
\delta &::= \top \mid \bot \mid c \mid \neg\delta \\
\alpha &::= \texttt{send}_{\texttt{id}\to\texttt{id}}\{\texttt{M}\} \\
\texttt{M} &::= \texttt{T} \mid \texttt{T}\,,\texttt{M} \mid \{\texttt{M}\} \\
\texttt{T} &::= (\texttt{cons} \mid \texttt{var})
\end{aligned}
$$

Here, α represents an interaction signature in an actor-based system, c is a boolean constraint, and M is the signature of a message. The modal operators Ω and Θ are explored in Section 3.3. The expression $\alpha : \delta$ is the core of WALTZ, and all verification will boil down to this expression. $\alpha : \delta$ indicates that a message in the trace with signature α must satisfy the constraint δ. If we want to simply capture a message without imposing any condition, we define $\alpha : \top$, as every message trivially satisfies $\top$.

A key construct in WALTZ is the chain operator, denoted by `;`, which sequences two formulas. For example, $\alpha_1 : \delta_1 \,;\, \alpha_2 : \delta_2$ specifies that a message matching α_1 must first be observed and satisfy δ_1, and only afterward should a message matching α_2 be observed and satisfy δ_2. This sequencing is essential for expressing the order in which messages should occur. An important note is that these sequences are "unbreakable", meaning that if we do not satisfy all the boolean constraints for each message in the chain, the property will not be satisfied.

A signature (α) in an actor-based system represents a pattern for a message involving two different actor entities and a `send` operation. For example, the signature $\texttt{send}_{A\to B}$ `{X, Y, {Z, ok}}` specifies the operation type (`send`), the source (`A`) and destination (`B`) actors, and the structure of the message along with its payload. At runtime, messages sent in the system are checked against these signatures to ensure they match the defined pattern, and whenever they do, the values will be bound to the variables at runtime.

Variables written in uppercase will be assigned values at runtime when a message with a specific signature is received. On the other hand, lowercase identifiers are fixed atoms that will be matched by the program at runtime for specific messages.

3.2.2 Semantics

An interaction with the system produces a trace composed of various events, some of which are causally or logically related. We refer to such related events as belonging to the same **context**. A trace may contain multiple contexts, each representing an independent

chain of events. Our goal is to verify properties over these individual contexts. Since event ordering is not always guaranteed and interactions may overlap, we assign a context identifier to each event to determine which events belong together. Formally, a message will be mapped to a context that represents a causal chain in the system. Let C be the set of all context identifiers (e.g., Δ_0, Δ_1). We introduce a function $\gamma : \Sigma \to C$ that maps each event (a message) in our trace to a unique context identifier. Intuitively, $\gamma(\sigma_i)$ tells us which context the event σ_i belongs to.

Δ_0 denotes the main context, which always exists, and additional contexts may be introduced depending on how our system evolves. Our monitors will also interpret them depending on how our properties are specified. Contexts in the system can be structured hierarchically, allowing for the modelling of nested behaviours. We define a hierarchical relation between two contexts using the notation $\Delta_b \blacktriangleright \Delta_a$, which denotes that context Δ_b is derived from context Δ_a. This hierarchical relationship captures the causal or structural dependency between contexts.

Definition 3.2.2 (WALTZ Semantics)**.** Let $\sigma = \sigma_1\sigma_2... \in \Sigma^\omega$ be an infinite sequence of events, C the set that represents the contexts and the relations between them in the given trace, γ the function that maps messages to a context, and Δ is the current context. Then the semantics of WALTZ is defined as follows:

$$\sigma, \Delta \models \alpha : \delta \text{ iff } \exists_{i \geq 0} : \sigma_i \models \delta \text{ and } \sigma_i \bowtie \alpha \text{ and } \gamma(\sigma_i) = \Delta$$
$$\sigma, \Delta \models \varphi \; ; \; \varphi' \text{ iff } \exists_{i \geq 0} : \sigma^i, \Delta \models \varphi \text{ and } \exists_{j \geq 0} : \sigma^j, \Delta \models \varphi' \text{ and } i < j$$
$$\sigma, \Delta \models \Omega(\varphi) \text{ iff } \forall_{\Delta_k \blacktriangleright \Delta} : \sigma, \Delta_k \models \varphi$$
$$\sigma, \Delta \models \Theta(\varphi) \text{ iff } \exists_{\Delta_k \blacktriangleright \Delta} : \sigma, \Delta_k \models \varphi$$

where $\bowtie$ is the operator that checks if a message has a given signature, and $\blacktriangleright$ is the operator that relates a sub-context with a higher one in the hierarchy.

Intuitively, given a trace σ and the existing contexts C, the following can be stated: $\sigma, \Delta \models \alpha : \delta$ means that three conditions must hold simultaneously. First, a message with signature α must be observed in the trace. Second, this message must satisfy the constraint δ. Finally, the message must belong to the context Δ in which it is being evaluated. Only when all three requirements are met is the property considered satisfied.

We say that σ, Δ satisfies $\varphi; \varphi'$ if there exists a suffix of the trace starting at position i that satisfies φ, and another suffix starting at position j that satisfies φ', where the event at position j occurs after the event at position i. The operator ; allows the definition of message chains by enforcing an order between events and specifying the boolean constraints that each message in the chain must satisfy. This mechanism enables the specification of properties involving multiple actors by relating the contents of message payloads across events. Importantly, all events in the same chain belong to the same context family.

The Ω and Θ operators are the most nuanced constructs in the language, as they introduce the evaluation over different contexts. Formally, a trace σ under the set of

contexts C, and current context Δ, satisfies $\Omega(\varphi)$ if, for all contexts Δ_k such that $\Delta_k \blacktriangleright \Delta$ (i.e., Δ_k was derived from Δ), the formula φ holds in each derived context. In short, $\Omega(\varphi)$ universally quantifies over all sub-contexts forked from the current one, requiring that φ be satisfied in each.

The Θ operator functions analogously to the Ω operator; however, instead of verifying the property across all possible contexts, it exhibits existential semantics, succeeding if there exists at least one context in which the property holds.

The verification process begins with a shared initial context, Δ_0, which serves as the root for all subsequent context derivations. As the system executes and contexts are assigned to messages, causal relationships are progressively established, enabling the monitor to accurately track dependencies and interactions throughout the system.

3.2.3 Monitors

A WALTZ monitor will behave precisely the same as this definition of a monitor given in Section 2.4.2; we will only be able to state a verdict whenever all the continuations of our trace allow us to give that verdict, that is, we can yield an irrevocable verdict. In the case of Ω, that will happen when one of the contexts violates the property, and in the case of Θ, that will happen when one of the contexts satisfies the property. Otherwise, we keep monitoring the system to yield an inconclusive verdict, indefinitely.

Certain properties may be impossible to evaluate to a definitive verdict at runtime due to the potentially infinite nature of the traces being analysed. The challenges of non-monitorability and the risk of becoming stuck with an inconclusive verdict are discussed in greater detail in Section 3.9.

3.2.4 Chains Of Messages

A sequence of messages can be expressed using the chain operator `;`, which concatenates individual messages to form a single message chain. By constructing such chains, certain guarantees about their structure and interpretation naturally arise:

- The order of the messages is preserved, that means, that whenever we verify a property we are expecting that the messages come in that following order and not any other.
- Chains are unbreakable. When verifying a chain of messages, the entire property is considered unsatisfied if even a single message in the chain fails to meet its boolean constraint. In other words, the violation of one element invalidates the chain as a whole.
- Even though we can have infinite different message chains in the system, the chains themselves have a beginning and an ending.

3.2.5 Chains in Action

In `Erlang`, a common interaction pattern consists of one process sending a request to another, which may subsequently issue further requests to additional processes. Eventually, a response propagates back to the original caller, completing the request–response cycle. This interaction is considered the message chain for that specific caller. Of course, multiple such requests can be performed, and we want to isolate each one of them in a bubble that does not interfere with any other message chain. Our properties will then be evaluated in these isolated chains, with the possibility of having hierarchical relations between them, as we will see ahead. There is a causality that is shared among all the messages in such chains, and we must have that in account when verifying our properties.

The interaction pattern considers the typical behaviour of `Erlang` systems, which will always depend on the specific design of the system. When defining our properties, it is essential to have a clear understanding of how messages are exchanged in the system we want to verify.

Example 7. In this system, a central server handles requests from multiple clients. Each client sends a number to the server, which then multiplies the number by two and returns the result to the originating client. The behaviour of this system is illustrated in Figure 3.1.

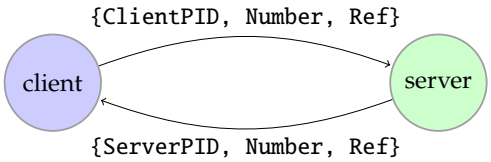


Figure 3.1: Multiplying number in request by 2

As we can see, this program is following the `Erlang` conventions of attaching references to the messages, so we can easily detect which messages are causally related. Therefore, we can clearly see which messages belong to the same chain of messages. This association is precisely what we mean by different chains of messages. Even if the same client interacts again with the system, the chain will still be considered an isolated chain from the others.

Consider σ_1 and σ_2, where the latter was produced by a faulty implementation of the `server` process.

$$\sigma_1 = \texttt{\{client1, 10, \#1\}}\,\texttt{\{client2, 20, \#2\}}\,\texttt{\{server, 40, \#2\}}\,\texttt{\{server, 20, \#1\}}$$

$$\sigma_2 = \texttt{\{client1, 10, \#1\}}\,\texttt{\{client2, 20, \#2\}}\,\texttt{\{server, 20, \#2\}}\,\texttt{\{server, 20, \#1\}}$$

Depending on whether the system is correctly implemented, it will adhere to the property that we expect, which is returning the original number multiplied by two. This can be specified by the following property:

$$\varphi = \Omega\left(\begin{array}{l}\texttt{send}_{\texttt{client}\to\texttt{server}}\texttt{\{\_, Number, Ref\}} : \top\,; \\ \quad \texttt{send}_{\texttt{server}\to\texttt{client}}\texttt{\{\_, NumberReturn, Ref\}} : \texttt{NumberReturn = 2 * Number}\end{array}\right)$$

Since every message chain operates within the same context, the Ω modal operator evaluates whether all contexts, that is, all individual message chains, satisfy a given property. By specifying the property φ, we can capture the desired system behaviour. In the runtime setting, the monitor generated from φ continuously checks for violations. If no violation occurs in the observed trace, the monitor outputs the inconclusive verdict "?".

Under this framework, the property φ is not violated in trace σ_1, whereas it is violated in trace σ_2 because one of the message chains fails to satisfy the boolean constraint defined by φ. This example illustrates how message chains are tracked and interpreted by the WALTZ monitor, highlighting the role of causal context in runtime verification.

3.2.6 Filtering Chains - Turning Infinity Into Finity

Before moving on, it is of great importance to understand how messages will exist within the different contexts. The trace is a potentially infinite entity that grows continuously as the system executes. However, for evaluations within different contexts, the message chain inside each context is treated as a finite trace.

We will be working with sub-traces, which compose the main trace being produced by the system. The internal mechanisms will automatically filter all the relevant messages present in the specification of a property to compose this sub-trace. We can think of this as messages flowing into the right place, that is, into the right context. Of course, message chains can be extremely long, but realistically they have a reasonable finite size.

This does not mean that the overall trace will be treated as finite. Rather, the intention is that the root context, Δ_0, may have infinitely many child contexts, but within each child context, the corresponding message chain is regarded as finite. Consider the trace $\sigma = \_\, \mathtt{m}_1 \,\_\, \mathtt{m}_2 \,\_\, \mathtt{m}_3 \,\_\, \mathtt{m}_4$ with the following context assignments $\gamma(\mathtt{m}_1) = \Delta_1, \gamma(\mathtt{m}_2) = \Delta_2, \gamma(\mathtt{m}_3) = \Delta_1, \gamma(\mathtt{m}_4) = \Delta_2$.

In this case, the messages belonging to the same context preserve their relative order. For example, within Δ_1, message $\mathtt{m}_1$ precedes $\mathtt{m}_3$, while in Δ_2, $\mathtt{m}_2$ precedes $\mathtt{m}_4$. Thus, although the global trace may be infinite, each context isolates a finite, ordered chain of messages that can be reasoned about independently. The structuring of our context tree C, is visible in Figure 3.2, which highlights the idea of the $\blacktriangleright$ association between the contexts in the system, in this case, $\Delta_1 \blacktriangleright \Delta_0$ and $\Delta_2 \blacktriangleright \Delta_0$.

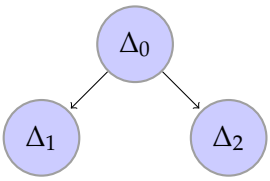
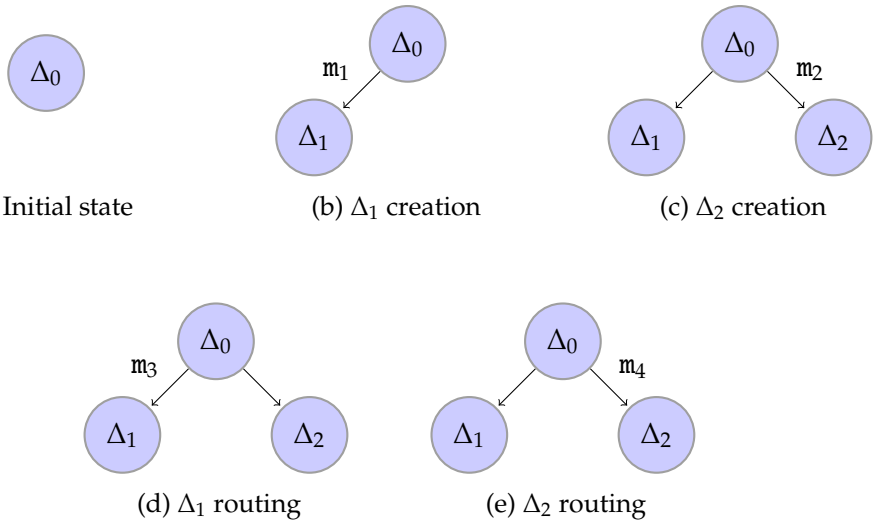


Figure 3.2: Context tree of the trace σ given by C

In order to have a clear understanding of how these contexts exist and are "bound" to message chains, consult Figure 3.3 which depicts a step by step context tree creation that eases the understanding of this concept. Each message is routed to its appropriate context,

and if no such context exists, a new one is created to host the corresponding message chain.



(a) Initial state (b) Δ_1 creation (c) Δ_2 creation

(d) Δ_1 routing (e) Δ_2 routing

Figure 3.3: Evolution of the context tree showing the dynamic construction process

Each sub-trace within a context has a well-defined beginning, marked by the anchor of its message chain, and a corresponding end, which enables the verification of properties over these finite segments. As we will see later, this property holds particularly for the leaf nodes of the context tree. Once hierarchical relations are introduced, however, new sub-traces may no longer have a natural endpoint and can instead grow indefinitely. This aspect will be examined in more detail in Section 3.4.

This constitutes our mechanism of verification: traversing the different contexts and checking properties over them. Depending on the chosen operator, verification may require ensuring compliance across all contexts or identifying at least one context where the property holds. These ideas anticipate the modal operators of WALTZ, which are discussed in detail in Section 3.3.

3.2.7 Why Context Matters?

The concept of a context is central to verifying the satisfiability of a WALTZ formula. As previously discussed, the main context, denoted Δ_0, always exists and can be regarded as representing the system. It acts as a primary reference point, or oracle, for verification. Whenever a new context is introduced, hierarchical relationships are automatically established between it and existing contexts. The evaluation of properties then depends on the chosen operator, such as Ω or Θ. Therefore, in order to evaluate a WALTZ formula, the entities that exist are the trace σ, the context tree C, and the mapping from the messages to the respective context function γ. For now, this is simply a mathematical function that maps messages to the respective context.

Understanding the context verification mechanism and how it aids and supports the property satisfiability is of great importance since it is the core of our language and

verification process in concurrent systems. In this section, whenever we associate a colour to a message it means that it belong to a given context, therefore, all messages sharing the same colour belong to the same context. We also use a simple notation to denote the satisfaction or violation of boolean constraints, using the symbol ✔ for satisfaction and ✘ for violation, and irrelevant messages produced by the system are symbolised by _.

Example 8. The system architecture is designed around a central server that routes incoming requests to specialized worker processes. In this simplified implementation, the server receives an input list and delegates it to a worker that removes duplicate values. The resulting duplicate-free list is then passed to another worker responsible for sorting. Together, these components ensure that the final output is both free of duplicates and properly ordered. The overall workflow is illustrated in Figure 3.4.

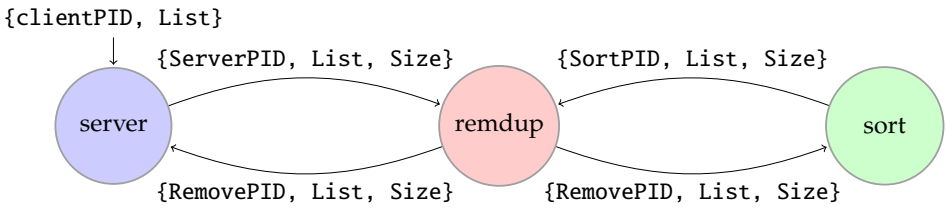


Figure 3.4: Actor-based system that operates over a list

We will use the dummy system from Figure 3.4 throughout this section to ease the explanation of context aware verification and the necessity of having it. At first, let us ignore completely the context management, that is, the function γ will assign the same global context to all the messages of our interest that come from the system trace. Consider the trace σ_1 depicted below:

$$\sigma_1 = \_\ \mathtt{m}_1\ \_\ \_\ \mathtt{m}_2\ \_\ \_\ \mathtt{m}_3\ \ldots$$

If we wanted to evaluate property $\varphi_1 = \Omega(\alpha : \delta)$, over trace σ_1, the evaluation of this property actually depends on how the context is assigned to each message. What we would like to capture with this property is that "for every context, we want to check that if there exists a message with signature α it adheres to the boolean constraint δ". In the case of our system, a property of such kind could be "all the lists exiting the `sort` process are sorted". Now here is the nuance, if the context mapping for the messages is $\gamma(\mathtt{m}_1) = \Delta_1$, $\gamma(\mathtt{m}_2) = \Delta_1$, and $\gamma(\mathtt{m}_3) = \Delta_1$, that is, they all share the same context, and assume that all the messages have signature α, which is the one we are trying to capture. What would result of the verification of trace σ_1 with the boolean constraints present in the trace σ_1 is the satisfaction of the property φ_1.

$$\sigma_1 = \_\ \mathtt{m}_1{}^{\text{✘}}\ \_\ \_\ \mathtt{m}_2{}^{\text{✘}}\ \_\ \_\ \mathtt{m}_3{}^{\text{✔}}\ldots$$

The property φ_1 is satisfied under the conditions present, due to the following semantic rule:

$$\sigma, \Delta \ \models\ \alpha : \delta \text{ iff } \exists_{i \geq 0} : \sigma_i \models \delta \text{ and } \sigma_i \bowtie \alpha \text{ and } \gamma(\sigma_i) = \Delta$$

In this case, we only need to identify a message within the context that fulfills the property. This is exactly what occurs in the trace, since $\mathtt{m}_3$ satisfies δ, thereby ensuring the property holds for the entire context. Drawing a parallel with our earlier example, under the current interpretation, where all messages are assigned to the same context, we risk allowing unsorted lists. This is because the property would be considered satisfied as long as at least one list is sorted, even if others are not.

However, that is not what we actually want to represent with property φ_1; we would like to have the verification that **every** message that has signature α, satisfies δ, essentially isolating every message with signature α in its own separate context from the others. To achieve this, we introduce the notion of context by redefining the context mapping as follows: $\gamma(\mathtt{m1}) = \Delta 1$, $\gamma(\mathtt{m2}) = \Delta 2$, and $\gamma(\mathtt{m3}) = \Delta 3$. With these mappings in place, we are now verifying the intended requirement: that for every message with signature α, the property must hold. In the case of the trace σ_1, this ensures the property is immediately violated, since two of the messages fail to satisfy δ.

$$\sigma_1 = \_\ \mathtt{m}_1{}^{\times}\ \_\ \_\ \mathtt{m}_2{}^{\times}\ \_\ \_\ \mathtt{m}_3{}^{\checkmark} \ldots$$

One might ask: what is the correct way to assign contexts? In fact, there is no single correct way, as context assignment occurs at runtime and establishes the causal relations between messages. The monitors, generated from the specified properties, then interpret these contexts according to how the property is defined. More precisely, they interpret the **chains** of messages that are causally related, as determined by the function γ. For now, we will treat this as given, but in Chapter 4 we will explore in detail how the γ operates.

Our context management strategy assigns the same context to all messages belonging to the same chain. Consequently, when we have $\alpha : \delta$, each message with signature α resides in its own context, since in this case the "chain" consists solely of the message itself. Recall that whenever we specify $\varphi_2 = \alpha_1 : \delta_1 \ ; \ \alpha_2 : \delta_2$, the chain becomes unbreakable. This means that if, at any point within the chain, a boolean constraint is not satisfied, the chain is considered broken, and verification must continue with another chain from a different context. Depending on the monitored property, such a break may constitute a violation. For instance, if we are monitoring $\Omega(\varphi_2)$, then the violation of a single chain would cause the entire property to be violated.

We can think of this in the same way as `Erlang` interactions, where each request lives in its own bubble, these bubbles correspond to our contexts. For every chain (interaction), we assign a dedicated bubble in which verification takes place.

To have another perspective on this and understand how it works for the Θ operator consider property $\varphi_2 = \Theta(\alpha_1 : \delta_1 \ ; \ \varphi_2 : \delta_2)$, over the following trace:

$$\sigma_2 = \_\ \mathtt{m}_1\ \_\ \_\ \mathtt{m}_2\ \_\ \_\ \mathtt{m}_3\ \_\ \_\ \mathtt{m}_4\ \ldots$$

and the context mapping: $\gamma(\mathtt{m}_1) = \Delta_1, \gamma(\mathtt{m}_2) = \Delta_2, \gamma(\mathtt{m}_3) = \Delta_1, \gamma(\mathtt{m}_4) = \Delta_2$, which is depicted by C in Figure 3.5.

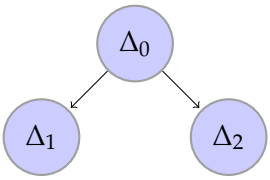


Figure 3.5: Context tree of the trace σ_2 given by C

Each context will have a dedicated monitor that checks the property $\alpha_1 : \delta_1$ and $\alpha_2 : \delta_2$ individually, recreating the chain. In the next chapter, we will explore how this process is carried out. Each monitor will receive only the filtered messages that are necessary for its contextual interpretation, resulting in finite sub-traces for evaluation, as illustrated in Figure 3.6. Within each context, we will assess the specified property; while the overall trace continuously grows, the interpretation for each message chain remains finite. Although the number of chains produced is infinite, we are grouping them and directing them to the appropriate monitor to verify property satisfaction. Thus, even if the root context has an infinite number of children, each child context will have a finite perspective on the global trace of the system.

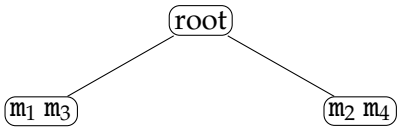


Figure 3.6: The monitor structure for property φ_2

Assume that $\mathtt{m}_1$, $\mathtt{m}2 \bowtie \alpha 1$ and $\mathtt{m}_3$, $\mathtt{m}4 \bowtie \alpha 2$, with the boolean constraints evaluated as follows:

$$\sigma_2 = \_\, \mathtt{m}_1{}^{\checkmark} \_\_ \mathtt{m}_2{}^{\checkmark} \_\_ \mathtt{m}_3{}^{\times} \_\_ \mathtt{m}_4{}^{\checkmark} \ldots$$

In this scenario, the property is satisfied because at least one context, namely Δ_2, adheres to it. However, if all these messages are placed in the same context, that is, no context management exists whatsoever, the situation changes. Consider the same trace, but with new boolean outcomes:

$$\sigma_2 = \_\, \mathtt{m}_1{}^{\checkmark} \_\_ \mathtt{m}_2{}^{\times} \_\_ \mathtt{m}_3{}^{\times} \_\_ \mathtt{m}_4{}^{\checkmark} \ldots$$

At first glance, this trace would be accepted as satisfying the property. Yet this interpretation is misleading. If $\mathtt{m}_1$ and $\mathtt{m}_3$ are causally related (thus belong to the same context), and likewise $\mathtt{m}_2$ and $\mathtt{m}_4$, then within each causal chain the property is never satisfied. In other words, under proper context management, the monitor should state inconclusiveness. Treating all messages as part of a single context therefore produces incorrect verdicts that do not reflect the true causal structure.

To draw a parallel with our example, consider the trace of interactions from two different clients with the system. These interactions can be represented by the sub-traces σ_{c1} and σ_{c2}. In this case, the traces reflect a buggy version of the system, where the lists

of data between the two clients are inadvertently swapped.

$$\sigma_{c1} = \texttt{\{<0.1.0>, [1,2,2], 3\} \{<0.2.0>, [5,1,3], 3\}}$$
$$\sigma_{c2} = \texttt{\{<0.1.0>, [5,1,3,5], 4\} \{<0.2.0>, [1,2], 2\}}$$

One property of interest is whether the `remdup` process correctly removes all duplicates from the original list, denoted by φ. This can be verified by comparing the list received by `remdup` from the `server` with the list it subsequently sends to `sort`, ensuring that all duplicates have been eliminated.

Considering that there is no context management, that is, all the messages belong to the same context, observing both sub-traces, we can see that the lists get swapped for both the clients, due to a buggy management of the `remdup` process. If we were to verify $\Theta(\varphi)$, the property would be satisfied, since all the messages are evaluated within the same context. But, that is definitely not the case, since individually, none of the client interactions respected the property φ.

The issue of not having context management is that we become prone to erroneous verdicts, as illustrated above. This underscores the necessity of relating each causal chain individually, so that information from one context is not mixed with another, avoiding false verdicts by the monitor. This is precisely why context plays a critical role in the verification process. We explore how context is managed in Chapter 4.

