Duncan Paul Attard

Runtime Monitoring for Asynchronous Reactive Components

Supervised by Adrian Francalanza, Luca Aceto, Anna Ingólfsdóttir

Submitted in partial fulfilment of the requirements for the degree of PhD in Computer Science

University of Malta and Reykjavik University · *February 8, 2024*

ISBN 978-9935-539-28-1 · Print version
ISBN 978-9935-539-29-8 · Electronic version

0000-0002-2448-5394 · Duncan Paul Attard

Declaration of Authenticity

I, the undersigned, declare that the dissertation titled

Runtime Monitoring for Asynchronous Reactive Components

is my work, except where acknowledged and referenced.

Duncan Paul Attard · *February 8, 2024*

Acknowledgements

First and foremost, I want to express my deepest gratitude to my supervisors, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. I have been singularly blessed to have worked with these three very remarkable people. Adrian, Luca, and Anna were present every step of the way, provided me with unwavering guidance and inspired me throughout these past years. They instilled in me a deep appreciation of the scientific pursuit not just through words but through their action, while at the same time, reminding me that there is more to life than just work. The three of them—each in their own manner—went above and beyond in nurturing and shaping my growth as a researcher. Luca, Anna, and especially, Adrian, thanks for being a steadfast source of moral support whenever my spirits flagged. I remain in your debt and aspire to follow in your footsteps. Thanks also to Antonis Achilleos, a friend and colleague who was always available to discuss ideas and give feedback during my time in Iceland.

My second thanks goes to my examiners, Emilio Tuosto, David Basin, and Keith Bugeja, for their constructive assessment and insightful comments, and for making my viva a very pleasant experience. Keith has been an invaluable reference when discussing implementation ideas and has been supportive throughout these past years. Emilio, whom I initially met in Leicester in the very first months of my programme, has always given me frank and constructive feedback whenever we chanced to meet at conferences or other academic settings. He is a person I respect and greatly admire. Another thanks goes to Simon Fowler and Phil Trinder for their suggestions on improving the presentation of my work and for helping me prepare for my viva.

This journey would have not been the same without the warm bonds of friendship. I count Adrian De Barro as one of the closest, together with Jasmine Xuereb and Elli Anastasiadi. Adrian has been a constant companion through the highs and lows of my academic and personal life. We spent countless nights working together until the wee hours of the morning. I cannot help but fear that I would have not met most of my deadlines had it not been for your honest encouragement and solidarity. Thank you, Jasmine, for not only being an incredible friend but also for generously dedicating your valuable time to proofread sections of this manuscript.

I extend my thanks to the inhabitants of our postgraduate office at Reykjavik University, including Joshua, Shalini, and Majd, in addition to colleagues from the University of Malta: Matthew, Gerard, Caroline, Chris, and Stefania, who kept me company from afar on numerous occasions. Thanks also to Andy, Mandy, Roderick, Josef, and Josef for checking in on me from time to time.

My parents, Raymond and Geltrude, as well as my siblings Daphne and Björn, have been a boundless source of motivation. They have endured my fair share of complaints and moments of despondency, which they repaid me tenfold with gentle words of care and encouragement. I feel incredibly fortunate to have you in my life. This PhD is yours as much as it is mine.

These past years have been a wonderful journey of personal growth and self-discovery. Pursuing a doctorate was always a vague notion that I entertained, albeit not one I believed in. Never have I thought that mid-life I could gather sufficient courage to quit my job, forgo a stable lifestyle, and embark on an undertaking about which I knew little and feared much. Reflecting on it, I suppose I would not have mustered the courage to take the leap. But I met Joe Cordina, a cherished friend from the past, whom I lastly thank. He urged me to follow my heart, rather than dwell on whether the path one travels in consequence leads to success or failure. For life is too short not to risk living it.

My research was supported by the Reykjavik University Research Fund, the Doctoral Student Grant (No: 207055) under the Icelandic Research Fund, and the ENDEAVOUR Scholarship Scheme (Group B - National Funds).

If you can dream—and not make dreams your master;
If you can think—and not make thoughts your aim;
If you can meet with Triumph and Disaster
And treat those two impostors just the same;

If you can fill the unforgiving minute
With sixty seconds' worth of distance run,
Yours is the Earth and everything that's in it,
And—which is more—you'll be a Man, my son!

(excerpt from *If*, by Rudyard Kipling)

Publications

These papers have been published as a result of the research work conducted in this thesis, and are used as its main contributing source. The list of publications is presented in reverse chronological order and includes a short description identifying my contributions.

- Luca Aceto, Antonis Achilleos, **Duncan Paul Attard**, Léo Exibard, Adrian Francalanza, and Anna Ingólfsdóttir. A Monitoring Tool for Linear-Time μHML. *Sci. Comput. Program.*, 232:103031, 2024

  Contribution Principal author of the paper and developer of the software artefact. This paper is the journal version of the one cited next.

- Luca Aceto, Antonis Achilleos, **Duncan Paul Attard**, Léo Exibard, Adrian Francalanza, and Anna Ingólfsdóttir. A Monitoring Tool for Linear-Time μHML. In *COORDINATION*, volume 13271 of *LNCS*, pages 200–219, 2022

  Contribution Principal author of the paper and developer of the software artefact and the accompanying tutorial. Section 2 of the paper forms a significant part of section 2.2 and chapter 3. Section 3 also contributes to chapter 3, whereas the implementation presented in section 4 of the paper provides the material for sections 4.1, 4.2, 4.5 and 4.6.

- Luca Aceto, **Duncan Paul Attard**, Adrian Francalanza, and Anna Ingólfsdóttir. On Benchmarking for Concurrent Runtime Verification. In *FASE*, volume 12649 of *LNCS*, pages 3–23, 2021

  Contribution Principal author of the paper and developer of the software artefact. This work has been conducted under the advice of my supervisors. Most of the material in the paper is integrated in chapter 6.

- Luca Aceto, **Duncan Paul Attard**, Adrian Francalanza, and Anna Ingólfsdóttir. A Choreographed Outline Instrumentation Algorithm for Asynchronous Components. Technical report, Reykjavik University, IS, 2021

  Contribution Principal author of the technical report and developer of the software artefact. This work has been conducted under the advice of my supervisors. The content of the technical report, apart from the empirical results section, forms the basis of chapter 5. Parts of the argumentation in chapter 7 is modelled on section 4 of the technical report, but the evaluation we present has been conducted again on newer hardware and extended further to a new direction.

- **Duncan Paul Attard**, Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Better Late than Never or: Verifying Asynchronous Components at Runtime. In

FORTE, volume 12719 of *LNCS*, pages 207–225, 2021

Contribution Principal author of the paper and developer of the software artefact, website and tutorial material. This work has been conducted under the advice of my supervisors. Some of the material in sections 4, 5 and 6 in the paper is integrated in chapter 4.

- Adrian Francalanza, Luca Aceto, Antonis Achilleos, **Duncan Paul Attard**, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A Foundation for Runtime Monitoring. In *RV*, volume 10548 of *LNCS*, pages 8–29, 2017

  Contribution Helped with writing sections of the paper, illustration of all the diagrams as well as poof reading. Parts of sections 3 and 4 in the paper are included in chapters 2 and 3.

- Ian Cassar, Adrian Francalanza, **Duncan Paul Attard**, Luca Aceto, and Anna Ingólfsdóttir. A Suite of Monitoring Tools for Erlang. In *RV-CuBES*, volume 3 of *Kalpa Publications in Computing*, pages 41–47, 2017

  Contribution Helped with writing sections of the paper as well as proofreading. This work has been conducted under the advice of my supervisors. None of this work is included in this thesis.

- **Duncan Paul Attard**, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Introduction to Runtime Verification. In *Behavioural Types: from Theory to Tools*, Automation, Control and Robotics, pages 49–76. River, 2017

  Contribution Principal author of the book chapter and developer of the software artefact and tutorial material. This work has been conducted under the advice of my supervisors. Small parts of section 1.1 in the book chapter are included in chapters 2 and 3, whereas section 1.2 contributes minimally to chapter 4.

- **Duncan Paul Attard** and Adrian Francalanza. Trace Partitioning and Local Monitoring for Asynchronous Components. In *SEFM*, volume 10469 of *LNCS*, pages 219–235, 2017

  Contribution Co-author with my supervisor and principal developer of the software artefact and tutorial material. This work has been conducted under the advice of my supervisors. Some of the material of section 2 in the paper contributes to chapters 2 and 3. Ideas in sections 3 and 4 of the paper have also been lifted and adapted to chapter 4.

Abstract

Modern software is built on reactive principles, where systems are responsive, resilient, elastic, and message-driven. Despite the benefits they engender, these aspects make the correctness of reactive systems in terms of their expected behaviour hard to ascertain statically. This thesis investigates how the correctness of reactive systems can be ascertained dynamically at runtime. It considers a lightweight monitoring technique, called runtime verification, that circumvents the issues associated with traditional pre-deployment techniques. One core challenge of runtime verification lies in choosing a monitoring approach that does not impinge on the reactive aspects of the system under scrutiny. Such a goal is met only if the monitoring system is itself reactive. We propose a novel monitoring approach grounded on this precept. It treats the system as a black box, instrumenting monitors dynamically and in an asynchronous fashion, which is in tune with the requirements of reactive architectures. Our development approach is systematic, mapping directly the constituent parts of our formal model to implementable modules. This gives assurances that the results obtained in the theory are preserved in the implementation.

The first part of the thesis builds on established theoretical results. It lifts these results to a first-order setting to accommodate scenarios where systems manipulate data. We define an asynchronous instrumentation relation that decouples the operation of the system from that of its monitors. This definition forms the basis of our decentralised outline monitoring algorithm presented in the second part of the thesis. Our algorithm employs a tracing infrastructure to collect trace events as the system executes and uses key events as cues to instrument new monitors or terminate redundant ones dynamically. It accounts for the interleaving of events that arises from the asynchronous execution of the system and monitors, guaranteeing that events are analysed by monitors in the correct sequence and without gaps.

Part three develops a runtime verification benchmarking framework that is tailored for reactive systems. The framework can generate models that faithfully capture the realistic behaviour of master-worker systems under typical load characteristics. Our tool collects different performance metrics suited to reactive applications, to give a multi-faceted depiction of the overhead induced by runtime monitoring tools. Part four of this thesis embarks on an extensive evaluation of our decentralised outline monitoring algorithm using the benchmarking tool developed in part three. The algorithm is compared against our implementation of inline and centralised monitoring—two prevalent methods used in state-of-the-art runtime verification tools. Apart from demonstrating that our monitoring algorithm is reactive, the experiments we conduct testify that it induces acceptable overhead that, in typical cases, is comparable to that of inlining. These results also confirm that centralised monitoring is prone to scalability issues, poor performance, and failure, making it generally inapplicable to reactive system settings. We are unaware of other comprehensive empirical runtime verification studies such as ours that compare decentralised, centralised, and inline monitoring.

Contents

Figures

Tables

Listings

Conventions and Notation

Textual content and illustrations in this thesis adopt the following conventions. Shadows are used to highlight illustration elements that are important in the surrounding context.

Text

Emphasised text denotes key concepts, phrases, and term definitions

Small Capitals identify process, function, or set names in mathematical notation

`Teletype` text identifies source code snippets or keywords

Sans Serif denotes functions or values in mathematical notation

‘*Quoted*’ italic text symbolises the textual description of correctness properties

Illustrations

$\overset{\curvearrowright}{x\;x}$ variable binding and scoping

ⓝ sequential steps in a figure or formula

⋯▷ (fork) creation of a child process

⋯∗ (exit) process termination

⟶ (send) uni-directional communication between processes

⤳ (trace) pairing between the process and the monitor tracing it

⇢ (read or write) read or write from or to queue

$\boxed{P}$ process or groups of processes of the SuS

$\boxed{T}$ tracer process

$\boxed{M}$ outlined monitor process

M inlined monitor code

$\boxed{e}$ trace event

✓ monitor verdict

⬚ process abstraction or system boundary

$\boxed{2}$ arbitrarily long queue of objects

ⓟ process state

1 Introduction

Modern software applications are architected in terms of *concurrent* components that execute independently to one another without recourse to a global clock or shared state [15, 140]. Instead, components interact together and with their environment via *non-blocking* messaging [136] to create a dynamic, loosely-coupled software organisation known as a *reactive system* [2, 153]. Reactive systems must:

- respond in a timely manner (be *responsive*),
- remain available in the face of failure (be *resilient*),
- grow and shrink to accommodate variable computational loads (be *elastic*), and
- react to inputs from users or their environment (be *message-driven*).

Such architectures facilitate incremental updates (*maintainability*) and permit the various constituent components to execute on different locations (*distribution*) [153, 83, 120]. At the same time, the benefits of reactive systems make the correctness in terms of their expected behaviour hard to verify statically [119].

This thesis investigates how the correctness of reactive systems can be established at runtime. We consider runtime verification (RV), which is a dynamic technique that checks the current execution of a system under scrutiny (SuS) to determine whether it satisfies or violates some correctness *property*. RV uses *monitors*—computational machines that are synthesised from formal property descriptions. Monitors are *instrumented* with the SuS to incrementally analyse its execution (expressed as a trace of events) and reach *verdicts* about its observed behaviour. We make the following contributions.

(i) Build on previous theoretical results [6, 8] and extend their specification language, monitor operational model, and monitor synthesis procedure with predicates to reason on the data carried by trace events. We implement these extensions and give a technique for instrumenting inline monitors. Additionally, we define an asynchronous instrumentation relation that decouples the operation of the SuS from that of its monitors, in line with a reactive approach.

(ii) Devise a decentralised outline monitoring algorithm that realises the asynchronous instrumentation definition of (i). Our algorithm accounts for the interleaving arising from asynchronous execution and guarantees that trace events are reported to monitors in the correct order and without loss.

(iii) Develop a configurable benchmarking framework that can generate synthetic SuS models which reproduce the realistic behaviour of master-worker systems. This tool collects various performance metrics to give a multi-faceted view of overhead that is relevant to reactive runtime monitoring.

(iv) Give a comprehensive empirical evaluation of the overhead induced by the instantiation of the formalisation developed in contribution (i) as the algorithm in (ii), using the benchmarking framework of (iii). We compare (ii) against our implementations of inline and centralised instrumentation—also based on contribution (ii)—to demonstrate that our decentralised approach induces feasible overhead that, in typical cases, is proportionate to, or outperforms the latter methods.

1.1 Motivation and Contributions Summary

Our ultimate research goal is to construct a suite of runtime monitoring tools for reactive systems founded on the contributions (i) to (iv). We use these tools as a vehicle to:

- demonstrate that the formalisation and method proposed in contributions (i) and (ii) can be implemented in a general-purpose language that targets reactive applications (chapters 4 and 5);
- debunk the commonly-held belief [90, 25] that decentralised outline instrumentation is necessarily infeasible (section 7.2) and show that in typical cases, inline and outline instrumentation induce comparable runtime overhead (section 7.3);
- confirm that centralised monitoring approaches are generally inapplicable in settings exhibiting moderate to high concurrency, and are prone to poor performance or failure (section 7.2).

Based on these conclusions, we immediately note that decentralised outline monitoring is the only viable approach when inlining cannot be employed (refer to discussion in section 2.1.4). Sections 1.1.1 to 1.1.4 respectively detail the research gaps that each of contributions (i) to (iv) addresses.

1.1.1 Asynchronous Runtime Monitoring with Data

RV approaches that are not equipped to handle data explicitly have very limited applicability in practice. For instance, the property stating *'always greater than zero'*, is easily expressed as the linear temporal logic (LTL) formula $\mathrm{G}\ 1 \vee \mathrm{G}\ 2$, when the set of actions that a SuS can exhibit is $\{0, 1, 2\}$. However, a generalisation of this requirement to the domain of integers cannot be expressed in a finite way. Equipping the specification logic with a predicate over data values and variables enables us to compactly represent this requirement using the formula $\mathrm{G}\,(x > 0)$. The same reasoning can be extended to monitors that runtime check such specifications against system executions.

Our work follows this route. It builds on the theoretical results of Aceto et al. [6, 8] that use the linear-time interpretation of the Hennessy-Milner logic with recursion (μHML), a highly-expressive modal logic that can encode other logics such as LTL. This gives our work a sufficiently-general basis. In *op. cit.*, the authors define an operational model of regular monitors and a compositional synthesis procedure that generates monitors from *monitorable* fragments of the logic. We lift their results and extend the logic, monitors, and synthesis procedure with predicates over data. One challenge that arises upon introducing data predicates is that of variable *binding* and *scoping*, that gives rise to subtle dependencies between sub-formulae and complicates their runtime checking. We address this aspect from two angles. First, our synthesis procedure generates parallel monitors whose constituent sub-monitors runtime check different sub-formulae and can reach independent verdicts. Second, the executable monitor code generated delegates the binding and scoping aspects to the implementation language to streamline the synthesis. In addition to augmenting the model of Aceto et al. [6, 8] with data predicates, we provide an alternative *asynchronous* instrumentation definition to the synchronous one given by the aforesaid authors. Our definition is preferable in reactive systems settings since the SuS and monitors can be organised into independent components. Separating the SuS and monitors minimises the dependencies between these entities and the risk that the system is impacted by the operation of monitors.

1.1.2 Decentralised Outline Monitor Instrumentation

We claim that reactive applications necessitate a RV monitoring set-up that is *itself* reactive and, crucially, does not impinge on any of the reactive characteristics of the SuS. One of the main challenges in constructing RV tools lies in choosing an instrumentation technique that suits the architecture of SuS one wants monitored. Intuitively, instrumentation can be seen as a procedure $\triangleleft$ that takes a SuS and its monitors, and composes them together as a *monitored system*, which we denote by

$$\mathit{Monitors} \triangleleft \mathit{SuS}$$

State-of-the-art approaches that focus on monolithic programs generally prefer synchronous instrumentation in the form of monitor *inlining* (see section 2.1.4 for details), since the targeted systems are typically single-threaded and do not scale (*e.g.* [197, 70, 68, 175, 24, 148, 138]). Numerous other works that consider decentralised or distributed systems and use synchronous or asynchronous instrumentation methods assume a *static* SuS whose number of components is known and remains fixed at runtime (*e.g.* [31, 45, 67, 122, 180, 203, 208, 219]). Observe that in both cases described, the SuS is not reactive as it is neither resilient (single-threaded) nor elastic (static).

The RV approaches that *do* support dynamic systems mostly adopt inline instrumentation. Inlining remains the predominant method used in decentralised and distributed RV (*e.g.* [60, 148, 89, 87, 34, 45, 110, 13]). One possible reason behind this is that most efforts extend mature tools that were originally conceived for monolithic RV, where inlining has traditionally performed well. It is, therefore, natural to want to extend this proven approach to a new domain such as decentralised monitoring, rather than abandon the prior implementation investment in favour of a completely new approach. However, inlining creates a tight dependency between the SuS and its monitors. This dependency is known to hamper the responsiveness of the SuS when the inlined monitors are slow in their runtime analysis [61, 51]; it can also impinge on the resiliency of the system when monitors suffer from faults or failures. For these reasons, we view inline instrumentation as producing a monitored system *i.e., $\mathit{Monitors} \triangleleft \mathit{SuS}$*, that might not be reactive.

Centralised monitoring is an approach occasionally adopted when inlining cannot be administered to the SuS (see section 2.1.4 for reasons). In a centralised set-up, it is often the case that a singleton monitor is instrumented to execute apart of the reactive SuS via *outlining*. Trace events exhibited by different components of the SuS are directed to a central collection point, such as a *queue*, that the monitor then accesses to analyse these events (*e.g.* [71, 21, 219, 113, 51, 52, 207, 101]). While the serialisation of events on the centralised monitor may facilitate the runtime analysis, it creates contention and sacrifices the scalability of the system. This means that a centralised monitoring set-up can experience diminishing returns as new computational resources are introduced [18]. Moreover, the reliance on one monitoring entity makes centralised set-ups susceptible to single point of failures (SPOFs) [153, 152]. We hold that these two shortcomings (evidence of both is given in our empirical investigation of chapter 7) renders the monitored system, $\mathit{Monitor} \triangleleft \mathit{Queue} \triangleleft \mathit{SuS}$, not reactive.

We propose an algorithm that dynamically instruments decentralised outline monitors as the SuS executes. The asynchronous instrumentation definition we give as part of the contribution outlined in section 1.1.1 is used as the basis of our decentralised algorithm. The algorithm generalises the configuration $\mathit{Monitor} \triangleleft \mathit{Queue} \triangleleft \mathit{SuS}$ to different SuS components, where each is organised with a *separate* monitor and trace event message queue:

$$(\mathit{Monitor} \triangleleft \mathit{Queue} \triangleleft \mathit{Component})_i$$

To the best of our knowledge, this approach is novel. In fact, the latest taxonomy of RV tools in Falcone et al. [100, Tables 3 and 4] shows that *none* of the works it catalogues use outlining combined with decentralisation[1]. Another recent classification for decentralised and distributed monitoring in Francalanza et al. [119, Tables 1 and 2] also indicates that the approach we propose remains unexplored[2]. One rationale why outlining is seldom considered for decentralised RV arises from its perceived infeasibly high overhead when compared to inlining. This is partly because inlining statically identifies the designated monitor instrumentation points within the SuS, whereas outlining defers this decision post-deployment. The perception about high overheads is reinforced when the overhead in decentralised RV is gauged in terms of criteria that are applicable to monolithic, batch-style systems (*e.g.* percentage slowdown) that are hardly relevant to reactive settings (see *e.g.* [158, 184, 185, 62, 61, 197, 43]). This lack of proper RV benchmarking tools for reactive systems motivates our third contribution of section 1.1.3.

However, the foremost reason for the scarce adoption of decentralised outline instrumentation is that reactive systems impose onerous terms that make it *hard* to build. Chief among these requirements is the capacity for a reactive system to grow and shrink in response to fluctuating computational demands, obliging the RV set-up to scale accordingly. With the use of inlining, such elastic behaviour emerges naturally as a byproduct of the monitor logic that is weaved into the components of the reactive system itself. By contrast, elasticity must be explicitly engineered in the decentralised outline case so that the instrumentation can reconfigure its monitoring set-up while the runtime analysis is underway. Decoupling the SuS from its monitors calls for the instrumentation to contend with the inherent race conditions (*e.g.* message reordering) that arise from the asynchronous execution of the SuS and monitors. As section 2.1.4 later stresses, instrumentation that is tailored for verification purposes must ensure that the trace events collected from the SuS are reported to the correct monitors in the proper order *and* with no loss, lest this invalidates the runtime analysis [25]. The lock-step execution of the weaved system-monitor components spares inline monitoring these complications. Despite the challenges that decentralised outline instrumentation poses, the monitored system that results from this set-up *is* reactive (refer to section 5.4).

1.1.3 Quantifying Runtime Overhead Reliably

The overhead induced by monitors is a manifestation of the formal framework that underpins the RV model *and* the implementation effort that instantiates it as a concrete software artefact. Runtime overhead is the litmus test that determines whether a monitoring tool is applicable in practice [25]. Benchmarking is a commonly-accepted practice of gauging runtime overhead in software [165] which is also adopted by the RV community [25, 119]. The usefulness of benchmarking tools rests on two aspects, namely, (i) the *coverage* of scenarios of interest, and (ii) the *quality* of runtime metrics collected by the benchmark harness [108]. Benchmarking tools (*e.g.* [215, 40, 212, 193]) generally employ third-party off-the-shelf (OTS) programs to capture scenarios of interest. OTS software is appealing, as it inherently

[1] While THEMIS [88] and StateRover [85] are marked as decentralised outline approaches in [100], both are simulation tools.

[2] The authors use the label 'Distributed Monitoring', but this refers to *concurrent* monitors on the same machine.

provides realistic scenarios and can be readily integrated within an existing benchmarking suite. In a bid to broaden and diversify the coverage of real-world scenarios, benchmarking tools rely on a range of OTS programs (*e.g.* DaCapo [40] uses 11 open-source libraries, Renaissance [193] uses 21). Yet, using such programs as benchmarks poses certain challenges. By design, OTS programs do not expose *hooks* that enable harnesses to easily and accurately gather the runtime metrics of interest. When OTS software is treated as a black box, benchmarks become harder to control, impacting their ability to produce repeatable results. OTS software-based benchmarks are also limited when inducing specific edge cases—this aspect is critical when assessing the safety of software, such as runtime monitors, that are often assumed to be *dependable* [25, 112]. Custom-built *synthetic programs* (*e.g.* Savina [137]) are an alternative way to perform benchmarking [46]. These tend to be less popular due to the perceived drawbacks associated with developing such programs from scratch and the lack of 'real-world' behaviour intrinsic to benchmarks based on OTS software. However, synthetic benchmarks offer benefits that offset these drawbacks. For example, specialised hooks can be built into the synthetic set-up to collect specific runtime metrics. Moreover, synthetic benchmarks can also be *parametrised* to emulate variations on the same core benchmark behaviour; this is usually harder to achieve via OTS programs that, often, implement very specific use cases.

Established benchmarking frameworks such as SPECjvm2008 [215], DaCapo [40], ScalaBench [212] and Savina [137]—developed for the Java virtual machine (JVM)—have been adopted by the RV community as the benchmarking tools of choice, *e.g.* see [185, 62, 61, 197, 43, 176, 124]. Apart from [176], the cited works assess the runtime overhead solely in terms of the *execution slowdown*, *i.e.,* the difference in running time between the system fitted with and without monitors. While this metric is suited to batch-style monolithic programs [68, 100], it is *inapplicable* to the reactive setting, where systems are engineered to *not* terminate. The *response time* (or *latency*) between communicating components is one of the fundamental aspects that quantifies the quality of a reactive system [153]. Concretely, it reflects the *responsiveness* from a client standpoint (*e.g.* interactive apps) [187, 217, 211, 73]; in the broader sense, it indicates the *service degradation* that one should manage to ensure adequate quality of service [49, 151]. The first competition on runtime verification (CRV) [26] advocates for the *memory consumption* as another measure that gives a more complete view of runtime overhead. However, the CRV disregards the *scheduler* (or CPU) utilisation that, for component-based applications, indicates how well the tool being benchmarked maximises the capacity of the processing elements provided by the host platform.

Arguably, benchmarking tools like the ones above (*e.g.* Savina) should provide even more. RV set-ups for reactive systems need to scale in response to dynamic changes, and the capacity for a benchmark to emulate *high loads* cannot be overstated. In practice, these loads assume characteristic *profiles* (*e.g.* spikes or uniform rates), which are hard to administer with the benchmarking tools mentioned earlier. The state of the art in benchmarking for concurrent RV suffers from another core issue. At one end, existing benchmarking tools are *repurposed* for RV, but are not made to account for concurrent scenarios where RV is realistically put to use. For instance, SPECjvm2008, DaCapo, and ScalaBench lack workloads that leverage the JVM concurrency primitives [193]; meanwhile, Blessing et al. [41] show that the Savina microbenchmarks are essentially sequential and that the rest of the programs in the suite are sufficiently simple to be regarded as microbenchmarks, too. This makes it challenging to generalise the results obtained from experiments based on these benchmarks. At the other end, the RV-centric CRV suite mostly targets *monolithic* software with limited concurrency, where the potential for scaling to high loads is, therefore, severely curbed. Its recent editions [98, 198, 27] acknowledge that concurrency remains uncatered for.

In the absence of a suitable solution that provides for reactive systems, we propose a synthetic benchmarking framework that addresses the deficiencies described above. The framework records three performance metrics—response time, memory consumption, and scheduler utilisation—that give a comprehensive depiction of runtime overhead. Our tool is configurable. It can generate different benchmarking models of master-worker systems based on various parameters and subject these models to load profiles that are typically observed in practice. Despite the synthetic nature of the tool, the models it generates capture the realistic behaviour of software which is conducive to reliably quantifying overhead. This improves the likelihood that conclusions drawn from the synthetic experiments are portable to real-world applications of the evaluated RV tool.