3.3 The Modal Operators Ω and Θ

Now that we have established the notions of message chains and the role of contexts, we can turn to the modal operators of WALTZ and examine how they enable the verification of different classes of properties. As discussed, multiple contexts may coexist throughout the lifespan of the system, and our verification process operates over these contexts.

In some cases, we may wish to express that all message chains must satisfy a given property; in others, it may be sufficient that at least one message chain does so. The modal operators provide exactly this expressive power, allowing us to capture such requirements in a precise and systematic way.

3.3.1 The Θ Operator

Whenever we use $\Theta(\varphi)$, we are stating that we want to see the satisfaction of φ in at least one context, out of all the existing ones in the system. Imagine that the contexts existing in our system have the structure depicted in Figure 3.7.

The operator Θ enables inspection across all contexts forked from the main context (Δ_0), checking whether the property holds within each of them. Since every context is associated with a finite trace, it can provide a verdict based solely on the events it observes.

If none of the current contexts satisfy the property under inspection, the monitor issues an inconclusive verdict. This arises because Θ requires that at least one context must

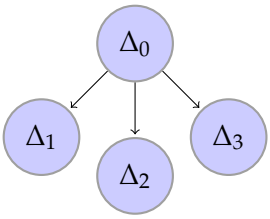


Figure 3.7: Context tree example

satisfy the property, and future child contexts of Δ_0 may still do so. Once a single context is found to satisfy the property, the verdict can be confidently concluded as satisfied, regardless of subsequent behaviour. Conversely, if no context meets the property, the monitor remains inconclusive, as illustrated in Figure 3.8.

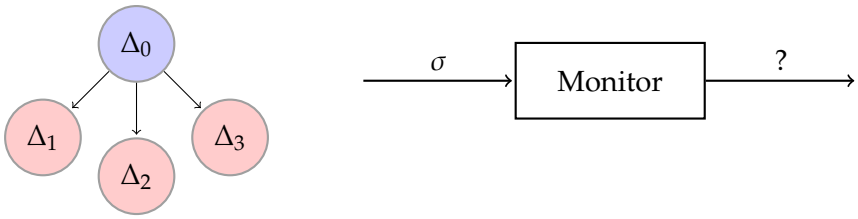


Figure 3.8: Inconclusive verdict for Θ

Now, suppose that one of the contexts successfully satisfies the property. In this case, the monitor will produce a conclusive verdict, indicating that the property has been satisfied. This behaviour is illustrated in Figure 3.9.

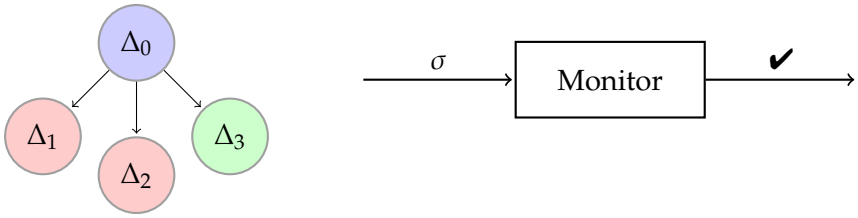


Figure 3.9: Conclusive verdict for Θ

Consider a concrete example. Since all new contexts in our system are forked from the initial context Δ_0, when verifying the property $\varphi = \Theta(\alpha : \delta)$ we check whether at least one of the forked contexts satisfies the property. According to the semantics defined previously, this is formalized as:

$$\sigma, \Delta \models \Theta(\alpha : \delta) \text{ iff } \exists_{\Delta_k \blacktriangleright \Delta} : \sigma, \Delta_k \models (\alpha : \delta)$$

In practice, this rule requires evaluating the formula for all contexts forked from Δ_0, namely Δ_1, Δ_2, and Δ_3, as illustrated in Figure 3.7. The verification then reduces to checking each context individually:

$$\varphi_{11} = \sigma, \Delta_1 \models (\alpha : \delta)$$
$$\varphi_{12} = \sigma, \Delta_2 \models (\alpha : \delta)$$
$$\varphi_{13} = \sigma, \Delta_3 \models (\alpha : \delta)$$

Since we are checking whether at least one message with signature α satisfies δ, the monitor can declare the property satisfied, as illustrated in Figure 3.9. Conversely, if none of the contexts comply with the defined property, the monitor will reach an inconclusive verdict, as shown in Figure 3.8. This is because, at runtime, the monitor continues to observe the system: it seeks satisfaction of the property, and if none is found yet, it keeps monitoring for potential future contexts that may satisfy the constraint.

Each sub-formula, φ_{11}, φ_{12}, and φ_{13}, is evaluated within its respective context. As noted earlier, every leaf context has a finite endpoint, so the verification of whether the required message exists will always yield a definitive success or failure verdict, there is no inconclusive verdict at this level of the context tree. In contrast, the property as a whole spans all possible incoming message chains, which may be potentially infinite. Within each child context, we therefore carry out the following verification procedure:

$$\sigma, \Delta \models \alpha : \delta \text{ iff } \exists_{i \geq 0} : \sigma_i \models \delta \text{ and } \sigma_i \bowtie \alpha \text{ and } \gamma(\sigma_i) = \Delta$$

Each context will individually evaluate the property and return either true or false. If any context evaluates to true, the property is satisfied for the system as a whole. If a context evaluates to false, the monitor continues checking the remaining contexts to determine whether at least one satisfies the property.

Consider the example presented in Figure 3.4 and the property φ_1, which uses the `isSorted` predicate to verify whether a list is sorted:

$$\varphi_1 = \Theta(\texttt{send}_{\texttt{sort}\rightarrow\texttt{remdup}}\{\_,\ \texttt{List},\ \_\} : \texttt{isSorted(List)})$$

In this example, we aim to identify at least one list that is sorted across all system interactions. Consider the following trace:

```
{sort, [9,5], 2}✘
{sort, [5,1,3,5], 4}✘
{sort, [1,2], 2}✔
```

Here, the property is satisfied because at least one context contains a sorted list (in this case, `[1,2]`). Essentially, the monitor evaluates each context individually, checking whether the list in that context satisfies the `isSorted` predicate. Once it finds a context where the predicate holds, the property is considered satisfied for the system as a whole. Otherwise, it emits the inconclusive verdict.

3.3.2 The Ω Operator

The Ω operator verifies whether all contexts forked from the main context satisfy the specified property. While it inspects all contexts like Θ, it differs in its satisfaction criteria, focusing on detecting violations rather than satisfactions.

If all current contexts satisfy the property, the monitor issues an inconclusive verdict. This occurs because Ω is concerned with identifying a context that violates the property.

The inconclusive verdict is not problematic: as soon as a context is found that does not satisfy the property, we can definitively conclude that the property is violated, regardless of future system behaviour, as seen in Figure 3.10.

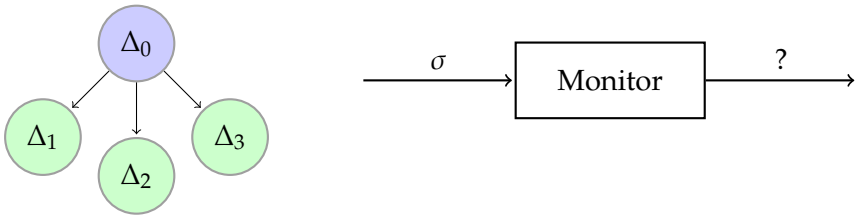


Figure 3.10: Inconclusive verdict for Ω

Now, suppose that one of the contexts violates the property. In this case, the monitor will produce a conclusive verdict indicating that the property has been violated, as illustrated in Figure 3.11.

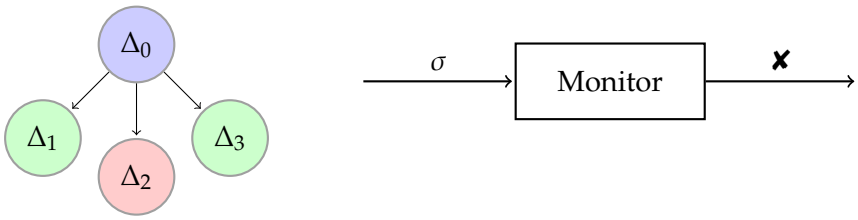


Figure 3.11: Conclusive verdict for Ω

To illustrate this with a concrete example, consider a system where all new contexts are forked from the initial context. When evaluating the property $\varphi = \Omega(\alpha : \delta)$ the monitor examines each forked context individually to determine whether it satisfies the property. According to the semantics defined earlier, this amounts to checking:

$$\sigma, \Delta \models \Omega(\alpha : \delta) \text{ iff } \forall_{\Delta_k \blacktriangleright \Delta} : \sigma, \Delta_k \models (\alpha : \delta)$$

This rule will result into the verification of the formula for all existing contexts that forked Δ_0, which are Δ_1, Δ_2, and Δ_3, as seen in Figure 3.7. Then the verification will boil down to:

$$\varphi_{11} = \sigma, \Delta_1 \models (\alpha : \delta)$$
$$\varphi_{12} = \sigma, \Delta_2 \models (\alpha : \delta)$$
$$\varphi_{13} = \sigma, \Delta_3 \models (\alpha : \delta)$$

Since we are verifying whether all messages with signature α satisfy the property δ, the monitor can issue a violation verdict as soon as it encounters a context where the property does not hold, as illustrated in Figure 3.11. If every context satisfies the property, the monitor produces an inconclusive verdict, as shown in Figure 3.10. This reflects the runtime behaviour: the monitor seeks violations, and if none are observed, it continues monitoring for potential future violations.

The evaluation of each sub-formula, φ_{11}, φ_{12}, and φ_{13}, proceeds similarly to the process described for Θ, but under the Ω semantics, the focus is on detecting violations rather than confirming satisfaction.

Consider the example presented in Figure 3.4 and the property φ_2, which uses the `isSorted` predicate again:

$$\varphi_2 = \Omega(\texttt{send}_{\texttt{sort}\rightarrow\texttt{remdup}}\{\_,\ \texttt{List},\ \_\} : \texttt{isSorted(List)})$$

In this case, we are interested in verifying whether all lists are sorted across the system interactions under observation. Consider the following trace:

```
{sort, [5,9], 2}✔
{sort, [1,3,5,5], 4}✔
{sort, [1,2], 2}✔
```

Up to this point, all lists observed are sorted, so no violation has been detected. However, since the system may generate additional contexts in the future, it is still possible that the property could be violated later. As a result, the monitor produces an inconclusive verdict, indicating that the property has not yet been violated, but a final determination cannot be made until a violation appears.

3.4 Nesting Modal Operators

It is also possible to nest our modal operators, which expands the expressiveness of the logic and allows us to capture more intricate relationships between contexts. At the same time, this introduces additional complexity in understanding how such properties are verified at runtime. Both Ω and Θ can be nested arbitrarily. However, care must be taken due to the nature of runtime verification, certain nested structures may in practice turn out to be not monitorable. We will return to this point in Section 3.9, where the monitorability of such properties is examined in detail. We now turn to a nesting of the same operator, where properties can be specified using nested modal operators. Consider the following property φ:

$$\varphi = \Omega(\alpha_1 : \delta_1\ ;\ \Omega(\alpha_2 : \delta_2))$$

This property illustrates how modal operators can be combined hierarchically to capture multi-level causal dependencies in the system. Intuitively, it states: for every context, δ_1 must hold, and for every child of that context, δ_2 must also hold.

With a single modal operator, verification was limited to checking properties over the immediate children of the root context Δ_0. By nesting modal operators, the scope extends further: not only are the direct children of Δ_0 checked, but also the children of those contexts, the "grandchildren" of Δ_0.

This hierarchical structure is especially valuable in systems where causal dependencies span multiple levels of interaction. It enables us to capture and enforce constraints across

chains of requests and their sub-requests, ensuring correctness across multiple layers of causality.

To ground the discussion, let us revisit the example introduced in Chapter 2, which describes a chat room system. In this setting, multiple clients may connect simultaneously, and once a connection is established, each client can issue several requests. Our objective is to capture the relationship between a client's initial connection and all subsequent requests they generate. Although these requests should be evaluated independently, they remain causally tied to the same client. This causal link allows us to reason not only about individual requests but also about how they relate back to the client session as a whole. An important property to monitor in this context can therefore be formulated as follows:

φ = For every client, after a connection, all subsequent requests are positive numbers

What we are essentially trying to capture is the following: for every client (each represented by its own context), once a successful connection has been established, all subsequent requests issued by that client must be positive numbers. Each of these requests is tied to a new context forked from the original connection context, thereby preserving the causal relationship between the client and their activity.

This perspective introduces a richer way of reasoning about properties at runtime. While the fundamental concepts of verification remain unchanged, the key difference is that we now need to account for the relationships between contexts, not just what happens within a single one.

To illustrate, imagine an interaction where two clients connect to the server and then proceed to issue a series of requests. The trace of such an execution can be represented as follows:

$$\sigma = \mathtt{m}_1 \_ \mathtt{m}_2 \_ \mathtt{m}_3 \_ \_ \mathtt{m}_4 \_ \mathtt{m}_5$$

Here, $\mathtt{m}_1$ represents the connection of the first client, while $\mathtt{m}_2$ and $\mathtt{m}_4$ correspond to subsequent requests made by that same client. Similarly, $\mathtt{m}_3$ is the connection of the second client, and $\mathtt{m}_5$ is one of its requests.

In this way, the messages naturally group into two chains: one for the first client ($\mathtt{m}_1$, $\mathtt{m}_2$, $\mathtt{m}_4$) and one for the second client ($\mathtt{m}_3$, $\mathtt{m}_5$). In each chain, the initial message has signature α_1 (indicating a connection), while the subsequent messages have signature α_2 (indicating requests). The resulting contexts and their relationships can be visualized in Figure 3.12.

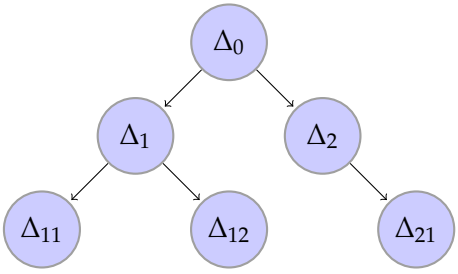


Figure 3.12: The contexts of the trace σ

This structuring of contexts introduces a new perspective: contexts are no longer limited to being direct children of the initial context Δ_0, but may themselves spawn further child contexts. In other words, we now allow a hierarchical organization where any context can act as a parent. While each context is still evaluated independently for property verification, it is nevertheless linked to its parent through this hierarchy. An important consequence of this extension is that chains no longer admit a simple finite representation, since a parent context may depend on the outcomes of multiple possible children. However, each individual context continues to be associated with its own finite sub-trace, ensuring that evaluation within that context remains well-defined.

To make this idea more concrete, let us examine how the context mapping function γ assigns messages to contexts. Consider the following attributions:

$$\gamma(\mathtt{m}_1) = \Delta_1 \; \gamma(\mathtt{m}_2) = \Delta_{11} \; \gamma(\mathtt{m}_3) = \Delta_2 \; \gamma(\mathtt{m}_4) = \Delta_{12} \; \gamma(\mathtt{m}_5) = \Delta_{21}$$

The satisfaction of boolean constraints determines whether a property is violated. In this setting, it is sufficient for a single client to issue a request that is not a positive number to trigger a violation. Assume the following assignments of boolean constraint satisfactions and violations:

$$\sigma = \mathtt{m}_1{}^{\checkmark} \_ \mathtt{m}_2{}^{\checkmark} \_ \mathtt{m}_3{}^{\checkmark} \_ \_ \mathtt{m}_4{}^{\checkmark} \_ \mathtt{m}_5{}^{\checkmark}$$

In such case, the property is not violated, and the monitor would keep monitoring the system. Using our semantics, we can check whether the trace adheres to the property. What we will perform at first is "unpack" the first Ω operator, which results in the verification of $\alpha_1 : \delta_1 \; ; \; \Omega(\alpha_2 : \delta_2)$ upon Δ_1 and Δ_2.

For each sub-formula, we evaluate whether the observed message α_1 satisfies the corresponding constraint δ_1 within both contexts. Because we are assessing a chain of messages, we apply the chain rule, which ensures that the context is consistently shared across all parts of the chain. Considering context Δ_1:

$$\sigma, \Delta_1 \; \models \; \varphi \; ; \; \varphi' \text{ iff } \exists_{i \geq 0} : \sigma^i, \Delta_1 \models \alpha_1 : \delta_1 \text{ and } \exists_{j \geq 0} : \sigma^j, \Delta_1 \models \Omega(\alpha_2 : \delta_2) \text{ and } i < j$$

The first part is straightforward to evaluate. Although Δ_1 may have potentially infinite children, the context itself contains a finite trace, consisting initially of $\mathtt{m}_1$. This message serves as the anchor of the chain, which can then grow indefinitely with all subsequent requests from the same client. In this way, $\mathtt{m}_1$ provides a fixed starting point for evaluating the chain, while the trace continuously expands as the client sends more messages.

The second formula evaluation, $\Delta_1 \models \Omega(\alpha_2 : \delta_2)$ will go through another Ω, which results into the evaluation of:

$$\sigma, \Delta_1 \; \models \; \Omega(\alpha_2 : \delta_2) \text{ iff } \forall_{\Delta_k \blacktriangleright \Delta_1} : \sigma, \Delta_k \models (\alpha_2 : \delta_2)$$

This, will check the children of Δ_1, which are Δ_{11} and Δ_{12}, leading into the evaluation of $\alpha_2 : \delta_2$ in both contexts.

$$\varphi_{11} = \sigma, \Delta_{11} \models (\alpha_2 : \delta_2)$$
$$\varphi_{12} = \sigma, \Delta_{12} \models (\alpha_2 : \delta_2)$$

The trace given with the boolean constraint satisfaction assures that φ_{11} and φ_{12} are satisfied. Due to this, inside the context Δ_1, no violation occurred, which means that all the client requests were indeed positive numbers. However, as we are monitoring that property within an Ω operator, we can only stop whenever a violation is detected; if no violation is detected, then we continue with an inconclusive verdict.

The same rules apply to context Δ_2 and its child Δ_{21}. It is important to remember that this entire verification is encapsulated within an Ω operator. If no client violates the property, the monitor continues observing the system and maintains an inconclusive verdict. This reflects the runtime nature of the system: new clients may still join, meaning Δ_0 could acquire additional children in the future, and connected clients may issue further requests. Consequently, the property must be continuously monitored as the system evolves.

The monitor would reach a verdict if one of the clients violates the property by either not connecting successfully or by sending a message afterwards with a negative payload. In both scenarios, the property will not be satisfied. Importantly, inside context Δ_1 "lives" message $\mathbb{m}_1$ and if it does not satisfy δ_1 then the property also fails, because the evaluation of the chain failed at the beginning.

For the nesting of the Θ operator, the context tree structure remains exactly the same as the one presented in Figure 3.12. The key difference lies in the semantics of the operator itself: unlike Ω, which searches for violations, Θ seeks to identify a satisfaction. Re-defining the property that we had before to:

φ = At least one client, after a connection, has at least one positive number request

Here, we aim to capture a context in which one of its children satisfies the given property. In WALTZ, this property is represented as $\Theta(\alpha_1 : \delta_1 \ ; \ \Theta(\alpha_2 : \delta_2))$, where α_1 denotes the connection signature and α_2 corresponds to the subsequent request with the number signature. By applying the semantics, we can determine whether a given trace satisfies the property, using the same unpacking techniques showcased for the Ω operator.

3.5 Under The Eye Of The Monitor

The interpretation of a formula, when evaluated over a trace and a context, can significantly influence what the monitor observes. Depending on the system, the way contexts are generated may vary, affecting how properties are verified.

Even if the set of contexts C is well-defined, the monitor can interpret them differently depending on the property being monitored. While the context assignment mechanism is independent, the interpretation of those contexts is property-dependent.

To accurately capture the desired behaviours, properties must be defined in alignment with how contexts are generated by the system. In other words, a deep understanding of the system's operational semantics is essential to ensure that properties are verified as

intended. The definition of a property directly shapes the interpretation of the context tree, making system insight crucial for correct verification.

Using the same example from Section 3.4, we can now examine in detail what possible interpretations the monitor perceives. The context tree and the corresponding trace are illustrated in Figure 3.13.

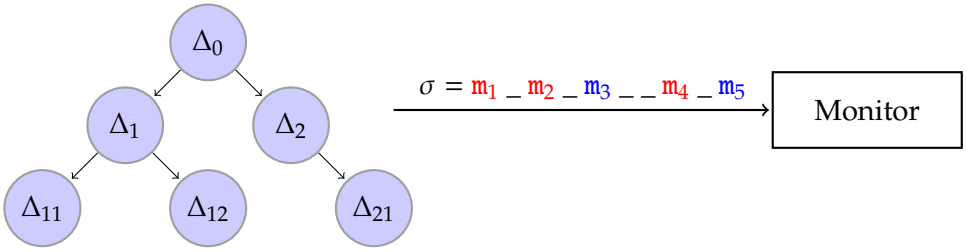


Figure 3.13: Context tree example and trace fed into monitor

The context tree itself does not change, the contexts are managed by γ and allow us to create C. The γ assignments will be the following:

$$\gamma(\mathtt{m}_1) = \Delta_1 \; \gamma(\mathtt{m}_2) = \Delta_{11} \; \gamma(\mathtt{m}_3) = \Delta_2 \; \gamma(\mathtt{m}_4) = \Delta_{12} \; \gamma(\mathtt{m}_5) = \Delta_{21}$$

What changes is the perspective from which the monitor interprets the context tree. In some cases, the properties may closely mirror the structure of the tree itself. However, there are situations where we do not want a one-to-one correspondence; instead, we aim to interpret the tree in alternative ways to capture different aspects of system behaviour.

Consider φ_1 = "At least one client, after a connection, has at least one message with a positive payload". One incorrect way to formalize this property is as follows:

$$\varphi_1 = \Theta(\alpha_1 : \delta_1 \; ; \; \alpha_2 : \delta_2)$$

Defining the property in this way would force the client to reconnect every time the property δ_2 is not satisfied by a message matching α_2, because the broken chain would need to be restarted (and if this were under Ω, the property would be immediately violated). This behaviour does not reflect the intended system semantics: a client connects once and then issues multiple requests, and we want to check whether at least one of those requests satisfies the property. Therefore, it is crucial to define properties carefully, taking into account the organization and behaviour of the system. The correct way to specify the property is as follows:

$$\varphi_1 = \Theta(\alpha_1 : \delta_1 \; ; \; \Theta(\alpha_2 : \delta_2))$$

By nesting Θ, we can capture the intended behaviour, as this allows us to "descend" the context tree to reach the contexts containing the messages corresponding to a client's requests.

But what if our goal is simply to verify messages with signature α_2, ignoring the initial connection? This approach suffices if we only want to ensure that at least one client has sent a request with a positive payload. Intuitively, we can focus directly on messages with

signature α_2 and check whether they satisfy δ_2. This approach is straightforward and should be feasible. The property can then be defined as follows:

$$\varphi_2 = \Theta(\alpha_2 : \delta_2)$$

At first glance, this approach seems natural: we define the signature we want to capture and then specify that we are interested in finding at least one context in which it is satisfied. However, this property does not fully capture our intended behaviour. By defining $\varphi_2 = \Theta(\alpha_2 : \delta_2)$ the semantics will evaluate the following:

$$\sigma, \Delta_0 \models \Theta(\alpha_2 : \delta_2) \text{ iff } \forall_{\Delta_k \blacktriangleright \Delta_0} : \sigma, \Delta_k \models (\alpha_2 : \delta_2)$$

The formula above results in the verification of the property $\alpha_2 : \delta_2$ over the contexts that are direct children of Δ_0, namely Δ_1 and Δ_2. The problem is that these are not the contexts in which we actually want to verify the property. Our goal is to check the children of Δ_1 and Δ_2, as these correspond to the contexts associated with the requests issued by clients after their initial connection to the system.

Our objective is to configure the monitor to focus exclusively on the messages associated with the second level of the context tree, as depicted in Figure 3.14.

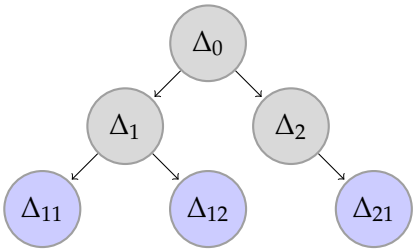


Figure 3.14: Monitor focus

Our goal is to interpret the context tree, from the monitor's perspective, as shown in Figure 3.15.

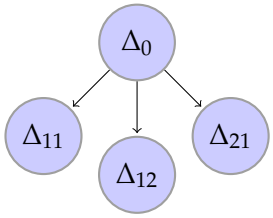


Figure 3.15: Under the eye of the monitor

If the context tree were structured as shown in Figure 3.15, then the property $\varphi_2 = \Theta(\alpha_2 : \delta_2)$ would correctly target the desired contexts. However, it is not feasible to construct a new context tree each time such a need arises.

The creation of contexts is tied to the actual interactions occurring in the system. For example, context Δ_1 is generated in response to the appearance of a specific message. Accordingly, we can visualize the context tree such that transitions between nodes

correspond to the messages that triggered the creation of each context, as illustrated in Figure 3.16.

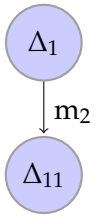


Figure 3.16: Message pipelining into contexts

All of these messages remain part of the system, and their corresponding contexts continue to be created. However, when our goal is to verify a property specifically over the children of Δ_1 and Δ_2, it becomes important to take into account the parent contexts and the messages that triggered their creation.

To address this, WALTZ provides a mechanism to selectively "ignore" certain contexts and focus only on the relevant ones, allowing us to specify the level of the context tree on which to operate. For example, by defining the property φ_2 as:

$$\varphi_2 = \Theta(\ \Theta(\alpha_2 : \delta_2))$$

we gain the ability to verify the property $\alpha_2 : \delta_2$ specifically within the contexts Δ_{11}, Δ_{12}, and Δ_{21}. Nesting the Θ operator in this way effectively directs the evaluation to a particular level of the context tree, while other contexts are ignored. These ignored contexts remain part of the system's logical structure, but they do not influence the current verification, allowing the monitor to concentrate solely on the relevant sub-traces.

This approach allows us to check whether at least one message from any client satisfies δ_2, even if a client fails to connect successfully, since we are only interested in the ones that did connect with success. Similarly, if we want to ensure that all messages from all clients satisfy δ_2, we can express this with a nested Ω operator, defining $\Omega(\ \Omega(\alpha_2 : \delta_2))$.

Returning to the property $\varphi_2 = \Theta(\ \Theta(\alpha_2 : \delta_2))$, and the contexts shown in Figure 3.13. We can see that the semantics correctly handle the evaluation, verifying the satisfaction of this formula at the intended level of the context tree.

We begin by evaluating $\Theta(\ \Theta(\alpha_2 : \delta_2))$, which corresponds to the following semantic check:

$$\sigma, \Delta_0 \ \models\ \Theta(\ \Theta(\alpha_2 : \delta_2)) \text{ iff } \exists_{\Delta_k \blacktriangleright \Delta_0} : \sigma, \Delta_k \models \Theta(\alpha_2 : \delta_2)$$

At this level, the outer Θ considers all contexts directly forked from Δ_0, which are Δ_1 and Δ_2. For each of these contexts, we now evaluate the inner formula $\Theta(\alpha_2 : \delta_2)$.

Focusing first on Δ_1, the inner Θ descends one level further to examine its child contexts, verifying whether any of them satisfy $\alpha_2 : \delta_2$. The same process is then applied to Δ_2 and its children. In this way, the nested structure of Θ ensures that the property is checked at the appropriate level of the context tree while ignoring irrelevant branches.

The semantic check for Δ_1 is:

$$\sigma, \Delta_1 \models \Theta(\alpha : \delta) \text{ iff } \exists_{\Delta_k \blacktriangleright \Delta_1} : \sigma, \Delta_k \models (\alpha_2 : \delta_2)$$

As we can observe, each time we unpack a modal operator, we descend one level in the context tree, examining the children of the current context under evaluation. This approach provides a powerful mechanism for focusing specifically on a single level of messages entering the system. With this step complete, the remaining verifications to be performed are:

$$\varphi_{11} = \sigma, \Delta_{11} \models (\alpha_2 : \delta_2)$$
$$\varphi_{12} = \sigma, \Delta_{12} \models (\alpha_2 : \delta_2)$$

The same verification will be applied to Δ_2. It is sufficient that at least one message within the contexts Δ_{11}, Δ_{12}, or Δ_{21} adheres to δ_2. If none of these messages satisfy the property, the monitor continues observing the system, issuing inconclusive verdicts, as is typical in runtime monitoring.

In conclusion, this example illustrates how different ways of specifying properties directly affect how the monitor interprets chains of messages. By navigating the context tree, we can choose to evaluate properties across the entire structure or focus on a specific level, as demonstrated in this section. This flexibility effectively shifts the monitor's perspective on the trace it witnesses, allowing it to check properties in a manner aligned with the intended system behaviour.

3.6 The Compilation Process

At the core of WALTZ is the idea that high-level specifications can be transformed into executable entities, running processes that continuously observe a system and determine whether it behaves as intended. This active component is referred to as the monitor, which, in our implementation, takes the form of executable `Erlang` code.

We begin by examining the compilation pipeline, implemented in `OCaml` [56] for its strong type system and functional programming capabilities, which make it well-suited for building a robust parser and compiler. The pipeline takes a WALTZ formula, provided as a simple `.txt` file, parses it according to our defined grammar, and translates it into optimized `Erlang` code ready for execution. The generated monitor is then attached to the system and runs alongside it, receiving relevant events to verify the defined properties. The details of this attachment and instrumentation process are discussed in Chapter 4.

To make this process more approachable (both for ourselves and for the tool), we slightly adapted the formal grammar described in the previous section. The semantics remain untouched, but we have introduced a few carefully chosen "quality-of-life" tokens. These small adjustments make the parser's job easier, reduce the chance of errors, and allow the code generator to work more smoothly, without compromising the expressiveness or rigor of WALTZ.

3.6.1 Practical Grammar Changes

In order to ease our parsing and compilation job, we add some useful tokens into our grammar. The new WALTZ grammar will be defined as:

Definition 3.6.1 (WALTZ Formula)**.** The set of WALTZ formulas is defined by the following grammar:

$$
\begin{aligned}
\varphi &::= \texttt{OMEGA}(\varphi) \mid \texttt{THETA}(\varphi) \mid \varphi \ ; \ \varphi \mid \alpha : \delta \\
\delta &::= \top \mid \bot \mid c \mid \neg\delta \\
\alpha &::= \texttt{send}_{\texttt{id}\to\texttt{id}}\ \texttt{WITH}\ \{\texttt{M}\} \\
\texttt{M} &::= \texttt{T} \mid \texttt{T}\ \texttt{,M} \mid \{\texttt{M}\} \\
\texttt{T} &::= (\texttt{cons} \mid \texttt{var})
\end{aligned}
$$

During this section we still might use the mathematical symbols of Ω and Θ, but have in mind that the specification files themselves, will work with the keyword `OMEGA` and `THETA`. Another useful addition is the `WITH` keyword, that is present in the signatures of the messages, separating the `send` request origin and destination from the signature of the message itself. More on this, can be consulted in the ACTORCHESTRA page, which has a comprehensive list of property definitions, and the full grammar for property specifications.

3.6.2 The General Monitor Structure

The design of our monitors is essential to ensure adherence to the rules we defined for property verification. A central concept in this design is the notion of context, which is critical to understanding how the monitor processes incoming traces from the system.

To simulate the consumption of contexts and message chains, the monitor compilation is implemented as a sequence of `receive` statements. These statements serve as the primary mechanism for capturing messages not only in the correct order of the chain but also within their respective contexts, as illustrated in Listing 3.1.

```
1    receive Signature1 ->
2      if BooleanConstraint1 ->
3        receive Signature2 ->
4          if BooleanConstraint2 ->
5            ...
```

Listing 3.1: Message Chains

As the monitor consumes a message chain, it progresses sequentially through the `receive` statements, checking property satisfaction at each step. Once the entire chain has been processed, the monitor either loops back to handle the next chain or produces a verdict, depending on the property being monitored. This architecture not only enforces

the order and unbreakability of messages but also preserves the context in which each event is evaluated and maintains the scope of all variables involved in the interaction.

Take for example the property $\Omega(\texttt{send}_{\texttt{actorA}\rightarrow\texttt{actorB}}\{\texttt{MyPID, Payload}\} : \{\texttt{Payload}\} > 0)$, which checks if the property is satisfied across all contexts. The monitor for this property can be visualized in Listing 3.2

```
loop():
  receive {actorA, actorB, Msg, Context} ->
    case Msg1 of {MyPid, Payload} ->
      if (Payload > 0) -> loop()
      true -> io:format("✘")
      end;
    end;
  end;
end.
```

Listing 3.2: Ω monitor example

Since this property is encapsulated within an Ω, the monitor focuses on detecting violations. If the current message chain satisfies the property, the monitor simply invokes `loop()` to evaluate the next chain. If the property is not satisfied, a violation is reported. Each call to `loop()` prepares the monitor to verify the property against a new context.

For the Θ operator, the overall structure of the monitor remains the same as before, with message chains captured in the same manner as shown in Listing 3.1. The key differences lie in the evaluation strategy and the stopping condition: rather than seeking violations, the monitor now continues processing messages until it finds a chain that satisfies the property. Once such a chain is detected, the property is considered satisfied, and the corresponding verdict is produced. As an example, the compiled monitor for the property $\Theta(\texttt{send}_{\texttt{actorA}\rightarrow\texttt{actorB}}\{\texttt{MyPID, Payload}\} : \{\texttt{Payload}\} > 0)$ is shown in Listing 3.3.

```
loop():
  receive {actorA, actorB, Msg, Context} ->
    case Msg1 of {MyPid, Payload} ->
      if (Payload > 0) -> io:format("✔")
      true -> loop()
      end;
    end;
  end;
end.
```

Listing 3.3: Θ monitor example

As we can see, the `Erlang` abstraction and design techniques make the transition of WALTZ semantics, into simple yet robust monitor executables that are able to capture these messages at runtime.

3.6.3 The Preservation of Context

The monitor expects to receive the following tuple at runtime:

```
{Sender, Receiver, Message, Context}
```

We will see in the next chapter how this is achieved, but for now, consider this as the signature produced by `Erlang`'s internal tracing mechanisms. It is important to note that, while the context appears as part of the message payload, the system itself is not required to explicitly include it in the messages. As we will explore in the next chapter, it is possible to build correct `Erlang` systems without manually managing the causality of messages by leveraging the OTP behaviour discussed previously in Section 2.8.

A natural question arises: how can we ensure that the monitor observes only the messages belonging to a specific chain? This is precisely where the context plays a crucial role. Consider the property φ:

$$\varphi = \Theta(\texttt{send}_{\texttt{actorA}\rightarrow\texttt{actorB}}\,\{\texttt{MyPID, X, Y}\} : \top\,;\, \texttt{send}_{\texttt{actorB}\rightarrow\texttt{actorC}}\,\{\texttt{MyPID2, Z}\} : \texttt{Z = X + Y})$$

The goal is to verify that at least one context, a chain of causally related messages, satisfies the defined property. Without proper context management, the monitor could mistakenly associate messages, as concurrent `Erlang` processes can produce events in many possible orders. For example, consider the following trace:

$$\begin{aligned} m_1 &= \texttt{send}_{\texttt{actorA}\rightarrow\texttt{actorB}}\,\{\texttt{actorA, 10, 10}\} \\ m_2 &= \texttt{send}_{\texttt{actorA}\rightarrow\texttt{actorB}}\,\{\texttt{actorA, 20, 10}\} \\ m_3 &= \texttt{send}_{\texttt{actorB}\rightarrow\texttt{actorC}}\,\{\texttt{actorB, 30}\} \\ m_4 &= \texttt{send}_{\texttt{actorB}\rightarrow\texttt{actorC}}\,\{\texttt{actorB, 20}\} \end{aligned}$$

The monitor generated from φ is presented in Listing 3.4.

```
1 loop():
2   receive {actorA, actorB, Msg1} ->
3     case Msg1 of {MyPid, X, Y} ->
4       receive {actorB, actorC, Msg2} ->
5         case Msg2 of {MyPid2, Z} ->
6           if (Z == X + Y) -> io:format("✔")
7           true -> loop()
8           ...
```

Listing 3.4: Wrong Monitor for φ

Without context awareness, it is insufficient to assume that the monitor will read messages from each context in the order the events occurred. While `Erlang` guarantees message ordering between two processes, this guarantee does not extend across multiple processes. Consequently, the monitor may encounter various interleavings of messages,

such as the following:

$$\mathtt{m}_1\ \mathtt{m}_4\ \mathtt{m}_2\ \mathtt{m}_3$$

$$\mathtt{m}_2\ \mathtt{m}_3\ \mathtt{m}_1\ \mathtt{m}_4$$

$$\mathtt{m}_2\ \mathtt{m}_1\ \mathtt{m}_4\ \mathtt{m}_3$$

$$\mathtt{m}_2\ \mathtt{m}_1\ \mathtt{m}_3\ \mathtt{m}_4$$

These are all valid interleavings that the monitor might observe, and in some cases, values from one causal chain, such as `X` and `Y`, could be incorrectly combined with `Z` from another chain. This is highly problematic, as it can lead to erroneous verdicts for the properties being monitored.

For example, suppose the monitor first captures message $\mathtt{m}_1$, and then, due to a possible interleaving, observes $\mathtt{m}_2$ followed by $\mathtt{m}_3$. In the code shown in Listing 3.4, $\mathtt{m}_1$ matches the first `receive` clause, assigning the runtime values `10` to both `X` and `Y`. When $\mathtt{m}_2$ arrives, it does not match the second `receive` clause for the chain and remains in the mailbox. Subsequently, $\mathtt{m}_3$ matches its expected signature, assigning `30` to `Z`. The verification of `Z == X + Y` then fails, since `10 + 10` $\neq$ `30`.

Then, the monitor might match $\mathtt{m}_2$ followed by $\mathtt{m}_4$, which do not belong to the same causal chain. This again can result in a property violation, even though, in reality, the property was was satisfied in both contexts.

This illustrates why attaching context to messages is absolutely crucial. With context information, the monitor can use pattern matching to "lock" onto a specific chain: once the first message matches the start of a `receive` chain, only messages belonging to the same context are accepted. Messages from other contexts may still arrive in the monitor's mailbox, but they will wait until the current chain has been fully processed.

3.6.4 Non Breakable Chains

As discussed in previous sections, message chains are unbreakable. By structuring the monitors as nested `receive` clauses and leveraging the context of each message, the monitor can process messages chain by chain, ensuring that payloads from different contexts are never mixed. If any message within a chain violates its boolean constraint, the monitor's next action depends on the type of property being monitored: it will either proceed to the next chain or immediately report a property violation. For instance, consider the following property:

$$\alpha_1 : \delta_1\ ;\ \alpha_2 : \delta_2\ ;\ \alpha_3 : \delta_3$$

In the generated monitor code, there are three nested `receive` clauses, as shown in Listing 3.5. To simplify the presentation of our monitors, we will use a combination of mathematical notation and pseudocode instead of strictly syntactically correct `Erlang` code. The behaviour of the monitor depends on the chosen modal operator. When monitoring the property $\Omega(\alpha_1 : \delta_1\ ;\ \alpha_2 : \delta_2\ ;\ \alpha_3 : \delta_3)$, a single broken message chain is sufficient to declare the property violated. In contrast, for the property $\Theta(\alpha_1 : \delta_1\ ;\ \alpha_2 : \delta_2\ ;\ \alpha_3 : \delta_3)$

the property is considered satisfied as soon as there exists one context in which the entire chain remains unbroken.

```
loop():
  receive α1 ->
    check δ1 ->
      receive α2 ->
        check δ2 ->
          receive α3 ->
            check δ3 ->
```

Listing 3.5: Nesting receive clauses

3.6.5 Compilation of Ω

As discussed previously, the Ω operator allows us to define a monitor that verifies whether a property holds in every context of the system; if any context fails, the property is considered violated. In general, we can define properties of increasing complexity, such as:

$$\varphi_1 = \Omega(\alpha : \delta)$$
$$\varphi_2 = \Omega(\alpha_1 : \delta_1 \; ; \; \alpha_2 : \delta_2)$$
$$\varphi_3 = \Omega(\alpha_1 : \delta_1 \; ; \; \alpha_2 : \delta_2 \; ; \; \alpha_3 : \delta_3)$$

To keep the discussion focused, we will set aside the nesting of modal operators for now, as they introduce additional complexity that we will address in a later section. In this compilation showcase, we concentrate on compiling monitors for single message chains without nested operators. Using this approach, the property φ_2 can be compiled into the `Erlang` monitor shown in Listing 3.6, which illustrates how multiple sequential messages within a single chain are handled.

```
monitor_φ2():
  receive α1 ->
    check δ1 ->
      if(!δ1) then ✘
      receive α2 ->
        check δ2 ->
          if(!δ2) then ✘
          else monitor_φ2()
```

Listing 3.6: Ω monitor generation

Each chain is going to be evaluated separately, flowing through the nested chain of `receive` statements, at each moment doing a sanity check to discover whether the property is being violated at that point or not. Considering property φ_2, and the trace $\sigma = \mathtt{m}_1{}^{\checkmark}\ \mathtt{m}_2{}^{\checkmark}\ \mathtt{m}_3{}^{\times}\ \mathtt{m}_4{}^{\checkmark}\ldots$, with the boolean constraints satisfactions indicated above it. Messages $\mathtt{m}_1$ and $\mathtt{m}_2$ share the signature α_1, while $\mathtt{m}_3$ and $\mathtt{m}_4$ have signature α_2. In

this scenario, the monitor processes each chain individually, ensuring that the property is evaluated correctly for each sequence of causally related messages, as illustrated in Figure 3.17.

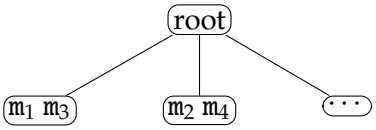


Figure 3.17: The monitor message flow for property φ_2

Each message in a chain is evaluated within its designated context, ensuring that it is checked against the specified property in the correct scope.

3.6.6 Compilation of Θ

The Θ operator allows verification of whether at least one context in the system satisfies a given property. Once any context meets the required condition, the property can be considered satisfied, enabling the monitor to halt further evaluation. This operator thus allows the definition of properties such as:

$$\begin{aligned}\varphi_1 &= \Theta(\alpha : \delta)\\ \varphi_2 &= \Theta(\alpha_1 : \delta_1\ ;\ \alpha_2 : \delta_2)\\ \varphi_3 &= \Theta(\alpha_1 : \delta_1\ ;\ \alpha_2 : \delta_2\ ;\ \alpha_3 : \delta_3)\end{aligned}$$

The monitor generated from property φ_3 is present in Listing 3.7.

```
monitor_φ3():
  receive α1 ->
    check δ1 ->
      if(!δ1) then flag
      receive α2 ->
        check δ2 ->
          if(!δ2) then flag
          receive α3 ->
            check δ3 ->
              if(!δ3) then flag
              if(flag) then monitor_φ3()
              else ✔
```

Listing 3.7: Θ behaviour with chains

We can observe that the overall pattern remains consistent: as chains grow longer, additional nested clauses are introduced to represent the extended sequence of messages. This structuring allows the monitor to traverse each chain and determine whether there exists at least one chain in which all messages satisfy their respective boolean constraints.

The `flag` is essential for detecting when a chain has been broken. It ensures that once a chain is invalidated, the monitor can correctly consume the remaining messages

without mistakenly treating them as contributing to property satisfaction. Without this mechanism, such as in the monitor generated in Listing 3.8, several potential errors could occur.

```
monitor_φ3():
  receive α1 ->
    check δ1 ->
      if(!δ1) then monitor_φ3()
      receive α2 ->
        check δ2 ->
          if(!δ2) then monitor_φ3()
          receive α3 ->
            check δ3 ->
              if(!δ3) then monitor_φ3()
              else ✔
```

Listing 3.8: Θ problematic monitor

Consider a scenario in which, for a given trace, the message matching α_1 fails to satisfy the boolean constraint δ_1. If we were to simply make a recursive call the subsequent messages of that interaction chain would remain in the mailbox. Repeating this process multiple times could result in the mailbox being filled with "garbage" messages.

The issue becomes particularly critical in systems where an entire client session is treated as a single causal chain. In these cases, multiple messages may belong to the same context, as discussed in Chapter 4, and failing to properly discard broken or invalid messages from the mailbox can result in inconsistencies.

By following this architecture for compiling the Θ operator, we not only prevent the accumulation of irrelevant messages but also safeguard against potential inconsistencies caused by leftover messages in the mailbox.

3.7 Compilation of Nested Modal Operators

It is also necessary to address the compilation of nested operators, as well as the structure of the monitors that result from such properties. Compared to the cases introduced earlier, these monitors exhibit higher complexity, but this complexity is essential: it allows us to analyse behavioural patterns that go beyond the verification of simple message chains defined by a single modal operator. Nesting enables properties to capture relationships across multiple levels of causality, reflecting scenarios where interactions are hierarchically dependent. To illustrate this idea, let us start by considering the following property:

$$\varphi_1 = \Theta(\alpha_1 : \delta_1 \; ; \; \Theta(\alpha_2 : \delta_2))$$

Recalling the example and the concepts presented in Section 3.4 and Section 3.5, we treat the messages within each context as finite chains. Nesting extends this model

by introducing a hierarchy and explicit relationships between contexts, allowing child contexts to inherit causal dependencies from their parents and enabling the monitor to correctly evaluate message chains across multiple levels. This extension allows us to capture dependencies between chains that belong to the same parent context. A representative example is the interaction where a client connects to the system, establishing the initial context, and subsequently issues multiple requests. Each request corresponds to a distinct event, yet all of them remain causally linked to the client's initial connection.

The monitor generated for φ_1 is shown in Listing 3.9. The fundamental idea here is the introduction of sub-monitors. Rather than relying on a single monitor, the system now maintains multiple monitors, one for each parent context. The central oracle, corresponding to Δ_0, assumes the responsibility of routing messages to the appropriate sub-monitor. Each sub-monitor then independently evaluates its portion of the property and reports back to its parent context whenever a satisfaction or violation is detected.

```
monitor_φ1():
  receive
    {Src, Dst, Msg, Context} ->
      if has_sub_monitor(Context)
        route({Src, Dst, Msg, Context})
      else main_monitor({Src, Dst, Msg, Context})

main_monitor({Src, Dst, Msg, Context}):
  case {Src, Dst, Msg} of
    α1 ->
      check δ1 ->
        if(!δ1) then flag
        if(flag) then monitor_φ1
        else
          spawn(sub_monitor_loop(), Δcurrent, env),
          monitor_φ1()
%sub_monitor_loop().erl
sub_monitor_loop():
  receive α2 ->
    check δ2 ->
      if(!δ2) then flag
      if(flag) then sub_monitor_loop()
      else ping(monitor_φ1, ✔)
```

Listing 3.9: Θ operator monitor nesting

As illustrated in Listing 3.9, the monitor now spawns an additional process responsible for handling the child contexts derived from a parent. This sub-monitor follows the same general structure we have already seen for the compilation of Ω and Θ, but with one important difference: whenever it reaches a verdict, it must communicate this outcome

back to the monitor that created it. This is achieved through a simple `ping` mechanism, enabling the parent to aggregate the results of its children and decide its own verdict.

A crucial detail here is that, besides inheriting the context upon which the child is spawned, denoted as Δ_{current}, the sub-monitor must also inherit the current `environment`. This ensures that any variable bindings established at the parent level remain available to the child monitors. Such sharing is essential because contexts, while evaluated independently, are still hierarchically related: together they form causal chains of events. By propagating both the context and the variable environment, we preserve the ability to reason consistently across parent–child relations, ensuring correctness when evaluating properties that depend on shared values.

It is also possible to nest Ω operators, though generating monitors for such properties is nontrivial. For instance, consider a property similar to φ_1, but expressed using the Ω operator, denoted as φ_2.

$$\varphi_2 = \Omega(\alpha_1 : \delta_1 \; ; \; \Omega(\alpha_2 : \delta_2))$$

What this property aims to capture is whether, for all contexts derived from Δ_0, namely Δ_1 and Δ_2, every one of their child contexts satisfies the condition δ_2. Intuitively, this can be read as: for every client connection, all subsequent requests must carry positive numbers. The monitor generated for this property is presented in Listing 3.10.

```
monitor_φ2():
  receive
    {Src, Dst, Msg, Context} ->
      if has_sub_monitor(Context)
        route({Src, Dst, Msg, Context})
      else main_monitor({Src, Dst, Msg, Context})

main_monitor({Src, Dst, Msg, Context}):
  case {Src, Dst, Msg} of
    α1 ->
      check δ1 ->
        if(!δ1) then ✘
        else
          spawn(sub_monitor_loop(), Δcurrent, env),
          monitor_φ2()
%sub_monitor_loop().erl
sub_monitor_loop():
  receive α2 ->
    check δ2 ->
      if(!δ2) then ping(monitor_φ2, ✘)
      else sub_monitor_loop()
```

Listing 3.10: Ω operator monitor nesting

This structuring allows us to faithfully reproduce the context hierarchies explored in the examples of the previous sections. For a complete generated `Erlang` monitor, see Section 3.8. Additional monitors and code examples can be found in the online repository.

3.7.1 Compilation of Different Nested Modal Operators

In WALTZ, as we have seen it is possible to define properties with nested modal operators. This greatly expands the expressiveness of the language but simultaneously opens the Pandora's box of non-monitorable properties. By the very nature of runtime verification, combining different quantifiers, such as $\forall$ and $\exists$, or in our case Ω and Θ, can lead to monitors that will never be able to produce a verdict during execution. To avoid such infinitely running monitors, WALTZ explicitly alerts the user whenever a given property is non-monitorable. This issue will be addressed in more detail in Section 3.9.

It is not difficult to see why such properties are problematic at runtime, even if they may be meaningful from a specification standpoint. Consider, for example, a property stating that "for every client, after a successful connection, at least one of the subsequent requests carries a positive number." The monitor that would be generated for this property, which in WALTZ can be written as $\varphi_3 = \Omega(\alpha_1 : \delta_1 \; ; \; \Theta(\alpha_2 : \delta_2))$ would result in the structure shown in Listing 3.11.

```
...
main_monitor({Src, Dst, Msg, Context}):
  case {Src, Dst, Msg} of
    α1 ->
      check δ1 ->
        if(!δ1) then ✘
        else
          spawn(sub_monitor_loop(), Δcurrent, env),
          monitor_φ2()
%sub_monitor_loop().erl
sub_monitor_loop():
  receive α2 ->
    check δ2 ->
      if(!δ2) then ping(monitor_φ2, ✔)
      else sub_monitor_loop()
```

Listing 3.11: Non monitorable property

In this case, we can never reach a conclusive verdict. Assuming all client connections are successful, the monitor will be forced to wait indefinitely: it cannot guarantee that for every client the property will hold. It might hold for some, or it might eventually be violated, but the monitor remains trapped in uncertainty. The problem lies in the mismatch between parent and child monitors, while the parent monitor expects a `ping` to signal a violation, the child monitors only notify their parent upon satisfaction. As a result, no definitive verdict is ever produced, and the monitor remains inconclusive.

3.8 A Monitor On The Go

Finally, knowing everything up until this point, we can now present a fully developed example, from specification to monitor generation and execution.

3.8.1 Chat Room System

Consider the example introduced in Chapter 2, a chat system where multiple clients can connect and post messages. To provide a more `Erlang` oriented view of the system, it is essential to clearly define the communication protocol, as this directly influences how properties should be specified and monitored.

In this system, clients first establish a connection, then enter a chat room, and finally post messages within that room. These interactions are illustrated in Figure 3.19, Figure 3.20, and Figure 3.18.