1.1.4 Evaluating Decentralised Outline Runtime Monitoring

The benchmarking tool developed in section 1.1.3 is used to empirically assess the three monitor instrumentation techniques, inline, outline decentralised, and outline centralised, mentioned in section 1.1.2. Our experiment set-up is extensive. It considers two configurations to model edge-case scenarios based on limited hardware, and general-case scenarios using modern hardware. We subject the three instrumentation algorithms to high loads that go beyond the state of the art and use realistic load profiles that, to wit, are not considered in the literature.

This empirical study shows that our decentralised instrumentation algorithm is, in fact, reactive, and does not impinge on the reactive characteristics of the SuS. It further deems the overhead our algorithm induces feasible for soft real-time applications [149]. We also certify that the known shortcomings of centralised architectures (see discussion in section 1.1.2) apply to our RV setting, too, where (i) the exhaustion of system resources leads the set-up to crash in the edge-case scenario due to its SPOF, and (ii) the central monitor does not avail of the ample hardware capacity provided by the general-case scenario. We are unaware of other comprehensive empirical RV studies such as ours that compare decentralised, centralised and inline monitoring.

1.2 Scope of the Study

We adopt the actor model of computation [133, 15] to conduct our scientific study. The actor model provides a simple, yet powerful paradigm to design and implement systems that follow the reactive principles introduced on page 1. *Actors*—the basic unit of decomposition in this model—are abstractions of concurrent entities that do not share mutable memory with other actors. Instead, actors interact through *asynchronous messaging* and alter their internal state based on messages they consume. Each actor is equipped with an incoming message buffer called the *mailbox*, from where messages deposited by other actors may be *selectively* read. Besides sending and receiving messages, actors can fork other actors. Actors are uniquely identifiable via their dynamically-assigned process identifier (PID) that they use to directly address one another.

The actor model is instantiated by a number of languages and frameworks, including Erlang [19, 57], Elixir [142], Akka [199] for Java [169], Thespian [194] for Python [173], and Pony [218]. We choose Erlang as our implementation language since it is *specifically engineered* for high-concurrency, soft real-time applications. BEAM, the Erlang virtual machine (EVM) implements actors as isolated lightweight processes which enables the remarkable scalability and fault tolerance of Erlang applications. The EVM uses *per-process* garbage collection that—unlike JVM implementations—does not subject the entire virtual machine to non-deterministic pauses [139, 188]. This aspect is particularly crucial to our empirical experiments conducted for the contribution of section 1.1.4 because it helps to stabilise the variance in our measurements. Conveniently, the EVM provides a native tracing infrastructure which tames the technical challenges that arise when implementing decentralised outline monitoring (see section 1.1.2). The terms *actor* and *process* are used synonymously in Erlang-related literature, and we adopt the same nomenclature in the rest of this thesis.

The inherent concurrency of components in reactive applications gives rise to natural *partitions* in the global execution of the SuS in the form of isolated sub-traces for each component. Our decentralised instrumentation algorithm exploits this view to generate trace partitions. These partitions make it possible to conceive of the overall system correctness as a collection of *local properties* that describe the behaviour of independent components. Such an approach gives certain advantages. It allows one to be *selective* about the SuS components that require runtime checking, and to specify properties accordingly. A similar technique called parametric trace slicing (PTS) [62, 201] is used in monolithic RV where properties are often specified on objects, the unit of decomposition of OOP paradigms [138, 176, 197]; by contrast, we focus on concurrent components. Being selective about the components to verify means that local properties need *only* be concerned about the trace events related to the component under scrutiny. This simplifies the corresponding specifications. The notion of local properties can be leveraged to dynamically instrument component replicas with monitors, free from assumptions about the number of components the SuS is expected to have, making the RV set-up elastic. Besides, the set-up benefits from a modicum of resiliency since failure in a system component or its corresponding monitor does not imperil the execution or runtime analysis of analogous components.

This thesis focusses on *online* RV [100], where the analysis that runtime monitors conduct takes place whilst the SuS executes. In this setting, we scope our study to reactive systems where failures do not arise, *i.e.*, we assume no link or communication omission failures [83], and no fail-stop or Byzantine failures [157].

1.3 Outline

The body of this thesis is organised into six main chapters. Chapter 2 introduces the classical RV set-up that assumes a single execution. Our development follows the *modular* approach advocated by Aceto et al. [6, 8] that delineates the semantics of the specification logic and the semantics of the monitor operational model. The chapter overviews the notions of monitors, monitorability in terms of soundness and completeness, and monitor instrumentation in the context of reactive systems. We lift definitions of these concepts from *op. cit.* and restate them as templates; these are instantiated w.r.t. a concrete definition of the logic and monitor model in chapter 3. Chapter 2 concludes with an outline of the linear-time and branching-time interpretations of the μHML. The logic is augmented with *symbolic actions*, consisting of variables and predicates that enable the reasoning about data carried by process actions; we refer to these extensions as μHML^{D}. This thesis adopts the linear-time semantics of the μHML^{D}.

The third chapter builds on the principles of chapter 2. It reviews the linear-time μHML$^{\text{D}}$ that is used to describe properties about the *current* execution, and shows how properties concerning data can be flexibly specified. We borrow the operational model of monitors used by Aceto et al. [6, 8] and extend it with the symbolic actions of chapter 2. The logic and monitor model, together with the synchronous instrumentation relation specified in the cited work suffice to give concrete definitions of soundness, completeness, and monitorability. Based on these concrete definitions, we restate the minimal and maximal monitorable fragments of μHML$^{\text{D}}$ that Aceto et al. [6] show to be *maximally-expressive*. Chapter 3 also adapts the synthesis procedure given in the latter work for the case of regular monitors to generate monitors that handle data. We define an alternative instrumentation relation to the one in Aceto et al. [6] that composes the SuS and monitors asynchronously. This asynchronous definition lays the foundation for our decentralised outline instrumentation algorithm described in chapter 5

Chapter 4 revisits the symbolic actions of chapter 2 and generalises them by introducing *pattern matching*, enabling the logic and monitors to reason on composite data types (*e.g.* tuples, lists, *etc.*). We use tuples to define a simple model that describes the process events: *fork* (process creation), *initialise* (process initialisation), *exit* (process termination), *send*, and *receive*. This chapter concretises our synthesis procedure of chapter 3 to generate executable monitors—these use a subset of the Erlang syntax to delegate variable binding, scoping, and pattern matching to the language runtime. The monitoring algorithm that we give encodes the monitor operational semantics defined in chapter 3 and is used to evaluate synthesised monitors. One aspect that the instrumentation relations of chapter 3 leave unspecified is how processes of the SuS can be *selectively* instrumented. We generalise our instrumentation definitions to make use of the *instrumentation map* that identifies the processes to be monitored based on the signature of the function used to fork them. Chapter 4 also details an implementation of synchronous instrumentation that instruments monitors selectively. The procedure inlines monitors by manipulating the abstract syntax tree (AST) of Erlang programs via source-level weaving, which results in a modified program.

Decentralised outline instrumentation adopts a non-invasive approach that treats the SuS and its components as a black box. Outlining assumes a tracing infrastructure that collects events from the running system. By contrast to inlining, which instruments monitors statically, our algorithm of chapter 5 uses key events in the execution trace as cues to instrument monitors dynamically. Decoupling the SuS and monitors introduces complications that arise due to the interleaved execution of the system and monitors. The main part of chapter 5 is devoted to describing the methods we use to overcome these challenges. We elucidate how our algorithm instantiates the instrumentation definition of chapter 3 while ensuring that the events reported to monitors are in the correct order and with no loss. Chapter 5 discusses briefly how the algorithm we give is mappable to Erlang actors, followed by a series of precautions taken to ensure its correct operation. Our implementation is validated further via the comprehensive empirical study of chapter 7.

Chapter 6 proposes a benchmarking framework that targets RV tools built for reactive systems. The framework follows the master-worker model—an architecture that is pervasive in both distributed and concurrent systems. Our tool is configurable and can generate different synthetic master-worker models for high loads and under commonly-observed load profiles. The benchmarking environment gathers different metrics (see contribution (iii)) that give a multi-faceted view of runtime overhead. In spite of the synthetic models it generates, we empirically show that our tool can be tuned to approximate the realistic behaviour of web server traffic with high degrees of fidelity and repeatability. We showcase the efficacy of our benchmarking tool via a two-part case study. First, we use our inline monitoring tool of chapter 4 to demonstrate how the framework can induce edge-case scenarios. The second case-study confirms that the results obtained from our experiments with a real-world use-case set up with OTS software coincide with the ones obtained by the synthetic experiments.

Chapter 7 presents an comprehensive evaluation of three instrumentation approaches: (i) our decentralised outline algorithm of chapter 5, (ii) its different configuration for centralised monitoring, and (iii) the inlining approach developed in chapter 4. Through our extensive experiment set-up, we show that decentralised outline monitoring is reactive and that it induces feasible runtime overhead that makes it practicable in soft real-time settings. By contrast, our configuration with centralised monitoring crashed when the resources were scarce, and failed to scale properly when additional resources were made available. Chapter 7 makes other observations as a byproduct of our experiments, *e.g.* a considerable amount of the monitoring overhead is carried by the instrumentation. In particular, we remark that in cases where the SuS does not continually create and terminate processes, decentralised outline monitoring induces comparable overhead to inline monitoring.

The main contributions of this thesis are found in chapters 4 to 7. Our extensions to the logic, monitor operational semantics and synthesis procedure of Aceto et al. [6] in chapters 2 and 3 are vehicles supporting the work in the aforementioned chapters; the definition of the asynchronous instrumentation, meanwhile, formalises part of the ideas of chapter 5.

1.3.1 How to Read this Thesis

Readers familiar with the fundamentals of RV may skip chapter 2 on first reading. Chapter 3 introduces the notions that chapter 4 and the initial part of chapter 5 build upon. Chapter 5 lists the pseudocode of our decentralised outline instrumentation algorithm, accompanied by the challenges that arise and the steps taken to address them. The material is technical and readers may find table 5.2 helpful to navigate through the sections in this chapter. Chapter 6 can be fully understood independently of the other chapters, and is likewise technical. Chapter 7 makes frequent references to the configuration parameters offered by our benchmarking framework of chapter 6. A summary of these parameters is provided in table 6.1 for convenience. Whilst discussing the results, chapter 7 mentions certain specifics of the algorithms developed in chapters 4 and 5. It is therefore advisable to embark on chapter 7 only after having read these chapters. Table 7.1 lists the set-ups used in our experiments, whereas table 7.3 summarises our claims and the outcomes expected from each experiment. Readers may find it helpful to consult these table when reading chapter 7. Supporting material for the algorithm of chapter 5 is provided in appendix A; additional results for chapter 7 may be found in appendix C. While reading these appendices is not necessary to understanding the work in the main text, one may benefit from skimming this content.

2 Preliminaries

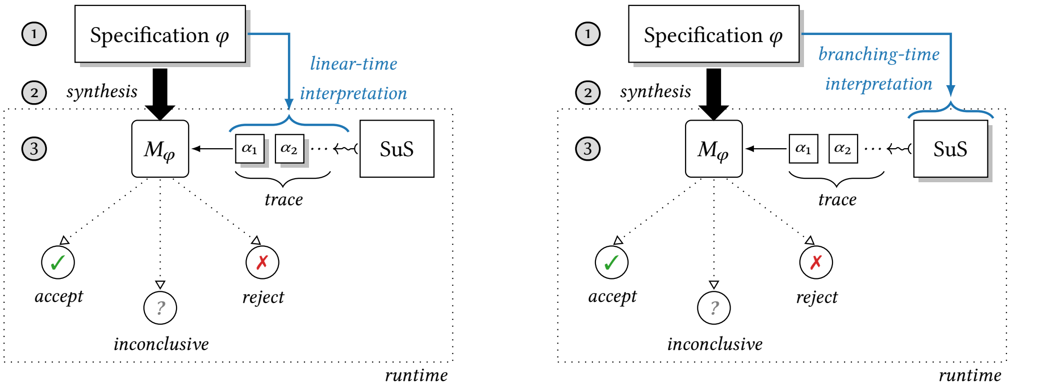
There are three key aspects to RV: the specification formalism used to express properties, the monitors that conduct the runtime checking, and the instrumentation that composes monitors with the SuS. These aspects are linked by the notion of monitorability that identifies what expressible properties can be runtime checked. This chapter adopts the modular approach advocated by Aceto et al. [6, 8], which delineates the semantics of the specification formalism, and the verdicts that monitors flag as a result of their runtime analysis. Following *op. cit.*, we regard monitors as machines that (i) analyse finite trace prefixes, and (ii) reach irrevocable verdicts, that once given, cannot be retracted. The unified monitorability definition of Aceto et al. [8] for the finite and infinite trace domain uses the notions of soundness and completeness which are based on two predicates that determine whether monitors accept or reject traces. We adapt these definitions to include the branching-time setting where specifications describe the execution graphs of processes [118, 6]. Our definitions are given as templates—they lay the foundation for chapter 3 where we instantiate them w.r.t. a concrete operational model of monitors that adheres to the requirements (i) and (ii) above. We:

- introduce the classical RV set-up assuming a single execution, overviewing the notions of monitorability and instrumentation in the context of reactive systems, Section 2.1;
- review the μHML, a highly-expressive modal logic that we extend and adopt as our specification formalism, Section 2.2.

Section 2.2 presents both the linear-time and branching-time semantics of the μHML. It gives a full account of the logic and draws contrast between the two interpretations. This thesis adopts the linear-time semantics of the μHML for the reasons discussed in the concluding section 2.5.

2.1 Runtime Verification

Traditional pre-deployment verification techniques have limited applicability to reactive applications. Commonly-used practices, such as testing [181], only reveal the presence of errors [82], whereas exhaustive approaches such as model checking [141] are laborious [65] and often scale poorly due to state explosion problems. Reactive settings pose even more challenges. For instance, static verification techniques often rely on having access to the system source code or model, which is not necessarily available when software is constructed from libraries or components that are subject to third-party restrictions. Moreover, certain components may be offered as services that are not always known pre-deployment but discovered dynamically at runtime. These aspects tend to increase the complexity of software and the resources required to verify it, while at the same time, decreasing the time available to conduct its verification.



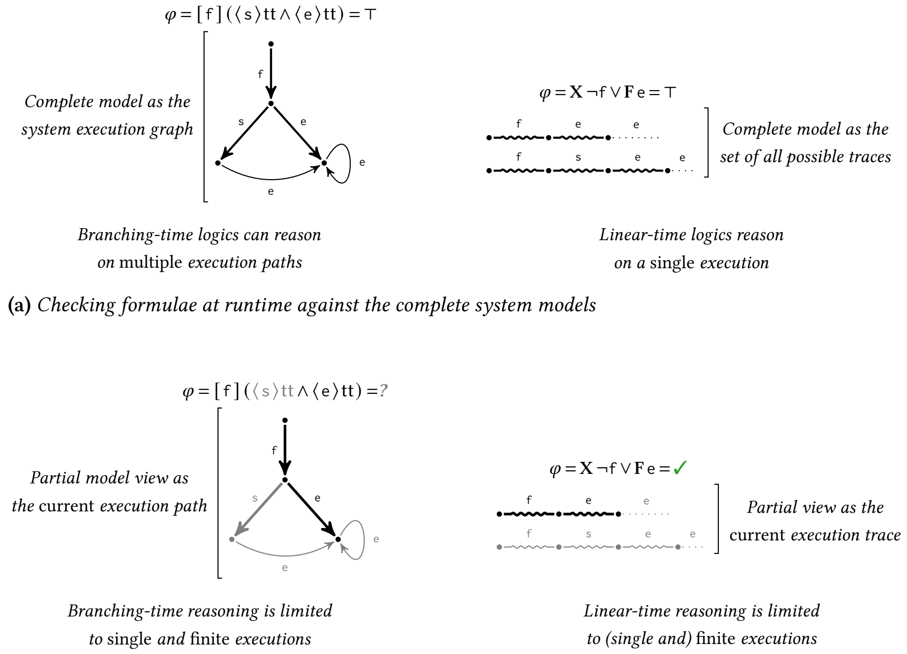
(a) *Property φ describes the current execution trace of the SuS***(b)** *Property φ describes the execution graph of the SuS*

Figure 2.1. *RV for the classical set-up with one execution trace*

RV is a post-deployment technique that can complement static techniques to increase correctness assurances about a program or SuS. It circumvents the obstacles of pre-deployment methods by *dynamically* checking the current execution to determine whether the SuS satisfies or violates some correctness requirement. These requirements are generally specified using a high-level formalism, *e.g.* logic, automata, *etc.*, to unambiguously specify *properties* about the behaviour of the SuS. RV synthesises correctness specifications into *monitors*—computational entities that are instrumented with the SuS to analyse its execution (expressed as a *trace* of events). Monitors typically analyse the trace incrementally up to the current point of execution to reach a *verdict*. Synthesising monitors from correctness specifications implies that, on some level, the meaning of a specification and the verdict that a synthesised monitor declares should *correspond*. Figure 2.1a depicts the traditional RV set-up where a specification φ (①) is synthesised into the monitor M_φ (②) that is instrumented with the SuS to analyse its execution as the events $\alpha_1, \alpha_2 \dots$ (③) until a satisfaction (✓) or rejection (✗) verdict is reached by M_φ.

2.1.1 Specification Logics

Various specification languages are employed to describe correctness properties of the SuS, ranging from temporal logics [36, 207, 210, 31, 118], to automata-based formalisms [24, 69, 197, 124, 166, 23] and (extensions of) regular expression (RE) [115, 206, 63, 23, 176]. Logics and regular expressions provide a 'declarative' way of expressing properties where specifications stipulate *what* to verify. Automata-based formalisms, meanwhile, tend to have a more 'imperative', operational flavour that is close to the verification technique, dictating *how* a property is verified. The former approaches benefit from *compositionality*, since complex specifications can be easily constructed from simpler terms. For instance, two formulae, φ_1 and φ_2, that express different requirements can be combined into a *new* specification, $\varphi_1 \wedge \varphi_2$, demanding that both formulae hold. This benefit also permeates to the verification layer, where constituent parts of a specification (*e.g.* φ_1 and φ_2) may be verified independently. By contrast, automata-based specification languages tend to lack these qualities. As an example, two automata M_1 and M_2 that respectively express the same requirements as the aforementioned formulae, φ_1 and φ_2, must be intersected to describe the requirement equivalent to $\varphi_1 \wedge \varphi_2$. This makes automata-based specifications


φ = [f] (⟨ s ⟩tt ∧ ⟨ e ⟩tt) = ⊤
Complete model as the system execution graph
f
s
e
e
e
φ = X ¬f ∨ F e = ⊤
f
e
e
f
s
e
e
Complete model as the set of all possible traces
Branching-time logics can reason on multiple execution paths
Linear-time logics reason on a single execution
φ = [f] (⟨ s ⟩tt ∧ ⟨ e ⟩tt) = ?
Partial model view as the current execution path
f
s
e
e
e
φ = X ¬f ∨ F e = ✓
f
e
e
f
s
e
e
Partial view as the current execution trace
Branching-time reasoning is limited to single and finite executions
Linear-time reasoning is limited to (single and) finite executions

(a) *Checking formulae at runtime against the complete system models*

(b) *Checking formulae at runtime against the current system execution*

Figure 2.2. *The interpretation of formal logics on system models and system executions*

monolithic, cumbersome to work with, and prone to state blow-ups. Declarative specifications also have an edge in terms of modularity: they make the formalism and verification technique amenable to separate study and development (see section 2.1.3). In RV, this formalism-verification gap is bridged by a synthesis procedure that is responsible for reconciling differences to preserve semantic correspondence. For the reasons mentioned, this thesis looks to *logics* as property specification languages, as these are also portable to other verification platforms, such as model checkers.

Temporal logics are generally categorised into two classes, based on their underlying notion of time [141, 156]. In *linear-time* logics such as LTL [141] and the μ-calculus with a linear-time interpretation [6], formulae describe the behaviour of sets of (possibly infinite) traces that a system model is able to generate. From a temporal perspective, each computational step that a system performs is considered to have one possible future. By contrast, *branching-time* logics such as computation tree logic (CTL) [141] and the μHML [159, 2] describe graphs of the system execution whose states may (non-deterministically) transition to many possible futures. Figure 2.2a (left) depicts a system execution graph that satisfies the branching-time specification given in HML; 2.2a (right) shows a set of traces that satisfy the linear-time specification given in LTL.

2.1.2 Monitors

Monitors are classified based on the timeliness with which execution traces are analysed [100, 25]. *Online* monitors actively analyse events the SuS exhibits while it executes; this analysis is deferred after the system terminates in the case of *offline* monitoring. Offline monitors have access to the complete trace, which enables them to move forward or backward along the execution timeline. Their online counterpart typically analyses the execution in a unidirectional fashion, discarding past events to keep the runtime analysis as *lightweight* as possible. Readers are referred to [100] for details.

The partial view of an execution that an *online* monitor has can be seen as a *prefix* of a larger (possibly infinite) trace, or of a finite *path* within the computation graph of the SuS. We shall refer to finite *or* infinite traces as *finfinite* traces [6]. A monitor is a machine (or a *sequence recogniser* [204, 166]), m, that analyses this prefix and determines set of traces or process states of the SuS that it *accepts* and *rejects* [8, 6]. The restriction on analysing finite traces stems from the online setting, where monitors are constrained to partial views of runs of the SuS that are current, up to the latest event. One non-negotiable requirement is that the verdicts flagged by monitors are *irrevocable*, since verdicts that are subject to revision depending on future trace events are ephemeral, thus *not* dependable. These two aspects distil the core monitor definitions found in the literature (*e.g.* [25, 35, 6]).

The set-ups of figure 2.1 are generalised by Aceto et al. [8] as a *monitoring system*, comprised of a non-empty set of monitors, $\textsc{Mon}$, and two predicates, acc and rej, defined over monitors $m \in \textsc{Mon}$, process states, and finfinite traces. Monitors determine whether to *accept* or *reject* traces or processes via acc and rej respectively. The interpretation of the trace prefix by acc and rej in definition 2.1 depends on the linear-time or branching-time semantics of the formalism used to express properties.

Definition 2.1 (Linear-time and branching-time acceptance and rejection [8, adapted from Definition 3.1]). A monitor m,

(i) for every process p and finite prefix s:
- accepts (resp. rejects) p along s, denoted as $\mathrm{acc}(m,p,s)$ (resp. $\mathrm{rej}(m,p,s)$), if for all of its finfinite continuations f, $\mathrm{acc}(m,p,sf)$ (resp.$\mathrm{rej}(m,p,sf)$)

(ii) for every process p and finfinite trace f:
- accepts (resp. rejects) f produced by p, denoted $\mathrm{acc}(m,p,f)$ (resp. $\mathrm{rej}(m,p,f)$), if there exists a finite prefix s of f and $\mathrm{acc}(m,p,s)$ (resp. $\mathrm{rej}(m,p,s)$)
- accepts (resp. rejects) p along f, denoted $\mathrm{acc}(m,p,f)$ (resp. $\mathrm{rej}(m,p,f)$), if there exists a finite prefix s of f and $\mathrm{acc}(m,p,s)$ (resp. $\mathrm{rej}(m,p,s)$) ■

Point (i) of definition 2.1 captures the notion of irrevocable verdicts, where monitors pass judgements w.r.t. trace prefixes and preserve it along *all* the (possibly infinite) trace continuations. It is worth mentioning that standard finite automata do not satisfy requirement (i): they do not operate on infinite traces and can transition from final to non-final states, which compromises verdict persistence. Point (ii) demands that the analysis that monitors conduct is necessarily finite. It expresses the notions of *good* and *bad* prefixes [155, 17]. Informally, a good prefix is a finite trace such that any of its infinite extensions is accepted; dually, a bad prefix is a finite trace such that any of its infinite extensions is rejected. Standard Büchi automata fail to meet condition (ii), since they require an infinite trace to be read before an acceptance or rejection verdict, can be flagged.

2.1.3 Monitorability

Not all expressible properties can be runtime checked in an online RV setting that is limited to a single, partial execution [25, 117, 94, 58]. For instance, the satisfaction of a (linear-time or branching-time) safety property, *i.e.,* ‘something bad does not happen’, cannot be determined by observing a finite trace, but its *violation* can. Figure 2.2b (left) gives another example of a branching-time property that requires certain behaviour to hold from the same state. Clearly, one execution will never suffice to deduce whether such a property holds.

This limitation is generally tackled in one of two ways. In the first approach, one either (i) restricts the expressive power of the specification language by adapting formalisms such as REs (*e.g.* [115, 192, 23]) or automata to describe finite executions (*e.g.* [68, 70, 176]), or (ii) redefines the semantics of existing logics to reflect the limitations of the runtime setting (*e.g.* [36, 35, 34, 128, 203, 182, 45]). The latter approach leaves the formalism unaltered and identifies *subsets* that can be verified at runtime (*e.g.* [118, 6, 8]).

Both strategies have their merits. The specification formalism in the former approach is closely linked with the monitors, thereby facilitating certain aspects of correctness. Semantics that are bespoke to the RV set-up, on the other hand, complicate its integration with other methods (*e.g.* model checking) that use standard formalisms (*e.g.* LTL). For instance, Bauer et al. [35, 36] adopt this approach, altering the semantics of LTL to assign the truth values $\top$ (satisfied), $\bot$ (violated), and ? (inconclusive) to formulae in their logic, LTL_3. The second strategy preserves the full expressive power of the formalism. Isolating the semantics of the formalism from the operational semantics of monitors makes it possible to establish what aspects of the SuS need to be verified, *agnostic* of the technique used for the verification task. Separating these concerns facilitates the construction of hybrid verification set-ups, where parts of a property can be runtime checked, and other parts verified through more powerful techniques [9, 179]. One body of work adhering to the second method is by [118, 6, 8] which we adopt and build upon in this thesis.

The second strategy also facilitates the study of monitorability. *Monitorability* concerns itself with delineating the properties that can be runtime checked and those that can not [25, 117, 6]. It is the study of the relationship between the semantics of the specification formalism on the one hand (*i.e.,* satisfactions and violations of logic formulae in our case), and the verdicts that are reached by monitors on the other (*i.e.,* acceptances and rejections). Monitorability relies on what a *correct* monitor for a given specification is, which, in turn, establishes what it means for that specification to be *monitorable.* Apart from providing the formal underpinning for monitor correctness [112, 111, 113, 160], monitorability instils a principled approach to constructing RV tools by guiding the development of automated syntheses procedures that generate monitors from specifications. Delimiting the monitorable properties from non-monitorable ones carries other practical advantages. For instance, the synthesis procedure can be optimised to generate monitors for monitorable properties *only.* In certain cases, syntactic characterisations of monitorable properties can be determined (*e.g.* [6, 118, 58]), which improves the usability of RV tools that reject non-monitorable properties via lightweight syntactic checks (*e.g.* [21, 221]). Most crucially, this guarantees that non-rejected specifications generate monitors that are *always* able to reach meaningful verdicts.

Aceto et al. [8] argue that monitorability comes in a spectrum which establishes a *trade-off* between the guarantees that monitors provide, and the properties that can be monitored under these guarantees. The least such non-negotiable guarantee is *soundness*, where the verdicts that monitors report do not contradict the meaning ascribed to the monitored specification φ. We define the predicate $\mathsf{sat}(\varphi,f)$ to denote that a finfinite trace f satisfies φ; analogously $\mathsf{sat}(\varphi,p)$ denotes that a process p satisfies φ.