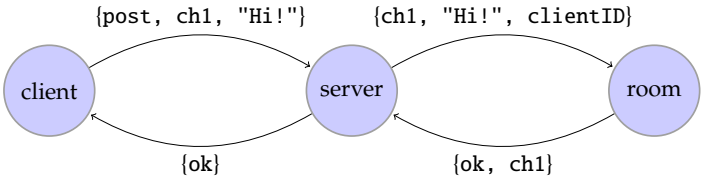


Figure 3.18: System behaviour for post action

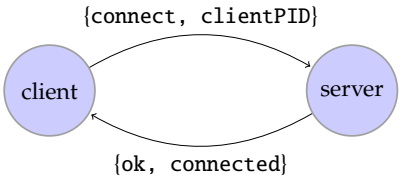


Figure 3.19: System behaviour for login action

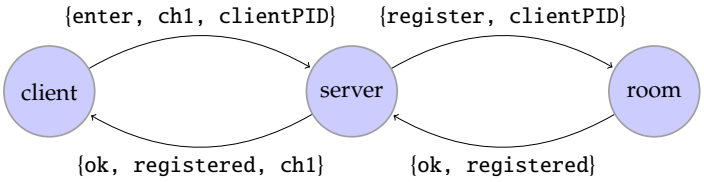


Figure 3.20: System behaviour for enter actions

This system exhibits a variety of interesting interactions that we can capture. For instance, we might want to verify that once a client enters a chat room, they can only post messages within that room and not elsewhere. In WALTZ, this property can be formalized as follows:

$$\varphi = \Omega\left(\begin{array}{l} \texttt{send}_{\texttt{server}\rightarrow\texttt{client}}\{\texttt{ok, registered, Room1}\} : \top\ ; \\ \quad \Omega(\texttt{send}_{\texttt{client}\rightarrow\texttt{server}}\{\texttt{post, Room2, Message}\} : \texttt{Room1 = Room2}) \end{array}\right)$$

This property allows us to capture a part of each client's interactions with the system, verifying that all clients adhere to the expected rules: specifically, no client can post a message in a channel where it is not registered. Consider the following trace, where each sequence corresponds to a different client communicating with the system:

$$\sigma_{c1} = \{\texttt{enter}, \texttt{ch1}, \texttt{ClientPID}\}\,\{\texttt{ok, registered, ch1}\}\,\{\texttt{post, ch1, Msg}\}$$
$$\sigma_{c2} = \{\texttt{enter}, \texttt{ch2}, \texttt{ClientPID}\}\,\{\texttt{ok, registered, ch2}\}\,\{\texttt{post, ch1, Msg}\}$$

By examining the trace, we can see that the first client behaves correctly: it enters a chat room and subsequently sends a message to that room. The issue arises with the second client, which enters a different chat room and attempts to send a message to a chat room it is not registered in, a clear violation of the property we intend to monitor. It is important to remember that both σ_{c1} and σ_{c2} are part of a single trace σ produced by the system. For simplicity and better clarity, we can treat them as independent, isolated traces. In practice, the monitor treats them as separate chains, verifying the property independently for each context.

```
monitor_loop(ContextMap) ->
  receive
    {Src, Dst, Msg, Context} ->
      case maps:get(Context, ContextMap, undefined) of
        undefined ->
          main_loop(Src, Dst, Msg, Context, ContextMap);
        PID ->
          PID ! {Src, Dst, Msg, Context},
          monitor_loop(ContextMap)
  end
end.

main_loop(Src, Dst, Msg, Context, ContextMap) ->
  case {Src, Dst, Msg} of
    {server, client, {ok, registered, Room1}} ->
        if true ->
          Environment = #{room1 => Room1},
          SubPID = spawn(sub_monitor, [Environment, Context,
              self()]),
          NewContextMap = ContextMap#{Context => SubPID},
          monitor_loop(NewContextMap),
          ok
        ; true ->
          io:format("✘")
        end;
end.
```

Listing 3.12: `main_monitor.erl`

The main monitoring loop that coordinates the verification process is shown in Listing 3.12. Here, the `main_monitor.erl` process receives all messages from the system, identifies or creates the appropriate sub-monitor for the given context, and forwards messages to it. Each new client connection triggers the creation of a sub-monitor, responsible for tracking that specific client's interactions.

Once a sub-monitor is spawned, it is responsible for evaluating all subsequent messages in its assigned context. Listing 3.13 shows the structure of a sub-monitor in `sub_monitor.erl`. This process continuously receives messages from the client, checks whether the client is posting to the registered room, and either loops to process further messages or reports a violation back to the main monitor. In our example, the second client attempting to post in an unregistered room would trigger the sub-monitor to notify the main monitor of the property violation.

```
1  sub_monitor(Environment, Context, ParentPID) ->
2    receive
3      {client, server, Msg, Context} ->
4        case Msg of
5          {post, Room2, _} ->
6            if maps:get(room1, Environment) == Room2 ->
7              sub_monitor(Environment, Context, ParentPID)
8            ; true ->
9              ParentPID ! {context_completed, Context, ✘}
10           end;
11         _ ->
12           sub_monitor(Environment, Context, ParentPID)
13       end;
14   end.
```

Listing 3.13: `sub_monitor.erl`

This monitor exemplifies the strengths of WALTZ, including hierarchical context management, correlation of messages across multiple interactions, and the evaluation of properties over all contexts present in the system. The use of nested modal operators necessitates more than a single monitor, as relying on just one would result in the monitor being indefinitely occupied with a single context. By spawning separate processes for sub-monitors, we enable concurrent monitoring of multiple contexts, while the main oracle orchestrates the routing of messages to their corresponding child monitors.

When examining the trace of the second client, σ_{c2}, the property violation becomes apparent. The sub-monitor evaluates the condition `Room1 == Room2`, which fails because the client attempts to post to a chat room that is different from the one it entered. This violation of property φ is detected at runtime, demonstrating how the WALTZ-generated monitor successfully enforces correctness across concurrent client interactions.

3.9 Monitorability of WALTZ

Like any specification language, WALTZ is not immune to the issue of non-monitorability. It is possible to restrict the language to a monitorable fragment, but doing so comes at the cost of expressiveness. For instance, consider the property φ_1 = "For every client, at least one of his requests is a positive number". Due to the inherently infinite nature of runtime traces, this property is non-monitorable, as discussed in [66]. Similarly, a property such as φ_2 = "At least for one client, all his messages have a positive number" is also non-monitorable. In this case, we depend on an event occurring infinitely often to guarantee satisfaction for at least one client, which, again, prevents runtime monitors from ever yielding a conclusive verdict.

There are properties that can be expressed in WALTZ and are meaningful from a specification perspective, but in practice, they are non-monitorable. This is because the monitor would never be able to reach a conclusive verdict, resulting in a continuously running process that consumes system resources, adds overhead, and ultimately provides no useful feedback. While there is ongoing research aimed at expanding the set of monitorable properties, and attempts to transform certain non-monitorable properties into monitorable ones [43, 43], some properties remain inherently non-monitorable unless additional assumptions or modifications are made.

For example, consider the property $\varphi_1 = \Omega(\alpha_1 : \delta_1 \ ; \ \Theta(\alpha_2 : \delta_2))$ whose context structure tree is illustrated in Figure 3.21. As we will see, the nesting between the Ω and Θ operators makes it impossible for a runtime monitor to ever produce a conclusive verdict in this scenario.

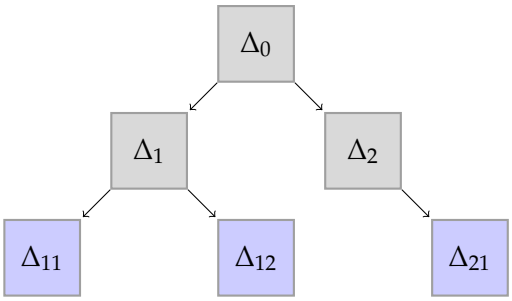


Figure 3.21: Monitor organization for φ_1

Examining the structure depicted in Figure 3.21, we can describe the waiting behaviour of each monitor within the hierarchy of contexts. In this setup, Δ_0, which is the parent of all contexts monitored by the Ω operator, waits for a property violation in order to produce a conclusive verdict. Meanwhile, the child contexts, such as those under Δ_1 and Δ_2, are monitored by the Θ operator. These sub-monitors are waiting for a satisfaction to produce a verdict, as this is the fundamental behaviour of Θ. The problem arises because Θ monitors can never output a negative verdict, they can only yield a positive verdict or remain inconclusive.

As a result, Δ_0 will never receive the violation signal it needs to reach a conclusive

verdict. Instead, it is left waiting indefinitely while its children either satisfy the property or remain inconclusive. This interplay of waiting and verdict-passing is clearly illustrated in Figure 3.22, highlighting why this property is inherently non-monitorable at runtime.

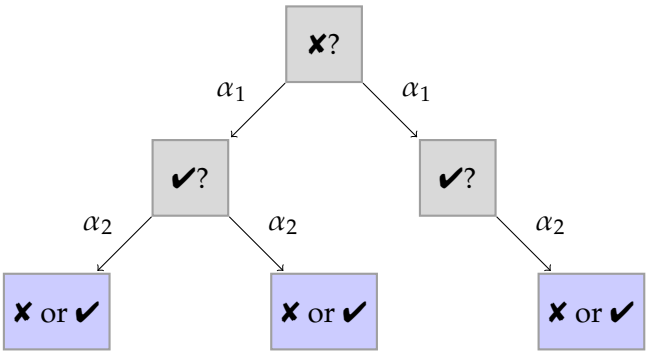


Figure 3.22: Non-monitorable structure for φ_1

A related, yet distinct, challenge arises when monitoring for properties that require at least one instance in the domain to satisfy a given condition. In this case, the monitor's ability to reach a verdict depends on an event occurring at some point in the future, potentially indefinitely, which we cannot guarantee at runtime. Even if a violation occurs, which might allow a partial conclusion, the monitor must continue observing, since we are specifically interested in finding at least one context that satisfies the property. This inherent uncertainty renders such properties non-monitorable. An example of this scenario is the property $\varphi_2 = \Theta(\alpha_1 : \delta_1 \,;\, \Omega(\alpha_2 : \delta_2))$ whose monitor structure is illustrated in Figure 3.23.

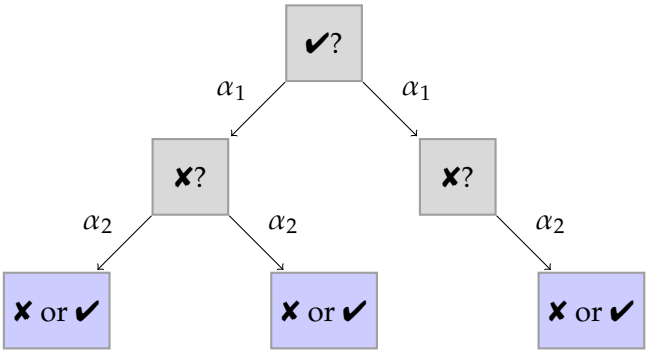


Figure 3.23: Non-monitorable structure for φ_2

As illustrated in Figure 3.23, the main monitor is waiting to detect a satisfaction, but none of its child monitors can provide this information. At most, they are able to detect violations, and under normal execution, they will continue emitting inconclusive verdicts (denoted by ?) indefinitely. As a result, the main monitor is never able to determine conclusively whether the property has been satisfied, leaving the system in a state of perpetual uncertainty.

To illustrate when a property is fully monitorable, we can consider the example of the property $\varphi_3 = \Omega(\alpha : \delta)$. The monitor structure for this property is shown in Figure 3.24.

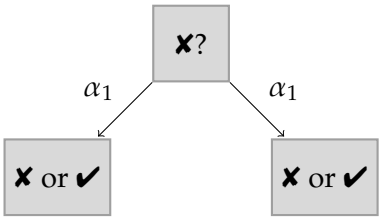


Figure 3.24: Monitorable property

As we can see from the figure, the oracle monitor is waiting for a potential violation. Each sub-monitor associated with a specific context either detects a violation or observes satisfaction. Because the property requires that all contexts satisfy the condition, a violation detected in any single leaf context is sufficient for the oracle to yield a conclusive verdict. This demonstrates how, in this scenario, the monitor can effectively reach a verdict, highlighting the monitorable nature of φ_3.

The issue of non-monitorability arises primarily because we are dealing with potentially infinite traces. If the trace were finite, however, all of these properties would become monitorable, since we could examine the trace in its entirety and produce a conclusive verdict. This approach, however, requires offline monitoring, which cannot alert the system to violations in real time. Later in this document, we outline concrete strategies, both in a general sense and specifically for actor-based systems, that enable monitoring of properties that would otherwise be non-monitorable at runtime.

CHAPTER 4

ACTORCHESTRA

In Chapter 3, we introduced the theoretical foundations of WALTZ, focusing on how formulas are evaluated and how runtime monitors verify these formulas against system execution traces. A central concern in this verification process is the notion of contexts, which capture the causal dependencies between messages. Importantly, such contextual information is not natively provided by the systems under observation.

This chapter turns to the problem of context management. We begin by examining how context information could, in principle, be generated and maintained directly by the system. We then show that, even in the absence of explicit support, correct `Erlang` programs can still be constructed by relying on the OTP model. However, OTP on its own imposes intrinsic limitations when it comes to building explicit causal chains between messages. To address these limitations, we propose a complementary approach based on the selective injection of code into the system. This method systematically enriches program execution with causal information, enabling precise reasoning about message dependencies and interactions.

4.1 Instrumentation

`ACTORCHESTRA` is the glue that holds everything together. It provides the necessary instrumentation to observe a system under scrutiny and to coordinate the interaction between the system and the monitors. Our primary objective is to enable monitoring without interfering with the system itself, following an outline monitoring approach in which monitors are external entities rather than being embedded directly into the system's code. This task becomes particularly challenging in distributed `Erlang` systems, where multiple components may be active simultaneously and asynchronous behaviour plays a central role. While this asynchrony is precisely what allows such systems to scale and

perform effectively, it also complicates the design of efficient and non-intrusive monitoring. In what follows, we focus on addressing these challenges in the context of asynchronous, distributed settings.

In this chapter, we address the three main entities that collaborate to achieve runtime verification: the system, the `conductor`, and the monitors. We begin by outlining the assumptions made about the system under observation and the standards it must follow. We then introduce the `conductor`, the central entity responsible for managing contexts. Finally, we show how these three entities interact, exchanging information and coordinating their roles to produce the "symphony" required to verify properties over an `Erlang` system.

4.2 Causality With Contexts

As established in Chapter 3, the function γ plays a central role in WALTZ by defining the relation between messages and their contexts. But how does γ manifest in practice? It is precisely through this function that we can construct the context tree C, which underpins property verification and serves as one of the key foundations of WALTZ.

At first glance, we might think of γ as the programmer himself. Indeed, when designing and implementing a correct `Erlang` program, reference management is unavoidable. Without it, we would not be adhering to `Erlang`'s standards, and our program would be incorrect by design. However, even with proper reference management, such as that enforced by OTP, causal chains are not automatically established, unless the programmer explicitly encodes them in the system's logic.

In our earlier exposition of WALTZ, we worked under the assumption that γ was already available and that messages in a trace carried explicit context information. In practice, however, this assumption does not hold: messages do not inherently "know" which context they belong to. Instead, an explicit mechanism is required to associate each message with the correct context.

What this means in practice is that context information must be explicitly propagated through the messages exchanged in the system. By doing so, and given that the monitors are already aware of the message signatures, they can reliably capture and associate the causal information required for verification. The following subsections examine different strategies for handling this causality: starting from manual management by the developer, moving to solutions that exploit the existing `Erlang`/OTP infrastructure, and culminating in our novel approach of automated context injection.

4.2.1 Developer Management

This represents raw `Erlang` programming, where developers themselves are responsible for managing causality without relying on OTP abstractions. In this style, processes interact exclusively through direct message passing, and correlation tokens must be

explicitly threaded through each interaction by the programmer. An example of such a design is shown in Listing 4.1.

In this raw approach, every message carries its correlation token explicitly. This allows the monitor to track message relationships precisely, since γ directly assigns tokens to messages. For instance, if a client issues multiple requests, the monitor can only determine how they are related if each one carries its associated context. By managing this manually, the programmer ensures that causality is preserved end-to-end. Crucially, this not only aids the monitor in correlating messages, but also guarantees the correct flow of the system itself.

```
start_client() ->
    CorrelationToken = make_ref(),
    AddPid ! {process, self(), 10, CorrelationToken},
    receive
        {result, Result, CorrelationToken} -> Result
    end.

add_loop() ->
    receive
        {process, ClientPid, Number, Context} ->
            MultPid ! {process, ClientPid, Number+10, Context},
            add_loop()
    end.

mult_loop() ->
    receive
        {process, ClientPid, Number, Context} ->
            ClientPid ! {result, Number*2, Context},
            mult_loop()
    end.
```

Listing 4.1: Manual Reference Management in `Erlang`

The function `make_ref()` generates a unique reference that guarantees each message chain can be distinguished, providing robustness against concurrency effects and message interleaving. This is a common distributed systems pattern for maintaining request integrity in asynchronous communication. To make this more concrete, let us assume that the trace σ consists of the messages m_1, m_2, and m_3, and unfolds as follows:

$$m_1 = \texttt{\{process, <1.0.0>, 10, \#Ref<1.2>\}}$$
$$m_2 = \texttt{\{process, <1.0.0>, 20, \#Ref<1.2>\}}$$
$$m_3 = \texttt{\{result, 40, \#Ref<1.2>\}}$$

Given this trace, we can state that $\gamma(m_1) = \gamma(m_2) = \gamma(m_3) = \texttt{Ref<1.2>}$, since all messages in the chain share the same correlation token. This guarantees perfect causality tracking but

comes at the cost of meticulous manual management, which quickly becomes error-prone in large distributed systems.

The design relies on explicit context threading: every message must carry its correlation token to preserve causal relationships across interactions. Developers are therefore responsible for propagating metadata in every request and response. This tight coupling between business logic and token management not only clutters the code but also amplifies the risk of subtle mistakes. A single omission, forgetting to forward or validate a token, breaks the causal chain, leaving downstream components unable to reconstruct the intended flow. Such errors are notoriously difficult to trace and debug.

Beyond fragility, this approach forfeits the reliability and fault-tolerance benefits of higher-level abstractions such as OTP. Because causality is handled ad-hoc, rather than through established supervisory patterns, the system cannot leverage OTP's guarantees and often ends up re-implementing them manually. The result is a brittle, maintenance-heavy architecture that demands rigorous developer discipline. While complete causality tracking is achieved, it comes at the price of ignoring OTP's proven design standards and making it harder to establish a uniform behaviour for verification, since each manually managed system tends to diverge significantly in style and structure.

4.2.2 OTP Management

To address the shortcomings of manual context management, the `Erlang`/OTP framework introduces built-in reference handling through its `gen_server` behaviour and message-passing primitives. In particular, every `gen_server:call` operation automatically generates a unique reference that binds a request to its corresponding response. This mechanism relieves developers from explicitly managing correlation tokens while still providing a degree of causality tracking. Concretely, when one process issues a `call` to another, the recipient responds via its `handle_call` function, and both sides of this interaction are internally linked by the same reference. In this way, the runtime itself ensures that each response can be unambiguously associated with the originating request, offering a standardized form of causality tracking without additional developer effort. This behaviour is illustrated in Listing 4.2.

```
% client call
Result = gen_server:call(add, {process, 10}),

handle_call({process, Number}, From, State) ->
    MultResult = gen_server:call(mult, {process, Number}),
    {reply, MultResult, State}.

handle_call({process, Number}, From, State) ->
    {reply, {ok, Number*2}, State}.
```

Listing 4.2: OTP Reference Management Example

OTP generates references using the `make_ref()` function, creating one for every `(request, response)` pair in order to correlate each request with its matching reply. In the example of Listing 4.2, a reference $\texttt{REF}_1$ is established between the `client` and the `add` interaction, and another, $\texttt{REF}_2$, is created for the interaction between `add` and `mult`. The abstraction underpinning this mechanism is straightforward: a process issues a request and expects a response. This is the natural behaviour of `call` and `handle_call` operations in OTP, which are designed primarily for synchronous communication.

That said, asynchronous designs can still benefit from OTP's reference management. By spawning a separate process to execute the `call` operation, and then manually returning the result using `gen_server:reply`, developers can integrate OTP's built-in correlation into an asynchronous workflow.

While OTP provides excellent local causality, ensuring that each individual request is correctly matched to its corresponding response, it does not extend this tracking across an entire chain of requests. The function γ, representing OTP's internal correlation mechanism, is thus limited to pairwise relationships:

$$\gamma(\texttt{request}_i, \texttt{response}_i) = \texttt{REF}_i$$

Due to the nature of these references, OTP cannot relate $\texttt{REF}_1$ and $\texttt{REF}_2$ as part of the same logical request without additional information.

This limitation becomes evident in complex distributed systems, where understanding end-to-end request flows is crucial for debugging, performance analysis, and, in our case, verification. A monitor observing OTP references only sees independent request–response pairs and cannot reconstruct complete causal chains spanning multiple actors. Consequently, the set of properties we can monitor is constrained, although this partial visibility is still valuable.

The OTP approach offers clear advantages. It imposes zero developer burden, since no manual context management is required, and ensures automatic correlation: request–response pairs are always properly linked. Moreover, OTP provides fault tolerance, handling failures and timeouts gracefully.

However, this model introduces notable limitations for system observability. Its causality tracking is local, limited to immediate request–response pairs, preventing true end-to-end tracing of user request flows. As a result, debugging operations that span multiple services, or, in our case, multiple actors, becomes more challenging. Nevertheless, OTP provides a standard foundation to work from, in contrast to the manual approach, which requires developers to explicitly manage correlations themselves.

4.2.3 No Management, Working With Wrong Programs

This represents the worst-case scenario: raw `Erlang` programs with no reference management at all. Such implementations violate fundamental principles of concurrent programming, resulting in systems that are unreliable, untraceable, and highly prone to

race conditions. We assume that we will never deal with systems of this type, since they fail to adhere to the basic structuring principles of `Erlang` programs.

Our focus is on verifying the logical properties of messages within a properly structured `Erlang` system, not on whether the program itself is designed correctly. One of the key assumptions we make about the systems we analyse is that they follow OTP standards. These standards are proven to be correct and provide a uniform structure, which greatly facilitates verification by ensuring predictable behaviour and a consistent coding style.

As a contrast, consider a poorly designed system using raw message passing without any correlation mechanisms, shown in Listing 4.3.

```
start_client() ->
    AddServer ! {process, self(), 10},
    receive
        {result, Result} -> Result  % Could be anyone's data
    end.
add_loop() ->
    receive
        {process, ClientPid, Number} ->
            MultPid ! {process, ClientPid, Number+10},
            add_loop()
    end.
mult_loop() ->
    receive
        {process, ClientPid, Number} ->
            ClientPid ! {result, Number*2},
            mult_loop()
    end.
malicious_loop() ->
  ClientPid ! {result, 100}.
```

Listing 4.3: No Reference Management - Broken Raw `Erlang`

In such broken systems, not only is causality completely lost, but the overall management of message flow collapses. Function γ becomes undefined or maps all messages to a meaningless global context, as discussed in Section 3.2.7.

This leads to severe systemic problems. Responses cannot be reliably correlated with their originating requests, client isolation may be violated, and debugging becomes nearly impossible. Non-deterministic behaviour can arise when identical inputs produce different outputs depending on timing. Additionally, malicious or misbehaving processes may send unexpected messages, for instance, a client might receive a message from an unintended source, such as `malicious_loop`. Without proper error handling, messages may reach the wrong recipients, causing processes to wait indefinitely and potentially stalling the entire system. Such systems are fundamentally broken and unfit for practical use. This underscores the necessity of proper reference management, whether manual or

through OTP, to maintain causality, ensure correct message flow, and enable meaningful runtime verification.

4.2.4 Why OTP Is Not Enough

As we have seen, using purely OTP, we are able to capture causal relations between pairs of actors. For simple properties, this may suffice. For example, consider the property $\varphi = \Omega(\texttt{send}_{\texttt{server} \rightarrow \texttt{client}}\ \texttt{\{ok, Result\}} : \texttt{Result} > \texttt{10})$, which checks that all responses from the server to clients have a result greater than 10. Following OTP standards, a simple monitor can successfully verify this property. Listing 4.4 shows what the trace looks like to the monitor using `Erlang`'s tracing mechanisms:

```
client sent message: [alias| #Ref<140989>, {echo,"Hello World!"}]
   to server
client received: [alias| #Ref<140989>, {echoed,"Hello World!"}]
   from server
```

Listing 4.4: OTP trace

As observed, the request from the client can be correlated with the server's response because both share the same reference. This is sufficient for monitoring simple interactions like the one above. The challenge arises in more complex systems with multiple interacting components. Consider the following scenario:

$$\texttt{client} \rightarrow \texttt{server1} \rightarrow \texttt{server2} \rightarrow \texttt{server3}$$

OTP internally maintains correct causal chains for each request–response pair. However, for an external observer, such as our monitor, each interaction generates a distinct reference. The simplified trace, represented in Listing 4.5, shows that while we can correlate `client` with `server1`, `server1` with `server2`, and `server2` with `server3`, there is no single reference linking the entire logical transaction.

```
%% Client calls Server1
*** SEND: Client -> Server1 with REF #Ref<A>
*** RECEIVE: Server1 received REF #Ref<A>
%% Server1 calls Server2 (NEW reference!)
*** SEND: Server1 -> Server2 with REF #Ref<B>
*** RECEIVE: Server2 received REF #Ref<B>
%% Server2 calls Server3 (ANOTHER new reference!)
*** SEND: Server2 -> Server3 with REF #Ref<C>
*** RECEIVE: Server3 received REF #Ref<C>
%% Responses come back with their respective refs
*** SEND: Server3 -> Server2 with REF #Ref<C>
*** SEND: Server2 -> Server1 with REF #Ref<B>
*** SEND: Server1 -> Client with REF #Ref<A>
```

Listing 4.5: Reference Correlation

From the perspective of the monitor, each pairwise correlation is visible, but the overall transaction remains fragmented. To enable end-to-end monitoring of complex properties across multiple actors, we require a global causal reference that spans the entire chain of messages belonging to the same logical transaction. This motivates the need for an entity capable of assigning and propagating such references, ensuring full causal correlation across all interactions.