Definition 2.2 (Linear-time and branching-time monitor soundness [8, adapted from Definition 3.3]). A monitor m is sound,

(i) for linear-time property φ if, for every process p and finfinite trace f:
- $\mathsf{acc}(m,p,f)$ implies $\mathsf{sat}(\varphi,f)$, and
- $\mathsf{rej}(m,p,f)$ implies $\neg\mathsf{sat}(\varphi,f)$

(ii) for branching-time property φ if, for every process p and finfinite trace f:
- $\mathsf{acc}(m,p,f)$ implies $\mathsf{sat}(\varphi,p)$, and
- $\mathsf{rej}(m,p,f)$ implies $\neg\mathsf{sat}(\varphi,p)$ ■

Monitors can easily fulfil the soundness condition by *not* producing a verdict. This calls for *completeness* guarantees that relate to the verdicts that monitors can reach. These guarantees depend on the requirements of the monitoring set-up. For example, a monitor that can reach a verdict at least once even though it might miss other viable detections, may be adequate for certain cases. Other scenarios could impose stricter constraints, such as being able to identify *all* possible satisfactions (satisfaction-completeness) or all possible violations (violation-completeness) for a property [6]. Generally, the stronger the completeness guarantees demanded, the smaller the set of monitorable properties (see [8] for more details).

Definition 2.3 (Linear-time and branching-time monitor completeness [8, adapted from Definition 3.5]). A monitor m is *satisfaction-complete*,

(i) for a linear-time property φ, if for all processes p and finfinite traces f:
- $\mathsf{sat}(\varphi,f)$ implies $\mathsf{acc}(m,p,f)$, and is *violation complete* if $\neg\mathsf{sat}(\varphi,f)$ implies $\mathsf{rej}(m,p,f)$

(ii) for a branching-time property φ, if for all processes p and finfinite traces f:
- $\mathsf{sat}(\varphi,p)$ implies $\mathsf{acc}(m,p,f)$, and is *violation complete* if $\neg\mathsf{sat}(\varphi,p)$ implies $\mathsf{rej}(m,p,f)$

A monitor is *complete*[1] for a property φ if it is both satisfaction-complete and violation-complete, and *partially-complete* if it is either. ■

In their general framework, Aceto et al. [8] give a unifying account of existing notions of monitorability for the linear-time domain over finfinite traces; monitorability for branching-time settings is studied in [118, 6]. The authors show that soundness and the various grades of completeness guarantees produce different monitorability definitions (*e.g.* informative monitorability, partially-complete monitorability, *etc.*). Recall that monitorability establishes how a finite execution prefix is to be interpreted by a monitor *and* correctly mapped to the property expressed by some specification φ. Intuitively, a monitor that checks for property satisfactions analyses the execution to find one witness confirming that the property holds. Dually, monitoring for property violations requires the monitor to find one counter witness confirming that the property does not hold. More formally, a linear-time property is a language over trace events, denoted as P_{LT}. By analysing events from the trace, a monitor determines whether the event sequence read so far constitutes the prefix of a *word* in the property language. Words in (resp. not in) P_{LT} denote property satisfactions (resp. violations). A branching-time property is a set of program states, denoted as P_{BT}, that correspond to the behaviour the system can exhibit. By analysing events, a monitor determines whether the event sequence read so far constitutes a *path* leading to program states described by the property. States in (resp. not in) P_{BT} denote property satisfactions (resp. violations).

[1] As Aceto et al. [8] show, full monitor completeness is only possible for trivial properties, namely all the formulae that are semantically equivalent to true or false.

Figure 2.2b sketches how branching-time and linear-time properties would be runtime checked against the current execution trace (*cf.* figure 2.2a that has access to complete models).

Aceto et al. [8] discuss that monitorability can be specified in terms of monitor soundness and different levels of strictness of completeness that depend on the guarantees expected of monitors. The approach taken in this body of work, by contrast to others in the field (*e.g.* [35, 34, 68, 24, 45, 203]), adheres to the tenets of modular verification advocated earlier in section 2.1.3. The authors consider the μHML as their touchstone specification formalism. The authors identify maximally-expressive (*i.e.,* characterises all semantically equivalent specifications) monitorable syntactic fragments of the μHML for the linear-time interpretation of their logic. We adopt their framework, and instantiate definitions 2.1 to 2.3 under specific completeness guarantees in chapter 3 w.r.t. a concrete operational model of monitors that builds on theirs. Chapter 3, also formalises the definitions of the predicates acc and rej via an instrumentation relation, followed by a synthesis procedure that generates correct monitors that can handle data. We start by concretising the abstract predicates sat mentioned above in section 2.2.

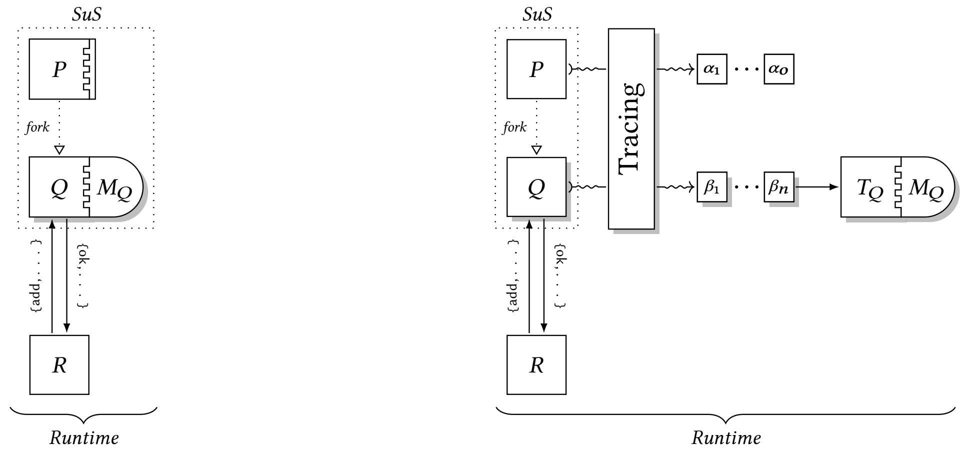
2.1.4 Instrumentation for Online Monitoring

Instrumentation lies at the heart of runtime monitoring [164, 117, 25]. It refers to the extraction of information from executing software and its reporting to monitors, following one of two approaches. In the *inline* approach, instrumentation is implemented by manually implanting the SuS with tracing instructions, or automatically, using aspect-oriented programming (AOP) [146] frameworks that inject the instrumentation code with the system via source or object code weaving (*e.g.* AspectJ [147], SpringAOP [223], BCEL [76], *etc.*). Inlining offers a number of benefits, such as timely detections of anomalous behaviour and the ability to intervene and steer the system execution if required. Nevertheless, these qualities do not necessarily make inlining the ideal approach for monitoring large-scale reactive systems. Despite its reputation for inducing low overhead, the synchronous coupling that inlining creates with the SuS can impinge on the operation of the system [61, 51, 25, 68], *e.g.* slow runtime analyses manifest as high response time latencies, faulty monitors may break the system, *etc.* Moreover, certain kinds of monitoring errors, such as deadlocks [61] or component crashes [221], may be difficult to detect since the monitoring logic shares the execution thread of the affected component. In cases where the SuS sources or binaries are unavailable (*e.g.* closed-source components, licensing agreements, third-party services, *etc.*), inlining *cannot* be used. Inlining is typically programming language-dependent, which limits its application to heterogeneous components. It is also hard to undo once administered, requiring restarts or redeployments of the SuS.

Outline instrumentation [100, 25] is an alternative approach to inlining, where the SuS and monitors are encapsulated into respective concurrent entities [15]. It leverages a tracing infrastructure that gathers information externally (*e.g.* DTrace [50], LTTng [80], Erlang Trace [57], OpenJ9 Trace [86]). This minimal coupling between the SuS and monitors begets a number of advantages that are attuned to the characteristics of reactive systems [153]. For instance, outline monitors can treat the SuS as a *black* (or *grey*) box and only react to certain events exhibited in the system execution trace. Besides serving the runtime analysis, the trace information can be leveraged to scale the instrumentation dynamically, proportionate to the computational demands of the SuS. Since tracing frameworks do not necessitate access to the SuS, it makes the set-up *language agnostic.* Additionally, monitors may be enabled and disabled on demand without system redeployments or restarts, which is invaluable when profiling or live debugging concurrency bugs that emerge for particular execution paths. Decoupling the SuS from monitors carries another advantage. It induces a degree of resiliency in the set-up in the forms of *partial failure* (faulty monitors do not compromise the system, and vice versa) and *monitor redundancy* (a failed monitor does not hamper other instances from monitoring replicas of the same component).

Tracing information reported by the instrumentation can assume different forms, and is often tailored to specific uses. For instance, coarse-grained or aggregated data suffices for compiling usage statistics or for application performance monitoring (APM) and tuning. Applications such as live debuggers, auditing or verification tools require data as program events that advertise *changes* in the state of the SuS. Our abstract definition of RV monitors from section 2.1.2 demands stringent guarantees from the instrumentation, namely that the (i) trace events reported to monitors are *consistent* with the order in which they are exhibited by the SuS, and (ii) that traces have *no missing events*.

The instrumentation determines how the SuS and monitor execution evolves as time progresses. *Synchronous* monitoring interleaves the SuS-monitor execution such that both run in lock-step, *i.e.,* the system is paused until the monitor completes its analysis. Synchronous monitoring is implemented using inlining [70, 13, 130, 148, 197, 88, 84]. Certain tools [60, 51, 52] externalise monitors as processes that synchronise with the SuS on each event it exhibits. While their authors refer to these monitors as 'outline', we classify them as inline since the instrumentation must modify the system to inject monitor synchronisation points. *Asynchronous* monitoring uses outline instrumentation, enabling the SuS to execute unencumbered by monitor computation. To the best of our knowledge, relatively few instances of asynchronous monitoring tools exist, some of which employ the Erlang tracing infrastructure to report events to a *central* monitor that executes alongside the SuS [71, 221, 219, 113]. Figure 2.3 illustrates typical monitor arrangements for the synchronous and asynchronous cases. Monitor M_Q is inlined as part of process Q (2.3a), whereas tracer T_Q obtains the events of process Q by way of the tracing infrastructure that acts as a middleware (2.3b). The events that tracer T_Q receives are, in turn, reported to monitor M_Q for analysis. The material in the rest of this thesis regards inline (resp. outline) instrumentation and synchronous (resp. asynchronous) monitoring as synonymous.



(a) *Synchronous monitors via weaving*

(b) *Asynchronous monitors via the tracing infrastructure*

Figure 2.3. *Inline (synchronous) and outline (asynchronous) instrumentation for process Q*

In principle, the instrumentation composes monitors with the SuS to yield a *monitored system* [118]. A monitored system could potentially manifest different behaviour to the unmonitored SuS—a product of (i) the instrumentation method adopted [112], *e.g.* outline, and (ii) the assurances given by monitors [160], *e.g.* passive monitors. Although core monitor concerns, such as correctness [210, 67, 70, 71, 68], efficiency [207, 208, 209, 175, 96, 24, 63], security [102, 91], and even failure [34, 180, 31], have been treated to different degrees in the RV literature, instrumentation has not been studied in its own right. This theme recurs in particular RV tool development practices, where instrumentation is occasionally portrayed to induce low overhead [95, 55, 68, 25, 100], albeit with no quantifiable backing [175, 209, 42, 61, 207, 99, 24] (we elaborate on these arguments in chapter 7 and in particular, section 7.4).

A recent body of work [118, 112, 6, 8] is one of the few notable efforts that investigates monitors in the context of an instrumented system set-up from a formal aspect. The operational definition of the instrumentation given relates SuS and monitor states to produce a monitored system where monitors are *passive*. Despite their passive role, [112] shows that certain monitors that behave inertly when considered in isolation *can* still interfere with an instrumented system. For instance, it is natural to expect the instrumentation not to prematurely terminate monitors before a verdict is flagged, but wait for their internal computation to complete. However, too lengthy or divergent computations can slow or even stall the SuS. The *execution slowdown* [26] observed in practice is a manifestation of this phenomenon, and is one of the main drawbacks of synchronous (*i.e.,* inline) approaches [61, 51, 25, 68]. Such subtle interdependencies that arise between the SuS and monitors are not edge-case scenarios, but practical issues that the design of monitoring tools must tackle from the outset. Particularly, [112, 6] make a strong case that the definition of correct monitors needs to comprise the instrumentation. As far as we can understand, the above-mentioned works that use *inlining* do not reconcile the gap between the monitor formalisations at one end, and the instrumentation aspect in their ensuing prototype tools at the other (*e.g.* [210, 67, 70, 71, 68, 207, 208, 209, 175, 96, 24, 63, 102, 91, 34, 180]).

2.2 The Hennessy-Milner Logic with Recursion

We overview our chosen logic [6, 8], μHML [159, 2], which we use to specify correctness properties. The μHML is a reformulation of the highly-expressive modal μ-calculus [150] that can embed other prevalent logics, such as CTL and LTL [141], making it suitable to express a wide range of properties. It has a branching-time semantics to specify properties about the execution graph of processes, and a linear-time semantics (adapted from the modal μ-calculus) describing properties of the *current* program execution (see section 2.1.1). The logic presented in Aceto et al. [6, 8] can express *regular* properties, which arguably limits its applicability to a broader setting where systems deal with data. We, therefore, extend the μHML of *op. cit.* to a first-order setting, where logic formulae can specify properties that reason about the data carried in trace events. Sections 2.3 and 2.4 recall the syntax and semantics of the logic and formalise the concepts of traces and processes introduced in section 2.1.2.

2.3 The Syntax of μHML$^{\mathrm{D}}$

Figure 2.4 shows our extension of μHML, called μHML$^{\mathrm{D}}$. It assumes a set of external actions, $\alpha,\beta \in \textsc{Act}$, together with a distinguished internal action $\tau \notin \textsc{Act}$ that represents one internal step of computation. External actions range over values taken from some (potentially infinite) data domain, $\mathbb{D}$. The μHML$^{\mathrm{D}}$

syntax also uses a denumerable set of propositional variables, $X, Y \in \textsc{PVar}$. In addition to the standard Boolean constructs, the logic can express recursive and least and greatest fixed point formulae, $\mathsf{min}X.(\varphi)$ and $\mathsf{max}X.(\varphi)$, that bind the free occurrences of X in φ. The existential and universal modalities, $\langle \boldsymbol{x}, b\rangle\varphi$ and $[\boldsymbol{x}, b]\varphi$, express the dual notions of *possibility* and *necessity* respectively. We augment these two modal constructs with *symbolic actions*, denoting them by $(\boldsymbol{x}, b)$, to enable reasoning on the data carried by external actions. Symbolic actions are pairs consisting of data binders, $x, y \in \textsc{DVar}$, and *decidable* Boolean constraint expressions, $b, c \in \textsc{BExp}$. Data binders also range over the domain $\mathbb{D}$ of data values, and bind the free occurrences of x in the expression b of the modality and in the continuation formula φ. The set BExp, defined over $\mathbb{D}$ and DVar, consists of the usual Boolean operators, including, $\neg$ and $\wedge$, together with a set of relational operators that depends on $\mathbb{D}$, and which we leave unspecified. For clarity, we omit writing the Boolean constraint expression b when $b = \mathsf{tt}$, and use ***bold*** italicised lettering to identify binders in symbolic actions.

In the sequel, the standard concepts of open and closed expressions, scoping, and formula equality up to alpha-conversion are used. A formula is said to be *guarded* if every fixed point variable X appears within the scope of a modality that is itself in the scope of X. For example, the formula $\mathsf{max}X.([\boldsymbol{x}]\mathsf{ff} \wedge [\boldsymbol{y}]X)$ is guarded, as is $\mathsf{max}X.\big([\boldsymbol{x}]([\boldsymbol{y}]\mathsf{ff} \wedge X)\big)$, while $[\boldsymbol{x}]\mathsf{max}X.([\boldsymbol{y}]\mathsf{ff} \wedge X)$ is not.

μHML$^{\mathrm{D}}$ Syntax

$$\varphi, \psi \in \mu\mathrm{HML}^{\mathrm{D}} ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle \boldsymbol{x}, b\rangle\varphi \mid [\boldsymbol{x}, b]\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathsf{min}X.(\varphi) \mid \mathsf{max}X.(\varphi) \mid X$$

μHML$^{\mathrm{D}}$ Linear-Time Semantics

$$[\![\mathsf{tt}, \sigma]\!]_{\mathrm{LT}} \triangleq \textsc{Act}^{\omega} \qquad [\![\mathsf{ff}, \sigma]\!]_{\mathrm{LT}} \triangleq \emptyset$$

$$[\![\langle \boldsymbol{x}, b\rangle\varphi, \sigma]\!]_{\mathrm{LT}} \triangleq \big\{t \mid (\exists u. \exists \alpha.\, t = \alpha u \text{ and } b[{}^{\alpha}\!/\!_{x}] \Downarrow \mathsf{tt} \text{ and } u \in [\![\varphi[{}^{\alpha}\!/\!_{x}], \sigma]\!]_{\mathrm{LT}})\big\}$$

$$[\![[\boldsymbol{x}, b]\varphi, \sigma]\!]_{\mathrm{LT}} \triangleq \big\{t \mid (\forall u. \forall \alpha.\, (t = \alpha u \text{ and } b[{}^{\alpha}\!/\!_{x}] \Downarrow \mathsf{tt}) \text{ implies } u \in [\![\varphi[{}^{\alpha}\!/\!_{x}], \sigma]\!]_{\mathrm{LT}})\big\}$$

$$[\![\varphi \vee \psi, \sigma]\!]_{\mathrm{LT}} \triangleq [\![\varphi, \sigma]\!]_{\mathrm{LT}} \cup [\![\psi, \sigma]\!]_{\mathrm{LT}} \qquad [\![\varphi \wedge \psi, \sigma]\!]_{\mathrm{LT}} \triangleq [\![\varphi, \sigma]\!]_{\mathrm{LT}} \cap [\![\psi, \sigma]\!]_{\mathrm{LT}}$$

$$[\![\mathsf{min}X.(\varphi), \sigma]\!]_{\mathrm{LT}} \triangleq \bigcap \big\{T \mid [\![\varphi, \sigma[X \mapsto T]]\!]_{\mathrm{LT}} \subseteq T\big\} \qquad [\![\mathsf{max}X.(\varphi), \sigma]\!]_{\mathrm{LT}} \triangleq \bigcup \big\{T \mid T \subseteq [\![\varphi, \sigma[X \mapsto T]]\!]_{\mathrm{LT}}\big\}$$

$$[\![X, \sigma]\!]_{\mathrm{LT}} \triangleq \sigma(X)$$

μHML$^{\mathrm{D}}$ Branching-Time Semantics

$$[\![\mathsf{tt}, \rho]\!]_{\mathrm{BT}} \triangleq \textsc{Prc} \qquad [\![\mathsf{ff}, \rho]\!]_{\mathrm{BT}} \triangleq \emptyset$$

$$[\![\langle \boldsymbol{x}, b\rangle\varphi, \rho]\!]_{\mathrm{BT}} \triangleq \big\{p \mid (\exists q. \exists \alpha.\, p \overset{\alpha}{\Longrightarrow} q \text{ and } b[{}^{\alpha}\!/\!_{x}] \Downarrow \mathsf{tt} \text{ and } q \in [\![\varphi[{}^{\alpha}\!/\!_{x}], \rho]\!]_{\mathrm{BT}})\big\}$$

$$[\![[\boldsymbol{x}, b]\varphi, \rho]\!]_{\mathrm{BT}} \triangleq \big\{p \mid (\forall q. \forall \alpha.\, (p \overset{\alpha}{\Longrightarrow} q \text{ and } b[{}^{\alpha}\!/\!_{x}] \Downarrow \mathsf{tt}) \text{ implies } q \in [\![\varphi[{}^{\alpha}\!/\!_{x}], \rho]\!]_{\mathrm{BT}})\big\}$$

$$[\![\varphi \vee \psi, \rho]\!]_{\mathrm{BT}} \triangleq [\![\varphi, \rho]\!]_{\mathrm{BT}} \cup [\![\psi, \rho]\!]_{\mathrm{BT}} \qquad [\![\varphi \wedge \psi, \rho]\!]_{\mathrm{BT}} \triangleq [\![\varphi, \rho]\!]_{\mathrm{BT}} \cap [\![\psi, \rho]\!]_{\mathrm{BT}}$$

$$[\![\mathsf{min}X.(\varphi), \rho]\!]_{\mathrm{BT}} \triangleq \bigcap \big\{P \mid [\![\varphi, \rho[X \mapsto P]]\!]_{\mathrm{BT}} \subseteq P\big\} \qquad [\![\mathsf{max}X.(\varphi), \rho]\!]_{\mathrm{BT}} \triangleq \bigcup \big\{P \mid P \subseteq [\![\varphi, \rho[X \mapsto P]]\!]_{\mathrm{BT}}\big\}$$

$$[\![X, \rho]\!]_{\mathrm{BT}} \triangleq \rho(X)$$

Figure 2.4. *Syntax, linear-time and branching-time semantics for the μHML^{D}*

2.4 The Semantics of μHML$^{\mathrm{D}}$

The linear-time interpretation of μHML$^{\mathrm{D}}$ is given by the denotational semantic function $[\![-]\!]_{\mathrm{LT}}$ that maps a formula to a set of executions. Executions (or traces) are *infinite* sequences of external system actions that abstractly represent *complete* system runs. We reserve the metavariables $t,u \in \textsc{Act}^{\omega}$ to range over infinite traces, $T \subseteq \textsc{Act}^{\omega}$ to range over sets of infinite traces, and use αt to denote an infinite trace that starts with α and continues with t. Finite traces, $s,r \in \textsc{Act}^{*}$, represent prefixes of infinite or finite executions.

The function $[\![-]\!]_{\mathrm{LT}}$ uses valuations, $\sigma : \mathrm{PV}\textsc{ar} \longrightarrow 2^{\textsc{Act}^{\omega}}$, to define the semantics inductively on the structure of formulae. The value $\sigma(X)$ is the set of traces that are assumed to satisfy X. In the definition of $[\![-]\!]_{\mathrm{LT}}$, modal formulae are interpreted w.r.t. symbolic actions. A symbolic action (x,b) describes a set of external system actions, referred to as an *action set.* An action α is in this set when the data value it carries satisfies the Boolean constraint expression b that is instantiated with the *applied substitution* $[{}^{\alpha}/_{x}]$, *i.e.,* $b[{}^{\alpha}/_{x}] \Downarrow \mathrm{tt}$ (see figure 2.4). The existential modality $\langle x,b\rangle\varphi$ denotes all the traces αu where α is in the action set (x,b) *and* u satisfies the continuation $\varphi[{}^{\alpha}/_{x}]$. Dually, $[x,b]\varphi$ denotes all the traces αu that, *if* prefixed by any α from the action set (x,b), u *then* satisfies $\varphi[{}^{\alpha}/_{x}]$. Note that if α is *not* in the action set, the trace αu satisfies $[x,b]\varphi$ trivially. The set of traces satisfying the least (resp. greatest) fixed point formulae $\mathsf{min}\,X.(\varphi)$ (resp.$\mathsf{max}\,X.(\varphi)$) is the intersection (resp. union) of all the pre-fixed (resp. post-fixed) point solutions, $T \subseteq \textsc{Act}^{\omega}$, of the function induced by the formula φ.

The branching-time interpretation of μHML$^{\mathrm{D}}$, denoted by $[\![-]\!]_{\mathrm{BT}}$, is defined over process states of a labelled transition system (LTS) [145]. A LTS is a triple, $\langle \textsc{Prc},(\textsc{Act}\cup\{\tau\}),\longrightarrow\rangle$, consisting of a set of process states, $p,q \in \textsc{Prc}$, a set of actions including τ, and a transition relation, $\longrightarrow \subseteq \textsc{Prc}\times(\textsc{Act}\cup\{\tau\})\times\textsc{Prc}$. The variable $\mu \in \textsc{Act}\cup\{\tau\}$ is reserved for external or internal actions, and $P \subseteq \textsc{Prc}$ for sets of processes. We use the suggestive notation $p \xrightarrow{\mu} p'$ to denote labelled state transitions, $\langle p,\mu,p\rangle \in\; \longrightarrow$, and $p \not\xrightarrow{\mu}$ to mean $\neg(\exists p' \cdot p \xrightarrow{\mu} p')$. Weak transitions, $p(\xrightarrow{\tau})^{*}p'$, are denoted as $p \Longrightarrow p'$, whereas $p \overset{\alpha}{\Longrightarrow} p'$ is written in lieu of $p \Longrightarrow \cdot \xrightarrow{\alpha} \cdot \Longrightarrow p'$, referring to p' as the α-*derivative* of p. A transition sequence, $p \overset{\alpha_1}{\Longrightarrow} \cdots \overset{\alpha_n}{\Longrightarrow} p'$, is compactly written as $p \overset{s}{\Longrightarrow} p'$, where $s=\alpha_1\cdots\alpha_n$ is a *finite* trace of external actions. We say that a process p generates the trace $t=\alpha_1\alpha_2\cdots$ if there is an infinite sequence $p_0,p_1,p_2,\ldots$ of processes such that $p=p_0$ and $p_0 \overset{\alpha_1}{\Longrightarrow} p_1 \overset{\alpha_2}{\Longrightarrow} p_2\cdots$.

Figure 2.4 also defines the branching-time semantics of μHML$^{\mathrm{D}}$ via the function $[\![-]\!]_{\mathrm{BT}}$ that uses valuations $\rho : \mathrm{PV}\textsc{ar} \longrightarrow 2^{\textsc{Prc}}$. Most cases follow the linear-time counterpart; the main differences are w.r.t. modal formulae. Existential modalities, $\langle x,b\rangle\varphi$, require *at least* one α-derivative of a process p for some α in the action set (x,b) to satisfy φ. Its dual, $[x,b]\varphi$, requires *all* the α-derivatives of p labelled by the actions in the set defined by (x,b) to satisfy φ.

Since the interpretation of *closed* formulae does not depend on the environment σ or ρ, we may use $[\![\varphi]\!]_{\mathrm{LT}}$ and $[\![\varphi]\!]_{\mathrm{BT}}$ in lieu of $[\![\varphi,\sigma]\!]_{\mathrm{LT}}$ and $[\![\varphi,\rho]\!]_{\mathrm{BT}}$ respectively. We also write $[\![\varphi]\!]$ instead of $[\![\varphi]\!]_{\mathrm{LT}}$ or $[\![\varphi]\!]_{\mathrm{BT}}$ whenever the correct semantic interpretation can be inferred from the surrounding context or is unimportant. A trace t (resp. process p) satisfies (the closed) formula φ when $t \in [\![\varphi]\!]_{\mathrm{LT}}$ (resp. $p \in [\![\varphi]\!]_{\mathrm{BT}}$), and violates φ when $t \notin [\![\varphi]\!]_{\mathrm{LT}}$ (resp. $p \notin [\![\varphi]\!]_{\mathrm{BT}}$). Unless otherwise indicated, we assume that all formulae considered are closed. To facilitate our exposition in this section and chapter 3, we let $\mathbb{D}=\mathbb{Z}$, and fix the set of operators used in BE$\textsc{xp}$ to $\neg$, $\wedge$ and $=$. Chapter 4 considers the general case where the data carried by external actions can consist of *composite* data types.