4.2.5 Out Take On It

Our approach bridges the gap between the fine-grained control of developer-managed contexts and the convenience of OTP's automatic reference management. We target systems that follow standard `Erlang`/OTP patterns, primarily leveraging the `gen_server` behaviour, and augment them with automated code instrumentation to enable end-to-end causality tracking without requiring any manual intervention from developers. This automated context injection relies on specific architectural and communication assumptions that are not arbitrary limitations but rather reflect best practices for building reliable, observable, and traceable distributed systems. In particular, we assume that all participating processes consistently follow the OTP `gen_server` behaviour, providing predictable request–response structures that our approach can instrument systematically. By grounding our method in these assumptions, we can achieve complete causal tracking while maintaining compatibility with existing OTP applications.

We leverage several key strengths of the OTP model. First, it benefits from standardized communication patterns: although both `gen_server:call` and `gen_server:cast` exist, we focus on `gen_server:call`, since it provides automatic pairwise reference management, ensuring that requests and responses are intrinsically correlated. Equally important is OTP's structured state management. Each `gen_server` maintains its own state, allowing our context injector to seamlessly embed causality information without interfering with the core business logic, since context data is simply added to the process state. Additionally, OTP provides a uniform message-handling interface through the `handle_call` callback, giving us a consistent entry point to systematically apply context injection across all services.

The proposed method presents several important advantages. It imposes no additional burden on developers, as existing OTP code requires no manual modifications. At the same time, it enables true end-to-end tracing, capturing complete request flows seamlessly across service boundaries. Automatic propagation ensures that contexts move transparently throughout the system, without any developer intervention. The solution is fully backward compatible, so systems continue functioning correctly even if the enhancement is not adopted.

In essence, our approach combines the best of both worlds: the complete causality and precision of manual context management, with the convenience and robustness of OTP's automatic handling, while avoiding the limitations inherent in each individual approach.

4.3 Assumptions and Design Principles

In addition to leveraging OTP's `gen_server:call` and `call` directives, our approach targets a specific architectural pattern to maintain clarity and enforce causal tracking: the client–server model. In this design, clients serve as distinct logical entities, such as users, external systems, or autonomous agents, that generate independent streams of work by initiating requests. Server processes handle these requests and may, in turn, issue requests to other servers, forming chains of causally related operations that span multiple components.

To provide a concrete boundary for context creation, we assume the presence of a `client.erl` module responsible for interacting with the system. Each client issues requests and awaits responses before initiating subsequent requests, ensuring that individual client sessions maintain a clear causal flow. While multiple clients can operate concurrently, the system is designed to handle their interactions asynchronously, without breaking the integrity of each client's request–response chain. This architectural assumption allows our automated context injection to anchor contexts at the client boundary, providing well-defined, traceable causal chains across the system.

This architecture naturally supports our context generation strategy: client interactions mark the boundaries of causal chains, with each new client request potentially spawning a fresh context. By anchoring context creation at the client boundary, we ensure that causally related operations are consistently tracked throughout the system.

We further assume that complex operations are constructed through chains of service calls. In this model, services communicate primarily using synchronous `gen_server:call` operations, while still allowing for asynchronous execution when necessary by spawning separate processes. Every service in the chain adds value in some form, whether through transformation, validation, persistence, or other business logic. Once the work is completed, the final result propagates back along the same causal chain, ultimately reaching the originating client. This preserves a clear and traceable flow of responsibility throughout the system.

4.3.1 The Asynchrony Problem and Context Necessity

The introduction of asynchrony, an essential feature for building responsive and scalable systems, disrupts the natural causality tracking inherent in synchronous execution. In our target systems, multiple forms of asynchrony are present, each requiring explicit context management to accurately relate messages and verify properties over them. This need for contextual tracking is exemplified in Listing 4.6, where message relationships must be preserved despite the non-deterministic order of execution.

Without explicit context tracking, a monitor observing these messages cannot determine which `store` operation corresponds to which client's original request, making it impossible to reason about their relationship. Using time as a heuristic is unreliable, as it

```
    % Time T1: Client A starts request
    ClientA -> ServiceX: {process, data_a}
    % Time T2: Client B starts request
    ClientB -> ServiceX: {process, data_b}
    % Time T3: ServiceX processes data_b first (scheduling)
    ServiceX -> DatabaseY: {store, data_b}
    % Time T4: ServiceX processes data_a
    ServiceX -> DatabaseY: {store, data_a}
```

Listing 4.6: Concurrent Client Requests

offers no guarantees and may incorrectly correlate messages.

To address this, we can leverage the `handle_call` reference generation mechanism, enabling us to maintain asynchrony in the system while spawning processes and replying later. This approach, illustrated in Listing 4.7, allows us to preserve causal relationships without sacrificing concurrency. The primary trade-off is the overhead of spawning multiple processes; however, since `Erlang` is designed for efficient handling of large numbers of processes, it can easily accommodate this approach, achieving both scalability and reliable message tracking.

```
handle_call({heavy_computation, Data}, From, State) ->
  % Spawn worker to avoid blocking the main service
  spawn(fun() ->
    Result = perform_computation(Data),
    gen_server:reply(From, Result)
  end),
  {noreply, State}.
```

Listing 4.7: Spawned Process Delegation

Our approach targets request-response distributed systems, which may incorporate both synchronous and asynchronous processing patterns. These systems are defined by operations where each request produces a corresponding response, providing a natural form of operational traceability regardless of the underlying processing model used by the target services.

The distinction lies in operational semantics rather than the execution model. Whether a service handles requests synchronously or delegates them to background workers is transparent to our context-assignment mechanism. What matters is that each request ultimately generates exactly one corresponding response, ensuring reliable tracking and reasoning about system behaviour.

4.3.2 The Context Injection Solution

Our automated context injection addresses common gaps in asynchrony and observability by transparently carrying causal information throughout system interactions. Instead of

requiring developers to manually manage correlation IDs or thread contexts across every hop, the system automatically injects and propagates lightweight context metadata along `gen_server:call` chains, ensuring that causally related operations are linked end-to-end.

From the moment a client issues a call, the context is attached to the message envelope and travels with the call/response pair. This makes end-to-end tracing straightforward: each service in the chain receives the context and returns results without any developer intervention to pass or reconstruct correlation information.

New contexts are established at client boundaries, treating each client interaction as the natural starting point of a causal chain. This design ensures that every logical request stream is associated with a single, well-defined context, simplifying trace interpretation by providing a clear entry point for each trace. It also reduces the risk of context contamination that can arise when internal services create contexts arbitrarily.

The client entities, including users, external APIs, scheduled jobs, or autonomous agents, naturally mark where new work enters the system. Typically, each client interaction corresponds to a single logical operation that must be traced from start to finish. By anchoring contexts at these boundaries, traces accurately reflect real-world operations rather than arbitrary internal events.

Generating fresh contexts at the client boundary also helps maintain separation between causal chains: unrelated client requests are not conflated simply because they pass through the same services. This design minimizes false correlations and makes it straightforward to answer questions such as "Which client triggered this sequence of events?" without having to sift through noisy, intermixed traces.

Context creation scales naturally with the number of clients. As concurrent client interactions increase, the system automatically produces independent causal chains without additional coordination, keeping the tracing layer horizontally scalable and aligned with the application's inherent concurrency.

In practice, this approach provides strong observability with minimal developer effort, though it carries pragmatic trade-offs. Maintaining context across spawned or delegated work requires disciplined wrapper usage to ensure every process preserves the context token. Additionally, achieving strict request isolation may necessitate some coordination on the client side, for example, requiring clients to wait for a response before issuing a follow-up request, which is the strategy employed in our work.

Our approach introduces several important advantages. It imposes zero burden on developers, as existing OTP code requires no manual modifications. It also enables true end-to-end tracing, allowing complete request flows to be captured seamlessly across service boundaries.

Another benefit of creating contexts at client boundaries is session awareness: multiple requests from the same client can be correlated as part of a session. However, strict request isolation is not fully guaranteed, since achieving it would require treating each request as a completely independent entity. In our approach, we accept the trade-off of using a single context for all interactions from a given client. The monitor interprets messages to

enforce logical isolation between requests, often requiring clients to wait for a response before issuing a subsequent request. This is necessary because, internally, all requests from the same client share the same context, even though the monitor treats them as separate logical units.

Our approach thus combines the best aspects of manual context management (complete causality) with the convenience of OTP (automatic handling), while avoiding the limitations of both approaches.

4.4 The Conductor

The `conductor` will be the central entity responsible for automatic context management in the system. It operates externally, capturing system messages and communicating with the monitor to provide a real-time view of system activity.

Serving as an oracle for every action in the system, the conductor ensures that all interactions pass through it before proceeding further. In this section, we explore how the conductor operates and the key nuances that must be considered when integrating it into the system.

Naturally, introducing the conductor introduces some overhead. In addition to the monitor that verifies system properties, the conductor executes the function γ, capturing all runtime messages and managing the contexts between them. While this adds computational cost, it provides precise causal tracking and consistent context propagation across all system interactions.

4.4.1 Redirecting Calls

Call redirection to the `conductor` is the pillar of the monitoring system, serving as the mechanism that preserves causal correlations between messages throughout the distributed system. Without programmatic injection of context-tracking code, it would be impossible to maintain causal consistency across actor interactions while keeping the original application logic intact.

Instead of manually modifying source code, our approach leverages `Erlang`'s Abstract Syntax Tree (AST) transformation capabilities via the `parse_transform` module. This enables precise, surgical modifications at the compiled representation level, ensuring that the original source files remain unaltered. The general structure is illustrated in Figure 4.1.

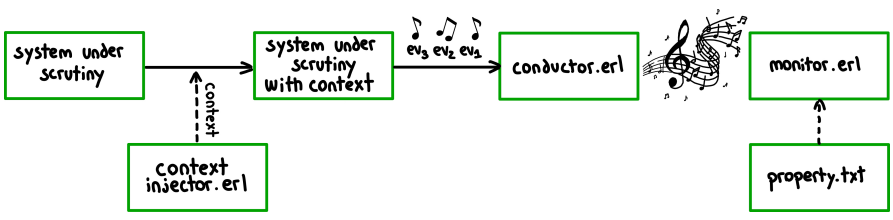


Figure 4.1: Instrumentation pipeline of `ACTORCHESTRA`

A key modification is the transformation of every `call` request. Instead of a message being sent directly to its original target, each call is redirected to the `conductor`, carrying both its original information and additional metadata required for context management. By intercepting all messages in the system, the `conductor` ensures that every communication passes through it. It assigns context references and delegates them whenever messages belong to the same causal chain, enabling consistent end-to-end tracking.

This approach embodies the principle of transparent context injection. To illustrate, consider a standard OTP-based client-server system shown in Listing 4.8 with the original code.

```
% Client code - unchanged
handle_call({send_number, Number}, _From, State) ->
    Result = gen_server:call(central_server, {process, self(),
        Number}),
    {reply, Result, State}.
```

Listing 4.8: Standard OTP System Before Enhancement

Through parse-time transformation, our context injector automatically enhances the code to re-direct the original call to the `conductor`, sharing the context and the module, as depicted in Listing 4.9.

```
% Client code - automatically transformed
handle_call({send_number, Number}, _From, State) ->
    Result = conductor:call(central_server, {process, self(),
        Number}, Context, client),
    {reply, Result, State}.
```

Listing 4.9: After Automatic Context Injection

This transformation allows the system to implement the function γ, which maps messages to their corresponding contexts. By routing all system actions through the `conductor`, the injector ensures that contexts are consistently assigned and related, providing accurate end-to-end tracking of causal chains.

$$\gamma(\texttt{message}) = \begin{cases} \texttt{Context} & \text{if message belongs to established context} \\ \texttt{gen\_context()} & \text{if message initiates new context} \end{cases}$$

The `conductor` forwards all messages to the monitor, generating specific signatures that the monitor anticipates. Thanks to this context management, the monitor can distinguish between different contexts and evaluate properties within each independently. In essence, the `conductor` effectively simulates the γ function.

4.4.2 Injection Mechanisms

To enable context injection, each `Erlang` module must include the following compiler directive in its header:

```
-compile(parse_transform, context_injector).
```

This flag instructs the `Erlang` compiler to process the module's AST according to the transformations defined in `context_injector.erl`, which analyses the existing code structure and applies the necessary modifications automatically.

The `context_injector` performs four primary categories of transformations: state management enhancement, context reception handling, message call transformation, and context extraction.

To start, the injector detects whether the module uses record-based or map-based state management. Following OTP conventions, developers may choose either approach for maintaining state. To accommodate this flexibility, the injector supports both record and map-based state representations, ensuring that all system files can be correctly transformed regardless of the underlying data structure.

In the OTP framework, a process's state is formally an opaque value passed through successive callback invocations. The runtime system imposes no restriction on its type: any `Erlang` term is admissible. In principle, a process may maintain its state as a single integer, a tuple, or even a list. However, both the `Erlang`/OTP standard library and idiomatic application design show that structured representations, specifically records and maps, are overwhelmingly preferred in practice [8, 29].

The rationale is twofold. First, records and maps provide semantic clarity by associating symbolic names with fields, avoiding the positional ambiguities inherent in tuples or lists. Second, they enhance reliability: records offer compile-time validation of field access, while maps provide runtime flexibility to accommodate evolving state schemas without extensive code modification. This duality reflects a key design trade-off in `Erlang`: records dominate the OTP infrastructure, whereas maps are adopted due to their dynamic evolution behaviour.

From a software engineering perspective, using records or maps balances efficiency, readability, and maintainability in concurrent systems. The literature consistently reinforces this consensus: while arbitrary terms are technically valid as state, disciplined use of records or maps is a hallmark of well-structured and robust OTP system [8].

Our context injection mechanism builds on these assumptions, specializing its transformations for the two idiomatic state representations in `Erlang`. In record-based modules, the injector augments the record definition with an additional `context` field. In map-based modules, context management is introduced via `maps:put` and `maps:get` operations.

This design ensures that the injected code remains aligned with established `Erlang` programming idioms, thereby minimizing cognitive overhead for developers and maximizing the maintainability of the transformed system. Crucially, it also reflects an aspect-oriented

approach at the compiler level: rather than introducing an alternative representation of process state, the injector integrates context propagation into the forms already recognized as best practice within the community.

By constraining the mechanism to operate over records and maps, we simultaneously achieve compatibility, clarity, and methodological soundness. While the `Erlang` runtime permits unconstrained state representations, restricting the injection mechanism to the dominant patterns constitutes a deliberate design choice: it anchors the system in conventions that have proven effective in the construction of reliable, large-scale concurrent systems. In doing so, it aligns the technical design of context injection with both the theoretical foundations of OTP's process model and the practical conventions of expert `Erlang` development.

Therefore, the injector augments process state to include a dedicated `context` field. If a state record is defined, a `context` field is added, and set up as `undefined` as its initial value. When clients interact with our system, this value will constantly change. The change can consulted in Listing 4.10.

```
-record(state, {
    existing_field1,
    existing_field2,
    context = undefined  % Injected field
}).
```

Listing 4.10: Context addition in record

For map-based state handling, the injector introduces context operations via `map:put` and `map:get` whenever we have to update the context value or retrieve, if it already exists. The injection can be visible at Listing 4.11.

```
% Context storage
maps:put(context, Context, State).
% Context retrieval
maps:get(context, State, undefined).
```

Listing 4.11: Context addition in map based structures

In addition to injecting the `context` field, each `gen_server` is augmented with a dedicated `handle_call` clause to receive and store context updates from the `conductor`. To ensure atomicity, these updates are processed as a single message. In this clause, the server receives the message from the `conductor`, updates the context variable, and then invokes the original function within the module that would have been called by the original requester.

The `context` value is replaced for each interaction, which is expected: as long as the original message and its associated context are signaled to the monitor by the `conductor` as an atomic pair, causal consistency is preserved. This mechanism is realized through the `with_context` handler, which receives both the original request and the context from the

conductor, updates the state accordingly, and executes the original function within the same module. An implementation example of this handler is presented in Listing 4.12.

```
handle_call({with_context, Context, Msg}, From, State) ->
    % Map-based:
    StateWithContext = maps:put(context, Context, State),
    % Record-based:
    StateWithContext = State#state{context = Context},
    ?MODULE:handle_call(Msg, From, StateWithContext);
```

Listing 4.12: `with-context` handler injection

4.4.3 Forward Propagation

As discussed, the most critical transformation involves rewriting all inter-process calls to pass through the `conductor`. Specifically, each original `call` is replaced with a corresponding `call` to the `conductor`. For example, consider the following original code:

```
Result = gen_server:call(TargetPid, process, Data).
```

we will have the following change:

```
Result = conductor:call(TargetPid, process, Data, Context, ?MODULE).
```

where the context is automatically extracted from the caller's state and the `?MODULE` identifies the calling module for monitoring and tracing purposes. When a chain exists in the system, the context is propagated forward throughout the interaction. Upon completion of an operation, the result has to be delivered to the client, and the client's context needs to update accordingly.

This architecture ensures that the context flows across the entire chain of interactions, with the `conductor` relaying all events to the running monitor. As the context propagates, the injected `with_context handle_call` functions assign the current context to each process, effectively maintaining context consistency across the full interaction chain of request and response calls.

4.4.4 Back-Propagation

Back-propagation of the context does not require updating the state in every process it passes through; this is already handled during forward propagation. The `conductor` maintains all necessary information to provide a complete execution trace to the monitor, so internal calls do not need to modify or store the context as results return in the chain.

However, the client process, being the anchor of the communication, must update its injected `context` field. Since the client cannot automatically update this field during the call, the context is piggybacked on the response. A special handling mechanism in the client module extracts the context from the result and updates the client's state accordingly.

This is illustrated in Listing 4.13, enabling correlation of all subsequent requests from the same client when needed.

```
handle_call(operation, _From, State) ->
    ...
    {Reply, NewContext} = conductor:call(...),
    {reply, Reply, maps:put(context, NewContext, State)}.
```

Listing 4.13: Client processing of context

This extraction ensures that any subsequent interaction from the same client uses the context established during the first interaction. Request isolation is preserved, as the monitor can interpret each context independently within separate `receive` chains, while still allowing correlation of messages originating from the same client.

4.4.5 Context Assignment

At the start of an actor's life cycle, the `context` variable is initialized to `undefined`. As the actor begins communicating within the system, the `conductor` assigns values to the `context` to establish causal relationships between messages.

The `conductor` accomplishes this by sending the `context` through the special handler injected into each actor (`with_context`). Upon receiving a message, the `conductor` checks whether the `context` is already set. If not, it generates a unique reference using `Erlang`'s built-in `make_ref()` function and shares this reference along with the original message with the target actor.

The `conductor` achieves this by sending the `context` through the special `handle_call` that we injected into each actor. The `conductor` receives a message, examines if the `context` field has a value to it, if not it generates a unique reference, using `Erlang` built in function `make_ref()`, and shares that reference with the target actor that was supposed to receive the initial message and the original message at the same time.

The key concern is the correct assignment and sharing of context at the time of interaction, since the `conductor` captures and replays this information to the monitor. Multiple clients may overwrite a component's context value, but as long as the pairing of message and context is accurately shared with the `conductor`, causal consistency is maintained.

Special attention is required when propagating context, both from the actors and the `conductor`. Each actor maintains a `context` field, which may be overwritten throughout its life cycle. As long as the correct context is associated with each message, causal consistency is preserved. To ensure this, the context must be extracted before sending it to the `conductor`.

In `Erlang`, it is considered good practice to avoid letting spawned processes access mutable state directly. The context must be captured before spawning, particularly when the system employs OTP `gen_server:call` and asynchronous behaviour, as discussed

in Section 4.3.1. Without this precaution, the system might inadvertently propagate an incorrect or outdated context.

To address this, we inject a single line of code that extracts the context at the beginning of the `handle_call`, ensuring that any spawned process receives the exact context active when the call was initially handled. The implementation is shown in Listing 4.14.

```
handle_call({process, Data}, _From, State) ->
  Context = State#state.context,
  spawn(fun() ->
    Result = conductor:call(Target, Msg, Context, ?MODULE),
    gen_server:reply(From, Result)
  end)
```

Listing 4.14: Correct context extraction

4.4.6 Concurrency and Dead-Locks

The `conductor` inevitably introduces some overhead, as all messages must pass through it. To minimize interference with the system, the `conductor` leverages asynchrony wherever possible, enabling multiple messages to be routed concurrently and making it suitable for highly concurrent scenarios.

Asynchrony is also essential to prevent deadlocks. Since all processes communicate with the `conductor`, a chain of dependent events could cause the `conductor` to block if it processed messages synchronously. For example, if a response depends on the actions of another process, a synchronous `conductor` might stall when the dependent process attempts to interact with it. By handling message redirection asynchronously, the `conductor` avoids such deadlocks. Thus, asynchrony is not merely an optimization for performance, it is a requirement for correct orchestration.

4.4.7 New Signatures

With the `conductor` managing contexts, the monitor no longer requires `Erlang`'s built-in `tracing` tool. The `conductor` handles message tracking and forwards only the relevant information to the monitor, specifically, the messages that the monitor is expecting according to the property being verified.

This approach significantly reduces overhead. In dynamic systems, traditional tracing mechanisms capture all system events and then filter out irrelevant ones, which is costly. With the `conductor`, only the relevant communications are sent to the monitor. Messages outside the scope of the property are automatically ignored, simplifying monitoring and focusing only on meaningful events. Instead of relying on the built-in tracing signatures,

such as:

```
{trace, Pid, send, Msg, To}
{trace, Pid, 'receive', Msg}
{trace, Pid, return_from, Module, Function, Arity, ReturnVal}
...
```

we can disable these tracing mechanisms and rely on the messages produced by the `conductor`, which follow a simple signature:

```
{Sender, Receiver, Message, Context}
```

This simplifies monitor design, as it only needs to handle messages in the `conductor`'s format, while the attached context enables causal reasoning between different messages in the system. Although it would be possible to retain the tracing mechanisms and augment each message with context, doing so would unnecessarily increase overhead, which is avoided by relying solely on the `conductor`.

4.5 Conductor re-direction

As introduced earlier, the `conductor` injects a `context` field into the system. This context may be overwritten multiple times, depending on the different interactions taking place. Naturally, it is crucial to ensure that each message is consistently associated with the correct context. To guarantee this, the `conductor` sends messages as tuples, preserving atomicity between the message and its context.

Consider a simple system with two modules: `client.erl` and `server.erl`. Clients can send messages to the server, which simply echoes them back. Even in this simple setup, concurrency issues can arise if context handling is not performed carefully.

Whenever a message is sent from the `client` to the `server`, the call is redirected to the `conductor`. The original `call` from the client is transformed into a `call` to the `conductor`, including both the message and the destination process. The following interaction illustrates this behaviour:

$$\texttt{client1} \rightarrow \texttt{server}$$
$$\texttt{client2} \rightarrow \texttt{server}$$

will be transformed into the following interactions with the `conductor`:

$$\texttt{client1} \rightarrow \texttt{conductor}$$
$$\texttt{conductor} \rightarrow \texttt{server}$$
$$\texttt{client2} \rightarrow \texttt{conductor}$$
$$\texttt{conductor} \rightarrow \texttt{server}$$

The `conductor` forwards messages to their intended recipients along with the associated context. To achieve this, the `conductor` first determines whether the message should be propagated with an existing context or if a new context should be generated for a fresh interaction. This is handled by a dedicated `call` endpoint, which all other processes invoke. The handler inspects the message to check for an existing context and either reuses it or generates a new one as needed, as illustrated in Listing 4.15.

```
call(To, Msg, undefined, Module) ->
    Context = make_ref(),
    gen_server:call(?MODULE, {send_with_context, To, Msg,
       Context, Module});
call(To, Msg, Context, Module) ->
    gen_server:call(?MODULE, {send_with_context, To, Msg,
       Context, Module}).
```

Listing 4.15: Conductor Context Assignment

Next, the `conductor` invokes its internal method responsible for redirecting the original message, capturing the response, and reporting all interactions to the running monitor. To avoid deadlocks and optimize performance, the `conductor` spawns worker processes to handle message redirection, which is especially important under high concurrency with multiple clients. This mechanism is illustrated in Listing 4.16. It is important to distinguish between the `client` module and other modules: for the `client`, the context must be propagated back so that it can be stored locally.

```
handle_call({send_with_context, To, Msg, Context, Module}, From,
    State) ->
    ToModule = get_target_module(To),
    monitor ! {Module, ToModule, Msg, Context},
    spawn(fun() ->
        worker_process(To, Msg, Context, Module, From, ToModule)
    end),
    {noreply, State}.

worker_process(To, Msg, Context, Module, From, ToModule) ->
  Reply = gen_server:call(To, {with_context, Context, Msg}),
  FinalReply = case is_client_module(Module) of
    true -> {Reply, Context};
    false -> Reply
  end,
  monitor ! {ToModule, Module, Reply, Context},
  gen_server:reply(From, FinalReply),
  end.
```

Listing 4.16: Conductor Re-direction of Messages

The proposed solution introduces a wrapper-based architecture that fundamentally changes how context propagation occurs in inter-process communication. Instead of transmitting context and original messages separately, which can create race conditions and synchronization complexities, the system embeds contextual information directly within the message payload using a specialized envelope pattern. This effectively transforms the traditional two-message protocol into a single atomic operation.

The core innovation is the context injection handler at the receiving process level. When a target process receives a message of the form `{with_context, Context, OriginalMessage}`, the injected handler immediately extracts the context, updates the process state with this information, and delegates the original message to the appropriate receiver.

4.5.1 Monitor's Melody

The monitoring component provides comprehensive observability into the causal flow of messages across the distributed system. The monitor receives notifications for each message transmission, including both outbound messages and their corresponding responses, along with the associated context identifier. This bidirectional tracking enables the construction of complete request-response cycles that can be correlated across arbitrary process topologies.

The monitor's effectiveness relies on the atomic nature of the context injection mechanism. Because the context and message are transmitted together as a single unit, the monitoring system obtains a consistent view of the causal chain without the temporal ambiguities that would arise from separate transmissions. Each monitored interaction includes the source module, destination module, message content, and context reference, providing sufficient information to reconstruct the full execution trace.

This approach is particularly valuable in complex distributed workflows, where a single client request may trigger cascading operations across multiple services. The monitor can follow these operations as they propagate through various processing stages, preserving causal linkages that support debugging, performance analysis, and end-to-end request tracing.