Definition 2.4 (Linear-time and branching-time formula satisfaction). The predicates $\mathsf{sat}(\varphi,f)$ and $\mathsf{sat}(\varphi,p)$ assumed in section 2.1.3 can now be defined. Since the linear-time interpretation of μHML$^{\text{D}}$ given in figure 2.4 assumes an infinite domain, $\mathsf{sat}(\varphi,f)$, is restricted to infinite traces, t.

$$\mathsf{sat}(\varphi,t) \triangleq t \in [\![\varphi]\!]_{\text{LT}} \qquad\qquad \mathsf{sat}(\varphi,p) \triangleq p \in [\![\varphi]\!]_{\text{BT}} \qquad \blacksquare$$

Example 2.1 (Interpretation and reasoning on data). Consider the formula:

$$[x, x=0]\,\mathsf{ff} \tag{φ_1}$$

The symbolic action $(x, x=0)$ defines the singleton set, $\{0\} \subset \mathbb{Z}$, of external system actions. In the linear-time interpretation, modal formulae $[x,b]\,\varphi$, state that, for *any* trace prefix α in the action set (x,b), the trace continuation u must satisfy φ. However, *no* trace satisfies ff, *i.e.*, $\forall u.\, u \notin [\![\mathsf{ff}]\!]_{\text{LT}}$. This means that traces that do not violate formula φ_1 are those starting with actions $\alpha \notin \{0\}$. The interpretation under the branching-time semantics is similar: $[x,b]\,\varphi$ requires that *all* the α-derivatives of a process p, where α is in the action set (x,b), reach some state p' that satisfies φ. Since $p' \notin [\![\mathsf{ff}]\!]_{\text{BT}}$ for any p', process p satisfies φ_1 only when it exhibits actions other than 0; this includes the deadlocked process that performs no action. $\blacksquare$

Example 2.2 (Comparison). Consider the two formulae φ_2 and φ_3, together with the trace $t_1 = (0.1)^{\omega}$ and the (non-deterministic process) given in CCS syntax [178], $p_1 = \mathsf{rec}\,X.(0.1.X + 0.0.X + 0.\mathsf{nil})$. Note that in particular, p_1 produces the infinite trace t_1.

$$(\varphi_2) \qquad [x,x=0]\,[y,y=0]\,\mathsf{ff} \qquad\qquad [x,x=0]\,(\langle \overset{\curvearrowright}{y,y}=0\rangle\mathsf{tt} \vee \langle \overset{\curvearrowright}{y,y} \neq 0\rangle\mathsf{tt}) \qquad (\varphi_3)$$

While $t_1 \in [\![\varphi_2]\!]_{\text{LT}}$, $p_1 \notin [\![\varphi_2]\!]_{\text{BT}}$ because p_1 performs the transition $p_1 \overset{0}{\Longrightarrow} 0.p_1$ along one branch, and the derived process state $0.p_1 \notin [\![\,[y,y=0]\,\mathsf{ff}]\!]_{\text{BT}}$ (see example 2.1). Under the linear-time interpretation, the equality $[\![\langle y,b\rangle\mathsf{tt} \vee \langle y,\neg b\rangle\mathsf{tt}]\!]_{\text{LT}} = [\![\mathsf{tt}]\!]_{\text{LT}}$ holds for every symbolic action (y,b). In our case, $(y, y=0)$ and $(y, y \neq 0)$ in formula φ_3 define the *complementary* action sets $\{0\}$ and $\mathbb{Z} \setminus \{0\}$ respectively. Now, every infinite trace *must* have a first element α that is either $\alpha \in \{0\}$ or $\alpha \in \mathbb{Z} \setminus \{0\}$. This means that $[\![\langle y,y=0\rangle\mathsf{tt} \vee \langle y,y\neq 0\rangle\mathsf{tt}]\!]_{\text{LT}} = [\![\mathsf{tt}]\!]_{\text{LT}}$. From the semantic definitions of figure 2.4, one can also deduce that $[\![\,[x,b]\,\mathsf{tt}]\!] = [\![\mathsf{tt}]\!]$ under *both* interpretations. As a result, φ_3 is equivalent to tt under the linear-time semantics, and thus, $t_1 \in [\![\varphi_3]\!]_{\text{LT}}$ for all traces t_1. In the branching-time setting, $[\![\langle y,y=0\rangle\mathsf{tt} \vee \langle y,y\neq 0\rangle\mathsf{tt}]\!]_{\text{BT}} \neq [\![\mathsf{tt}]\!]_{\text{BT}}$. One witness for this inequality is the process nil, where $\mathsf{nil} \in [\![\mathsf{tt}]\!]_{\text{BT}}$, *but* $\mathsf{nil} \notin [\![\langle y,y=0\rangle\mathsf{tt} \vee \langle y,y\neq 0\rangle\mathsf{tt}]\!]_{\text{BT}}$ since $\mathsf{nil} \overset{\alpha}{\not\longrightarrow}$. In fact, the transition $p_1 \overset{0}{\Longrightarrow} \mathsf{nil}$ does not fulfil the semantic condition of φ_3 that all α-derivatives of p_1, where $\alpha \in \{0\}$, reach a state p_1' that satisfies the continuation formula (clearly, nil does not). Consequently, $p_1 \notin [\![\varphi_3]\!]_{\text{BT}}$. Note that the binders y in $\langle y,y=0\rangle\mathsf{tt}$ and $\langle y,y\neq 0\rangle\mathsf{tt}$ of formula φ_3 have *different* scopes. $\blacksquare$

Example 2.3 shows how μHML can encode the core operators of LTL, a temporal logic which is widely-adopted by the RV community, and that most tooling efforts employ as their specification formalism (*e.g.* [35, 36, 34, 45, 31, 128, 208, 210, 203]).

Example 2.3 (Expressiveness). The core LTL operators, next and until, can be encoded thus [141]:

$$\mathsf{X}\,\varphi \triangleq \langle x\rangle\mathsf{tt} \qquad\qquad \varphi\,\mathsf{U}\,\psi \triangleq \min Y.\big(\psi \vee (\varphi \wedge \langle x\rangle Y)\big) \qquad \blacksquare$$

Despite its widespread use, LTL has limited expressiveness. For instance, it cannot describe properties such as *'every* even *position in the execution satisfies some proposition p'* [225, 8]. Such properties can be easily expressed in μHML^{D} (see example 3.3 on page 25).

2.5 Discussion

Runtime monitoring is amenable to lightweight verification settings where traditional approaches cannot be used (*e.g.* expensive, not scalable). Despite the advantages it offers, the technique suffers from limited expressiveness, where certain properties cannot be runtime checked. This constraint arises from the partial view that monitors have of the SuS, which is limited to a single and finite execution—one of the possible paths the system follows at runtime. Monitorability provides a principled method to identify properties that can be monitored from those that cannot. This, in turn, gives a precise meaning of what it means to monitor for a property correctly. Monitorability is underpinned by the notion of a monitor [8]: a machine that analyses finite event sequences to accept (acc) or reject (rej) finfinite traces or process states of the SuS w.r.t. specific guarantees. We expect two least guarantees, namely that (i) the verdicts that monitors report do not contradict the meaning ascribed to specifications (*soundness*), and (ii) under some criterion, the monitor can perform detections (*completeness*). We adopt the unified monitorability view of Aceto et al. [8], where soundness and completeness are defined operationally in terms of the predicates acc and rej; these predicates (definition 2.1), together with definitions 2.2 and 2.3 are concretised in chapter 3.

This chapter also discusses the instrumentation that composes monitors with the SuS in inline (*synchronous*) or outline (*asynchronous*) fashion. In spite of its importance to RV, the instrumentation is given limited consideration in the literature, with much of the work focussing on the monitors, studied in a vacuum [210, 67, 70, 71, 68, 207, 208, 209, 175, 96, 24, 63, 102, 91, 34, 180, 31]; Aceto et al. [8] together with [118, 6] are few notable exceptions that give the instrumentation a central role. Particularly, Aceto et al. [8] shows that passive monitors can still produce side effects when instrumented with the SuS. The authors make a strong case that the definitions of monitorability and monitor correctness should incorporate the instrumentation.

The ongoing line of work by [1, 118, 3, 4, 5, 6, 7] studies the branching-time μHML in the context of RV and hybrid approaches [9], and parts of the results have been instantiated in a number of tools, *i.e.,* the set-up of figure 2.1b. Readers are referred to [21, 219, 220, 221, 53, 51, 52, 113] for more details. In this thesis, we adopt the linear-time interpretation of μHML where specifications express properties on the *current* system execution (figure 2.1a). Example 2.3 shows that the logic easily embeds other logics and can express a wider range of properties; this gives us a good level of generality in our results. The aforecited tools focussing on the branching-time interpretation of the μHML employ the *same* operational model of monitors given in [118, 6], which we extend in chapter 3. As a result, our synthesis procedure can generate executable monitor code from linear-time specifications that is identical to monitors obtained from branching-time specifications. This portability makes our subsequent results of chapters 6 and 7 applicable to the tools mentioned, *i.e.,* [21, 219, 220, 221, 53, 51, 52, 113].

3 Monitors and Instrumentation

Properties may be expressed using different formalisms. We adopt the linear-time μHML that describes properties about the *current* execution trace (refer to section 2.2). Section 2.1.3 establishes that the runtime setting limits what properties can be monitored for under the constraints of a single, incomplete trace that is incrementally extended as the execution of the SuS unfolds. This chapter instantiates the concepts introduced there. It pins down a formal operational model of monitors whose description can be executed. We give concrete definitions for the *acceptance* and *rejection* predicates, acc and rej, w.r.t. the irrevocable verdicts that these monitors can reach. Our definitions make use of a synchronous instrumentation relation that composes the monitors and SuS, dictating how these verdicts are reached at runtime. Using acc and rej, we formalise the notions of monitor *soundness* and *completeness* to recall the monitorability definition for the linear-time μHML [6, 8], together with two maximally-expressive monitorable logic fragments (refer to section 2.5 for reasons why we adopt the linear-time μHML).

Our work builds on the theoretical foundations of Aceto et al. [6, 8] that give an operational model of regular monitors and a compositional synthesis procedure that generates correct monitors from the aforementioned monitorable fragments of μHML. We lift the results of that study to a first-order setting and extend the monitoring model and synthesis procedure with symbolic actions introduced in section 2.2 to account for data payloads carried by trace events. Our adaptation of the monitor synthesis closely follows the one of *op. cit.*, giving us high assurances that the corresponding monitors are correct. The modular approach followed by the authors has been translated to different implementations [21, 219, 221, 13, 114], including detectEr [221], a RV tool that targets programs written for the Erlang/OTP platform. One aspect that Aceto et al. [6, 8] do not tackle is how the SuS and monitors can be composed *asynchronously* to mitigate the issues with lock-step execution and monitor inlining mentioned in section 2.1.4. This chapter addresses this gap and gives an alternative instrumentation that disconnects the SuS from its monitors. Crucially, our asynchronous instrumentation definition remains *compatible* with the requirements that Aceto et al. [6, 8] expect of the monitoring model, making their correctness results transferable to our framework as well. We:

(i) demonstrate how properties on the current execution can be flexibly expressed via the linear-time μHML^{D}, Section 3.1;

(ii) overview our extended monitoring model and the synchronous instrumentation relation of Aceto et al. [6, 8], Section 3.2;

(iii) define soundness, completeness, and monitorability w.r.t. the logic of (i) and models of (ii), and recall the monitorable fragments of the linear-time μHML^{D}, Section 3.3;

(iv) outline our adaptation of the monitor synthesis procedure that generates parallel monitors from monitorable linear-time μHML^{D} fragments, Section 3.4;

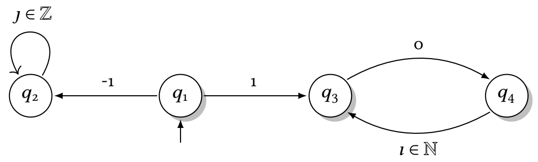


Figure 3.1. *Token server that issues integer identification tokens to client programs*

(v) define an instrumentation relation that composes monitor and SuS processes in asynchronous fashion, Section 3.5.

3.1 Trace Properties

Figure 3.1 depicts a generalisation of process p_1 from example 2.2, $q_1 = 1.\text{rec}\, X.(0.\imath.X) + -1.\text{rec}\, Y.(\jmath.Y)$. The process q_1 models a reactive token server that issues client programs with identification tokens that they use as an alias to write logs to a remote logging server. Clients request an identifier by issuing the command 0, which the server then fulfils by replying with a new token, $\imath \in \mathbb{N}$. Since the server is itself a program that also uses the remote logging service, it is launched with its (reserved) identification token 1. Figure 3.1 shows that from its initial state q_1, the token server either: (i) starts up with the token 1 and transitions to q_3, where it awaits incoming client requests, or (ii) fails to start and transitions with a status of -1 to the sink q_2, thereafter exhibiting *undefined behaviour*, $\jmath \in \mathbb{Z}$. There are a number of properties we want *executions* of this token server to observe.

Example 3.1 (Necessity). One rudimentary property that the current execution of the server q_1 should uphold is that *'no failure occurs at start up'*. This safety requirement is expressed as follows:

$$[x, x = -1]\,\text{ff} \qquad (\varphi_4)$$

The symbolic action $(x, x = -1)$ defines the singleton set $\{-1\} \subset \mathbb{Z}$ of external system actions. This means that in order for server traces not to violate formula φ_4, they must start with actions $\alpha \notin \{-1\}$. The set of traces $1.(0.\mathbb{N})^\omega$ exhibited by q_1 satisfies this property, whereas $-1.\mathbb{Z}^\omega$ does not. ■

Example 3.2 (Necessity and possibility). Further to the stipulation of example 3.2, we require that *'the server is initialised with the identification token 1'*, expressed as:

$$[x, x = -1]\,\text{ff} \wedge \langle x, x = 1 \rangle\,\text{tt} \qquad (\varphi_5)$$

The conjunct $[x, x = -1]\,\text{ff}$ guards against traces of q_1 exhibiting failure when loading; $\langle x, x = 1 \rangle\,\text{tt}$ asserts that the trace exhibits 1 at start-up, indicating a successful initialisation of the server. Formula φ_5 is satisfied exactly by server traces of the form $1.\mathbb{N}^\omega$. ■

The symbolic actions of examples 3.1 and 3.2 define sets of external actions by specifying *literal* values (*e.g.* 1 and −1). Action sets can be more generally defined via constraint expressions that refer to other data variables within the same scope.

Example 3.3 (Recursion). Amongst the executions satisfying φ_5 are those where the server accidentally returns its identifier token 1 in reply to client requests. We, therefore, demand that *'the server private identification token* 1 *is not leaked in client replies'*. Formula φ_6 expresses this recursive property in a general way, *i.e.,* it does not hardcode the token value 1. Note that the Boolean constraint expressions $b = \mathsf{tt}$ are elided.

$$[x]\,\mathsf{max}X.\big([y]\,([z,x=z]\,\mathsf{ff}\wedge[z,x\neq z]X)\big) \qquad (\varphi_6)$$

The symbolic action (x,tt) in the first necessity defines the set of external actions $\mathbb{Z}$. Its binder, x, binds the variable x in $\mathsf{max}X.\big([y]\,([z,x=z]\,\mathsf{ff}\wedge[z,x\neq z]X)\big)$ (marked in φ_6). For some initial server action $\alpha\in\mathbb{Z}$, applying the substitution $[^{\alpha}/_{x}]$ to this continuation, followed by a single unfolding of the recursion variable, yields the residual formula:

$$[y]\Big([\overparen{z,\alpha=z}]\,\mathsf{ff}\wedge[\overparen{z,\alpha\neq z}]\,\mathsf{max}X.\big([y]\,([\overparen{z,\alpha=z}]\,\mathsf{ff}\wedge[\overparen{z,\alpha\neq z}]X)\big)\Big) \qquad (\varphi_6')$$

Necessity $[y]$ maps y to the second server action $\beta\in\mathbb{Z}$ in the trace, *i.e.,* $[^{\beta}/_{y}]$. Applying the substitution $[^{\beta}/_{y}]$ to $[z,\alpha=z]\,\mathsf{ff}$ and $[z,\alpha\neq z]\,\mathsf{max}X.\big([y]\,([z,\alpha=z]\,\mathsf{ff}\wedge[z,\alpha\neq z]X)\big)$ leaves both sub-formulae *unchanged*, since y binds no variables in either. For the third server action γ, the modalities $[z,\alpha=z]$ and $[z,\alpha\neq z]$ map z to γ. Formula φ_6 is violated, ff, when the constraint $\alpha=z[^{\gamma}/_{z}]$ holds. Crucially, a *fresh* scope for data variables is created upon each unfolding of X, such that y and z can be mapped to new values. By contrast, the value in x is substituted for *once* in φ_6' and remains fixed when X is unfolded.

Formula φ_6 compares actions at *every* odd position in the trace against the one at the head. When φ_6 is interpreted over all the possible traces that the token server generates upon successful initialisation, the binder x in the modal construct $[x]$ becomes instantiated to the value 1. This ensures that, in particular, the set of traces $1.(0.\{\imath\in\mathbb{N}\mid\imath\neq 1\})^{*}.(0.1).\mathbb{N}^{\omega}$ are violating. Note that this property is not *not* expressible in LTL. ■

3.2 Synchronous Runtime Monitoring

Monitors may be viewed as processes via the syntax given in figure 3.2. This syntax differs from its regular counterpart of Aceto et al. [6, 8] in that it augments the prefixing construct with symbolic actions, (x,b) (*cf.* section 2.2). Besides the prefixing, external choice, and recursion constructs of CCS [178], the syntax of figure 3.2 includes disjunctive, $\oplus$, and conjunctive, $\otimes$, *parallel composition*. We use the symbol $\odot$ to refer to both $\oplus$ and $\otimes$ when needed. Monitor verdict states, $v\in\textsc{Vrd}$, are expressed as yes, no, and end respectively denoting the *accept*, *reject* and *inconclusive* verdicts.

Figure 3.2 outlines the behaviour of monitors, where the transition rules $\textsc{mRec}$, $\textsc{mChs}_{\mathrm{L}}$, and its symmetric case $\textsc{mChs}_{\mathrm{R}}$ (omitted), are standard. Rule $\textsc{mAct}$ describes the analysis that monitors perform, where the binder x in the symbolic action (x,b) is mapped to an external system action α, yielding the substitution $[^{\alpha}/_{x}]$ that is applied to the *decidable* Boolean constraint expression b. The monitor $(x,b).m$ analyses α *only if* the instantiated constraint $b[^{\alpha}/_{x}]$ is satisfied, whereupon α is substituted for the *free* occurrences of the variable x in the body m. When the premise $b[^{\alpha}/_{x}]$ does not hold, the monitor action

Monitor Syntax

$$m,n \in \textsc{Mon} ::= v \quad\mid\quad (x,b).m \quad\mid\quad m+n \quad\mid\quad m\oplus n \quad\mid\quad m\otimes n \quad\mid\quad \mathsf{rec}X.(m) \quad\mid\quad X$$

$$v \in \textsc{Vrd} ::= \mathsf{yes} \quad\mid\quad \mathsf{no} \quad\mid\quad \mathsf{end}$$

Monitor Small-Step Semantics

$$\textsc{mVrd}\;\frac{}{v \xrightarrow{\alpha} v} \qquad \textsc{mAct}\;\frac{b[{}^{\alpha}/_{x}] \Downarrow \mathsf{tt}}{(x,b).m \xrightarrow{\alpha} m[{}^{\alpha}/_{x}]} \qquad \textsc{mChs}_{\mathrm{L}}\;\frac{m \xrightarrow{\alpha} m'}{m+n \xrightarrow{\alpha} m'}$$

$$\textsc{mTau}_{\mathrm{L}}\;\frac{m \xrightarrow{\tau} m'}{m \odot n \xrightarrow{\tau} m' \odot n} \qquad \textsc{mPar}\;\frac{m \xrightarrow{\alpha} m' \qquad n \xrightarrow{\alpha} n'}{m \odot n \xrightarrow{\alpha} m' \odot n'} \qquad \textsc{mVrdE}\;\frac{}{\mathsf{end} \odot \mathsf{end} \xrightarrow{\tau} \mathsf{end}}$$

$$\textsc{mDisY}_{\mathrm{L}}\;\frac{}{\mathsf{yes} \oplus m \xrightarrow{\tau} \mathsf{yes}} \qquad \textsc{mDisN}_{\mathrm{L}}\;\frac{}{\mathsf{no} \oplus m \xrightarrow{\tau} m} \qquad \textsc{mConY}_{\mathrm{L}}\;\frac{}{\mathsf{yes} \otimes m \xrightarrow{\tau} m} \qquad \textsc{mConN}_{\mathrm{L}}\;\frac{}{\mathsf{no} \otimes m \xrightarrow{\tau} \mathsf{no}}$$

$$\textsc{mRec}\;\frac{}{\mathsf{rec}X.(m) \xrightarrow{\tau} m[{}^{\mathsf{rec}X.(m)}/_{X}]}$$

Monitor Instrumentation

$$\textsc{iMon}\;\frac{p \xrightarrow{\alpha} p' \qquad m \xrightarrow{\alpha} m'}{m \triangleleft p \xrightarrow{\alpha} m' \triangleleft p'} \qquad \textsc{iTer}\;\frac{p \xrightarrow{\alpha} p' \qquad m \not\xrightarrow{\alpha} \qquad m \not\xrightarrow{\tau}}{m \triangleleft p \xrightarrow{\alpha} \mathsf{end} \triangleleft p'}$$

$$\textsc{iAsyP}\;\frac{p \xrightarrow{\tau} p'}{m \triangleleft p \xrightarrow{\tau} m \triangleleft p'} \qquad \textsc{iAsyM}\;\frac{m \xrightarrow{\tau} m'}{m \triangleleft p \xrightarrow{\tau} m' \triangleleft p}$$

Figure 3.2. *Syntax, small-step semantics for parallel monitors, and synchronous instrumentation*

α is disabled. Verdict irrevocability is modelled by mVrd, where once in a verdict state v, any action can be analysed by monitors without altering v. Rule mPar enables parallel sub-monitors to transition in lock-step when they analyse the *same* action α, while mVrdE consolidates parallel inconclusive verdicts. The rest of the rules (omitting the obvious symmetric cases) cater to the internal reconfiguration of monitors. For instance, rules $\textsc{mDisY}_{\mathrm{L}}$ and $\textsc{mDisN}_{\mathrm{L}}$ state that in disjunctive parallelism, yes supersedes the verdicts of other monitors, whilst no does not affect the verdicts of other monitors; $\textsc{mConY}_{\mathrm{L}}$ and $\textsc{mConN}_{\mathrm{L}}$ express the dual case for parallel conjunctions. Finally, $\textsc{mTau}_{\mathrm{L}}$ and its symmetric analogue permit sub-monitors to execute internal reconfigurations independently.

Monitors execute together with the SuS to analyse its actions. Figure 3.2 recalls the instrumentation transition relation defined in Aceto et al. [6] that composes a monitor m with a system process p to yield a *monitored system*, denoted as $m \triangleleft p$. The relation $\triangleleft$ is parametric w.r.t. the transition semantics of processes and monitors, providing the latter supports the inconclusive verdict end. This instrumentation definition gives monitors a *passive* role, whereby $m \triangleleft p$ transitions via an external action only when p transitions with that action. Rules iMon and iTer capture this notion. iMon describes the *analysis* that monitors perform. It dictates that whenever a process p transitions via α to some p' and the monitor can analyse α and transition to m', the monitored system transitions in lock-step to $m' \triangleleft p'$. Monitors that are unable to analyse actions, nor unfold internally, are *terminated* by the instrumentation with an inconclusive state, as iTer states (note that iTer still permits the system process to resume its execution). The remaining rules, iAsyP and iAsyM, enable system and monitor processes to transition internally.

Example 3.4 (Synchronous instrumentation). The monitor $(x, x=1).\mathsf{rec}X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)$ that rejects traces of the form $1.0^{*}.1.\mathbb{Z}^{\omega}$, is instrumented with the server of figure 3.1. When the server leaks its identification token 1, this monitor reaches a rejection verdict along the transitions:

$$
\begin{aligned}
& (x,x=1).\mathsf{rec}X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big) \triangleleft 1.\mathsf{rec}X.(0.\imath.X)+-1.\mathsf{rec}\,Y.(\jmath.Y) \\
\xrightarrow{1}\; & \mathsf{rec}X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big) \triangleleft \mathsf{rec}X.(0.\imath.X) \\
\Longrightarrow\; & (y,y=0).\mathsf{rec}X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)+(y,y=1).\mathsf{no} \triangleleft 0.\imath.\mathsf{rec}X.(0.\imath.X) \\
\xrightarrow{0}\; & \mathsf{rec}X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big) \triangleleft \imath.\mathsf{rec}X.(0.\imath.X) \\
\xrightarrow{\tau}\; & (y,y=0).\mathsf{rec}X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)+(y,y=1).\mathsf{no} \triangleleft \imath.\mathsf{rec}X.(0.\imath.X) \\
\xrightarrow{1}\; & \mathsf{no} \triangleleft \mathsf{rec}X.(0.\imath.X) \xrightarrow{\tau} \cdots
\end{aligned}
$$

However, for a different execution where the server replies to a client with the identification token 2, the same monitor flags an inconclusive verdict.

$$
\begin{aligned}
& (x,x=1).\mathsf{rec}X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big) \triangleleft 1.\mathsf{rec}X.(0.\imath.X)+-1.\mathsf{rec}\,Y.(\jmath.Y) \\
\overset{1.0.2}{\Longrightarrow}\; & \mathsf{end} \triangleleft \mathsf{rec}X.(0.\imath.X) \xrightarrow{\tau} \cdots
\end{aligned}
$$

The concluding transition, $\cdots \xrightarrow{2} \mathsf{end} \triangleleft \mathsf{rec}X.(0.\imath.X)$, is obtained via the rule iTer, at which point the token value 2 issued by the server cannot be analysed by the monitor (it can only analyse either the action 0 or 1). Observe that the monitor does not interfere with the operation of the server. Henceforth, the instrumented system transitions exclusively through iMon, whereby any action that the server exhibits is analysed by the monitor (rule mVrd) which *persists* in flagging the same verdict end. Rule mVrd enables our monitors to meet the verdict irrevocability requirement (i) of definition 2.1. ■

3.3 Monitorable Logic Fragments

Accept and reject verdicts establish the monitoring counterpart to satisfactions and violations of $\mu\mathrm{HML}^{\mathrm{D}}$ formulae. Our definition of the accept and reject predicates, acc and rej, from definition 2.1 is given for finfinite (*i.e.,* finite or infinite) traces. Since the linear-time semantics of the $\mu\mathrm{HML}^{\mathrm{D}}$ is defined over *infinite* traces, we instantiate definition 2.1 of chapter 2 w.r.t. to this domain using our operational model of monitors and the instrumentation relation of figure 3.2.

Definition 3.1 (Linear-time acceptance and rejection [6, adapted from Definition 3.3]). A monitor m,

(i) for every process p and finite prefix s:

- accepts (resp. rejects) p along s, denoted as $\mathrm{acc}(m,p,s)$ (resp. $\mathsf{rej}(m,p,s)$), if $m \triangleleft p \overset{s}{\Longrightarrow} \mathsf{yes} \triangleleft p'$ (resp. $m \triangleleft p \overset{s}{\Longrightarrow} \mathsf{no} \triangleleft p'$) for some p'

We say that 'm accepts s' to mean $\forall p.\, \mathrm{acc}(m,p,s)$, and '$m$ rejects s' to mean that $\exists p.\, \mathsf{rej}(m,p,s)$.

(ii) for every process p and infinite trace t:

- accepts (resp. rejects) t produced by p, denoted $\mathrm{acc}(m,p,t)$ (resp. $\mathsf{rej}(m,p,t)$), if $\exists s.\, \exists u$ such that $t=su$ and $\mathrm{acc}(m,p,s)$ (resp. $\mathsf{rej}(m,p,s)$)

We abuse notation and use $\mathsf{acc}(m,t)$ as a shorthand for $\mathsf{acc}(m,p,t)$; similarly, $\mathsf{rej}(m,t)$ is used to denote $\mathsf{rej}(m,p,t)$. ■

Our concrete formalisation of soundness that instantiates definition 2.2 of chapter 2 uses the predicate sat given earlier in definition 2.4. Recall that the predicate $\mathsf{sat}(\varphi,t)$ determines whether an infinite trace t satisfies the linear-time μHML$^{\mathrm{D}}$ formula φ, *i.e.,* $t \in [\![\varphi]\!]$. We restate the soundness as follows.