4.6 System Symphony

Our monitoring approach addresses the fundamental challenge of maintaining coherent operation traceability in distributed systems through a deliberate architectural choice that combines context management with mandatory client API abstraction layers. This design reflects the recognition that effective distributed system monitoring requires careful consideration of both technical implementation constraints and natural system interaction patterns.

Depending on the system that we are working with, the way that we perform this causality tracking changes drastically, and in some systems, such as publish-subscribe

architectures, our approach breaks completely. Therefore, we must focus on one type of architecture, since tackling all the possible ones would be nearly impossible.

Our approach targets distributed systems that follow the actor-based model, specifically the client-server interaction pattern implemented using `Erlang` OTP `gen_server:call`. These systems exhibit request-response interaction patterns, where each operation has an initiator (the client) that waits for a response. Examples include traditional client-server systems, service interactions with request-response cycles, pipeline architectures, and microservice architectures. The key characteristic is that all supported systems share the fundamental property that each request produces a response, enabling traceability and context management.

Clients serve as the points where new contexts enter the system, providing a controlled mechanism for context creation. Without this client-driven context injection model, server processes would retain static contexts indefinitely, leading to context pollution, where unrelated operations become incorrectly associated across different client sessions. By isolating contexts at the client level, context boundaries align naturally with client interactions, preventing context staleness in long-running server processes.

This architectural choice addresses a critical challenge that arises in systems lacking client-level context management. In scenarios where clients interact directly with the system, such as via terminal interfaces or direct protocol access, the system might rely on internal databases or session stores for client identification. However, without client-side context preservation, each interaction is treated as an isolated event. When multiple requests originate from the same logical client session, the lack of maintained context state prevents the establishment of proper causal relationships, even when these relationships reflect important operational dependencies.

Despite the constraints imposed by sequential operations at the client level, this approach effectively addresses a substantial portion of distributed system architectures commonly encountered in enterprise environments. Examples include HTTP API interactions, service-to-service communications, database transaction sequences, and traditional request-response patterns. These scenarios exhibit natural session boundaries that align well with our monitoring model, providing comprehensive observability while maintaining clear boundaries regarding its intended application domain.

Chapter 5

Evaluation

We perform two different evaluations, in two different systems: a simple calculator system performing addition and multiplication, and a more complex chat room management system, similar to the one introduced in Chapter 2. The goal is to quantify the performance overhead of introducing the `conductor` and monitors alongside the system and understand the trade-offs between runtime property verification and unmonitored execution.

5.1 Experimental Set-up

The evaluation is conducted by running the same deterministic workloads on both a baseline system (unaltered original code) and its instrumented counterpart, which includes code injection, the `conductor`, and monitors. To reduce bias, the execution order of baseline and instrumented versions is randomized. Each experiment is repeated 10 times, and results are reported as mean values. For both systems, we measured the total time required to perform the requests in order to calculate the overall overhead, latency degradation, and throughput degradation. In the context of concurrent systems, these metrics provide more meaningful insights than execution time alone. For the calculator system, we developed a script that connects multiple clients to the system, and then for each will execute a determinant amount of send requests to the system. We perform this evaluation to different numbers of clients and requests per client. As for the chat system, we simulate a more real interaction, where we have multiple clients, each joining a given room and posting messages into it, trying to simulate a real chat room with clients. The benchmarks were conducted on a MacBook Air equipped with an M1 chip.

When benchmarking `Erlang` systems, certain precautions are essential to ensure reliable measurements. One of the most critical aspects is accounting for the warm-up phase of each run. Executing benchmarks immediately after startup can introduce

measurement artifacts, as the `Erlang` VM employs a lazy-start architecture [10]. Allowing the system to warm up stabilizes performance and produces results that more accurately reflect steady-state behaviour. This is easily spotted in plots that exhibit performance spikes in the initial set of data points. Such erroneous readings can be minimised or eliminated if a series of warm-up requests are performed before the actual experiment started. Doing this ensures that we reduce the erroneous readings.

5.2 Addition and Multiplication

Consider a system where a `client` interacts with an entry point, which is the `central` process. A client will be able to send a `connect` request to this given system, and while being connected he can send numbers to it. This system is supposed to add `10` to the number the client sent and then multiply it by `2`. The structure, alongside with the signatures of the messages to perform the number addition and multiplication is depicted in Figure 5.1.

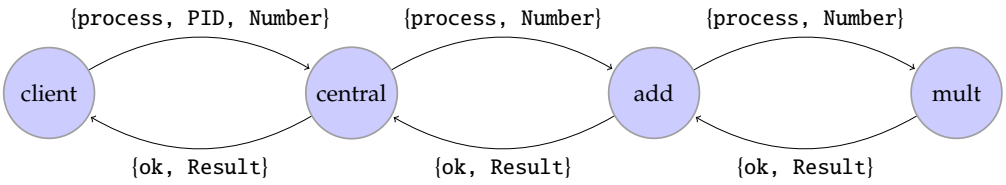


Figure 5.1: Addition and multiplication system

An interaction with this system can be observed by the following trace:

σ = {process, <0.1.0>, 10}{process, 10} {process, 20} {ok, 40} {ok, 40} {ok, 40}

This systems is structured as a pipeline, which is one of the classic architectures in some `Erlang` systems, where a process makes a requests and multiple processes work together in an established order to perform something over that request. The result will then piggyback to the client after all the interactions completed. Note that, even if the client has to wait before performing another request, the system itself is able to deal asynchronously with multiple clients by spawning processes to deal with each client request, responding when such process finishes its job.

After augmenting the system with our code injection, the communication flow changes significantly. Instead of processes interacting directly, all messages are now routed through the `conductor`. Every occurrence of `gen_server:call` is transformed into `conductor:call`, effectively re-directing communication through this intermediary. The `conductor` does not alter the contents of the messages; it simply forwards them to their intended recipients. At the same time, it reports every observed interaction to a dedicated monitor. This monitor is deployed alongside the system and receives the messages with their associated context, enabling runtime property verification. The resulting architecture of the instrumented system is illustrated in Figure 5.2.

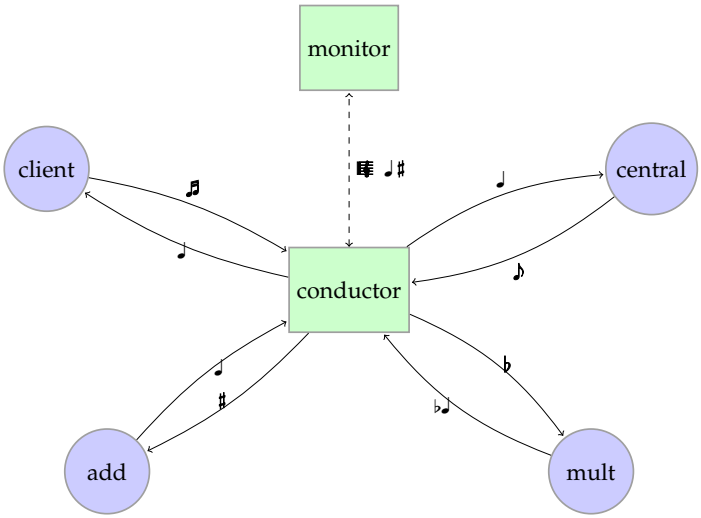


Figure 5.2: Conductor as hub routing all messages

Depending on the property that we are monitoring for, the monitor can have a more complex structure, or follow a simpler `receive` chain structuring. A natural property is to verify if the system as a whole behaves correctly, that is, if the number that a client sends is indeed proceed as expected. In order to do that we could specify the following property in `WATLZ`:

```
OMEGA(
  send {client -> central} WITH {process, _, Number} : TOP;
  send {central -> add} WITH {process, Number1} : TOP;
  send {add -> mult} WITH {process, Number2} : Number2 ==
     Number1+10;
  send {mult -> add} WITH {ok, Result} : Result == Number2 * 2
)
```

The monitor generated for this property is straightforward: it consists of a sequence of `receive` statements, each responsible for checking the property at its corresponding step in the chain.

The experiments were conducted first with a single client interacting with the system and then with multiple concurrent clients. Across both scenarios, the outcome is consistent: `ACTORCHESTRA` introduces noticeable overhead. This result is expected, as every message must be routed through the `conductor`, which assigns context, forwards the message to the intended process, and simultaneously reports it to the monitor. While this makes the `conductor` a performance bottleneck, it is a necessary trade-off for guaranteeing causality tracking and propagation. The results, shown in Figure 5.3 and Table 5.1, confirm the theoretical expectation: any architecture that centralizes message handling and couples it with monitoring inevitably incurs additional cost. Systems that natively embed causality within their design would experience lower overhead, but in our case, this mechanism is essential to ensure correctness.

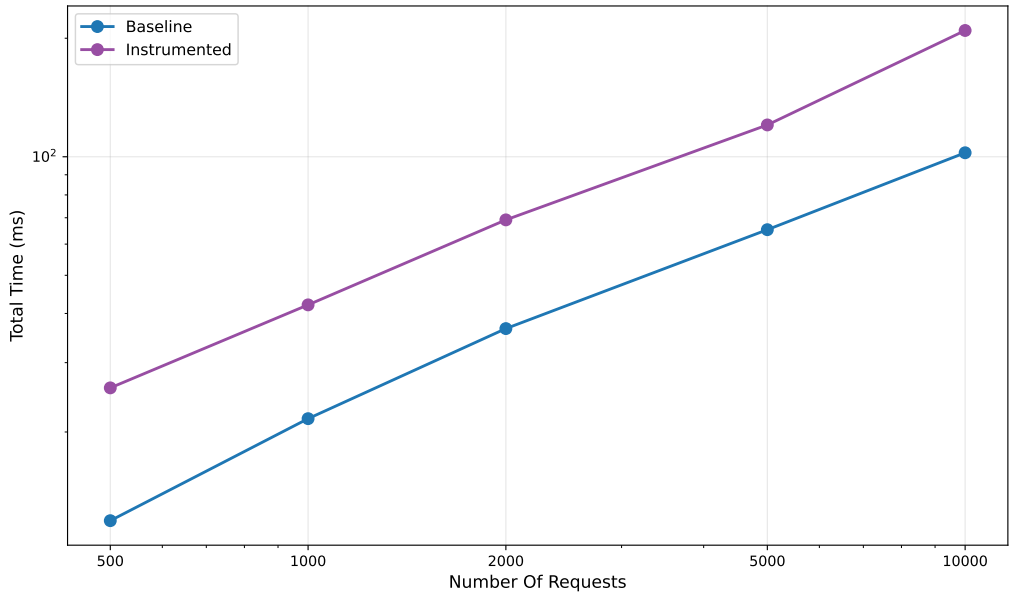


Figure 5.3: Single Client Performance - Baseline vs Instrumented

Clients	Req/Client	Total Req	Baseline (ms)	Instrumented (ms)	Overhead (%)
1	500	500	11.9	25.9	119.0
1	1000	1000	21.6	42.1	95.5
1	2000	2000	36.6	69.1	89.6
1	5000	5000	65.3	120.5	85.3
1	10000	10000	102.4	209.3	105.4

Table 5.1: Single-client benchmark results

The calculator system evaluation reveals important characteristics of our instrumentation overhead. Under single-client conditions, we observe an initial overhead of 119% for light loads (500 requests), which decreases to approximately 105% as the system handles heavier loads (10,000 requests). This trend indicates that our instrumentation exhibits favorable scaling properties, the relative cost of causal tracking and monitoring does not spike exponentially as the system processes more requests. The performance of our system with multiple clients can be visualized in Figure 5.4 and the detailed data in Table 5.2.

The multi-client results demonstrate how our instrumentation behaves under concurrent access patterns. Across different client configurations (2, 4, 8, and 16 clients), the overhead stabilizes in the 65-131% range, with most configurations converging around 100-130% overhead. Notably, the overhead does not exhibit exponential growth with client count, indicating our coordination mechanism scales well with concurrency.

The process of assigning causal context to each message introduces computational overhead that varies significantly based on request patterns. The data suggests this overhead is not simply proportional to request volume, but rather depends on the temporal distribution of requests and the resulting process scheduling behaviour. Real distributed

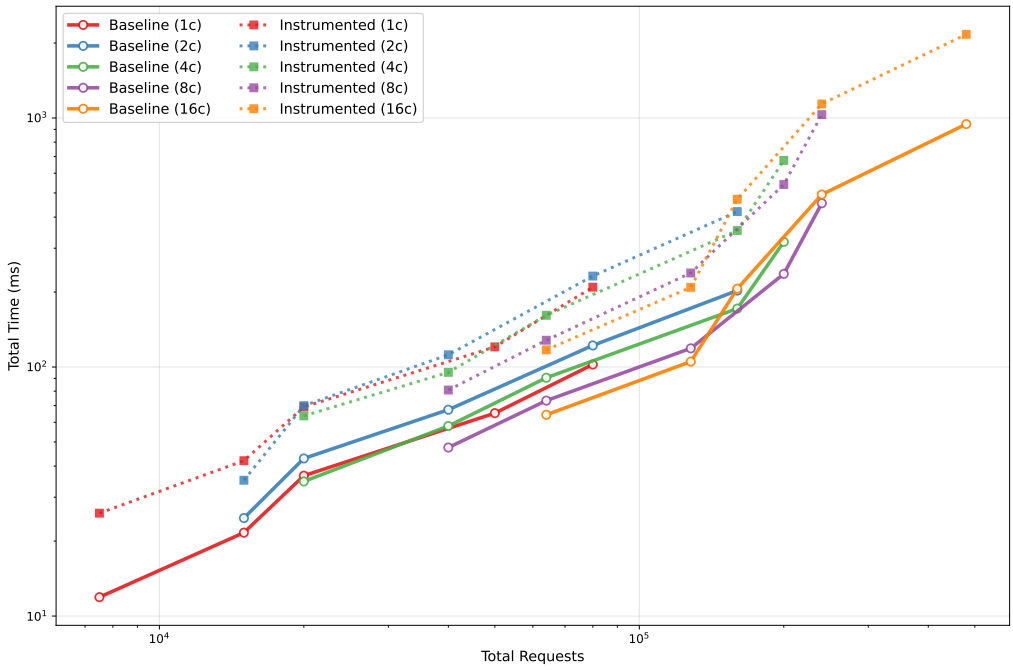


Figure 5.4: Multiple Clients Performance - Baseline vs Instrumented

Clients	Req/Client	Total Req	Baseline (ms)	Instrumented (ms)	Overhead (%)
2	500	1000	24.8	35.1	42.5
2	1000	2000	42.9	70.0	63.6
2	5000	8000	122.1	231.9	91.5
2	10000	16000	202.8	420.8	108.1
4	500	2000	34.7	63.7	84.4
4	1000	4000	57.9	95.2	65.1
4	5000	16000	171.7	353.3	106.5
4	10000	20000	318.0	675.6	113.1
8	500	4000	47.5	80.8	71.1
8	1000	6400	73.3	128.1	75.2
8	5000	20000	236.6	540.5	129.7
8	10000	24000	454.6	1031.9	128.5
16	500	6400	64.3	117.2	83.6
16	1000	12800	105.1	208.9	99.7
16	5000	24000	492.8	1138.5	131.7
16	10000	48000	945.3	2166.7	129.6

Table 5.2: Multi-client benchmark results for calculator system

systems are inherently concurrent, and our framework demonstrates that instrumentation overhead remains bounded and predictable under multi-client scenarios, ranging from 42.5% to 131.7%. Crucially, the overhead does not grow exponentially with increased concurrency, suggesting that the architecture scales in a controlled manner even under realistic, concurrent workloads.

5.3 Chat System

In order to study the impact of our framework in a more complex system, we will also benchmark the performance overhead over a chat room system, similar to the one provided in Section 3.8.

The idea of this system is that the clients will register into a main server, and then they will be able to join different chat rooms and send messages to those chat rooms. The system would be composed of three main components, the `chat_server` that is the entity communicating with the client, the `chat_room` which will correspond to each chat room created by the `chat_server`, and lastly a `registry_server` that will register and manage the clients into the system. As stated, there will always be the `client` entity which serves as the entry point to the communication with the system. What we will study in this evaluation, are two different version of this chat system.

The first one, called `no-broadcast` version, will behave differently from the `broadcast` version. Simply put, in the `no-broadcast` version, whenever a client posts a message it will simply be logged into a file, and in the `broadcast` version we will actually broadcast the message to all the clients in the room in an asynchronous fashion, using the `gen_server:cast` directive. The interaction schema and the monitored property are the same as those introduced in Section 3.8.1, namely verifying that all registered clients in the system post messages exclusively in the rooms they have joined.

5.3.1 No Broadcast Version

Our evaluation across different client configurations reveals that the message processing overhead maintains remarkable consistency. As the system scales from 5 clients sending 30 messages each to 200 clients sending 10 messages each, the overhead stabilizes between 127% and 145%. This narrow range demonstrates that our architecture handles the increased coordination complexity without exponential performance degradation.

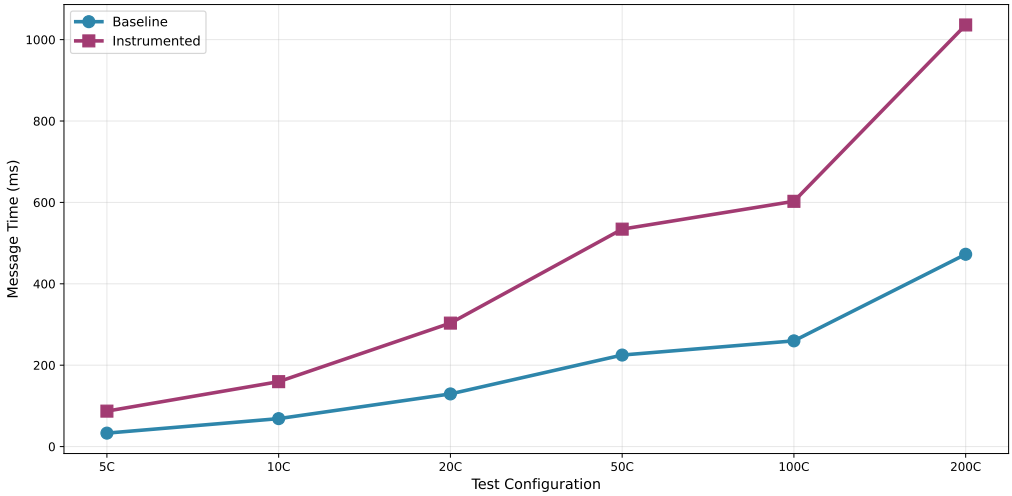


Figure 5.5: No-Broadcast System: Message Processing Time

The configuration with 200 clients shows the highest overhead at 144.9%, which is expected as the conductor must coordinate a larger number of concurrent message streams. However, the fact that this represents only a 17.7% increase over the lightest configuration (5 clients at 127.2% overhead) indicates excellent scalability characteristics for our causal tracking mechanism. The results of the chat system benchmarks are visible in Figure 5.5 and Table 5.3.

Clients	Msg/Client	Total Msg	Baseline (ms)	Instrumented (ms)	Overhead (%)
5	30	150	38.3	87.0	127.2
10	30	300	68.7	159.5	132.2
20	30	600	129.4	303.4	134.5
50	20	1000	224.9	534.3	137.6
100	10	1000	259.8	602.7	132.0
200	10	2000	428.1	1048.4	144.9

Table 5.3: No-Broadcast System: Message Processing Time Table

The results are slightly higher compared to the addition and multiplication system, primarily because that system was a "toy example," whereas the chat system reflects the complexity of a real-world distributed environment. Unlike the simpler case, the chat system requires explicit state management, more intricate message patterns across processes, and persistent storage, all of which contribute additional overhead.

Each user triggers multiple instrumented operations across the system, such as connecting, joining a room, sending messages within that room, joining another room, and eventually disconnecting. These interaction patterns are considerably more complex, leading to a higher volume of messages sent to the `conductor`.

The increased complexity of the monitor itself also contributes to the overall system overhead. Unlike in simpler settings, the monitor in this scenario can spawn sub-monitors to check property satisfaction and route messages to the appropriate instance. Combined with the higher volume of messages being funneled through the `conductor` and the intricate structure of the monitoring framework, these factors compound the cost. In effect, both the system's orchestration and the monitoring infrastructure become more demanding, amplifying the overhead in this already complex environment.

The latency measurements, presented in Figure 5.6 and Table 5.4, reveal the most direct impact of our instrumentation on user experience. Across all test configurations, the instrumented version consistently exhibits approximately double the latency of the baseline system. The latency and throughput measures, were calculated with the time of the messaging phase of the system, in order to ignore potential noise and focus on the actual work of the system.

The chat system evaluation provides additional insights into real-world performance characteristics. Message processing overhead ranges from 127% to 145%, while latency

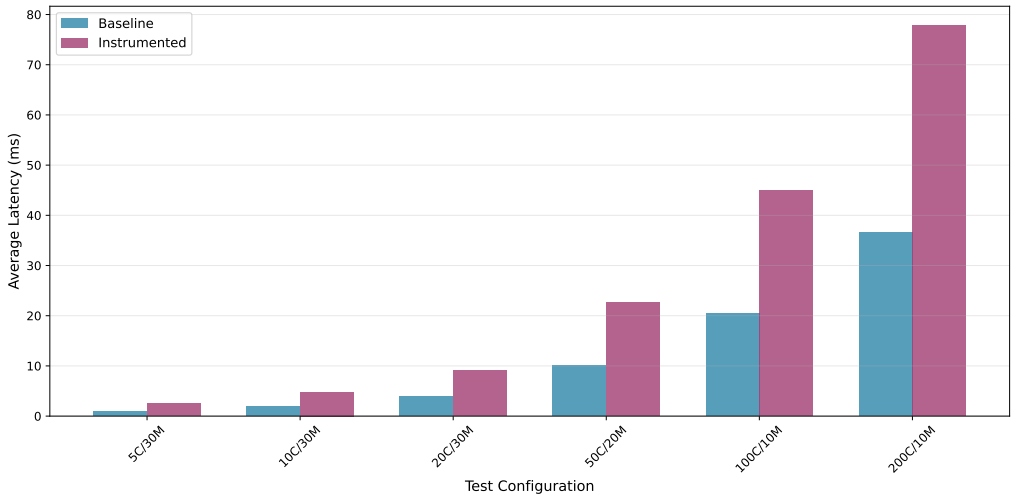


Figure 5.6: No-Broadcast System: Average Latency Comparison

consistently doubles compared to the baseline. Latency measurements show the instrumented version achieving approximately 50% of baseline performance, which represents the most significant impact of our approach.

Clients	Msg/Client	Total Msg	Base Lat (ms)	Instr. Lat (ms)	Δ Lat (%)
5	30	150	1.0	2.6	61.5
10	30	300	2.0	4.8	58.3
20	30	600	3.9	9.2	57.6
50	20	1000	10.1	22.6	55.3
100	10	1000	20.4	45.0	54.6
200	10	2000	36.6	77.8	52.9

Table 5.4: No-Broadcast System: Average Latency Table. Δ Lat (%) = degradation from baseline.

The throughput analysis, depicted in Figure 5.7 and Table 5.5, provides insight into the overall system capacity under our instrumentation. The baseline chat system achieves throughput ranging from approximately 4,800 to 5,300 messages per second across different client configurations. Under instrumentation, this drops to a consistent range of 2,000 to 2,500 messages per second.

The throughput limitation stems from the sequential nature of causal context assignment in the `conductor`, each message must be processed to determine its place in the causal chain before it can be forwarded.

Despite this reduction, the instrumented system still maintains substantial capacity for development-time evaluation. Processing 2,000+ messages per second provides ample performance for testing complex interaction scenarios, load testing with realistic user counts, and validating system behaviour under stress conditions.

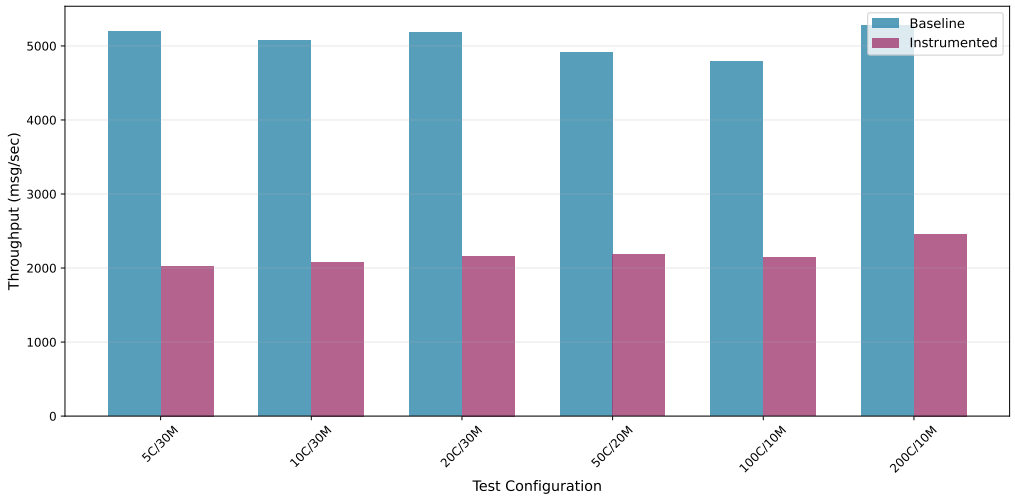


Figure 5.7: No-Broadcast System: Throughput Comparison

Clients	Msg/Client	Total Msg	Base Thpt (msg/s)	Instr. Thpt (msg/s)	Δ Thpt (%)
5	30	150.0	5193.5	2026.2	61.0
10	30	300.0	5074.3	2082.2	59.0
20	30	600.0	5176.9	2162.6	58.2
50	20	1000.0	4907.6	2180.2	55.6
100	10	1000.0	4792.2	2136.9	55.4
200	10	2000.0	5272.3	2460.7	53.3

Table 5.5: No-Broadcast System: Throughput Table. Δ Thpt (%) = degradation from baseline

5.3.2 Broadcast Version

When benchmarking our system, we depared ourselves with some very interesting results regarding the broadcast version of the chat system, which was the initial implementation that we intended for it. We conducted the same experiments, with the same configurations and deterministic actions for this version, and the conclusion can be visible in Figure 5.8 and Table 5.6.

Clients	Msg/Client	Total Msg	Baseline (ms)	Instrumented (ms)	Overhead (%)
5	30	150	97.5	141.5	45.0
10	30	300	285.5	348.6	22.1
20	30	600	930.5	1112.4	19.5
50	20	1000	3675.9	4080.0	11.0
100	10	1000	7524.5	8017.5	6.6
200	10	2000	31 209.0	32 467.3	4.0

Table 5.6: Broadcast System: Message Processing Time Table

Examining the data reveals a markedly different outcome compared to the `no-broadcast`

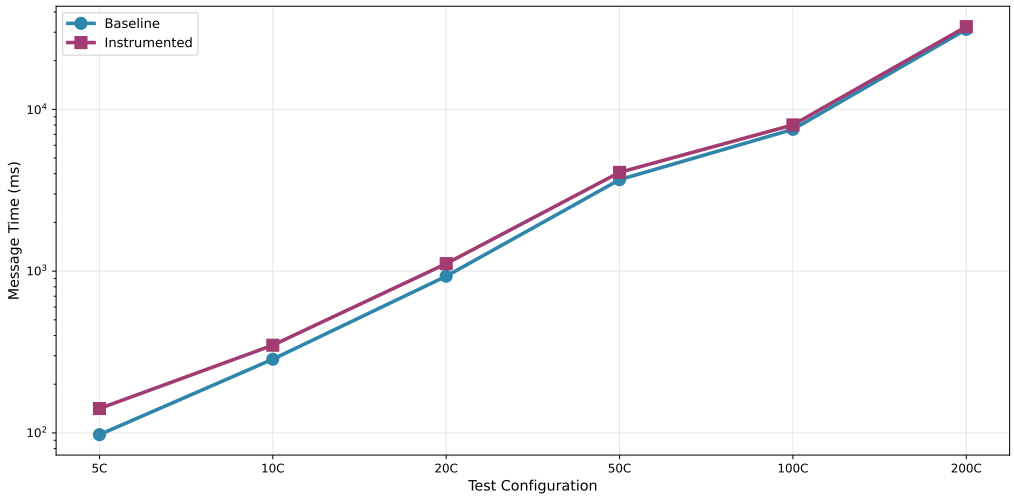


Figure 5.8: Broadcast System: Message Processing Time

version: the observed overhead is significantly lower and decreases as the workload increases. This behaviour can be attributed to the amortization of the instrumentation cost across a higher volume of system activity. In this broadcast scenario, each message is sent to all clients in the chat room, causing the system's core workload to grow more rapidly than the instrumentation overhead. As a result, the relative cost of monitoring diminishes over time, a dynamic that does not occur in the simpler `no-broadcast` version, where instrumentation dominates due to the lower overall workload.

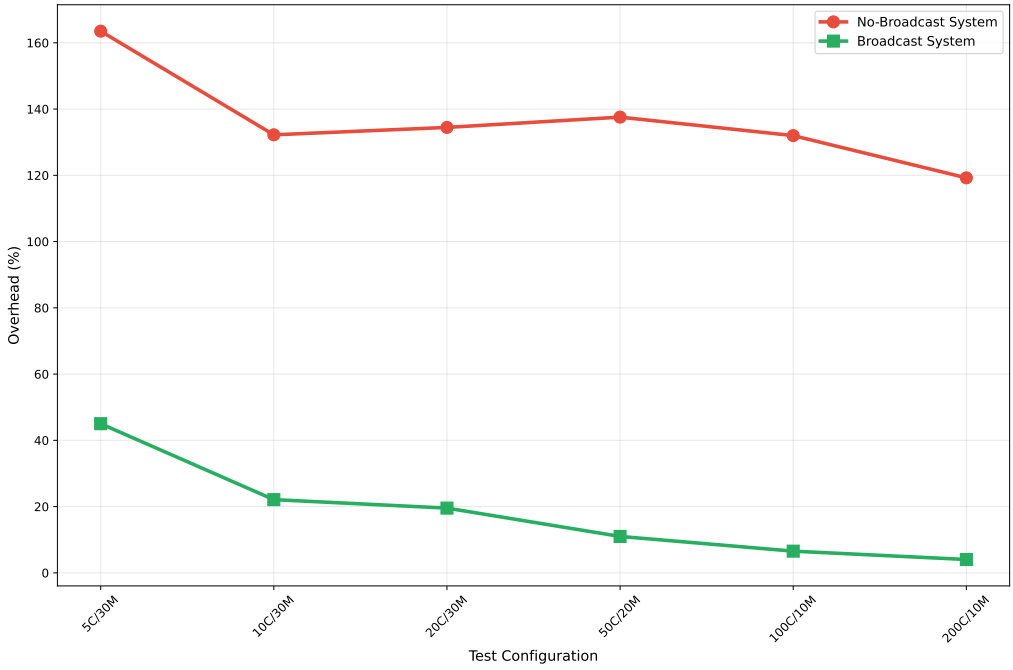


Figure 5.9: Instrumentation Overhead: Amortization Analysis

Here, the additional work associated with broadcasting messages effectively spreads out the monitoring cost of `ACTORCHESTRA`, demonstrating that in real-world concurrent

systems, the impact of instrumentation can be mitigated. This efficiency gain is further facilitated by the fact that broadcasts are handled via the `cast` OTP directive, a "fire-and-forget" operation that bypasses the `conductor`, avoiding extra routing overhead. The effect of this design choice is illustrated in Figure 5.9.

This is also observable in the throughput analysis, as in this broadcast version the system suffers in the throughput, since the work per message escalates with the client count, making the system saturated with the broadcast work, as visible in Figure 5.10 and Table 5.7. Throughput collapses because each message triggers massive coordination, compared to is counter part, the no broadcast version, where the system can leverage from concurrency better at scale due to the non-existent coordination overhead between clients.

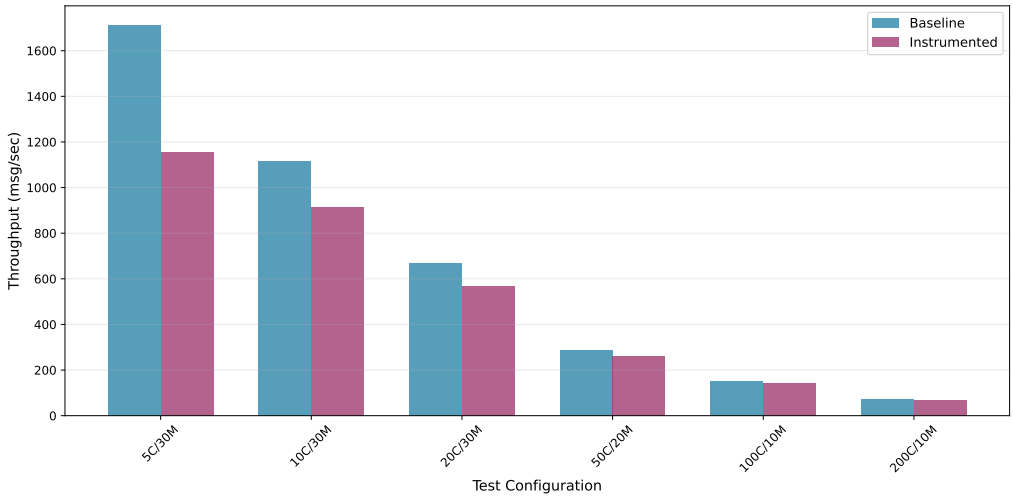


Figure 5.10: Broadcast System: Throughput Comparison

Clients	Msg/Client	Total Msg	Base Thpt (msg/s)	Instr. Thpt (msg/s)	Δ Thpt (%)
5	30	150.0	1711.1	1156.5	32.4
10	30	300.0	1116.7	913.4	18.2
20	30	600.0	670.0	566.6	15.5
50	20	1000.0	287.7	262.4	8.8
100	10	1000.0	149.1	140.8	5.6
200	10	2000.0	71.4	68.7	3.7

Table 5.7: Broadcast System: Throughput Table. Δ Thpt (%) = degradation from baseline

These results reveal an important insight: as the workload grows, the cost of broadcasting messages dominates system performance, gradually overshadowing the overhead introduced by instrumentation. In practice, this means both versions of the system converge in runtime behaviour, since broadcasting is the primary bottleneck. This also accounts for the observed throughput trends, where scaling introduces a natural degradation effect as message dissemination becomes increasingly expensive.

5.4 Conclusions

These performance characteristics must be contextualized within the intended use case of our instrumentation framework. The primary value proposition lies not in production deployment, but in development-time verification where runtime property checking provides immediate feedback on system correctness.

During development, engineers routinely tolerate substantial performance penalties from debugging tools, profilers, and testing frameworks in exchange for deeper insight into system behaviour. In this context, our measured overhead of 100–150% is far from prohibitive, particularly given the unique advantage it offers: the ability to detect causal property violations at runtime. Without such monitoring, these issues might remain hidden until late in deployment or emerge as subtle, hard-to-diagnose bugs in production.

The introduction of an intermediary routing entity fundamentally changes `Erlang`'s native message passing semantics. Instead of direct process-to-process communication, all messages must traverse the instrumentation layer, introducing additional copying and routing decisions. Although runtime property verification introduces some instrumentation overhead, the benefits far outweigh the costs. It enables early detection by catching issues during development rather than letting them surface as production failures. It also provides rich causal context, recording the full sequence of events that led to a violation so developers can understand not just what went wrong but why. Verification happens automatically, eliminating the need to manually construct exhaustive test cases for every execution path. As a result, developers gain greater confidence that their distributed systems preserve the desired properties, even in complex and highly concurrent scenarios.

More importantly, in the broadcast version of the chat system the overhead decreases drastically, ranging from 45%-4%, due to the amortized cost of the underlying operations that a system performs. The conclusion we can take from this is the overhead impact of our instrumentation depends not only on the workload applied to the system, but also the whole system architecture itself, where in some systems the instrumentation cost is what makes a bigger impact into the system performance, where in others it gets amortized by the true work load of the system. This makes our framework suitable in real world distributed systems, where we can mask the potential degradation that the instrumentation implies.

The same instrumentation framework exhibits dramatically different overhead profiles when applies to architectures with different computational complexities per operation. This demonstrates that architecture-workload interaction is a critical factor in the differences we have seen in the time executions, where lightweight systems show poor instrumentation amortization, yet still acceptable for the development and debugging phase, while in more complex distributed systems, with asynchronous background work, it shows a better amortization.

`ACTORCHESTRA` is not a framework to be compared to profiling tools that do not impact the system in such a way. It is a framework that is used in the production and debugging

phases of development where speed and latency are not as important as in production. The tool provides complete causality tracking between messages, and property verification at runtime, catching violations of properties that other traditional methods might miss. It opens the door to formal verification in distributed system resolving around the actor-based model.

Our results also reveal clear avenues for optimization to reduce overhead. The throughput limitations observed in the chat system indicate bottlenecks within the monitoring pipeline, which could be alleviated through approaches such as decentralized conductors or distributed worker pools. Furthermore, the relatively stable overhead across varying client counts demonstrates the framework's inherent architectural scalability, highlighting significant potential for further refinement and performance improvements.

Rather than serving as a general-purpose commercial monitoring solution, our framework is intended to complement manual testing and post-hoc debugging. In distributed systems, where diagnosing subtle bugs can consume days of investigation, the additional overhead incurred during controlled verification phases is a worthwhile trade-off. The key consideration is not the overhead itself, but the value of detecting causality violations early in development. For many systems, this benefit is substantial. In summary, the framework should be viewed as a specialized runtime verification tool, tailored for development and testing scenarios, providing guarantees and insights that conventional debugging and production monitoring alone cannot deliver.

CHAPTER 6

RELATED WORK

This chapter positions `WALTZ` and `ACTORCHESTRA` within the broader runtime verification landscape, highlighting how our context-aware approach overcomes fundamental limitations in monitoring concurrent actor systems. We evaluate existing specification languages and tools along three key dimensions: their capacity to process execution traces obtained from real running systems, their support for context-aware verification, and their mechanisms for correlating variables across interacting actors.

6.1 Runtime Verification Landscape

Runtime verification tools can be classified along several critical dimensions that determine their suitability for actor-based systems. One axis is the monitoring approach: some tools operate on live execution traces directly from a running system, while others analyse pre-defined logs of actions, even if those logs originate from real executions. Another dimension is concurrency support, which ranges from tools that assume sequential execution to those capable of context-aware verification that accounts for asynchronous message passing and causal relationships. A third dimension concerns property scope, spanning from monitoring single components in isolation to verifying system-wide invariants that involve multiple interacting actors.

Most existing RV approaches struggle when applied to actor-based systems because they either presuppose sequential execution or restrict verification to individual components. Actor systems, however, are inherently concurrent, built on asynchronous communication and causal message relationships that may extend across multiple hops in an interaction chain. The remainder of this chapter surveys existing languages and tools along these dimensions, setting the stage for our novel approach, which enables practical runtime verification of context-aware properties in concurrent actor-based systems.

6.2 Specification Languages

This section provides an overview of various specification languages introduced by the RV community, highlighting their characteristics and applications.

6.2.1 Linear Temporal Logics

Linear Temporal Logic (LTL) [67] and its extensions express system properties through temporal operators over global state sequences. While powerful in sequential contexts, these approaches face fundamental challenges when applied to actor systems, as discussed in Section 3.1.

LTL$_4$ [42] introduces the idea of revocable verdicts. In standard runtime monitoring, verdicts must be absolute, once produced, they cannot change. LTL$_4$ extends the verdict domain from three values ($\top$, $\bot$, "?") to four: true ($\top$), false ($\bot$), potentially true ($\top_c$), and potentially false ($\bot_c$). This allows monitors to capture a broader range of properties, though at the cost of sometimes never reaching a definitive conclusion. The guarantees remain limited to soundness [2], making LTL$_4$ more of a semantic refinement than a fundamental solution to the limitations of temporal logics.

Metric Temporal Logic (MTL) augments LTL with explicit timing constraints [53]. For example, in the running example from Chapter 2, we could define:

$$\Box\,(enter \rightarrow (\Diamond_{(0,20)}\; post))$$

Here, $\Diamond_{(0,20)}$ specifies that an event must occur within 20 time units. While this enables reasoning about deadlines and bounded response times, features absent in basic LTL, the reliance on global timing assumptions limits its applicability in distributed actor systems, where message delays vary unpredictably.

Metric First-Order Temporal Logic (MFOTL) [13] extends MTL by adding data parameters and first-order quantifiers, significantly increasing expressiveness. Tools such as MonPoly [14] and WHYMON [61] demonstrate MFOTL's practical utility. The monitorable fragment of MFOTL, formally defined in [13], imposes restrictions on quantifier usage while still allowing a wide range of properties to be monitored at runtime. For instance, in the running example, we could write:

$$\Box \forall x \forall y : entered(x, y) \rightarrow \Diamond_{(0,20)}\; post(x)$$

This property expresses that in every room, if a person x entered it, then x must post within 20 time units. Despite this expressive power, MFOTL inherits temporal semantics that assume perfect global synchronization, an unrealistic requirement in distributed systems. Alternatives like Distributed Temporal Logic (DTL) attempt to address this but assume sequential execution with synchronous event sharing [12].

In summary, while LTL and its extensions (LTL$_4$, MTL, MFOTL) expand the expressiveness and monitorability of temporal specifications, they remain grounded in assumptions

that do not align with the realities of actor-based distributed systems. Global time, sequential execution, and perfect synchronization are either impractical or outright impossible in asynchronous environments where causality is determined by message passing. As a result, these logics struggle to capture the fine-grained, context-dependent properties that arise in concurrent actor interactions. This motivates the need for the approaches such as `WALTZ` and `ACTORCHESTRA`, which explicitly embrace actor semantics and context propagation as first-class principles in their design and verification.

6.2.2 μHML and mHML

Efforts to create a specification language have been presented in various studies [18, 7], allowing for the definition of properties that can be monitored in a runtime context. Recent research has also explored monitors for specification languages not based on LTL, such as the modal μ-calculus with a linear-time interpretation. Additionally, a recent study has focused on Hennessy-Milner logic with recursion, known as μHML [1, 2], which is a reformulation of the μ-calculus. Instead of redefining the semantics of a specification logic to adhere to the necessities of RV like it was performed for LTL_3, the objective is to create a specification language with the sole purpose of runtime verification.

Certain properties expressed in μHML are not inherently monitorable. To address this limitation, Francalanza [44] analysed the monitorable fragment of μHML and identified a subset of the logic that ensures monitorability. Building on this insight, he introduced a specification logic called mHML, which guarantees that any property expressed within it can be monitored at runtime. The formal definition of this logic is provided in Definition 6.2.1.

Definition 6.2.1 (mHML [44]). $\psi, \chi \in \text{MHML}^{\text{def}} = \text{sHML} \cup \text{cHML}$ where:

$$\theta, \vartheta \in \text{sHML} ::= \text{tt} \mid \text{ff} \mid [\alpha]\theta \mid \theta \wedge \vartheta \mid \max X.\theta \mid X$$

$$\pi, \varpi \in \text{cHML} ::= \text{tt} \mid \text{ff} \mid \langle\alpha\rangle\pi \mid \pi \vee \varpi \mid \min X.\pi \mid X$$

where sHML is the safe property fragment and the cHML is the co-safe fragment of the logic. An important note is that we can only choose one or another, not both, at the same time to define monitorable properties with this specification language. An example of a property that can be defined with this specification language is stating that, for instance, our system cannot perform action ϕ, then the formula would be $[\phi]\texttt{ff}$.

While mHML aligns more closely with process semantics, it faces two major limitations in the context of actor-based systems. First, it lacks mechanisms for context-aware verification, preventing monitors from distinguishing between independent causal chains. Second, it requires the explicit specification of all possible interleavings, a burden that quickly becomes impractical in real-world interaction scenarios where concurrency patterns multiply rapidly.

These specification languages, while sophisticated, rest on assumptions that limit their applicability to actor-based systems: either a reliance on global state evolution or the need for explicit enumeration of all possible interleavings. In contrast, actor-based systems demand specification frameworks that can span multiple locality levels, capturing local actor state, actor-to-actor relationships, and system-wide invariants, while natively handling causal contexts and asynchronous interactions.

6.3 RV Tools

In this section, we present and examine several tools developed within the field of RV. These tools provide support for specifying system properties, monitoring executions, and analyzing system behaviour, enabling the early detection of deviations from expected properties and improving the reliability of concurrent and distributed systems.

6.3.1 Log Examination Tools

MonPoly [14] is a runtime verification tool built with `OCaml` [56] that analyses traces and properties defined in metric first-order temporal logic (MFOTL) [13]. Not all MFOTL properties are monitorable. Therefore, a monitorable safety fragment was defined for this language [13], which is used in MonPoly. This specification language is helpful since it can carry data on it, adding more expressiveness to the language, coming at the cost that only a fragment of it is monitorable [13]. A later version of MonPoly was released that supports full MFOTL but comes at the cost of substantial efficiency problems [69].

It is worth mentioning that monitors used in MonPoly compute sets of satisfying variable assignments instead of simple boolean verdicts, which is helpful information compared to simply knowing if a given property was violated; it is possible to know what the assignment of values that caused that violation. A notable limitation of MonPoly lies in the manual definition of traces. Although the tool's algorithm processes traces incrementally, the traces themselves must be read from pre-generated log files, which adhere to a grammar specific to that type of log file. Extracting traces from a running system would be more interesting than reading them from a log file that was not produced by a real running system. It does function in an online environment, but the trace is read from a log file that is generated based on a grammar and defined rules.

WHYMON [61] takes a similar approach to MonPoly, working with the same specification language (MFOTL) but presenting differences towards the MonPoly tool. To begin with, WHYMON uses full MFOTL, as it has no restrictions on negation and universal quantifiers, allowing arbitrary free variables, and supports past and future temporal operators. A novel feature of this tool is that the output value given by the monitor is richer; more specifically, the outputs of the monitors in WHYMON are partitioned decision trees that resemble MTL's semantics. These trees are used to explain why a formula is satisfied or violated, enhancing explainability in the verdicts by outputting explanations. The

tool provides a much more detailed explanation for every satisfying and violating assignment [62]. However, even if this tool can be applied online, the traces must be manually created by the user using their grammar, similar to how MonPoly does it. But, it also examines real-world traces besides the synthetic examples they provide. The tool is more advanced than MonPoly because it offers explainable verdicts, and we can incrementally examine the trace to observe those verdicts and the assignments bound to the variables in the formulas and the incoming trace. Compared to MonPoly, WHYMON takes much more time to perform the same analysis [62], but we must consider the different outputs that each monitor does; the WHYMON monitor's outputs carry much more information and explainability compared to the MonPoly ones.

Lima et al. [61] assume sequential, totally ordered event streams, processing events one at a time in timestamp order. Their approach works best for centralized systems with sequential logs, or distributed systems where events can be accurately timestamped and merged without losing causal information. While limited in concurrency handling, the tool greatly advances the state of the art by providing richer and more explainable verdicts.

6.4 RV in Actor-Based Models

Ian Cassar and Adrian Francalanza [28] outlined a spectrum of instrumentation techniques for online monitoring that can be integrated into runtime verification tools. This spectrum spans from fully asynchronous monitors, which may suffer from delayed detection of violations, to tightly coupled monitors, which provide fine-grained control over system behaviour but introduce significant overhead due to the required synchrony. Additionally, subsequent work examined the application of these instrumentation strategies specifically within the actor-based model [24].

6.4.1 Elarva

`ELARVA` [34] extends `LARVA` [35], a Java-based runtime verification tool that relies on Dynamic Automata with Timers and Events (DATE) for specification. While `LARVA` operates synchronously and lacks support for concurrency or distribution, `ELARVA` adapts the approach to the actor model, performing online monitoring in a fully asynchronous fashion. It leverages `Erlang`'s tracing mechanisms to observe communications, but suffers from late detection and limited fault-prevention capabilities. The crucial distinction is scope: `ELARVA` restricts properties to single `Erlang` components [47], whereas `ACTORCHESTRA` targets inter-component interactions. Moreover, while `ELARVA` relies on DATEs for specification, WALTZ introduces a modal, action-driven logic that abstracts away interleaving complexity and evaluates properties directly within their causal contexts.

6.4.2 Multiparty Session Types

Fowler [22], building on Yoshida's work on multiparty session types [65], proposed `monitored-session-erlang`, a runtime verification framework for `Erlang` OTP applications. His approach shares our assumptions, since it also targets OTP-compliant `Erlang` programs. The methodology centers on multiparty session types, where protocols define sequences of typed interactions among multiple participants.

By leveraging multiparty session types, Fowler's work introduces a notion of causality within actor interactions through the internal use of a `sessionID` that relates all the events within the same session. The `monitored-session-erlang` framework validates message exchanges at runtime against predefined session type specifications that describe the expected participants, sequence, and structure of interactions. To ensure correct routing of messages to their corresponding monitors in multiparty sessions, the framework generates opaque values that act as conversation keys. While this mechanism supports multiple concurrent sessions, the responsibility for switching between and managing these sessions rests on the developer. Messages are verified by checking whether valid transitions exist from the current session state, with violations resulting in runtime exceptions. The framework is implemented as an `Erlang` library, built on top of `gen_server` behaviour and supervision hierarchies. Participating actors are required to implement specific directives for session initiation, joining conversations, and handling typed communications, ensuring structured and reliable interactions within the system.

Although conceptually related, Fowler's framework differs from `ACTORCHESTRA` in several key ways. It relies on session types to capture causality, whereas our approach uses context injection, avoiding the rigidity of protocol typing. Additionally, Fowler's framework requires manual insertion of session-management directives, while `ACTORCHESTRA` automates this via code injection, reducing developer effort and allowing greater flexibility for future extensions.

Most importantly, the expressiveness of `monitored-session-erlang` is limited compared to `ACTORCHESTRA`. It ensures protocol conformance, e.g, that actor A sends a message of type X to actor B, but cannot reason about message payloads or their relationships across interactions. In contrast, `ACTORCHESTRA` supports context-aware verification of both communication flow and data content, enabling richer properties beyond protocol compliance.

6.4.3 detectEr

`detectEr` [9] is a runtime verification tool for `Erlang` programs that leverages the language's native tracing mechanisms to capture execution events. It uses the monitorable fragment of μHML to define safety properties and automatically generates monitors that run online alongside the system. This capability allows `detectEr` to operate on real execution traces, in contrast to many earlier tools that relied on pre-recorded logs. The tool supports multiple instrumentation strategies, including inline and outline deployment, and recent extensions allow it to handle the maxHML^D fragment, a maximally expressive

syntactic subset of μHML with data [1]. The tool produces boolean verdicts depending on whether the property was violated or not, logging which interaction was observed to cause it.

`detectEr` represents an important step forward in bringing runtime verification to actor systems. At the same time, it has certain limitations when applied to modern actor-based applications. The tool primarily supports monitoring of individual processes in isolation, which makes it challenging to express properties that span multiple actors, such as system-wide invariants or multi-actor protocols. Moreover, although the underlying logic can, in principle, capture cross-actor properties, the implemented fragment focuses on single-process monitoring. As a result, events that are causally related across asynchronous chains cannot be directly correlated.

In practice, applying `detectEr` to scenarios involving multiple actors can be challenging, as it would require a synchronous model that does not align naturally with the asynchronous nature of `Erlang`'s actor system. Moreover, due to `Erlang`'s concurrency model, users must explicitly enumerate possible message interleavings. As the number of actors and concurrent interactions increases, this task quickly becomes complex and difficult to scale. As a result, `detectEr` is particularly well suited for monitoring local request–response interactions but less equipped to capture longer causal chains or variable correlations across multiple interactions in large-scale distributed systems.

`ACTORCHESTRA` overcomes these limitations through its context-aware approach. By embedding causal contexts directly into the language semantics, WALTZ allows properties to be defined not just over individual messages, but across entire chains of actor interactions. This enables the specification and verification of properties that span multiple actors and multiple interactions, capturing complex causal relationships that arise in real-world distributed systems. The system automatically manages these contexts, eliminating the need for manual interleaving specification and allowing messages within the same causal chain to be correlated seamlessly. These capabilities make it possible to perform intra-actor and context-aware verification in concurrent, asynchronous environments. `detectEr`'s process-centric does not directly address context-awareness, which limits its applicability. In contrast, `ACTORCHESTRA` enables precise monitoring of complex distributed properties that naturally span multiple actors and interactions, providing a level of insight and correctness assurance aligned with the realities of modern actor-based concurrency.