Definition 3.2 (Linear-time soundness [6, adapted from Definition 4.1]). A monitor m is *sound* for a linear-time formula $\varphi \in \mu\mathrm{HML}^{\mathrm{D}}$ if, for every infinite trace t:

- $\mathsf{acc}(m,t)$ implies $t \in [\![\varphi]\!]$, and
- $\mathsf{rej}(m,t)$ implies $t \notin [\![\varphi]\!]$. ■

As section 2.1.3 argues, soundness is the least requirement expected from RV monitors since it ensures that verdicts reached by monitors do not contradict the corresponding logic semantics. Recall that different grades of completeness may be deemed adequate (refer to section 2.1.3), depending on the requirements of RV set-up. These requirements inform the definition of monitorability that identifies the logic fragments that can be accordingly runtime checked. We focus on partially-complete monitors which are satisfaction-complete or violation-complete for the formulae they monitor for, but are not required to be both.

Definition 3.3 (Linear-time completeness [6, adapted from Definition 4.1]). A monitor m for a linear-time formula $\varphi \in \mu\mathrm{HML}^{\mathrm{D}}$ and for every trace t is,

- *satisfaction-complete* if $t \in [\![\varphi]\!]$ implies $\mathsf{acc}(m,t)$, and
- *violation complete* if $t \notin [\![\varphi]\!]$ implies $\mathsf{rej}(m,t)$.

A monitor m is *complete* for a linear-time formula φ if it is both satisfaction-complete and violation-complete for φ, and *partially-complete* if it is either. ■

Monitorability for linear-time μHML$^{\mathrm{D}}$ formulae follows from definitions 3.2 and 3.3.

Definition 3.4 (Monitorability [6, adapted from Definition 4.10]). A formula $\varphi \in \mu\mathrm{HML}^{\mathrm{D}}$ is monitorable for *satisfaction* (resp. *violation*) iff there exists a monitor m that is a sound and satisfaction-complete (resp. violation-complete) monitor for φ. Formula m is *partially-monitorable* when it is monitorable for satisfaction or for violation. ■

Definition 3.5 gives the two fragments of the linear-time μHMLwith data (μHML$^{\mathrm{D}}$) that are partially monitorable [6]: $\textsc{min}$HML$^{\mathrm{D}}$, which is monitorable for satisfaction, and $\textsc{max}$HML$^{\mathrm{D}}$, which is monitorable for violation. Our definition shows the fragments extended with the data predicates presented in section 2.2 for the μHML$^{\mathrm{D}}$.

Definition 3.5 ($\textsc{min}$ and $\textsc{max}$ fragments of the μHML$^{\mathrm{D}}$ [6, adapted from Definition 4.11]). The least and greatest fixed point monitorable fragments of the μHML$^{\mathrm{D}}$ are respectively:

$$\varphi,\psi \in \textsc{min}\mathrm{HML}^{\mathrm{D}} ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle x,b\rangle\varphi \mid [x,b]\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathsf{min}X.(\varphi) \mid X$$

$$\varphi,\psi \in \textsc{max}\mathrm{HML}^{\mathrm{D}} ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle x,b\rangle\varphi \mid [x,b]\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathsf{max}X.(\varphi) \mid X$$

Both fragments are *maximally-expressive*, *i.e.*, for any $\varphi \in \mu\text{HML}^{\text{D}}$, if φ monitorable for satisfaction (resp. violation), then there exists some $\psi \in \textsc{min}\text{HML}^{\text{D}}$ (resp. $\psi \in \textsc{max}\text{HML}^{\text{D}}$) such that $[\![\varphi]\!] = [\![\psi]\!]$. ■

This means that up to logical equivalence, $\textsc{min}\text{HML}^{\text{D}}$ is the largest fragment of the μHML^{D} that is monitorable for satisfactions; dually, $\textsc{max}\text{HML}^{\text{D}}$ is the largest fragment that is monitorable for violations.

Example 3.5 (Non-monitorable linear-time properties). The property *'the token server must eventually issue the identification token* 100*'*, expressible in $\textsc{min}\text{HML}^{\text{D}}$ as $\varphi_7 = \mathsf{min}X.(\langle x,x=100\rangle\mathsf{tt} \vee \langle x,x\neq 100\rangle X)$, is not monitorable for violations. For if it were, a monitor m_{φ_7} that runtime checks for φ_7 should be able to flag a violation after analysing some finite execution s that does not contain the token 100. However, our token server will always be in a position to extend any such witness s that m_{φ_7} observes with one new action that exhibits the value 100, which would satisfy φ_7. Formula φ_7 is nevertheless monitorable for satisfactions since the monitor only commits itself to flag a satisfaction once the token server provides the required witness. Dually, formula φ_6 of example 3.3 *i.e.*, $[x]\mathsf{max}X.\big([y]([z,x=z]\mathsf{ff} \wedge [z,x\neq z]X)\big)$, is not monitorable for satisfactions, since the server can always present the monitor m_{φ_6} for formula φ_6 with a violating trace continuation after m_{φ_6} flags a satisfaction.

The liveness LTL formula $\mathsf{GF}\varphi$ that describes the behaviour 'φ holds infinitely often' is not monitorable [36]. For if a corresponding monitor exists, then this must check that at every position in the execution, $\mathsf{F}\varphi$ holds. For any finite trace prefix s where $\mathsf{GF}\varphi$ is declared satisfied, s can be extended by one action, obliging the monitor to check for $\mathsf{F}\varphi$ anew. Note that $\mathsf{GF}\varphi$ is expressible in μHML^{D} as $\mathsf{max}X.\big(\mathsf{min}\,Y.(\varphi \vee \langle x\rangle Y) \wedge [x]X\big)$, but in neither of the monitorable fragments of definition 3.5. ■

The formulae seen thus far in examples 2.1, 2.2 and 3.1 to 3.3 are in $\textsc{max}\text{HML}^{\text{D}}$. We adopt $\textsc{max}\text{HML}^{\text{D}}$ in the sequel and chapter 4, noting that the forthcoming synthesis procedure of section 3.4 generates identical monitors from $\textsc{min}\text{HML}^{\text{D}}$ and $\textsc{max}\text{HML}^{\text{D}}$ formulae.

3.4 Monitor Synthesis

Our adaptation $(\!|-|\!)$ of the synthesis procedure for regular monitors [6, 8] is given in definition 3.6. It generates monitors for $\varphi \in \textsc{min}\text{HML}^{\text{D}} \cup \textsc{max}\text{HML}^{\text{D}}$, following the inductive structure of formulae. The translation for truth and falsehood, and the least and greatest fixed point and recursion variable constructs is direct; disjunction and conjunction are transformed to their parallel counterparts. Modal constructs map to *deterministic* external choices, where the left summand handles the case where a system action α is in the set described by the symbolic action (x,b), and the right summand, the case where α is *not* in this set. This embodies the duality of possibility and necessity: when α is not in the action set (x,b), the formula $\langle x,b\rangle\varphi$ is violated, whereas $[x,b]\varphi$ is trivially satisfied.

Definition 3.6 (Monitor synthesis procedure for $\textsc{min}\text{HML}^{\text{D}}$ and $\textsc{max}\text{HML}^{\text{D}}$).

$$
\begin{array}{ll}
(\!|\mathsf{tt}|\!) = \mathsf{yes} & (\!|\mathsf{ff}|\!) = \mathsf{no} \\
(\!|\langle x,b\rangle\varphi|\!) = (x,b).(\!|\varphi|\!) + (x,\neg b).\mathsf{no} & (\!|[x,b]\varphi|\!) = (x,b).(\!|\varphi|\!) + (x,\neg b).\mathsf{yes} \\
(\!|\varphi \vee \psi|\!) = (\!|\varphi|\!) \oplus (\!|\psi|\!) & (\!|\varphi \wedge \psi|\!) = (\!|\varphi|\!) \otimes (\!|\psi|\!) \\
\left.\begin{array}{l}(\!|\mathsf{min}X.(\varphi)|\!) \\ (\!|\mathsf{max}X.(\varphi)|\!)\end{array}\right\} = \mathsf{rec}X.((\!|\varphi|\!)) & (\!|X|\!) = X
\end{array}
$$

■

Definition 3.6 makes use of the disjunctive, $\oplus$, and conjunctive, $\otimes$, parallel composition constructs of figure 3.2. These constructs are a convenient calculus for building monitors in a compositional fashion, making it possible to view a monitor as a system of sub-monitors that check for different sub-formulae. One byproduct of this construction is that it facilitates the definition of our synthesis procedure and ensuing executable monitor code (see section 4.2). The parallel transition rules mDisY$_{\mathrm{L}}$, mDisN$_{\mathrm{L}}$, mConY$_{\mathrm{L}}$, and mConN$_{\mathrm{L}}$ (and their symmetric counterparts) obviate the need for the instrumentation rule iTer that terminates monitors, and consequently, the use of the monitor transition rule mVrdE and the inconclusive verdict end. Note that our model can handle formulae such as, $\langle x, x=1\rangle\mathsf{tt} \wedge \langle x, x\neq 1\rangle\mathsf{tt}$, where the monitor generated, $\big((x, x=1).\mathsf{yes} + (x, x\neq 1).\mathsf{no}\big) \otimes \big((x, x\neq 1).\mathsf{yes} + (x, x=1).\mathsf{no}\big)$, together with the rules (mConN$_{\mathrm{L}}$ and mConN$_{\mathrm{R}}$ in this case) make the verdict flagged (*i.e.,* no) in line with the semantics of the logic.

Our monitor model assumes an infinite domain of data elements that—combined with the variable binding and lexical scoping induced by symbolic actions—makes monitors *not possible* to determinise in general (see example 3.6). At runtime, the view of monitors is limited to a single finite trace prefix, one of the many possible paths the SuS takes while executing. We exploit this partial view and use parallel monitors as a best-effort strategy to unfold and lazily analyse the events for the *current* trace observed. This may be seen as 'determinising on the fly', and contrasts with static determinisation that computes all the possible paths that a monitor *can* take *a priori*, only to follow a specific one at runtime.

Parallel monitors naturally handle the scoping and binding of variables between different sub-monitor hierarchies by following the syntactic structure of formulae. The rules mDisY$_{\mathrm{L}}$, mDisN$_{\mathrm{L}}$, mConY$_{\mathrm{L}}$, mConN$_{\mathrm{L}}$, and their analogues ensure that the sub-monitor hierarchies that result from $\oplus$ and $\otimes$ are kept compact by terminating superfluous monitor branches. Using flat, automata-like approaches to manage the variable scoping and binding aspects (*e.g.* register automata [143, 124, 104]) makes it hard to reason about monitors compositionally. These challenges concerning data binding and scoping do not arise in the framework of Aceto et al. [6, 8] that study *regular* monitors.

Example 3.6 (Non-determinisable monitors). Consider the property about our token server of figure 3.1 stating that *'when the server behaves erratically, it always generates distinct error codes'.* This can be expressed as the maxHML$^{\mathrm{D}}$ formula:

$$[x, x=-1]\,\max X.\Big([y]\big(\max Y.([z, y=z]\,\mathsf{ff} \wedge [z, y\neq z]\,Y) \wedge X\big)\Big) \tag{φ_8}$$

Formula φ_8 cannot be synthesised into a monitor that is determinisable. The binder y binds the variables y inside the greatest fixed point $\max Y.([z,y=z]\,\mathsf{ff} \wedge [z,y\neq z]\,Y)$, creating a *dependency* between the inner scope under variable Y and the outer scope under X (marked by arrows). This dependency obliges the monitors to reserve an unbounded number of variables (y in φ_8), one for each action analysed. It is necessary so that the values *of all* the different instantiations of y can be compared against future values in the trace through the recursive sub-formula $\max Y.([z,y=z]\,\mathsf{ff} \wedge [z,y\neq z]\,Y)$. Unfolding φ_8 once highlights the variables y (α-renamed to y_1 and y_2 for clarity) that track every action in the execution.

$$\begin{aligned}&[y_1]\Big(\max Y.([z,y_1=z]\,\mathsf{ff}\wedge[z,y_1\neq z]\,Y)\wedge\\ &\qquad\qquad \max X.\Big([y_2]\big(\max Y.([z,y_2=z]\,\mathsf{ff}\wedge[z,y_2\neq z]\,Y)\wedge X\big)\Big)\Big) \qquad (\varphi'_8)\end{aligned}$$

Each of $y_1,y_2,\ldots$ is respectively instantiated with the server error code value carried by actions in a trace $\alpha_1,\alpha_2,\ldots$. This makes the size of the monitor dependent on the length of its input, which results in a monitor whose number of states can grow indefinitely. ∎

Example 3.7 (Parallel monitors). Synthesising formula φ_5 produces the monitor m_{φ_5}:

$$\begin{aligned}(\!|\varphi_5|\!) &= (\!|[x,x=-1]\,\mathsf{ff}\wedge\langle x,x=1\rangle\mathsf{tt}|\!) = (\!|[x,x=-1]\,\mathsf{ff}|\!)\otimes(\!|\langle x,x=1\rangle\mathsf{tt}|\!)\\ &= \big((x,x=-1).\mathsf{no}+(x,x\neq-1).\mathsf{yes}\big)\otimes\big((x,x=1).\mathsf{yes}+(x,x\neq 1).\mathsf{no}\big) \qquad (m_{\varphi_5})\end{aligned}$$

When analysing the server traces $-1.\mathbb{Z}^{\omega}$, monitor m_{φ_5} reduces to $\mathsf{no}\otimes\mathsf{no}$ via the rule mPar. Its premises are obtained by applying the mChs$_\mathrm{L}$ and mAct to the left sub-monitor, and mChs$_\mathrm{R}$ and mAct to the right sub-monitor, giving:

$$(x,x=-1).\mathsf{no}+(x,x\neq-1).\mathsf{yes}\xrightarrow{-1}\mathsf{no}\qquad\text{and}\qquad(x,x=1).\mathsf{yes}+(x,x\neq1).\mathsf{no}\xrightarrow{-1}\mathsf{no}$$

The monitor $\mathsf{no}\otimes\mathsf{no}$ then transitions internally, $\mathsf{no}\otimes\mathsf{no}\xrightarrow{\tau}\mathsf{no}$, via either mConN$_\mathrm{L}$ or mConN$_\mathrm{R}$. Analogously, m_{φ_5} reaches yes when analysing the server traces $1.\mathbb{N}^{\omega}$. Recall that from a verdict state, a monitor can *always* analyse future actions via mVrd, flagging the *same* outcome. The behaviour of m_{φ_5} corresponds to the property that φ_5 describes. ∎

Example 3.8 (Lazy unfolding). Consider the recursive monitor m_{φ_6} synthesised from φ_6:

$$(x).\mathsf{rec}X.\Big((y).\Big(\big((z,x=z).\mathsf{no}+(z,x\neq z).\mathsf{yes}\big)\otimes\big((z,x\neq z).X+(z,x=z).\mathsf{yes}\big)\Big)\Big) \qquad (m_{\varphi_6})$$

For the server traces $1.0.2.0.1.(0.\mathbb{N})^{\omega}$, m_{φ_6} instantiates the binder x to the value 1 at the head, and applies the substitution $[1/x]$ to the residual monitor, giving:

$$\mathsf{rec}X.\Big((y).\Big(\big((z,1=z).\mathsf{no}+(z,1\neq z).\mathsf{yes}\big)\otimes\big((z,1\neq z).X+(z,1=z).\mathsf{yes}\big)\Big)\Big) \qquad (m'_{\varphi_6})$$

Hereafter, m'_{φ_6} unfolds continually, ensuring that no action carries the value 1 observed at the head of the trace. At every even position, y is instantiated to 0, whereas the binders z in each of the parallel sub-monitors compare the value carried by actions occurring at odd trace positions against 1. Monitor m'_{φ_6} reaches the verdict no via these reductions:

$$\begin{aligned}m'_{\varphi_6}&\xrightarrow{\tau}(y).\Big(\big((z,1=z).\mathsf{no}+(z,1\neq z).\mathsf{yes}\big)\otimes\big((z,1\neq z).m'_{\varphi_6}+(z,1=z).\mathsf{yes}\big)\Big) &&(m''_{\varphi_6})\\ &\xrightarrow{0}\big((z,1=z).\mathsf{no}+(z,1\neq z).\mathsf{yes}\big)\otimes\big((z,1\neq z).m'_{\varphi_6}+(z,1=z).\mathsf{yes}\big) &&(m'''_{\varphi_6})\\ &\xrightarrow{2}\mathsf{yes}\otimes m'_{\varphi_6}\xrightarrow{\tau}m'_{\varphi_6}\xrightarrow{\tau}m''_{\varphi_6}\xrightarrow{0}m'''_{\varphi_6}\xrightarrow{1}\mathsf{no}\otimes\mathsf{yes}\xrightarrow{\tau}\mathsf{no}\xrightarrow{\iota}\mathsf{no}\xrightarrow{\iota}\cdots\end{aligned}$$

For the satisfying server traces $1.(0.\{\imath \in \mathbb{N} \mid \imath \neq 1\})^{\omega}$, m'_{φ_6} visits the state $\text{yes} \otimes m'_{\varphi_6}$ indefinitely, where m'_{φ_6} supersedes the uninfluential verdict yes following the rule mConY_{L}. ■

Readers may find it instructive to consult the definition of satisfaction-complete and violation-complete monitors for the *branching-time* interpretation of the μHML, [6, Definition 5.1]. The latter definition demands that, whenever a monitor is presented with a satisfying (resp. violating) *process* state, it reaches an accept (resp. reject) verdict. Similarly, definition 3.3 above states that, whenever the monitor is presented with a satisfying (resp.violating) trace, it reaches an accept (resp. reject) verdict. Yet, there is a subtle distinction in the way the execution trace of the SuS is interpreted. In the branching-time setting, where the logic describes properties of *execution graphs*, a monitor may not reach an acceptance (or rejection) verdict about some φ. This happens when the current execution of the SuS does not provide evidence of satisfying (or violating) behaviour such that it enables the monitor to come to a definitive conclusion. In such cases, the monitor withholds its judgement (by flagging end) since there might be other unseen executions of the SuS that possibly contain the evidence required. By contrast, the linear-time interpretation of μHML^{D} concerns the *current execution.* The current execution provides the monitors that we synthesise from our monitorable fragments (see definition 3.6) with sufficient information to enable them to always flag a satisfaction or violation verdict.

3.5 Asynchronous Runtime Monitoring

The instrumentation relation of figure 3.2 is synchronous [118, 6], entwining the monitors and SuS such that the monitored system, $m \triangleleft p$, evolves as a single entity. Synchronous instrumentation is often implemented as inlined monitor code and is the *de facto* technique for monolithic settings where systems execute in a single thread. For the reasons given in section 2.1.4, the benefits of inlining are less suited to reactive settings where the SuS is comprised of multiple independently executing processes. Asynchronous instrumentation decouples monitors from SuS by introducing an intermediary buffer (or *queue*) where trace events can be deposited in *non-blocking* fashion. Decentralised monitoring set-ups replicate this arrangement: *each* monitor is equipped with a uniquely-addressable queue that it uses to analyse events *independent* of other monitors and out-of-sync with the SuS. This makes the technique less invasive and limits the side effects of monitors, *e.g.* a slow analysis does not impede the system from resuming its execution. Outline monitors are an embodiment of this approach, where individual monitor queues are connected to the tracing infrastructure that provides independent streams of system events (see section 2.1.4). One feature distinguishing synchronous and asynchronous instrumentation is that the latter form gives rise to multiple executions as a consequence of separating the monitor and SuS processes. Chapter 5 details the complications that arise in asynchronous decentralised set-ups and gives an algorithm that guarantees the correct order of events for each monitor. In this section, we reformulate the instrumentation semantics of figure 3.2 and define the asynchronous interaction between monitors and system processes.

Figure 3.3 gives the transition rules for our asynchronous instrumentation relation, $m \triangleleft \kappa \triangleleft p$. It assumes a FIFO queue of unbounded length, κ. We use the *cons* operator : and write $\alpha : \kappa$ to denote a queue of arbitrary length with head α, and $\kappa : \alpha$ to denote a queue of arbitrary length with α at its tail. An empty queue is denoted by ε. Same as the instrumentation given in [118, 6], our relation $m \triangleleft \kappa \triangleleft p$ is parametric w.r.t. the transition semantics of processes and monitors, *also* relegating monitors to a passive role. The

$$\textsc{aiPrc}\ \frac{p \xrightarrow{\alpha} p'}{m \triangleleft \kappa \triangleleft p \xrightarrow{\alpha} m \triangleleft \kappa{:}\alpha \triangleleft p'} \qquad \textsc{aiMon}\ \frac{m \xrightarrow{\alpha} m'}{m \triangleleft \alpha{:}\kappa \triangleleft p \xrightarrow{\tau} m' \triangleleft \kappa \triangleleft p}$$

$$\textsc{aiAsyP}\ \frac{p \xrightarrow{\tau} p'}{m \triangleleft \kappa \triangleleft p \xrightarrow{\tau} m \triangleleft \kappa \triangleleft p'} \qquad \textsc{aiAsyM}\ \frac{m \xrightarrow{\tau} m'}{m \triangleleft \kappa \triangleleft p \xrightarrow{\tau} m' \triangleleft \kappa \triangleleft p}$$

Figure 3.3. *Small-step semantics for asynchronous instrumentation*

rules aiPrc and aiMon capture the asynchronous operation of the system and monitors. Rule aiPrc always enables system processes p to transition to some p' via an action α that is deposited in the queue, *i.e.,* $\kappa{:}\alpha$. An action α from the queue $\alpha : \kappa$ is taken out by a monitor whenever it can analyse α, transitioning silently to m', as aiMon indicates. The remaining rules, aiAsyP and aiAsyM, allow processes and monitors to transition internally.

The rules of figure 3.3 highlight the minimal interference that monitors have. Rule aiPrc states that a monitored system $m \triangleleft \kappa \triangleleft p$ exhibits actions *as soon as* processes perform them; monitors, however, conduct their asynchronous analysis *silently.* One *may* consider an alternative formulation of aiPrc and aiMon that reverses the roles of processes and monitors in the monitored system $m \triangleleft \kappa \triangleleft p$. In this definition, processes can exhibit external actions α via aiPrc, but contrary to our rules of figure 3.3, the monitored system transitions internally, *i.e.,* $m \triangleleft \kappa \triangleleft p \xrightarrow{\tau} m \triangleleft \kappa{:}\alpha \triangleleft p'$. The conclusion of rule aiMon would then state that the monitored system emits external system actions only when these have been analysed by monitors, *i.e.,* $m \triangleleft \alpha{:}\kappa \triangleleft p \xrightarrow{\alpha} m' \triangleleft \kappa \triangleleft p$. While this variation seems innocuous (asynchrony is still preserved), the rules subtly alter the behaviour of the monitored system. Concretely, slowdowns (or deadlocks) in monitors delay (or prevent) the monitored system from reporting actions to the external environment promptly. This counters our aim of fully decoupling the SuS and monitors to induce minimal interference.

Finally, the definitions of figure 3.3 omit the analogue to iTer (*cf.* figure 3.2), which terminates monitors that cannot analyse events or transition internally. Rule iTer is required in a *synchronous* setting, otherwise, the system cannot progress when the monitor is stuck. From a formal standpoint, eliding this rule in the asynchronous case does not affect the overall behaviour, as the system can progress regardless of whether a monitor is terminated or cannot analyse actions. Yet, terminating redundant monitors is crucial for *implementing* tools that minimise the performance impact monitors have on the system. Rule aiTer below accomplishes this task, providing a basis upon which the garbage collection in our decentralised instrumentation algorithm of chapter 5 is built.

$$\textsc{aiTer}\ \frac{m \overset{\alpha}{\not\longrightarrow} \qquad m \overset{\tau}{\not\longrightarrow}}{m \triangleleft \alpha{:}\kappa \triangleleft p \xrightarrow{\tau} \mathsf{end} \triangleleft \varepsilon \triangleleft p}$$

Example 3.9 (Asynchronous instrumentation). Monitor $(x, x=1).\mathsf{rec}X.\big((y, y=0).X + (y, y=1).\mathsf{no}\big)$ from example 3.9 is instrumented with the token server of figure 3.1. When the server leaks its identification token 1, a rejection verdict can be reached as follows:

$$\begin{aligned}
&(x,x=1).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft\varepsilon\triangleleft 1.\mathsf{rec}\,X.(0.\imath.X)+-1.\mathsf{rec}\,Y.(\jmath.Y)\\
\xrightarrow{1}\;&(x,x=1).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft\varepsilon{:}1\triangleleft\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft\varepsilon\triangleleft\mathsf{rec}\,X.(0.\imath.X)\\
\xRightarrow{\tau}\;&(y,y=0).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)+(y,y=1).\mathsf{no}\triangleleft\varepsilon\triangleleft 0.\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{0}\;&(y,y=0).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)+(y,y=1).\mathsf{no}\triangleleft\varepsilon{:}0\triangleleft\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft\varepsilon\triangleleft\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&(y,y=0).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)+(y,y=1).\mathsf{no}\triangleleft\varepsilon\triangleleft\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{1}\;&(y,y=0).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)+(y,y=1).\mathsf{no}\triangleleft\varepsilon{:}1\triangleleft\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&\mathsf{no}\triangleleft\varepsilon\triangleleft\mathsf{rec}\,X.(0.\imath.X)\xrightarrow{\tau}\cdots
\end{aligned}$$

This transition sequence depicts the case where the token server advances by *one* step, and waits for the monitor to catch up and analyse the action deposited in the queue κ before proceeding with the next transition (*i.e.,* κ emulates a single-place buffer). It is but one of various interleaved executions that the monitored system can exhibit. We give it to elucidate how the intermediary queue κ that decouples the token server from its monitor, evolves as the latter effects its analysis. The execution obtained is similar to the synchronous run given in example 3.4, albeit interleaved with extra internal transitions performed by the monitor to reach a state where it is ready to analyse the next action. The following run shows the token server executing ahead and completing one request-response cycle before the monitor commences its analysis:

$$\begin{aligned}
&(x,x=1).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft\varepsilon\triangleleft 1.\mathsf{rec}\,X.(0.\imath.X)+-1.\mathsf{rec}\,Y.(\jmath.Y)\\
\xrightarrow{1}\;&(x,x=1).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft\varepsilon{:}1\triangleleft\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&(x,x=1).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft 1\triangleleft 0.\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{0}\;&(x,x=1).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft 1{:}0\triangleleft\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{1}\;&(x,x=1).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft 1.0{:}1\triangleleft\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&(x,x=1).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft 1.0.1\triangleleft 0.\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft 0.1\triangleleft 0.\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&(y,y=0).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)+(y,y=1).\mathsf{no}\triangleleft 0.1\triangleleft 0.\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)\triangleleft 1\triangleleft 0.\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&(y,y=0).\mathsf{rec}\,X.\big((y,y=0).X+(y,y=1).\mathsf{no}\big)+(y,y=1).\mathsf{no}\triangleleft 1\triangleleft 0.\imath.\mathsf{rec}\,X.(0.\imath.X)\\
\xrightarrow{\tau}\;&\mathsf{no}\triangleleft\varepsilon\triangleleft 0.\imath.\mathsf{rec}\,X.(0.\imath.X)\xrightarrow{0}\cdots
\end{aligned}$$

The last five τ-transitions showcase asynchronous instrumentation, which permits monitors to analyse the events accumulated in the queue independently of the server. Yet, the price of this benefit is paid in terms of possible delays when flagging verdicts. ■

Example 3.10 (Monitor termination). For an alternate run where the token server emits the trace 1.0.2.0.3…, the monitor of examples 3.4 and 3.9 gets stuck, as it cannot analyse actions that carry values other than 0 or 1. While unanalysed actions accumulate in the instrumentation queue, the server execution is not hampered from progressing.

$$\begin{aligned}
&(x, x=1).\mathsf{rec}\,X.\big((y, y=0).X + (y, y=1).\mathsf{no}\big) \triangleleft \varepsilon \triangleleft 1.\mathsf{rec}\,X.(0.\imath.X) + -1.\mathsf{rec}\,Y.(\jmath.Y)\\
\overset{1.0.2}{\Longrightarrow}\ &(y, y=0).\mathsf{rec}\,X.\big((y, y=0).X + (y, y=1).\mathsf{no}\big) + (y, y=1).\mathsf{no} \triangleleft 2 \triangleleft \mathsf{rec}\,X.(0.\imath.X)\\
\overset{0.3}{\Longrightarrow}\ &(y, y=0).\mathsf{rec}\,X.\big((y, y=0).X + (y, y=1).\mathsf{no}\big) + (y, y=1).\mathsf{no} \triangleleft 2.0.3 \triangleleft \mathsf{rec}\,X.(0.\imath.X) \xrightarrow{\tau} \cdots
\end{aligned}$$

In practice, the steadily increasing queue size is detrimental to the runtime performance, and rule aiTer is used to prematurely terminate the stuck monitor.

$$\begin{aligned}
&(x, x=1).\mathsf{rec}\,X.\big((y, y=0).X + (y, y=1).\mathsf{no}\big) \triangleleft \varepsilon \triangleleft 1.\mathsf{rec}\,X.(0.\imath.X) + -1.\mathsf{rec}\,Y.(\jmath.Y)\\
\overset{1.0.2}{\Longrightarrow}\ &\mathsf{end} \triangleleft \varepsilon \triangleleft \mathsf{rec}\,X.(0.\imath.X) \xrightarrow{\tau} \cdots
\end{aligned}$$

In principle, the asynchrony of our instrumentation naturally safeguards the SuS from problematic monitors (*e.g.* the divergent monitors explored in [112]). Observe that this does not necessarily apply in practice. For instance, events can accumulate in the queue when the monitor is slow to analyse them. This can induce considerable overhead that indirectly affects the applications being monitored. Section 7.2.2 demonstrates such an occurrence, wherein an asynchronous centralised monitor is inefficient to the point that it crashes the SuS. ■

3.6 Discussion

Organising the RV set-up into distinct components with cleanly delineated responsibilities is the core theme of this chapter. The formalism, in our case, a logic, provides a language through which properties can be expressed *independently* of the underlying verification technique. RV monitors are *instrumented* with the SuS and tasked with runtime checking properties against the trace that the system exhibits while executing. Monitorability bridges these two aspects: satisfactions and violations of properties in the logic on the one hand, and acceptances and rejections flagged by monitors on the other [8]. It establishes what it means for a monitor to be correct, which in turn, determines the fragments of the logic that can be runtime checked. This correspondence between these two distinct semantics can be *mechanised* into an automated synthesis procedure that generates *correct* monitors from logic formulae [113].