6.5 WALTZ and ACTORCHESTRA

The analysis above highlights a clear gap in the RV landscape: while existing approaches have advanced the field, they either lack full support for actor-specific concurrency or face constraints that reduce their effectiveness in real-world distributed systems. WALTZ and `ACTORCHESTRA` address these challenges through three core innovations.

Traditional runtime verification approaches struggle in actor systems because concurrency obscures causal relationships between messages. WALTZ overcomes this limitation

by embedding causal context directly into the language semantics, allowing the properties to track which events are truly related. This enables verification of properties that depend on causal chains, rather than just sequential order, making it possible to specify and check meaningful correctness conditions even in highly concurrent actor-based systems. Unlike process-centric tools such as `detectEr` or global-state approaches like MonPoly, `ACTORCHESTRA` explicitly models the causal relationships between messages, allowing monitors to reason about multi-actor interactions to verify properties.

Beyond context management, real-world actor systems often require verification of properties that span chains of interactions across multiple processes. Existing tools either restrict monitoring to single processes, as in `detectEr`, or rely on manual correlation mechanisms, such as multiparty session types. WALTZ overcomes these limitations by allowing natural specification of properties that encompass entire interaction chains. Causal relationships are automatically preserved, and variables can be correlated across different actors and interaction steps. This capability enables the expression of complex distributed protocols, such as end-to-end request guarantees or multi-actor coordination workflows, without requiring developers to manually track correlations.

Finally, practical deployment of runtime verification requires integration with live systems. Many sophisticated RV tools, such as MonPoly or WHYMON, operate on pre-recorded log files, restricting their usefulness to post-hoc analysis rather than proactive failure prevention. `ACTORCHESTRA` integrates directly with running actor systems, extracting traces in real-time, and providing immediate feedback. It automatically injects the necessary instrumentation code, minimizing developer overhead while enabling practical deployment in production environments. This combination of context-aware specification and real-time monitoring transforms runtime verification from a passive debugging tool into an active mechanism for maintaining system reliability.

The general comparison between our tool and others in the field is visible in Table 6.1.

Tool	Running trace	Distributed	Context-aware	Rich Verdicts
MonPoly	✘	✘	✘	✔
WHYMON	✘	✘	✘	✔
`detectEr`	✔	✔	✘	✔
ELARVA	✔	✔	✘	✘
Session Types[1]	✔	✔	✔	✘
`ACTORCHESTRA`	✔	✔	✔	✔

Table 6.1: Comparison of RV approaches

The runtime verification landscape shows a clear evolution from general temporal logics toward domain-specific approaches tailored to particular computational models. Nevertheless, existing tools struggle with the core challenges of monitoring concurrent

[1]Compared to our framework, Fowler [22]'s approach lacks automation and does not support properties that capture or reason about data relationships across interactions due to the nature of session types.

actor systems: context-aware evaluation, multi-actor property correlation, and real-time integration with running systems.

WALTZ advances the state of the art by embedding context-awareness directly into the specification language semantics, automatically handling the complexities that render verification of concurrent systems intractable with conventional methods. `ACTORCHESTRA` complements this with practical real-time monitoring capabilities, enabling deployment in production actor systems.

While tools such as `detectEr` pioneered actor-based runtime verification and proved its practical feasibility, WALTZ takes this further by introducing context-aware verification and supporting message correlation both within and across actors. This significantly broadens the spectrum of properties that can be expressed and checked, enabling the verification of richer and more realistic behaviours in modern distributed systems.

By combining domain-specific language design, context-aware semantics, and real-time system integration, WALTZ and `ACTORCHESTRA` open new possibilities for ensuring correctness in concurrent actor systems, capabilities that were previously either theoretically intractable or practically infeasible. `ACTORCHESTRA` functions as a passive, online, outlined, and asynchronous runtime verification tool, leveraging a centralized `conductor` along with decentralized monitors where needed. This combination enables scalable, context-sensitive verification that operates alongside the system without interfering with its execution, making it a practical and robust solution for detecting user defined property violations or satisfactions.

Chapter 7

Conclusions

This chapter finalizes the document with a summary of what this work tackled and providing future paths to improvement and extensions to the work.

7.1 Summary

We developed a framework for runtime verification of `Erlang` systems that adhere to OTP standards and follow client-server architectures. The effort involved studying the theoretical foundations of runtime verification, analysing existing tools, particularly those targeting actor-based systems, and identifying their capabilities and limitations. Building on this foundation, we developed three key components: a specification language for defining properties, a compiler that generates running `Erlang` monitors from these specifications, and a causality-tracking entity explicitly designed for client-server OTP systems.

WALTZ is a specification language designed explicitly for actor-based systems. It allows users to define properties over chains of interactions between different processes by capturing the messages deemed relevant. Built with the concurrency model of `Erlang` in mind, WALTZ evaluates properties not only over the observed trace but also within the causal context of each message. This context-awareness is essential for correctly verifying properties in concurrent systems, as it prevents the unintended mixing of messages from different causal chains. One of WALTZ's key strengths is its ability to specify properties that span multiple actor interactions, enabling users to define comprehensive, system-wide specifications or focus on specific interaction patterns. Equally important is its context-aware semantics, which are critical for verification in highly concurrent environments.

A central goal of our work was compiling WALTZ properties into running monitors. The compilation process leverages the close alignment between WALTZ's semantics and

the actor-based model. The generated monitors are scalable, tracking distinct message chains and evaluating them independently, producing verdicts upon detecting either a violation or satisfaction. These monitors operate alongside the system in a structured manner, without interfering with its execution, acting as oracles that receive only the relevant traces and evaluate them against the formally defined properties.

The `conductor` is the entity responsible for tracking all the interaction in the system, and assigning a causal reference, essentially the context, to each message that is observed. It is the primary entity that enables monitors to receive a trace with the context directly attached to the message payload, thereby avoiding concurrency problems on the monitor's side, as all important information is sent atomically. The `conductor` focuses on client–server applications built with the OTP standard, enabling context injection by providing a well-defined framework to build upon.

Together, these components form `ACTORCHESTRA`, a comprehensive runtime verification framework for verifying context-aware properties in actor-based systems, enabling the specification and evaluation of properties that span multiple interacting actors.

7.2 Future Work

There are several potential extensions and improvements to our work. For WALTZ, one such enhancement would be the ability to capture multiple events co-occurring as part of a property. Currently, the language supports only ordered chains of messages, but in some scenarios, it may be necessary to express properties involving concurrent events. Supporting this would require introducing a new operator to the language to handle these simultaneous interactions. Additionally, we can introduce logical conjunction and disjunction operators to increase the expressiveness of the language. Their inclusion requires the compilation of multiple monitors that coordinate and interact with one another, such as the handling of nested formulas.

Adding time to properties is also a path to follow, as with time, we will be able to escape the non-monitorability issue, since we are setting a beginning and an end to our properties. Another interesting extension would be the definition of properties that compare or relate two or more different contexts, essentially allowing inter-context property definitions to enrich our language. Exploring the notions of unbreakability of chains might also be an interesting topic, since `Erlang` is built around a fault-tolerant design, and sometimes it is normal for things to fail. In our case, we do not permit failures because of the non-breakable chains of messages. An interesting direction for future work would be to relax this constraint and define properties that tolerate failures, provided that the failed event eventually succeeds.

Regarding the `conductor` and `ACTORCHESTRA`, our current implementation targets a specific system architecture. As a result, certain types of systems, such as publish-subscribe architectures, do not fit within our existing assumptions. There is significant potential for improvement by investigating such systems and developing methods to

manage causality effectively within them. Given the wide variety of system architectures and behaviours, tracking and handling all of them imposes a great challenge. A standard reference is necessary, which is why we focused on the specific target system used in this work. A natural direction for future work is to extend `ACTORCHESTRA` to support multiple architectures, enabling it to manage causality across a broader range of system designs.

There is also room for improvement in the generated monitors, as specific properties, especially nested properties, could be enhanced to allow for more parallelism in verification and verdict notification. Not only improvement in the monitor organisation, but also the entire instrumentation pipeline, in order to reduce the overhead added to the monitored system.

Finally, given the structure of our specification language and the design of the generated monitors, extending the framework to include verdict explainability could significantly enhance the utility of the results. Such an extension would enable more profound insights into property violations and open avenues for research into automated blame assignment.

Bibliography

[1] L. Aceto et al. "Adventures in monitorability: from branching to linear time and back again". In: 3.POPL (2019-01). DOI: 10.1145/3290365 (cit. on pp. 110, 114).

[2] L. Aceto et al. "An Operational Guide to Monitorability". In: *Software Engineering and Formal Methods*. Ed. by P. C. Ölveczky and G. Salaün. Cham: Springer International Publishing, 2019, pp. 433–453. ISBN: 978-3-030-30446-1 (cit. on pp. 20, 109, 110).

[3] L. Aceto et al. "On Runtime Enforcement via Suppressions". en. In: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. DOI: 10.4230/LIPICS.CONCUR.2018.34 (cit. on p. 27).

[4] G. A. Agha et al. "Actors: a model for reasoning about open distributed systems". In: *Formal methods for distributed processing: a survey of object-oriented approaches* (2001), pp. 155–176 (cit. on pp. 3, 28).

[5] D. Albuquerque et al. "Quantifying usability of domain-specific languages: An empirical study on software maintenance". In: *J. Syst. Softw.* 101 (2015), pp. 245–259. DOI: 10.1016/J.JSS.2014.11.051 (cit. on p. 35).

[6] R. Amjad, R. van Glabbeek, and L. O'Connor. "Semantics for Linear-time Temporal Logic with Finite Observations". In: *Electronic Proceedings in Theoretical Computer Science* 412 (2024-11), 35–50. ISSN: 2075-2180. DOI: 10.4204/eptcs.412.4 (cit. on p. 16).

[7] D. Ancona et al. "RML: Theory and practice of a domain specific language for runtime verification". In: *Science of Computer Programming* 205 (2021), p. 102610. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102610 (cit. on p. 110).

[8] J. Armstrong. *Programming Erlang : Software for a Concurrent World*. Raleigh, NC : The Pragmatic Bookshelf, 2013. ISBN: 9781680504330. URL: http://digital.casalini.it/9781680504330 (cit. on pp. 4, 28, 30, 33, 86).

[9] D. P. Attard et al. "A runtime monitoring tool for actor-based systems". In: *Behavioural Types: from Theory to Tools* (2017), pp. 49–76 (cit. on pp. 26, 113).

[10] E. Barrett et al. "Virtual machine warmup blows hot and cold". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017-10). DOI: 10.1145/3133876 (cit. on p. 96).

[11] E. Bartocci et al. "Introduction to Runtime Verification". In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by E. Bartocci and Y. Falcone. Cham: Springer International Publishing, 2018, pp. 1–33. ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_1 (cit. on pp. 2, 6–11, 13–15, 17, 22, 23, 25–28).

[12] D. Basin et al. "Distributed temporal logic for the analysis of security protocol models". In: *Theoretical Computer Science* 412.31 (2011), pp. 4007–4043 (cit. on p. 109).

[13] D. Basin et al. "Monitoring Metric First-Order Temporal Properties". In: *J. ACM* 62.2 (2015-05). ISSN: 0004-5411. DOI: 10.1145/2699444 (cit. on pp. 109, 111).

[14] D. A. Basin, F. Klaedtke, and E. Zalinescu. "The MonPoly Monitoring Tool." In: *RV-CuBES* 3 (2017), pp. 19–28 (cit. on pp. 109, 111).

[15] A. Bauer, M. Leucker, and C. Schallhart. "Comparing LTL Semantics for Runtime Verification". In: *Journal of Logic and Computation* 20.3 (2010-02), pp. 651–674. ISSN: 0955-792X. DOI: 10.1093/logcom/exn075 (cit. on p. 16).

[16] A. Bauer, M. Leucker, and C. Schallhart. "Monitoring of Real-Time Properties". In: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*. Ed. by S. Arun-Kumar and N. Garg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 260–272. ISBN: 978-3-540-49995-4 (cit. on p. 17).

[17] A. Bauer, M. Leucker, and C. Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (2011-09). ISSN: 1049-331X. DOI: 10.1145/2000799.2000800 (cit. on pp. 2, 6, 11, 12, 14–18, 22, 26).

[18] A. Bauer, M. Leucker, and C. Schallhart. "The Good, the Bad, and the Ugly, But How Ugly Is Ugly?" In: *Runtime Verification*. Ed. by O. Sokolsky and S. Taşıran. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 126–138. ISBN: 978-3-540-77395-5 (cit. on pp. 19, 110).

[19] S. Berkovich, B. Bonakdarpour, and S. Fischmeister. "Runtime verification with minimal intrusion through parallelism". In: *Formal Methods Syst. Des.* 46.3 (2015), pp. 317–348. DOI: 10.1007/S10703-015-0226-3 (cit. on p. 26).

[20] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013 (cit. on p. 1).

[21] N. Bielova and F. Massacci. "Do You Really Mean What You Actually Enforced?" In: *Formal Aspects in Security and Trust*. Ed. by P. Degano, J. Guttman, and F. Martinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 287–301. ISBN: 978-3-642-01465-9 (cit. on p. 23).

[22] M. A. L. Brun, S. Fowler, and O. Dardha. "Multiparty Session Types with a Bang!" In: *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*. Ed. by V. Vafeiadis. Vol. 15695. Lecture Notes in Computer Science. Springer, 2025, pp. 125–153. DOI: 10.1007/978-3-031-91121-7_6 (cit. on pp. 113, 115).

[23] I. Cassar and A. Francalanza. "On Implementing a Monitor-Oriented Programming Framework for Actor Systems". In: *Integrated Formal Methods*. Ed. by E. Ábrahám and M. Huisman. Cham: Springer International Publishing, 2016, pp. 176–192. ISBN: 978-3-319-33693-0 (cit. on p. 23).

[24] I. Cassar and A. Francalanza. "On Synchronous and Asynchronous Monitor Instrumentation for Actor-based systems". In: *Electronic Proceedings in Theoretical Computer Science* 175 (2015-02), 54–68. ISSN: 2075-2180. DOI: 10.4204/eptcs.175.4 (cit. on pp. 25, 112).

[25] I. Cassar and A. Francalanza. "Runtime Adaptation for Actor Systems". In: *Runtime Verification*. Ed. by E. Bartocci and R. Majumdar. Cham: Springer International Publishing, 2015, pp. 38–54. ISBN: 978-3-319-23820-3 (cit. on p. 27).

[26] I. Cassar, A. Francalanza, and S. Said. "Improving Runtime Overheads for detectEr". In: *Electronic Proceedings in Theoretical Computer Science* 178 (2015-03), 1–8. ISSN: 2075-2180. DOI: 10.4204/eptcs.178.1 (cit. on p. 25).

[27] I. Cassar et al. "A Suite of Monitoring Tools for Erlang". In: *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*. Ed. by G. Reger and K. Havelund. Vol. 3. Kalpa Publications in Computing. EasyChair, 2017, pp. 41–47. DOI: 10.29007/7lrd (cit. on p. 25).

[28] I. Cassar et al. "A Survey of Runtime Monitoring Instrumentation Techniques". In: *Electronic Proceedings in Theoretical Computer Science* 254 (2017-08), 15–28. ISSN: 2075-2180. DOI: 10.4204/eptcs.254.2 (cit. on pp. 23, 26, 112).

[29] F. Cesarini and S. Thompson. *Erlang Programming - A Concurrent Approach to Software Development*. O'Reilly, 2009. ISBN: 978-0-596-51818-9. URL: http://www.oreilly.de/catalog/9780596518189/index.html (cit. on pp. 4, 30, 33, 86).

[30] F. Chen and G. Roşu. "Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation". In: *Electronic Notes in Theoretical Computer Science* 89.2 (2003). RV '2003, Run-time Verification (Satellite Workshop of CAV '03), pp. 108–127. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)81045-4 (cit. on p. 23).

[31] Z. Chen et al. "Deciding weak monitorability for runtime verification". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 163–164. ISBN: 9781450356633. DOI: 10.1145/3183440.3195077 (cit. on p. 21).

[32] E. M. Clarke and J. M. Wing. "Formal methods: state of the art and future directions". In: *ACM Comput. Surv.* 28.4 (1996-12), 626–643. ISSN: 0360-0300. DOI: 10.1145/242223.242257 (cit. on p. 1).

[33] E. Clarke et al. "State space reduction using partial order reduction". In: *International Journal on Software Tools for Technology Transfer* 2 (1999-01), pp. 279–287. DOI: 10.1007/s100090050035 (cit. on p. 1).

[34] C. Colombo, A. Francalanza, and R. Gatt. "Elarva: A Monitoring Tool for Erlang". In: *Runtime Verification*. Ed. by S. Khurshid and K. Sen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 370–374. ISBN: 978-3-642-29860-8 (cit. on p. 112).

[35] C. Colombo, G. Pace, and G. Schneider. "Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties". In: vol. 5596. 2008-09, pp. 135–149. ISBN: 978-3-642-03239-4. DOI: 10.1007/978-3-642-03240-0_13 (cit. on p. 112).

[36] M. d'Amorim and G. Roşu. "Efficient Monitoring of ω-Languages". In: *Computer Aided Verification*. Ed. by K. Etessami and S. K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 364–378. ISBN: 978-3-540-31686-2 (cit. on p. 18).

[37] A. C. Dias Neto et al. "A survey on model-based testing approaches: a systematic review". In: *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. WEASELTech '07. Atlanta, Georgia: Association for Computing Machinery, 2007, 31–36. ISBN: 9781595938800. DOI: 10.1145/1353673.1353681 (cit. on p. 1).

[38] Elixir Core Team. *Elixir*. Version v1.18. URL: https://elixir-lang.org/ (cit. on pp. 3, 28).

[39] U. Erlingsson. "The inlined reference monitor approach to security policy enforcement". AAI3114521. PhD thesis. USA, 2004 (cit. on p. 26).

[40] Y. Falcone. "You Should Better Enforce Than Verify". In: *Runtime Verification*. Ed. by H. Barringer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 89–105. ISBN: 978-3-642-16612-9 (cit. on p. 27).

[41] Y. Falcone et al. "A taxonomy for classifying runtime verification tools". In: *Int. J. Softw. Tools Technol. Transf.* 23.2 (2021-04), 255–284. ISSN: 1433-2779. DOI: 10.1007/s10009-021-00609-z (cit. on pp. 2, 3, 7, 10, 11, 23, 26, 28).

[42] Y. Falcone, J.-C. Fernandez, and L. Mounier. "What can You Verify and Enforce at Runtime?" In: *International Journal on Software Tools for Technology Transfer* 14 (2011-06). DOI: 10.1007/s10009-011-0196-8 (cit. on pp. 2, 6, 24, 27, 109).

[43] A. Ferrando and R. C. Cardoso. "Towards partial monitoring: Never too early to give in". In: *Science of Computer Programming* 240 (2025), p. 103220. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2024.103220 (cit. on pp. 7, 17, 20, 21, 70).

[44] A. Francalanza, L. Aceto, and A. Ingolfsdottir. "On Verifying Hennessy-Milner Logic with Recursion at Runtime". In: *Runtime Verification*. Ed. by E. Bartocci and R. Majumdar. Cham: Springer International Publishing, 2015, pp. 71–86. ISBN: 978-3-319-23820-3 (cit. on p. 110).

[45] A. Francalanza, A. Gauci, and G. J. Pace. "Distributed system contract monitoring". In: *The Journal of Logic and Algebraic Programming* 82.5 (2013). Formal Languages and Analysis of Contract-Oriented Software (FLACOS'11), pp. 186–215. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2013.04.001 (cit. on p. 27).

[46] A. Francalanza, J. A. Pérez, and C. Sánchez. "Runtime Verification for Decentralised and Distributed Systems". In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by E. Bartocci and Y. Falcone. Cham: Springer International Publishing, 2018, pp. 176–210. DOI: 10.1007/978-3-319-75632-5_6 (cit. on p. 25).

[47] I. Galea. "polyLARVA plugin for Erlang". PhD thesis. University of Malta, 2013 (cit. on p. 112).

[48] R. Gerth et al. "Simple On-the-fly Automatic Verification of Linear Temporal Logic". In: *Protocol Specification, Testing and Verification XV: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*. Ed. by P. Dembiński and M. Średniawa. Boston, MA: Springer US, 1996, pp. 3–18. ISBN: 978-0-387-34892-6. DOI: 10.1007/978-0-387-34892-6_1 (cit. on p. 18).

[49] K. Guan and O. Legunsen. "An In-Depth Study of Runtime Verification Overheads during Software Testing". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, 1798–1810. ISBN: 9798400706127. DOI: 10.1145/3650212.3680400 (cit. on p. 25).

[50] F. Hebert. *Learn you some Erlang for great good!: a beginner's guide*. No Starch Press, 2013 (cit. on pp. 4, 30).

[51] D. Jin et al. "JavaMOP: Efficient parametric runtime monitoring framework". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 1427–1430. DOI: 10.1109/ICSE.2012.6227231 (cit. on pp. 23, 26).

[52] J. A. W. Kamp. *Tense logic and the theory of linear order*. University of California, Los Angeles, 1968 (cit. on p. 16).

[53] R. Koymans. "Specifying real-time properties with metric temporal logic". In: *Real-time systems* 2.4 (1990), pp. 255–299 (cit. on p. 109).

[54] O. Kupferman and M. Vardi. "Model Checking of Safety Properties". In: *Computer Aided Verification*. Ed. by N. Halbwachs and D. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 172–183. ISBN: 978-3-540-48683-1 (cit. on p. 22).

[55] C. Lemieux, D. Park, and I. Beschastnikh. "General LTL Specification Mining (T)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 81–92. DOI: 10.1109/ASE.2015.71 (cit. on p. 10).

[56] X. Leroy et al. "The OCaml system: Documentation and user's manual". In: *INRIA* 3 (), p. 42 (cit. on pp. 56, 111).

[57] M. Leucker. "Teaching Runtime Verification". In: *Runtime Verification*. Ed. by S. Khurshid and K. Sen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 34–48. ISBN: 978-3-642-29860-8 (cit. on pp. 7, 14).

[58] M. Leucker and C. Schallhart. "A brief account of runtime verification". In: *The Journal of Logic and Algebraic Programming* 78.5 (2009). The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), pp. 293–303. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2008.08.004 (cit. on pp. 2, 6, 7, 9, 17, 19, 22, 24).

[59] L. Libkin, T. Tan, and D. Vrgoč. "Regular expressions for data words". In: *Journal of Computer and System Sciences* 81.7 (2015), pp. 1278–1297. ISSN: 0022-0000. DOI: https://doi.org/10.1016/j.jcss.2015.03.005 (cit. on p. 15).

[60] J. Ligatti, L. Bauer, and D. Walker. "Edit automata: Enforcement mechanisms for run-time security policies". In: *International Journal of Information Security* 4 (2005), pp. 2–16 (cit. on p. 27).

[61] L. Lima, J. Huerta y Munive, and D. Traytel. *Artifact for "WhyMon: A Runtime Monitoring Tool with Explanations as Verdicts"*. Version 0.1.5. 2024-08. DOI: 10.5281/zenodo.13361497 (cit. on pp. 109, 111, 112).

[62] L. Lima, J. J. Huerta y Munive, and D. Traytel. "Explainable Online Monitoring of Metric First-Order Temporal Logic". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by B. Finkbeiner and L. Kovács. Cham: Springer Nature Switzerland, 2024, pp. 288–307. ISBN: 978-3-031-57246-3 (cit. on p. 112).

[63] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[64] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962 (cit. on p. 1).

[65] R. Neykova and N. Yoshida. "Multiparty Session Actors". In: *Log. Methods Comput. Sci.* 13.1 (2017). DOI: 10.23638/LMCS-13(1:17)2017 (cit. on p. 113).

[66] A. Pnueli and A. Zaks. "PSL Model Checking and Run-Time Verification Via Testers". In: *FM 2006: Formal Methods*. Ed. by J. Misra, T. Nipkow, and E. Sekerinski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 573–586. ISBN: 978-3-540-37216-5 (cit. on pp. 2, 20, 22, 70).

[67] A. Pnueli. "The temporal logic of programs". In: *18th annual symposium on foundations of computer science (sfcs 1977)*. ieee. 1977, pp. 46–57 (cit. on pp. 2, 10, 12, 15, 109).

[68] J. Richard Büchi. "Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic". In: *Logic, Methodology and Philosophy of Science*. Ed. by E. Nagel, P. Suppes, and A. Tarski. Vol. 44. Studies in Logic and the Foundations of Mathematics. Elsevier, 1966, pp. 1–11. DOI: https://doi.org/10.1016/S0049-237X(09)70564-6 (cit. on p. 14).

[69] J. Schneider et al. "A Formally Verified Monitor for Metric First-Order Temporal Logic". In: *Runtime Verification*. Ed. by B. Finkbeiner and L. Mariani. Cham: Springer International Publishing, 2019, pp. 310–328. ISBN: 978-3-030-32079-9 (cit. on p. 111).

[70] F. Somenzi and R. Bloem. "Efficient Büchi Automata from LTL Formulae". In: *Computer Aided Verification*. Ed. by E. A. Emerson and A. P. Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 248–263. ISBN: 978-3-540-45047-4 (cit. on p. 14).

[71] "Timed regular expressions". In: *J. ACM* 49.2 (2002-03), 172–206. ISSN: 0004-5411. DOI: 10.1145/506147.506151 (cit. on p. 15).

[72] M. Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Automatic Program Verification". English. In: *Proceedings of the First Symposium on Logic in Computer Science*. Cambridge, United States - Massachusetts: IEEE Computer Society, 1986 (cit. on p. 14).

[73] R. Virding et al. *Concurrent programming in ERLANG (2nd ed.)* GBR: Prentice Hall International (UK) Ltd., 1996. ISBN: 013508301X (cit. on pp. 3, 28).

[74] M. Viswanathan and M. Kim. "Foundations for the Run-Time Monitoring of Reactive Systems – Fundamentals of the MaC Language". In: *Theoretical Aspects of Computing - ICTAC 2004*. Ed. by Z. Liu and K. Araki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 543–556. ISBN: 978-3-540-31862-0 (cit. on pp. 20, 22).