This modular design [6, 8] is reflected in our approach. We choose a logic—the highly-expressive linear-time μHML$^{\mathrm{D}}$ that describes properties of the current execution—and show how properties that reason on the data carried by trace events can be flexibly specified. We establish an operational model of parallel monitors [6] extended with data predicates, that fulfils two requirements, namely that (i) monitors analyse finite trace *prefixes*, and (ii) produce *irrevocable* accept or reject verdicts about these traces. Together with the instrumentation relation [6], this model suffices to concretely define the notions of trace acceptance and rejection, given by the predicates acc and rej. Monitor soundness and completeness are specified in terms of acc and rej, and a definition of monitorability for *partially-complete* monitoring follows as a result. Our compositional synthesis procedure translates *monitorable* linear-time μHML$^{\mathrm{D}}$ fragments to parallel monitors comprised of sub-monitors that check for corresponding sub-formulae. We define an *asynchronous* instrumentation relation alternative to the one of Aceto et al. [6, 8] to decouple the execution of the monitors and SuS. Our definition follows the same assumptions as their synchronous instrumentation, making it *compatible* with that framework.

One distinct advantage that this separation of concerns has over other bodies of work (*e.g.* [210, 67, 70, 68, 24, 63, 34, 197]) when it comes to tool construction is that every layer mentioned above is *directly* mappable to modular code. This provides high assurances that the correctness results obtained in theory are transferred to the implementation. Besides correctness, modularity makes it possible to *reuse* previously-established results and by extension, existing tools. For instance, our framework easily supports the monitorable fragments of the *branching-time* μHML^{D} since the respective synthesis procedure of [118, Definition 7] generates monitors described in a subset of the monitor calculus and operational semantics given in figure 3.2[1]. The instrumentation also benefits since the *same* synthesised monitor (code) can be instrumented with the SuS in synchronous or asynchronous modes. We highlight the indispensability of this aspect in section 4.7 and showcase it in chapter 7.

The asynchronous instrumentation we give in section 3.5 fits well the reactive systems setting. It keeps the SuS and monitors separate, in line with the concurrency-oriented programming tenets [19], where different responsibilities are organised into independent concurrent units. This fine-grained concurrent design increases the potential for parallelisation since the monitor code is not embedded into the SuS. Our monitored system, $m \triangleleft \kappa \triangleleft p$, that results from asynchronous instrumentation preserves the reactive qualities of the uninstrumented SuS:

- the queue κ enables the SuS to execute without waiting on monitors (*responsive*, example 3.9),
- monitors can fail with minimal impacts on the SuS (*resilient*, example 3.10)
- monitors only analyse the events communicated by the instrumentation over the queue κ (*message-driven*, examples 3.9 and 3.10)

Asynchronous instrumentation also opens the possibility for the monitored system to exhibit *elastic* behaviour. While this is not evident in our simplified system set-ups of examples 3.9 and 3.10, we detail how elasticity is attained via our decentralised monitoring algorithm of chapter 5.

[1] This approach is, in fact, already implemented in the detectEr tool.

4 Runtime Monitoring

Developing the theoretical foundations of runtime monitoring in a modular approach provides a blueprint against which RV tools can be systematically implemented and evolved. As section 3.6 argues, delineating the key components of the RV set-up not only facilitates their translation to code with minimal adaptation but gives increased assurances that such translations are correct. In addition, limiting the assumptions that each RV aspect makes on others (*e.g.* adopting a general logic that embeds other less-expressive ones, decoupling the logic from the verification method, using a common monitor calculus, *etc.*) makes it possible to reuse existing results and tools to assemble verification set-ups that suit particular requirements. This chapter details how each RV aspect of the model developed in chapter 3 can be mapped into its implementation equivalent. Figure 4.1a outlines the different components of our theoretical set-up and their implementation counterparts we present in this chapter, figure 4.1b (highlighted). While Erlang is our implementation language of choice (see discussion in section 1.2), the techniques in this chapter are not particularly tied to actor-oriented frameworks (*e.g.* Akka), but can also be applied to monolithic programs (Java, Python, *etc.*). We:

(i) augment the notion of symbolic actions given in section 2.2 with pattern matching, enabling the logic and monitors to reason on composite data types, which we use to define a simple model of events that capture the actions performed by processes, Section 4.1;

(ii) concretise the synthesis procedure stated in definition 3.6 to produce executable Erlang monitor code, Section 4.2;

(iii) encode the small-step rules given in figure 3.2 as an algorithm that operates on monitors generated by our synthesis, Section 4.3;

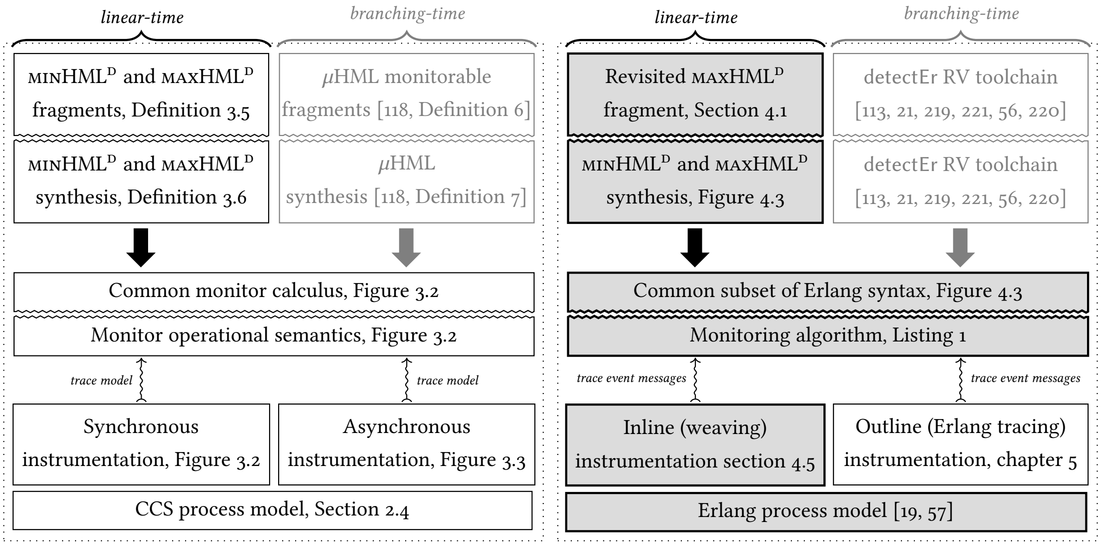
(iv) generalise the synchronous and asynchronous instrumentation relations of figures 3.2 and 3.3 to support selective process instrumentation, Section 4.4;

(v) detail an implementation of the synchronous instrumentation definition of (iv) based on source-level weaving, Section 4.5.

Our subsequent case study in section 4.6 demonstrates how properties can be flexibly specified to instrument and runtime check third-party concurrent applications built on top of the Erlang OTP middleware.

4.1 Revisiting the Data Model

We revise our definition of symbolic actions introduced section 2.2 to fit the Erlang use-case, where data can consist of composite types, such as tuples and lists [19, 57]. Let $\ell \in \mathcal{L}$ be a finite set of *action labels*, and $d_1, d_2, \ldots$ be data values taken from a set of data domains, $\mathcal{D} = \bigcup_{i \in \mathbb{N}} \mathbb{D}_i$ (*e.g.* integers, PIDs, tuples,


linear-time
branching-time
minHML^D and maxHML^D fragments, Definition 3.5
μHML monitorable fragments [118, Definition 6]
minHML^D and maxHML^D synthesis, Definition 3.6
μHML synthesis [118, Definition 7]
Common monitor calculus, Figure 3.2
Monitor operational semantics, Figure 3.2
trace model
trace model
Synchronous instrumentation, Figure 3.2
Asynchronous instrumentation, Figure 3.3
CCS process model, Section 2.4
linear-time
branching-time
Revisited maxHML^D fragment, Section 4.1
detectEr RV toolchain [113, 21, 219, 221, 56, 220]
minHML^D and maxHML^D synthesis, Figure 4.3
detectEr RV toolchain [113, 21, 219, 221, 56, 220]
Common subset of Erlang syntax, Figure 4.3
Monitoring algorithm, Listing 1
trace event messages
trace event messages
Inline (weaving) instrumentation section 4.5
Outline (Erlang tracing) instrumentation, chapter 5
Erlang process model [19, 57]

(a) *Modular theoretical RV set-up of chapters 2 and 3*

(b) *Implementation components reflecting the modules of 4.1a*

Figure 4.1. *Theoretical and corresponding implementation RV set-ups*

lists, *etc.*). An external action, α, is redefined as a tuple, $\langle \ell, d_2, \ldots, d_n \rangle$, where the first element $d_1 = \ell$ is the label of α and $d_2, \ldots, d_n$ is the *data payload* carried by α. We use the notation $\ell\langle d_2, \ldots, d_n \rangle$ to write α.

Patterns, $e \in \textsc{Pat}$, are counterparts to external system actions. These are defined as tuples, $\langle \ell, x_2, \ldots, x_n \rangle$ (written as $\ell\langle x_2, \ldots, x_n \rangle$), where $x_2, \ldots, x_n$ are *pairwise-distinct* data variables names ranging over $\mathcal{D}$. Our revised definition of symbolic actions in the modal constructs $\langle e, b \rangle \varphi$ and $[e, b]\varphi$ uses these patterns instead of variables (*cf.* section 2.2). The binders $x_2, \ldots, x_n$ in e bind the free occurrences of $x_2, \ldots, x_n$ in the Boolean constraint b, and in the continuation φ. We define the function, $\mathsf{match}(e, \alpha)$, to handle *pattern matching*. This function returns a substitution, $\pi : \textsc{DVar} \rightharpoonup \mathcal{D}$, that maps the variables in e to the corresponding data values in the payload carried by α when the shape of the pattern matches that of the action, or $\bot$ if the match is unsuccessful. Analogous to the symbolic actions of section 2.2, (e, b) describes a set of actions. An action α is in this set if (i) the patten match succeeds, *i.e.*, $\mathsf{match}(e, \alpha) = \pi$, and (ii) the *instantiated* Boolean constraint expression $b\pi$ holds.

We use the action label set $\mathcal{L} = \{\rightarrowtriangle, \leftarrowtriangle, *, !, ?\}$, that captures the lifecycle of, and interaction between processes. The *fork* action, $\rightarrowtriangle$, is exhibited by a process when it creates a child; its dual, $\leftarrowtriangle$, is exhibited by the child process upon *initialisation*. An *exit* action, $*$, signals process termination; *send* and *receive*, respectively $!$ and $?$, denote interaction. Table 4.1 details the actions related to these labels and the data payload they carry.

Our token server of figure 3.1 is readily translatable to Erlang, as figure 4.2 shows. The server starts when its main function, `loop`, in the Erlang module `ts` is invoked (state q_1, line 2). From q_1, it transitions to q_3 (line 4), exhibiting the initialisation event $\leftarrowtriangle\langle \mathtt{PID_S}, \mathtt{PID_P}, \mathtt{ts}, \mathtt{loop}, [1,2] \rangle$; the placeholders $\mathtt{PID_S}$ and $\mathtt{PID_P}$ respectively denote the PID values of the token server process and of the parent process forking the server. At q_3, the server accepts client requests, consisting of the tuple $\{\mathtt{PID_C}, \mathtt{0}\}$, where $\mathtt{PID_C}$ is the PID of the client, and `0`, the command requesting a new identification token, line 5. From state q_4, the server replies with the new token value *NextTok* on line 6, and transitions back to q_3. This client-server

Action α	Action pattern e	Variables	Description
fork	$\rightarrowtriangle\langle x_1,x_2,y_1,y_2,y_3\rangle$	x_1	PID of the parent process forking x_2
initialise	$\leftarrowtriangle\langle x_2,x_1,y_1,y_2,y_3\rangle$	x_2	PID of the child process forked by x_1
		y_1,y_2,y_3	Function signature forked by x_1
exit	$*\langle x_1,y_1\rangle$	x_1	PID of the terminated process
		y_1	Error datum, *e.g.* error reason, *etc.*
send	$!\,\langle x_1,x_2,y_1\rangle$	x_1	PID of the process sending the message
		x_2	PID of the recipient process
		y_1	Message datum, *e.g.* integer, tuple, *etc.*
receive	$?\langle x_2,y_1\rangle$	x_2	PID of the recipient process
		y_1	Message datum, *e.g.* integer, tuple, *etc.*

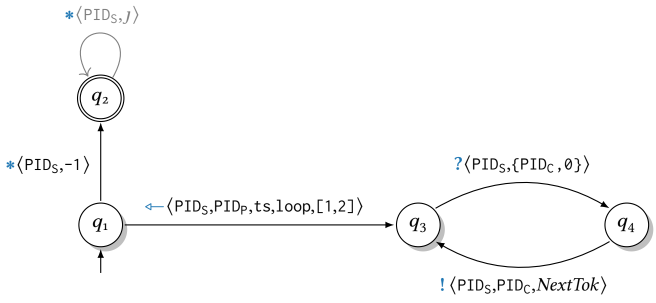
Table 4.1. *Actions capturing the behaviour exhibited by Erlang processes*

interaction emits the server events $?\langle\texttt{PID}_\texttt{S},\{\texttt{PID}_\texttt{C},\texttt{0}\}\rangle$ and $!\,\langle\texttt{PID}_\texttt{S},\texttt{PID}_\texttt{C},\mathit{NextTok}\rangle$. When the server fails at startup, it exhibits abnormal behaviour, shown as $*\langle\texttt{PID}_\texttt{S},\texttt{-1}\rangle$, and terminates, state q_3. Note that our translation of the server abstraction of figure 3.1 transforms the sink q_3 to a final state and removes its self-loop. This coincides with our token server implementation of figure 4.2b which exits when errors arise. While this adaptation prohibits the server from generating infinitely long executions, one may still interpret termination as the trace $-1.\mathbb{Z}^\omega$, indicating that once terminated, the server is permanently trapped in that state, q_2.

Example 4.1. (Pattern matching) Formula φ_5 can be reformulated to fit the implementation of figure 4.2:

$$[\,*\langle x_1,x_2\rangle, x_2=\texttt{-1}\,]\,\mathsf{ff}\wedge\langle\,\leftarrowtriangle\langle x_1,x_2,x_3,x_4,[x_5,y_6]\rangle, x_5=1\,\rangle\mathsf{tt} \tag{φ_9}$$

The patterns in the left and right conjuncts of φ_9 match the exit and initialisation events respectively. When q_1 crashes at start-up, $\texttt{match}(*\langle x_1,x_2\rangle,*\langle\texttt{PID}_\texttt{S},\texttt{-1}\rangle)$ yields the substitution $\pi=[{}^{\texttt{PID}_\texttt{S}}/_{x_1},{}^{\texttt{-1}}/_{x_2}]$, and the instantiated constraint $(x_2\texttt{=-1})\pi$ holds. For the same event, $\texttt{match}\big(\leftarrowtriangle\langle x_1,x_2,x_3,x_4,[x_5,x_6]\rangle,*\langle\texttt{PID}_\texttt{S},\texttt{-1}\rangle\big)=\bot$



$*\langle\texttt{PID}_\texttt{S},\_\rangle$
q_2
$*\langle\texttt{PID}_\texttt{S},\texttt{-1}\rangle$
q_1
$\leftarrowtriangle\langle\texttt{PID}_\texttt{S},\texttt{PID}_\texttt{P},\texttt{ts},\texttt{loop},\texttt{[1,2]}\rangle$
q_3
$?\langle\texttt{PID}_\texttt{S},\{\texttt{PID}_\texttt{C},\texttt{0}\}\rangle$
q_4
$!\,\langle\texttt{PID}_\texttt{S},\texttt{PID}_\texttt{C},\mathit{NextTok}\rangle$

(a) *Token server model updated with concrete Erlang process actions*

```
start(Tok) →
  spawn(ts, loop, [Tok, Tok + 1]).

loop(Tok, NextTok) when Tok = 1 →
  receive
  {Clt, 0} →
      Clt ! NextTok,
      loop(Tok, NextTok + 1)
  end.
```

(b) *Erlang implementation as module* `ts` *(excerpt)*

Figure 4.2. *Erlang adaptation of the token server of figure 3.1*

in the right conjunct, leading to a violation of formula φ_9. The reverse argument applies when q_1 loads successfully, where φ_9 is satisfied. In φ_9, the pattern variables x_1 in $*\langle x_1,x_2\rangle$, and x_1,x_2,x_3,x_4,x_6 in $\leftarrow\!\langle x_1,x_2,x_3,x_4,[x_5,x_6]\rangle$ are *redundant*.

$$[\leftarrow\!\langle \_,\_,\_,\_,[x_5,\_]\rangle]\,\mathsf{max}\,X.\Big([\_]\,([\,!\,\langle \_,\_,z_3\rangle, x_5 = z_3]\,\mathsf{ff} \wedge [\,!\,\langle \_,\_,z_3\rangle, x_5 \neq z_3]\,X)\Big) \qquad (\varphi_{10})$$

Formula φ_{10} restates φ_6 with pattern matching. It uses the 'don't care' pattern _, that matches *arbitrary* values, eliding redundant patterns and variables. ■

4.2 Synthesising Erlang Monitors

Our synthesis from maxHML$^{\mathrm{D}}$ specifications to executable Erlang monitors follows that of definition 3.6. Figure 4.3 omits the cases for the falsity, necessity, and conjunction constructs, as these are analogous to the ones for tt, $\langle e,b\rangle\varphi$ and $\varphi \vee \psi$. The translation from specifications to monitors is executed in three stages. First, a formula is parsed into its equivalent AST. This is then passed to the code generator that visits each of its nodes, mapping it to a *monitor description* as per the rules of figure 4.3. The monitor description is encoded as an Erlang AST to simplify its handling. In the final stage, this AST is processed by the Erlang compiler to emit the monitor source code or a BEAM [57] executable.

In this definition of $\llparenthesis - \rrparenthesis$, tt (resp. ff) is translated to the Erlang *atom* `yes` (resp. `no`) that indicates acceptance (resp. rejection). The remaining cases generate Erlang tuples whose first element, called the *tag*, is an atom that identifies the kind of monitor. Disjunctions (resp. conjunctions) are translated to the tuple tagged with `or` (resp. `and`), combining two sub-monitor descriptions. Greatest fixed point constructs, $\mathsf{max}X.(\varphi)$, are mapped to `rec` tuples consisting of *named* functions, `fun X()` $\rightarrow \llparenthesis\varphi\rrparenthesis$ `end`, that can be referenced by $\llparenthesis X\rrparenthesis$. Modal constructs are synthesised as a choice with *left* and *right* actions. An action tuple, `act`, combines a *predicate* function and an associated *monitor body* that is unfolded when the predicate is `true`. The predicate function encodes the pattern matching *and* Boolean constraint evaluation as one operation, using two *clauses*. Its first clause, `fun(`e`) when` b, tests the constraint b w.r.t. the variables in the pattern e that become *dynamically* instantiated with the data values carried by an action α at runtime. The second catch-all clause `(_)` covers the remaining cases, namely when: (i) either the action under analysis fails to match the pattern, or (ii) the pattern matches *but* the Boolean constraint does *not* hold. For the left action, the predicate clause `fun(`e`) when` b returns `true` when the

$$\llparenthesis \mathsf{tt} \rrparenthesis = \texttt{yes} \qquad\qquad \llparenthesis \varphi \vee \psi \rrparenthesis = \{\texttt{or},\llparenthesis\varphi\rrparenthesis,\llparenthesis\psi\rrparenthesis\}$$

$$\llparenthesis \mathsf{max}X.(\varphi) \rrparenthesis = \{\texttt{rec, fun X()} \rightarrow \llparenthesis\varphi\rrparenthesis\ \texttt{end}\} \qquad\qquad \llparenthesis X \rrparenthesis = \{\texttt{rec, X}\}$$

$$\llparenthesis \langle e,b\rangle\varphi \rrparenthesis = \begin{cases} \{\texttt{chs}, \\ \quad \left.\begin{array}{l} \{\texttt{act}, \overbrace{\texttt{fun}(e)\ \texttt{when}\ b \rightarrow \texttt{true};\ (\_) \rightarrow \texttt{false end}}^{\textit{predicate}}, \\ \texttt{fun}(e) \rightarrow \llparenthesis\varphi\rrparenthesis\ \texttt{end}\}, \end{array}\right\} \textit{left action} \\ \quad \left.\begin{array}{l} \{\texttt{act}, \texttt{fun}(e)\ \texttt{when}\ b \rightarrow \texttt{false};\ (\_) \rightarrow \texttt{true end}, \\ \underbrace{\texttt{fun}(\_) \rightarrow \texttt{no end}\}}_{\textit{monitor body}} \end{array}\right\} \textit{right action} \\ \} \end{cases}$$

Figure 4.3. *Translation from* maxHML^{D} *formulae to Erlang code (excerpt)*

pattern match and guard test succeed, and `false` otherwise, *i.e.,* `(_)`. This condition is inverted for the right action, modelling cases (i) and (ii) just described. Our encoding of the aforementioned predicate in terms of Erlang function clauses spares us from implementing the pattern matching and constraint evaluation mechanism. It also enables monitors to support most of the Erlang data types and its full range of Boolean constraint expression syntax [19]. For similar reasons, $\llparenthesis\langle e,b\rangle\varphi\rrparenthesis$ encodes the monitor body as `fun(`e`)` $\rightarrow$ $\llparenthesis\varphi\rrparenthesis$ `end` to delegate scoping to the Erlang language. This facilitates our synthesis and optimises the memory management of monitors by offloading this aspect onto the language runtime.

4.3 The Monitoring Algorithm

The synthesis procedure of definition 3.6 generates monitors that can runtime check formulae in parallel against the same position in the trace via disjunctive and conjunctive parallel composition. Our tool is however engineered to *emulate* parallel monitors, rather than forking processes and delegating their execution to the Erlang runtime. While the latter method tends to simplify the synthesis and runtime monitoring, we adopt the *former* approach for two reasons.

(i) Previous empirical evidence suggests that parallelising via processes can induce high overhead when the RV set-up is considerably scaled [219, 53]. A process-free design may render this overhead more manageable [10].

(ii) Emulating parallel monitors requires us to tease apart the synthesised monitor description from its operational semantics, which makes our set-up in line with the definitions of figure 3.2.

Our monitoring algorithm (listing 1) takes a monitor description m generated by $\llparenthesis-\rrparenthesis$, and performs successive reductions by applying m to events from the trace until a verdict is reached. Simultaneously, the algorithm maintains all the possible *active* states of the monitor as this is evolved from one state to the next. Listing 1 encodes this reduction strategy using a series of **case** statements (lines 2 to 15 and 20 to 32), following the operational semantics of figure 3.2. Each **case** maps the first part of a rule conclusion to a *pattern*, enabling the monitoring algorithm to unambiguously **match** the rule to apply. The body of **case**s consists of a **return** statement that corresponds to the outcome dictated by the rule. Rules with premises (*e.g.* mChs$_\text{L}$, mPar, *etc.*) are reduced *recursively* by reapplying rules until an axiom is met, whereas axioms (*e.g.* mVrd, mDisN$_\text{L}$, *etc.*) reduce immediately. For example, the pattern `{chs,` m`,` n`}` on line 7 specifies that mChs$_\text{L}$ and mChs$_\text{R}$ only apply to monitors of the form $m+n$. Selecting whether to reduce the left or right sub-monitor by analysing α is delegated to the function Holds. This instantiates the predicate encoded in `act` tuples with the data from α (see figure 4.3), returning the result of the predicate test. When the condition Holds$(\alpha,m)\wedge\neg$Holds(α,n) is `true`, $m+n$ is reduced to m, equivalent to the application of mChs$_\text{L}$; the argument for mChs$_\text{R}$ is symmetric.

The function AnalyseAct of listing 1 conducts the runtime analysis. It ensures that once an action is analysed, the monitor is left in a state where it is *ready* to analyse the next action. We implement this logic by organising the application of the operational rules of figure 3.2 into two functions, DeriveAct and DeriveTau, according to the kind of action used to reduce the monitor. DeriveAct on line 17 reduces the monitor *once* by applying it to the action under analysis, yielding m'. Subsequently, ReduceTau reapplies the function DeriveTau until all the internal transitions of the monitor are exhausted (lines 34 to 37). The cases on lines 21 to 24, corresponding to the axioms mDisY$_\text{L}$, mDisN$_\text{L}$, mConY$_\text{L}$, mConN$_\text{L}$,

```
1  def DeriveAct($\alpha$,$o$)
2    match $o$ do
3      case yes ∨ no :
4        print ‘Verdict reached’
5      case {act, $Pred$, $m$} :
6        return $m(\alpha)$ # Apply $m$ to trace event $\alpha$
7      case {chs, $m$, $n$} :
8        if (Holds($\alpha$,$m$) ∧ ¬Holds($\alpha$,$n$))
9          return DeriveAct($\alpha$,$m$)
10       else if ¬Holds($\alpha$,$m$) ∧ Holds($\alpha$,$n$)
11         return DeriveAct($\alpha$,$n$)
12     case {$Op$, $m$, $n$} ∧ $Op$ ∈ {or, and} :
13       $m'$ = DeriveAct($\alpha$,$m$)
14       $n'$ = DeriveAct($\alpha$,$n$)
15       return {$Op$, $m'$, $n'$}

Expect: Monitor is in a ready state
16 def AnalyseAct($\alpha$,$m$)
17   $m'$ = DeriveAct($\alpha$,$m$)
18   return ReduceTau($m'$)
```

```
19 def DeriveTau($o$)
20   match $o$ do
21     case {or, yes, $m$} : return yes
22     case {or, no, $m$} : return $m$
23     case {and, yes, $m$} : return $m$
24     case {and, no, $m$} : return no
25     case {rec, $m$} :
26       return $m()$ # Unfold monitor
27     case {$Op$, $m$, $n$} ∧ $Op$ ∈ {or, and} :
28       if ($m'$ = DeriveTau($m$) ∧ $m' \neq \bot$)
29         return $m'$
30       else
31         return DeriveTau($n$)
32     case Otherwise : return $\bot$

33 def ReduceTau($m$)
34   if ($m'$ = DeriveTau($m$) ∧ $m' \neq \bot$)
35     return ReduceTau($m'$)
36   else
37     return $m$ # No more $\tau$ reductions
```

Listing 1. *Monitoring algorithm that reduces monitors following the small-step rules of figure 3.2*

terminate redundant monitor states, and may be seen as a form of *garbage collection* (DeriveTau omits the cases symmetric to those of lines 21 to 24).

4.4 Selective Instrumentation

Concurrent RV requires a mechanism whereby monitors can be *selectively* instrumented with different processes of the SuS. This set-up generalises the concept of a monitored system induced by the instrumentation relation definitions of figures 3.2 and 3.3 (*i.e.,* $m \triangleleft p$ and $m \triangleleft \kappa \triangleleft p$) to *independent* system processes. Localising the instrumentation on the basis of processes naturally partitions the global trace of SuS events into isolated *sub-traces* that each corresponds to a process under scrutiny. These trace partitions [219] (or slices [62, 196]) permit monitors to consider only the trace events associated with a particular system component, and spares them from handling extraneous events not relevant to the property being checked (refer to motivation in section 1.2).

We model selective instrumentation via the notion of an *instrumentation map*, $\Phi : \textsc{Sig} \rightharpoonup \textsc{Mon}$, from function signatures, $g \in \textsc{Sig}$, to monitors, $m \in \textsc{Mon}$. Signatures g are triples, $\langle M,F,A \rangle$, comprised of the *atomic* module and function names, M and F, and the list of arguments, $A = [d_1,\ldots,d_n]$, used to launch g to execute as a process, $p \in \textsc{Prc}$.

Definition 4.1 (Selective instrumentation). A monitor m is instrumented with a function signature g that is launched as the process p whenever $\Phi(g) = m$, giving the instrumented process $(m \triangleleft p)_g$ in the synchronous case and $(m \triangleleft \kappa \triangleleft p)_g$ in the asynchronous case. ■

We implement selective instrumentation via the meta keywords with and check. These enable us to specify instances of Φ via the syntax: with $\langle M,F,A \rangle_1$ check $\varphi_1,\ldots,$with $\langle M,F,A \rangle_n$ check φ_n, where $\varphi_i \in$

$\textsc{maxHML}^{\mathrm{D}}$. Our implementation translates these statements to the map $\Phi = [\llparenthesis\varphi_1\rrparenthesis/\langle M,F,A\rangle_1, \ldots, \llparenthesis\varphi_n\rrparenthesis/\langle M,F,A\rangle_n]$, where $\llparenthesis\varphi_i\rrparenthesis$ is the Erlang function encoding of the monitor synthesised by the procedure of figure 4.3. We abuse notation and denote the Erlang monitor *code* $\llparenthesis\varphi\rrparenthesis$ simply as m.

4.5 Inline Instrumentation

To the best of our knowledge, there currently exists no inlining framework or library for the Erlang ecosystem, apart from the AOP prototype developed by Cassar et al. [54] called eAOP. Rather than adopting this framework, we opted to design our own instrumentation library since eAOP suffers from a number of shortcomings. For instance, the code that it generates gives rise to certain subtle bugs and the resulting weaved code is inefficient. Efficiency is a key concern of our empirical studies of chapters 6 and 7, because we need to scale our experiment to considerably high loads without risking biasing our results due to superfluous inline instrumentation overhead. The eAOP library is no longer maintained, lacks support for core or newer Erlang data types (*e.g.* binaries and maps), and is unable to instrument applications built on the OTP middleware. We required the latter feature to instrument third-party software, which we used in our case study of section 6.5.

Our inline instrumentation library assumes access to the source code of the SuS. It instruments invocations to the function AnalyseAct discussed in listing 1 via code injection by manipulating the program AST. We leverage the Erlang compilation pipeline that includes a *parse transformation* phase [57] which offers an optional hook whereby the AST can be processed externally, prior to code generation. This program code modification procedure is outlined in figure 4.4. In step ①, the Erlang program source code is preprocessed and parsed into the corresponding AST, step ②. Subsequently, the AST is passed to the parse transformer in step ③: this invokes our custom-built weaver (step ④) that produces the modified AST′ in step ⑤. The decorated AST is compiled by the Erlang compiler into the program binary in the final stage, step ⑥. Note that this compilation phase, as well as the SuS, assume two core dependencies, namely the (i) implementation equivalent of the monitoring algorithm of listing 1, and (ii) monitor executable generated by our synthesis given in figure 4.3.

Step ④ in figure 4.4 performs two transformations on the program AST (shown in brown). Its first transformation initialises the monitor (encoded as an Erlang function by the synthesis procedure of figure 4.3) and stores it in the process dictionary (PD) of the instrumented process. PDs are process-local, mutable key-value stores that every Erlang actor owns [19, 57]. The weaver identifies calls to the Erlang

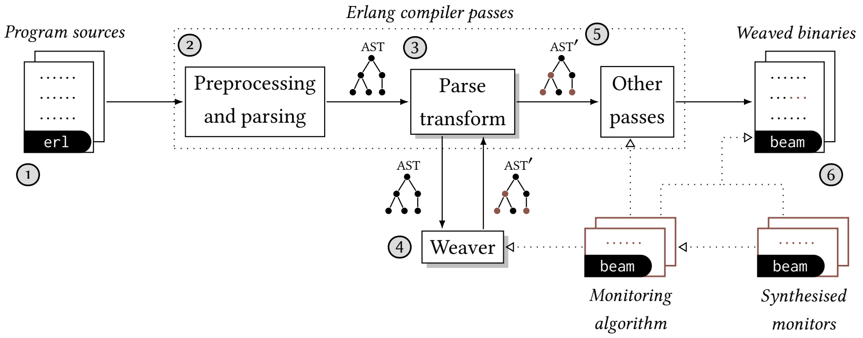


Figure 4.4. *Instrumentation pipeline for inlined monitors using Erlang source-level weaving*

```
1 start(Tok) →
2   spawn(ts, loop, [Tok, Tok + 1]).
```

```
1  start(Tok) →
2    MFA = {M0 = ts, F0 = loop,
3      A0 = [Tok, Tok + 1]},
4    MonFun0 = load_mon_for(. . . )
5    P1 = self(),
6    P0 = spawn(
7      fun() →
8        put($mon_fun, MonFun0),
9        dispatch({⇽, self(), P1, MFA}),
10       apply(M0, F0, A0)
11     end)
12   dispatch({⇾, self(), P0, MFA}),
13   P0.
```

(a) *Server initialised with analyser function*

```
1 loop(Tok, NexTok) when Tok = 1 →
2 receive
3   M2 = {Clt, 0} →
4     dispatch(?, self(), M2),
5     (P1 = Clt) ! M1 = NextTok,
6     dispatch(!, self(), P1, M1),
7     loop(Tok, NextTok + 1)
8 end.
```

(b) *Weaved analysis code in token server loop*

```
1 dispatch(Act) →
2   MonFun0 = get($mon_fun)
3   MonFun1 = analyse_act(Act, MonFun0)
4   put($mon_fun, MonFun1)
```

(c) *Analysis done by* AnalyseAct *of listing 1 (excerpt)*

Figure 4.5. *Transformations to the AST of the* `ts` *program (shown as code)*

built-in function (BIF) `spawn()` that carries the signature of the function that is forked to execute as a new process. Our weaver replaces every `spawn()` with an overloaded version [19] that accepts an anonymous function, `fun(`e`)` → … `end`. This anonymous function is implemented such that it: (i) embeds the monitor function in the PD, and (ii) applies the function specified in the original call to `spawn()`.

Figure 4.5a (top) recalls the function `start()` that forks our token server loop. The weaved counterpart of its AST—given as Erlang code for illustration in figure 4.5a (bottom)—performs the initialisation described (i) and (ii), as follows. Line 2 constructs the Erlang triple `MFA`, initialising the variables `M0`, `F0`, and `A0` with the atoms `ts` and `loop`, and the argument list `[`*Tok*, *Tok* + `1]`. Observe that `MFA` corresponds to the function forked by the call to `spawn()` on line 2 in figure 4.5a (top). Next, the function `load_mon_fun()` on line 4 is used to determine whether a specific `spawn()` call should be instrumented or skipped. It encapsulates the (omitted) boilerplate logic for the instrumentation map Φ described in section 4.4. For example, if $\Phi = [{}^{m}/_{\langle \mathtt{ts},\mathtt{loop},[\_,\_]\rangle}]$, `load_mon_fun()` returns the Erlang monitor code m for the triple `MFA`. When no mapping can be found, *i.e.*, $\Phi(g) = \bot$, the atom `undef` is returned. Lines 6 to 11 replace the original call to `spawn()` of line 2 in figure 4.5a (top) with the aforementioned anonymous function that:

(i) stores the monitor m in the PD via the BIF invocation `put($mon_fun, MonFun0)`, and
(ii) applies the signature `{M0, F0, A0}` to replicate the original `spawn()` invocation mentioned earlier.

The second transformation decorates the program AST with calls at points of interest: these correspond to the actions catalogued in table 4.1. Each call constructs an intermediate trace event description that is dispatched to the monitor for analysis. Lines 9 and 12 in figure 4.5a forward the events ⇽ and ⇾ to the monitor using the function `dispatch()` defined in figure 4.5c. The function `dispatch()`,

(i) retrieves the monitor function m from the PD via the BIF invocation `get($mon_fun)`,
(ii) analyses `Act` by delegating to `analyse_act()` that implements AnalyseAct of listing 1, and
(iii) writes the residual monitor `MonFun1` back to the PD, *i.e.*, `put($mon_fun, MonFun1)`.

Figure 4.5c omits the logic where the retrieved monitor function is equivalent to the atoms `undef` (mapping in Φ was not defined) or `end` (monitor terminated), in which case the analysis step on line 3 is bypassed. The events ? and ! are analogously handled on lines 4 and 6 in figure 4.5b.

Our monitoring algorithm, the choice of process events to collect, together with the two AST transformations discussed, reflect the operational rules of the synchronous instrumentation defined in figure 3.2. The monitoring algorithm of listing 1 ensures that a monitor is *fully* unfolded and left in a ready state, which captures rule iAsyM. Weaving particular points in the AST that correspond to the events of table 4.1 models the case where a process can transition internally (*e.g.* call other functions, write to standard output, *etc.*) via the rule iAsyP. The function `dispatch()` combines the rules iMon and iTer that *always* permit the monitored system $m \triangleleft p$ to transition to a next state, providing the system process *can* perform an action (*i.e.,* the premise $p \xrightarrow{\alpha} p'$). Note that the state reached by $m \triangleleft p$ is dictated by whether the monitor can analyse the exhibited process action (iMon) or is stuck (iTer). In the former case, the function `analyse_act()` on line 3 is invoked; in the latter, the atom `end` is returned and future analyses are *skipped* by `dispatch()` (code omitted).

4.6 Case Study: Monitoring the Cowboy-Ranch Protocol

We demonstrate the usability of inline monitoring by applying it to an off-the-shelf Erlang webserver called Cowboy [134]. Cowboy delegates its socket management to Ranch (a socket acceptor pool for TCP protocols [135]), but forwards incoming HTTP client requests to *protocol handlers* that are forked dynamically by the webserver to service requests independently. Our aim is to runtime check fragments of the request handling protocol between the Cowboy and Ranch components to:

- demonstrate the *expressiveness* of our extended logic maxHML$^{\text{D}}$ by capturing properties of real-world software (section 4.1), and
- validate the *applicability* of our monitoring and inline instrumentation technique to third-party applications built on top of the Erlang/OTP middleware (sections 4.2 and 4.5).

Details of this protocol can be found in appendix B.2. The implementation of inline monitoring, along with the properties discussed, are further validated in chapters 6 and 7 through extensive empirical tests.

For this case study, we redesign the token server of figure 4.2 as a REST web service that is deployed on Cowboy. The server generates identification tokens in one of two formats, UUID, or short alphanumeric strings. Clients request new tokens by issuing GET requests with the parameter, `type=uuid` or `type=short`, specifying the token format required. The web service offers a standard interface: (i) it returns HTTP 200 when requests are properly formatted, (ii) HTTP 400 when the `type` parameter is omitted from the request, and (iii) HTTP 500 when an unsupported `type` is used. We also simulate intermittent faults in Cowboy components by *injecting* random process crashes based on a fair Bernoulli trial [191]. This enables us to formulate properties that describe process termination. Our case study considers a selection of properties that describe the Cowboy-Ranch request handling protocol; the full list of properties may be found in appendix B.3.

Example 4.2 (Cowboy-Ranch protocol). One such property, φ_{RP}, concerns Cowboy *request processes* that service client requests. It states that in its (current) execution, *'a request process does not issue HTTP* *responses with code 500, nor does it crash'.*

$$\mathsf{max}X.\left(\begin{array}{l} [\,!\,\langle \mathit{rprc},\_,\{\mathit{tag},\ \mathit{code},\ .\ .\ .\ \}\rangle, \mathit{tag} = \texttt{resp} \wedge \mathit{code} = \texttt{200}]\,X \wedge \\ [\,!\,\langle \mathit{rprc},\_,\{\mathit{tag},\ \mathit{code},\ .\ .\ .\ \}\rangle, \mathit{tag} = \texttt{resp} \wedge \mathit{code} = \texttt{500}]\,\mathsf{ff} \wedge \\ [\,*\langle \mathit{rprc},\mathit{stat}\rangle, \mathit{stat} = \texttt{crash}]\,\mathsf{ff} \end{array}\right) \qquad (\varphi_{\text{RP}})$$

In φ_{RP}, the binders *tag* and *code* become instantiated with the atom `resp` designating a response message, and the HTTP code of the response returned to requesting clients. Besides ensuring that response messages sent by request processes do not contain the code 500, *i.e.,* $\mathit{tag} = \texttt{resp} \wedge \mathit{code} = \texttt{500}$, formula φ_{RP} also asserts that these processes do not `crash`, *i.e.,* $\mathit{stat} = \texttt{crash}$. The binder *rprc*, referring to the request process PID, is included in φ_{RP} for clarity. ■

4.7 Discussion

This chapter details an implementation of the core building blocks that comprise a RV set-up following the modular blueprint established in chapter 3. We use Erlang as a vehicle to concretise these formal concepts in terms of different software components that fit together according to the schematic of figure 4.1b. The account we give makes minimal assumptions on the underlying implementation framework and can be instantiated to other languages such as Java.

We extend the notion of symbolic actions from section 2.2 with pattern matching to reason about composite data types (*e.g.* tuples and lists), and define a basic model of events that suffices to capture the core behaviour of system processes. Section 4.2 replicates the synthesis procedure of definition 3.6 to generate executable Erlang monitors. It leverages the standard concepts of functional paradigms (*e.g.* pattern matching, variable scoping) to streamline the synthesis and delegate these aspects to the programming language, thereby minimising the chances of translation errors. The resulting monitors emulate parallelism, in that these simultaneously explore the possible paths that can lead monitors to reach a verdict. Our choice to forego parallel monitors stems from the overhead that these induce [219, 53]. While fine-grained concurrency *does* advocate for decomposing multiple tasks into processes, forking a process for every parallel operator (that may be potentially nested into recursive constructs) rapidly increases the consumption of memory. Moreover, sub-monitor processes are typically short-lived, which would result in the continual triggering of the Erlang garbage collector, provoking further scheduler utilisation. Consolidating the different verdicts reached by sub-monitors requires additional communication that further aggravates the overhead.

The core monitor calculus of figure 3.2 that the synthesised monitors and monitoring algorithm assume is crucial: it acts as an *intermediate encoding* that enables the monitoring algorithm to operate on any monitor expressed in that calculus (see figure 4.1). There are two advantages to this scheme. First, the semantics of monitors are not reliant on the specification formalism (the formalism-to-monitor mapping is handled by the synthesis). Second, the monitors *and* monitoring algorithm that interprets them can be treated as a *black box* that fulfils our general definition of a runtime monitor proposed in section 2.1.2, *i.e.,* a monitor is a machine *m* (or sequence recogniser) that analyses finite trace prefixes and reaches irrevocable verdicts.

One challenging aspect in implementing the instrumentation is to provide a *standard* mechanism via which monitors can be selectively attached to the SuS. Section 4.4 defines the notion of an *instrumentation* *map*, Φ, that generalises the instrumentation relations of figures 3.2 and 3.3. Instances of Φ designate particular points in the system execution at which monitors are to be instrumented. For our concurrency use case, we specify instrumentation points as function signatures that are launched by the SuS to execute as independent processes. The same scheme can also be adapted to (monolithic) object-oriented scenarios where monitors are often instrumented with class constructors. Our monitor inlining procedure implements selective instrumentation through source-level weaving by manipulating the AST of Erlang programs. It adheres to the instrumentation rules of figure 3.2 and is compatible with applications that are built atop the Erlang OTP libraries; see section 4.6.

In chapter 5, we show how the same definition of the instrumentation map is implemented for the case of *outline* monitoring (figure 4.1b, bottom right). The common interface that selective instrumentation establishes between the SuS and monitors, together with our treatment of monitors as black-box machines, makes the ensuing Erlang monitors 'synthesise once, instrument anywhere'. This aspect is key to our empirical experiments of chapters 6 and 7, where using the *same* monitor executable with both inlined and outlined benchmarks eliminates the possibility of inducing runtime biases that could arise from disparities in the synthesised monitor code.

4.7.1 Related Work

Our synthesis procedure of section 4.2 contrasts with another alluded to in sections 2.2, 2.5 and 3.6 that operates on the monitorable fragments of the branching-time μHML [116, 118, 7, 4] (figure 4.1a, top right). The latter synthesis generates monitors with non-deterministic behaviour that, while sufficient for the theoretical results required in *op. cit.*, may lead to missed detections in practice. An early materialisation of [116, 118] as the tool detectEr [21, 220, 56, 113] addresses this shortcoming by parallelising monitors using processes, enabling them to reach verdicts along all possible paths. The monitors in these studies use a subset of the core calculus defined in figure 3.2, making them compatible with our framework (see component labelled 'detectEr' in figure 4.1b, top right). While effective, [219, 53] show that these monitors scale poorly.

There are other approaches to monitoring systems with events that carry data, *e.g.*, [30, 33, 131, 128, 129, 37, 216]. One work that shares characteristics with ours is PTS [62], where the global trace is projected into local sub-traces called *slices*, based on parametric specifications. These are properties specified in terms of *symbolic events* whose parameters are instantiated to values from events in the global trace. Our mechanism of the instrumentation map identifies the SuS components to be instrumented and filters out events to obtain trace slices (see section 4.4). PTS is adopted by a number of RV tools that handle data (see *e.g.*, [16, 78]), notably JavaMOP [176, 138, 61] and MarQ [197, 24] for Java, and Elarva [71] for Erlang. JavaMOP and MarQ use inlining to instrument Java objects with local monitors to obtain trace slices naturally. Both of these tools target monolithic architectures and do not provide support for concurrent RV. Elarva takes a different strategy to PTS. It uses the Erlang tracing infrastructure to centrally collect trace events that are demultiplexed between monitors, thereby fabricating slices at runtime. Due to its centralised architecture, this technique is susceptible to suffering from considerable overheads and is unable to scale in practice. As we show in chapter 7, centralised approaches such as these are bound to fail.

5 Decentralised Outline Instrumentation

Outlining is an alternative instrumentation method that circumvents the limitations of inlining discussed in section 2.1.4. It decouples the SuS from its monitors and treats it as a black box, which makes it the only viable option when the system cannot be modified through inlining. This chapter devises a first, general, *reactive* algorithm that instantiates the asynchronous instrumentation definition formalised in section 3.5, extending it to *decentralised* components. In our study, we delineate instrumentation and monitor analysis to: (i) isolate and address the complications of instrumenting decentralised outline monitors, and (ii) understand the impact of separating the instrumentation and analysis w.r.t. overhead (refer to section 7.2.3). This adheres to our modular set-up of figure 4.1b where outline instrumentation is encapsulated as a separate component (bottom right) that provides the monitoring layer with trace events. Our algorithm assumes a tracing infrastructure, such as the ones discussed in section 2.1.4, to reap the benefits of outlining. This design choice, however, complicates the collection and reporting of trace events to outline monitors due to the interleaved execution of the SuS and the instrumentation processes. We:

- detail how our algorithm overcomes the challenges of scaling the monitoring set-up with the SuS, elaborating on the issues that stem from the dynamic reconfiguration of outline monitors in our asynchronous setting, Section 5.1;
- demonstrate its implementability by overviewing our tool that monitors programs written for the EVM, and discuss how the correctness of our implementation is validated via rigorous invariant testing, Section 5.3.

Chapter 7 validates our implementation further by subjecting it to a comprehensive empirical evaluation that gives us high assurances of its correctness and feasibility in practice.

5.1 Modelling Decentralised Outline Instrumentation

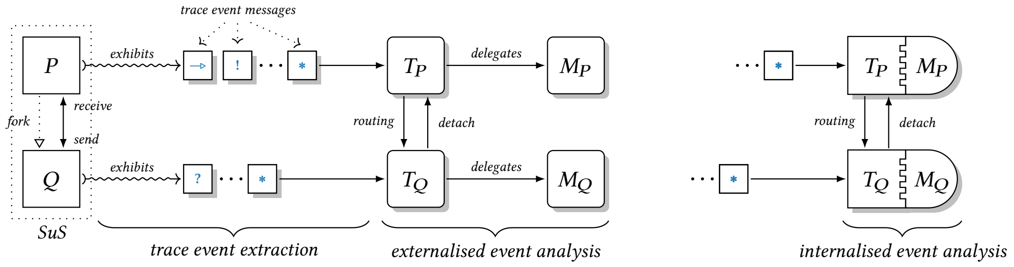
The decentralised outline algorithm we propose addresses the instrumentation gap identified in section 1.1.2. There are several constraints that the reactive system setting necessarily imposes on our operational model of processes and monitors:

C_1 *Local clocks.* Components do not share a common global clock.

C_2 *Elastic.* The number of components fluctuates.

C_3 *Point-to-point messaging.* A sender component interacts directly with one receiver at a time.

C_4 *Message reordering.* The order of messages as sent from different components is not guaranteed at the recipient end. This does not apply to point-to-point messaging, *i.e.,* successive messages exchanged between pairs of components are delivered in the same sequence issued.



(a) *Tracer and monitor organised as separate processes (*external*)* **(b)** *Merged tracer and monitor processes (*internal*)*

Figure 5.1. *Decentralised outline monitoring set-up consisting of tracer and monitor roles*

Online monitors are instrumented to run with the SuS. A reactive system, therefore, entails that the monitoring set-up is itself reactive, which further requires the runtime analysis to be:

C_5 *Decentralised.* No central entity coordinates monitors so that the set-up is scalable and not susceptible to SPOFs.

C_6 *Passive.* Monitors react to SuS events but do not steer or block its execution.

C_7 *Reliable.* Trace events are not lost, nor reported to monitors out of order.

Since our study considers neither failure nor security aspects (refer to section 1.2), we assume:

A_1 *Reliable components.* Components are not subject to fail-stops or Byzantine failures.

A_2 *Reliable communication.* Messages are not tampered with, always delivered, and never duplicated.

The design of our instrumentation approach abides by constraints C_1 to C_7. Our definition of monitors as sequence recognisers (refer to section 2.1.2) satisfies constraint C_6. The algorithm instruments monitors to run asynchronously with the SuS, in line with constraint C_1; this turns out to be the general case for distributed set-ups. Note that distribution can be obtained by weakening assumptions A_1 and A_2. Constraints C_2 and C_5 call for the instrumentation to scale dynamically by continually reconfiguring the monitoring set-up in response to changes in the SuS. Finally, constraint C_7 guards against issues arising from constraint C_4, which is vital for analyses that are sensitive to the temporal ordering of trace events, as argued in section 2.1.4. C_7 enables to pin down our notion of *valid* traces.

Definition 5.1 (Valid trace). A finite trace s is said to be *valid* w.r.t. a process p iff

- s contains *all* the trace events exhibited by p so far, *i.e.,* no events are missing, and
- the order of these events corresponds to the one in which these occur locally at p. ■

Figure 5.1 shows the variants of outline instrumentation that we consider. It depicts a two-process SuS where the trace events (encoded as messages) of processes P and Q are respectively directed to *tracers* T_P and T_Q and analysed by monitors M_P and M_Q. The externalised analysis (*external*) arrangement in figure 5.1a consists of independent tracer and monitor processes. It teases apart the tasks of trace event handing and monitor reorganisation, performed by tracers, T, from the task of event analysis, effected by monitors, M. Decoupling the tracers from monitors follows the *single responsibility* tenet advocated in fine-grained concurrency design [15, 19], but at the expense of introducing a separate monitor component.

The internalised analysis variant (*internal*) merges the tracer and monitor to forgo this extra component (figure 5.1b). Our algorithm relies on a tracing infrastructure, such as the ones mentioned in section 2.1.4, to gather streams of event messages for the traced SuS components. Tracers can start and stop these event streams at runtime. The model also assumes that:

A_3 *System processes may share tracers.* A tracer can trace multiple processes simultaneously. This makes it possible for monitors to treat multiple processes of the SuS as one component[1].

A_4 *Tracers do not share system processes.* A process of the SuS is traced by *one* tracer at any point in time. This keeps our core logic manageable. If multiple monitors need to analyse the behaviour of the same component, the tracer can duplicate the events and report them to the monitors accordingly.

A_5 *System processes inherit tracers.* A newly forked process in the SuS is automatically assigned the tracer of its parent. This behaviour facilitates assumption A_3 as it allows tracers to consider sets of processes as a unit by default.

Assumption A_5 requires a tracer to intervene if it needs to monitor a particular process independently from others: it must first *stop* the active tracer before it can take over and resume tracing this process itself. In the absence of such interventions, the SuS is implicitly traced as one entity by the (central) tracer, which is instrumented with the *root* system process. This design choice follows the approach of existing *centralised* monitoring tools, *e.g.*, [21, 53, 71, 180].

5.1.1 Processes and Trace Events

Our model of processes and trace events builds on the one introduced in sections 4.1 and 4.4. It assumes a denumerable set of PIDs to reference processes. We distinguish between system, tracer, and monitor process forms, denoting them respectively by the sets $\textsc{Pid}_{\mathrm{S}}$, $\textsc{Pid}_{\mathrm{T}}$ and $\textsc{Pid}_{\mathrm{M}}$, where $p_{\mathrm{S}} \in \textsc{Pid}_{\mathrm{S}}$, $p_{\mathrm{T}} \in \textsc{Pid}_{\mathrm{T}}$, $p_{\mathrm{M}} \in \textsc{Pid}_{\mathrm{M}}$. Processes are created via the function $\mathsf{fork}(g)$ that takes the signature of the code to be run by the forked process, $g \in \textsc{Sig}$, and returns its *fresh* PID. We refer to the process invoking fork as the *parent*, and to the forked process as the *child*. To create monitor processes, the function fork is overloaded to accept executable monitor code, m, and return the corresponding PID, p_{M}. Tracer processes are forked analogously. Recall that the code m is generated by the synthesis procedure described in section 4.2 from some $\textsc{maxHML}^{\mathrm{D}}$ specification φ that one wishes to runtime check. Since our account focusses mostly on the tracing aspect, we use the terms *tracer* and *monitor* interchangeably whenever the distinction is unimportant. We refer to a grouping of one or more processes of the SuS as a *component*.

Following a reactive model, our processes communicate via asynchronous messages. Each process owns a *message buffer*, κ, from where it can read messages *out-of-order* and in *non-blocking* fashion. Messages, $k \in \textsc{Msg}$, adopt an analogous definition to the trace events given in section 4.1. They are tuples, $\langle \partial, d_2, \ldots, d_n \rangle$, where the first element $d_1 = \partial$ is the qualifier and $d_2, \ldots, d_n \in \mathcal{D}$ (see section 4.1) is the data payload carried by the message. The message qualifier, $\partial \in \{\mathsf{evt}, \mathsf{dtc}, \mathsf{rtd}\}$, indicates the type of message:

- evt : *trace event* message collected by the tracing infrastructure,
- dtc : *detach command* message that tracers exchange to reorganise the monitor choreography, and
- rtd : *routing* message that embeds evt or dtc messages forwarded between tracers.

[1] This is something that is not easily achieved with inlining.

Event	Action (e.act)	Field name	Description
fork	$\rightarrowtriangle$	src	PID of the parent process forking e.tgt via fork(g)
		tgt	PID of the child process forked by e.src
		sig	Function signature g forked by e.tgt
exit	$*$	src	PID of the terminated process
send	!	src	PID of the process sending the message
		tgt	PID of the recipient process
receive	?	src	PID of the recipient process

Table 5.1. *Trace event messages, action label, and data field names*

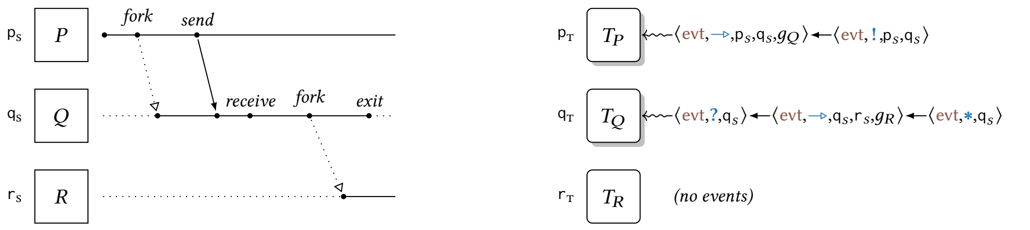
We use the dot notation (.) to access elements of the *data payload* carried in messages, $d_1, d_2, \ldots, d_n$, via indexable field names, *e.g.* the message qualifier ∂ is read through k.type. The metavariables e, c, and r are reserved for message types evt, dtc, and rtd respectively.

Trace events are encoded as messages, $\langle \mathsf{evt}, \ell, \ldots, d_n \rangle$, where the label, $d_2 = \ell \in \mathcal{L}$, identifies the action exhibited by the SuS, and the remainder, $\ldots, d_n$, is the action payload described in section 4.1. The event *action label* is accessed using e.act. As in section 4.1, we let $\mathcal{L} = \{\rightarrowtriangle, *, !, ?\}$ denote process actions *fork* ($\rightarrowtriangle$), *exit* ($*$), *send* (!) and *receive* (?); *initialise* ($\leftarrowtriangle$) is omitted since this is not used by our algorithm. We use the action label ℓ in lieu of the full trace event message payload (*i.e.*, omitting ∂ and $\ldots, d_n$) to simplify our exposition when suitable. Table 5.1 adapts table 4.1 and catalogues the relevant trace events and corresponding data.

5.2 The Instrumentation Algorithm

Our algorithm covers the two variants of figure 5.1. Listings 2 to 4 describe the core logic found in each tracer. Every tracer maintains an internal state, ς, that consists of three maps:

(i) the *routing map*, Π, governing how events are routed to other tracers,
(ii) the *instrumentation map* from section 4.4, Φ, that enables selective instrumentation, and
(iii) the *traced-component map*, Γ, maintaining processes of the SuS that the tracer currently tracks.


p_S
P
fork
send
q_S
Q
receive
fork
exit
r_S
R
p_T
T_P
$\langle \mathsf{evt}, \rightarrowtriangle, p_S, q_S, g_Q \rangle$
$\langle \mathsf{evt}, !, p_S, q_S \rangle$
q_T
T_Q
$\langle \mathsf{evt}, ?, q_S \rangle$
$\langle \mathsf{evt}, \rightarrowtriangle, q_S, r_S, g_R \rangle$
$\langle \mathsf{evt}, *, q_S \rangle$
r_T
T_R
(no events)

(a) *Interaction sequence of P, Q and R*

(b) *Trace partitions for T_P, T_Q and T_R (monitors omitted)*

Figure 5.2. *SuS with processes P, Q, and R instrumented with three independent monitors*

Recall that our monitors are sequence recognisers, which allows tracers to remain *agnostic* to their encapsulated analysis logic. We overload the function AnalyseAct described in listing 1 to link the tracing and monitors. The algorithm we give uses these overloads to analyse events by forwarding them to a monitor *externalised* in its own process (figure 5.1a), or analyse them *internally* (figure 5.1b), following the exact method detailed in the function `dispatch()` of figure 4.5c.

The message buffer that tracer processes are equipped with is a materialisation of the queue κ that our asynchronous instrumentation definition given in section 3.5 uses to decouple the SuS from its monitors. This buffer enables the system to execute without waiting for the monitors to complete their analysis, in agreement with rule aiPrc of figure 3.3. The instrumentation infrastructure tends to the collection of the trace events exhibited by the SuS and their delivery to the message buffer of the appropriate tracer. Tracers can independently analyse their buffer of trace events through invocations of the function AnalyseAct; this corresponds to rule aiMon. Rule aiAsyM is embodied by our monitoring algorithm of listing 1 that always unfolds monitors to a ready state. Analogous to the reasoning of section 3.5, capturing only specific events (*i.e.,* fork, exit, send, and receive) models the unobservable transitions that processes can follow via aiAsyP. The rule aiTer given in section 3.5 is central to garbage collection, where redundant tracers are terminated. While aiTer is specific to analyses that reach the inconclusive verdict (end), our algorithm extends this rule to handle (yes) and reject (no) verdicts. The reason for this is that the verdicts flagged by monitors are irrevocable, which permits monitors to terminate, knowing that future analyses can never overturn the verdict flagged. Note that verdict flagging alone does not decide whether tracers are terminated; section 5.2.7 outlines other conditions that our algorithm considers during garbage collection.

Example 5.1. Consider a SuS consisting of three processes, $\{P,Q,R\}$. P forks process Q and communicates with it; afterwards, Q forks R and terminates. P, Q, and R are assigned PIDs p_s, q_s, and r_s respectively. This interaction, captured in figure 5.2a, is fundamentally *sequential* due to the synchronous dependency between processes: *e.g.*, Q is created by P, and R is forked by Q only after Q receives the message from P.

There are a number of ways in which this system can be instrumented with monitors (or *tracers*). For instance, a central tracer may be set up for all of $\{P,Q,R\}$; alternatively one could choose to trace $\{P,Q\}$ as a single component and use a separate tracer for the singleton process $\{R\}$, *etc.* In this example, we instrument the SuS with independent tracer, one for each of $\{P\}$, $\{Q\}$, and $\{R\}$. Figure 5.2b shows these tracers labelled as T_P, T_Q and T_R, their corresponding PIDs, and the valid sequence of events (definition 5.1) each tracer is meant to analyse. ■

Despite its small size and sequential operation, the SuS and monitoring set-up of example 5.1 may still be subject to multiple interleaved executions. This results from the asynchronous organisation of the SuS and monitor components, whose execution depends on external factors such as process scheduling.

Table 5.2 summarises the challenges inherent to decentralised outline monitoring we tackle in the forthcoming sections 5.2.1 to 5.2.7. These sections detail how our algorithm reports trace events to independent monitors while abiding by the reliability guarantees that RV requires, *i.e.,* trace events are not lost, nor reported out of order (definition 5.1). Along with ensuring these guarantees, we elaborate on the technique our algorithm uses to achieve elastic behaviour via dynamic instrumentation and garbage collection of monitors.

Challenge	Solution
Non-invasive monitors	Collecting trace events from the SuS via asynchronous tracing, Section 5.2.1
Scaling up the set-up	Instrumenting new monitors dynamically for partitioned traces, Section 5.2.2
No trace event loss	Routing trace events to deliver them to the correct monitors, Section 5.2.3
No trace event reordering	Prioritising forwarded events before analysing any other event, Section 5.2.4
Independent monitors	Detach tracers from their ancestors once all the trace events have been forwarded, Section 5.2.5
Targeted monitoring	Selective instrumentation of forked processes, Section 5.2.6
Scaling down the set-up	Garbage collecting redundant monitors, Section 5.2.7

Table 5.2. *Challenges addressed by decentralised outline monitoring to ensure correct and elastic runtime analyses*

5.2.1 Tracing

The operations Trace, Clear and Preempt provide access to the underlying tracing infrastructure. Trace enables a tracer with PID p_{T} to register its interest in receiving trace events (in the form of messages) of a system process with PID p_{S}. This operation can be undone using Clear, which *blocks* the calling tracer p_{T} and returns once all the event messages for p_{S} that are in transit to p_{T} have been delivered (assumption A_2). Preempt combines Clear and Trace, enabling a different tracer p'_{T} to take over the tracing of process p_{S} from the current tracer, p_{T}. Tracing is *inherited* by every child process that a traced system process forks, following assumption A_5. Clear or Preempt can be used to alter this arrangement, as section 5.2.2 explains. Readers are referred to listing 7 for specifics on these operations.

5.2.2 Trace Partitioning

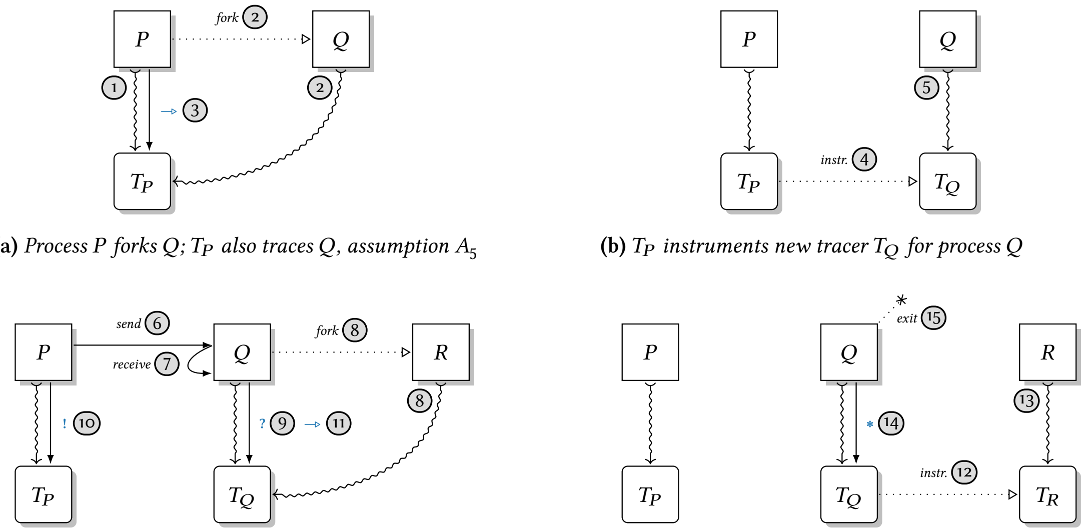
Processes (or threads) originate as a hierarchy, starting from the root process that forks child processes, and so forth, *e.g.* `CreateThread()` in Windows [177], `pthread_create()` for POSIX threads [48], `ActorContext.spawn()` in Akka [199], and `spawn()` in Erlang [57] and Elixir [142]. We borrow standard terminology to describe process relationships in this hierarchy (*e.g.* parent, ancestor, descendant, *etc.*). In our algorithm, tracers are programmed to react to *fork* (⇾) and *exit* (∗) events in the trace. Figure 5.3 illustrates how the hierarchical process creation sequence of the SuS is exploited to instrument tracers. A tracer instruments other tracers whenever it encounters ⇾ events in the execution. In figure 5.3a, the root tracer T_P analyses the top-level process P, step ①. It instruments a new tracer, T_Q, for process Q when it observes the fork event $\langle \mathrm{evt}, ⇾, \mathrm{p}_{\mathrm{S}}, \mathrm{q}_{\mathrm{S}}, g_Q \rangle$ exhibited by P in step ③. The field e.tgt (refer to table 4.1) carried by ⇾ designates the SuS process (ID) to be instrumented with the new tracer, q_{S} in this case. From this point onwards, T_Q *takes over* the tracing of process Q from T_P by invoking Preempt to trace Q *independently* of T_P, as shown in steps ④ and ⑤ of figure 5.3b. Meanwhile, T_P resumes its analysis and receives the send event $\langle \mathrm{evt}, !, \mathrm{p}_{\mathrm{S}}, \mathrm{q}_{\mathrm{S}} \rangle$ in step ⑩ after P messages Q in step ⑥ of figure 5.3c. Subsequent ⇾ events observed by T_P and T_Q are handled as described earlier in steps ③ to ⑤. Figures 5.3c and 5.3d show how the final tracer, T_R, is instrumented as Q forks its child R. It is worth mentioning that prior to instrumenting T_Q in step ④, process Q automatically inherits tracer T_P of its parent P in step ②, following assumption A_5. T_Q is analogously assigned to process R in step ⑧ before T_Q instruments the new tracer T_R for R in step ⑫.

5.2.3 Trace Event Routing

The asynchrony between the SuS and tracer components may give rise to different interleaved executions. Figure 5.4 shows an interleaving *alternative* to that of figures 5.3b to 5.3d. In figure 5.4a, T_P is slow to handle the fork event of Q (received in step ③ in figure 5.3a), and fails to instrument T_Q promptly. As a result, the events ? and ⇾ exhibited by Q are received by T_P in steps ⑦ and ⑨. Figure 5.4a shows the case where ⟨evt,?,q_s⟩ is processed by T_P, step ⑪, rather than by the *correct* tracer T_Q that would be eventually instrumented by T_P. This interleaving corrupts the runtime analysis, as the events that should be processed by one tracer reach unintended ones.

To address this issue, tracers *keep* the events they should handle and *forward* the rest to neighbouring tracers. This scheme follows *hop-by-hop* routing used in IP networks [174]. We define the notion of a *router tracer* as one that receives the trace events of a system process that are meant to be handled by *another* tracer. The role of router tracers (or *routers* for short) is to (i) embed trace events evt or detach commands dtc into routing messages, rtd, and (ii) dispatch them to neighbouring tracers. Routing messages are transmitted in a hop-by-hop fashion by tracers until they reach their destination tracer. For instance, T_P in figure 5.4a becomes the router tracer for Q since it initially receives the events ? and ⇾ of Q (steps ⑦ and ⑨), although these are meant to be handled by T_Q. T_P routes these events as follows. It first instruments T_Q with Q in step ⑪. Next, it prepares ⟨evt,?,q_s⟩ and ⟨evt,⇾,q_s,r_s,g_R⟩ for transmission by embedding them in rtd messages (steps ⑭ and ⑱), forwarding them to T_Q in steps ⑮ and ⑲. The event ⟨evt,!,p_s,q_s⟩ is handled by T_P, step ⑰. Concurrently, T_Q acts on the forwarded events ? and ⇾ in steps ⑯ and ㉑, and instruments T_R with R as a result in step ㉒.

Tracers determine which events to keep or forward by means of the *routing map*, $\Pi : \textsc{Pid}_S \rightharpoonup \textsc{Pid}_T$, that relates SuS and tracer PIDs. Each tracer queries its routing map for every event e it processes using the



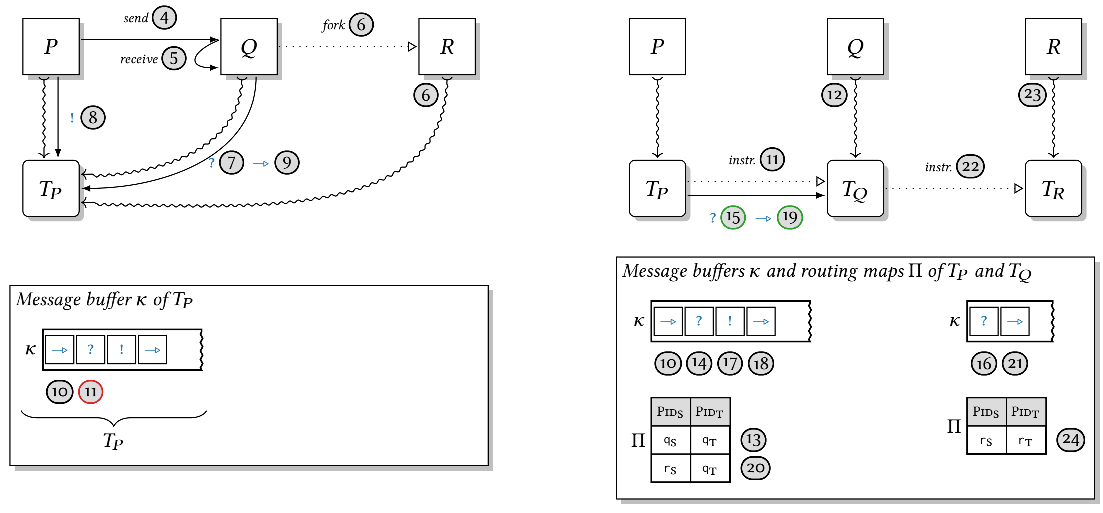
(a) *Process P forks Q; T_P also traces Q, assumption A_5*

(b) *T_P instruments new tracer T_Q for process Q*

(c) *T_P and T_Q analyse trace events independently*

(d) *Processes P, Q, R and corresponding tracers*

Figure 5.3. *Outline tracer instrumentation for processes P, Q and Q (monitors omitted)*



(a) *Trace events for P, Q, and R observed by T_P*

(b) *Trace events for Q routed from T_P to T_Q*

Figure 5.4. *Hop-by-hop trace event routing using tracer routing maps Π (monitors omitted)*

source PID, e.src. An event is forwarded to a tracer with PID p_T *only if* $\Pi(e.\mathrm{src}) = p_\mathrm{T}$, otherwise it is handled by the tracer itself since a route for the event does not exist, *i.e.,* $\Pi(e.\mathrm{src}) = \bot$. HandleFork, HandleExit and HandleComm in listing 3 implement this forwarding logic on lines 19, 27 and 35.

A tracer populates its routing map Π whenever it processes a fork event $\langle \mathrm{evt}, \rightarrow, p_\mathrm{S}, p'_\mathrm{S}, g\rangle$. It considers *one* of two cases for the originator of the event, PID p_S:

$\mathrm{C_K}$ $\Pi(p_\mathrm{S}) = \bot$. This is a cue to adapt the monitor choreography to account for the forked process p'_S. The tracer *keeps* the fork event and instruments a second tracer $T_{P'}$ with PID p'_T for the process p'_S. It then adds the mapping $p'_\mathrm{S} \mapsto p'_\mathrm{T}$ to its routing map Π.

$\mathrm{C_F}$ $\Pi(p_\mathrm{S}) = p'_\mathrm{T}$. A route to the neighbouring tracer p'_T exists for trace events originating from the process with PID p_S. This informs the tracer that the event is meant for another tracer. The tracer *forwards* the fork event of process p_S to tracer p'_T, and adds the mapping $p'_\mathrm{S} \mapsto p'_\mathrm{T}$ to its routing map Π.

In cases $\mathrm{C_K}$ and $\mathrm{C_F}$, the addition of $p'_\mathrm{S} \mapsto p'_\mathrm{T}$ ensures that future events originating from p'_S can always be

```
Expect: $e.\mathrm{act} = \rightarrow$
def Instrument$_\circ(\varsigma, e, p_\mathrm{T})$
  $p_\mathrm{S} \leftarrow e.\mathrm{tgt}$
  if $((m \leftarrow \varsigma.\Phi(e.\mathrm{sig})) \neq \bot)$
    $p'_\mathrm{T} \leftarrow$ fork(Tracer$(\varsigma, m, p_\mathrm{S}, p_\mathrm{T})$)
    $\varsigma.\Pi \leftarrow \varsigma.\Pi \cup \{\langle p_\mathrm{S}, p'_\mathrm{T}\rangle\}$
  else
    # In ∘mode, there is no PID $p_S$ to detach
    # from a router; add $p_S$ to Γ in ∘mode
    $\varsigma.\Gamma \leftarrow \varsigma.\Gamma \cup \{\langle p_\mathrm{S}, \circ\rangle\}$
  return $\varsigma$
```

```
Expect: $e.\mathrm{act} = \rightarrow$
def Instrument$_\bullet(\varsigma, e, p_\mathrm{T})$
  $p_\mathrm{S} \leftarrow e.\mathrm{tgt}$
  if $((m \leftarrow \varsigma.\Phi(e.\mathrm{sig})) \neq \bot)$
    $p'_\mathrm{T} \leftarrow$ fork(Tracer$(\varsigma, m, p_\mathrm{S}, p_\mathrm{T})$)
    $\varsigma.\Pi \leftarrow \varsigma.\Pi \cup \{\langle p_\mathrm{S}, p'_\mathrm{T}\rangle\}$
  else
    # Detach PID $p_S$ from router $p_T$
    Detach$(p_\mathrm{S}, p_\mathrm{T})$
    $\varsigma.\Gamma \leftarrow \varsigma.\Gamma \cup \{\langle p_\mathrm{S}, \bullet\rangle\}$ # Add $p_S$ in •mode
  return $\varsigma$
```

Listing 2. *Instrumentation operations for direct and priority tracer modes*

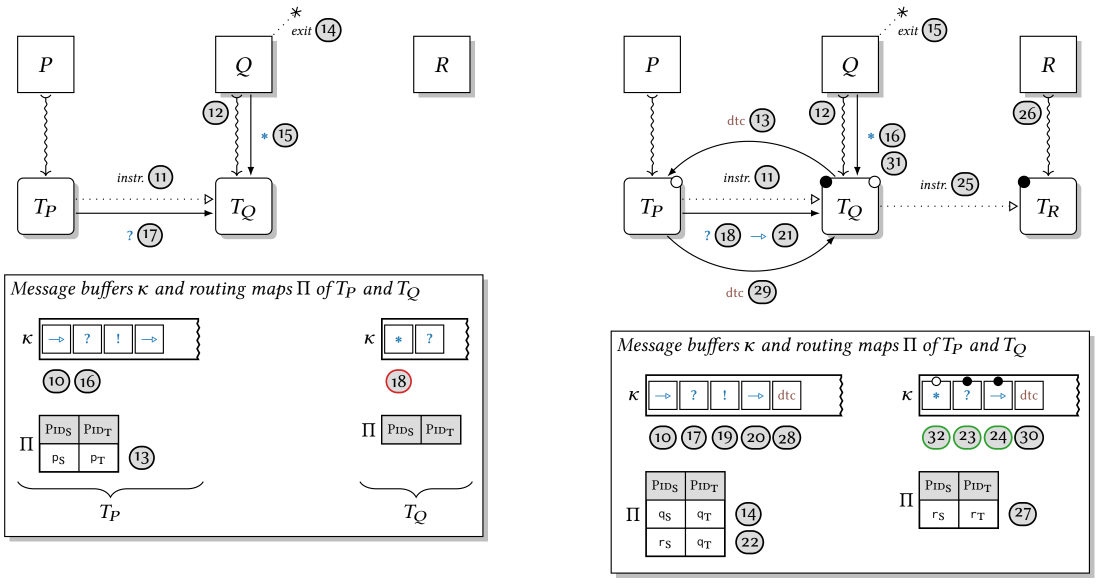
```
1  def Loop$_\circ$($\varsigma$,$p_\mathrm{M}$)
2    forever do
3      $k \leftarrow$ next message from buffer $\kappa$
4      if ($k$.type = evt)
5        $\varsigma \leftarrow$ HandleEvent$_\circ$($\varsigma$,$k$,$p_\mathrm{M}$)
6      else if $k$.type = dtc
           # route dtc back to issuer
7        $\varsigma \leftarrow$ RouteDtc($\varsigma$,$k$,$p_\mathrm{M}$)
8      else if $k$.type = rtd
9        $\varsigma \leftarrow$ ForwdRtd$_\circ$($\varsigma$,$k$,$p_\mathrm{M}$)

10 def HandleEvent$_\circ$($\varsigma$,$e$,$p_\mathrm{M}$)
11   if ($e$.act = $\rightarrowtriangle$)
12     $\varsigma \leftarrow$ HandleFork$_\circ$($\varsigma$,$e$,$p_\mathrm{M}$)
13   else if $e$.act = $*$
14     $\varsigma \leftarrow$ HandleExit$_\circ$($\varsigma$,$e$,$p_\mathrm{M}$)
15   else if $e$.act $\in$ {!,?}
16     HandleComm$_\circ$($\varsigma$,$e$,$p_\mathrm{M}$)
17   return $\varsigma$

18 def HandleFork$_\circ$($\varsigma$,$e$,$p_\mathrm{M}$)
19   if (($p_\mathrm{T} \leftarrow \varsigma.\Pi(e.\mathrm{src})$) $\neq \bot$)
20     Route($e$,$p_\mathrm{T}$)
       # Route for e.tgt goes via the tracer of e.src
21     $\varsigma.\Pi \leftarrow \varsigma.\Pi \cup \{\langle e.\mathrm{tgt}, p_\mathrm{T}\rangle\}$
22   else
23     AnalyseAct($e$,$p_\mathrm{M}$) # Analyse event
24     $\varsigma \leftarrow$ Instrument$_\circ$($\varsigma$,$e$,self())
25   return $\varsigma$

26 def HandleExit$_\circ$($\varsigma$,$e$,$p_\mathrm{M}$)
27   if (($p_\mathrm{T} \leftarrow \varsigma.\Pi(e.\mathrm{src})$) $\neq \bot$)
28     Route($e$,$p_\mathrm{T}$)
29   else
30     AnalyseAct($e$,$p_\mathrm{M}$) # Analyse event
31     $\varsigma.\Gamma \leftarrow \varsigma.\Gamma \setminus \{\langle e.\mathrm{src}, \circ\rangle\}$ # Remove dead e.src
32     TryGC($\varsigma$,$p_\mathrm{M}$)
33   return $\varsigma$
```

```
34 def HandleComm$_\circ$($\varsigma$,$e$,$p_\mathrm{M}$)
35   if (($p_\mathrm{T} \leftarrow \varsigma.\Pi(e.\mathrm{src})$) $\neq \bot$)
36     Route($e$,$p_\mathrm{T}$)
37   else
38     AnalyseAct($e$,$p_\mathrm{M}$) # Analyse event

39 def RouteDtc($\varsigma$,$c$,$p_\mathrm{M}$)
40   if (($p_\mathrm{T} \leftarrow \varsigma.\Pi(c.\mathrm{tgt})$) $\neq \bot$)
41     Route($c$,$p_\mathrm{T}$)
42     $\varsigma.\Pi \leftarrow \varsigma.\Pi \setminus \{\langle c.\mathrm{tgt}, p_\mathrm{T}\rangle\}$ # Clear c.tgt route
43     TryGC($\varsigma$,$p_\mathrm{M}$)
44   return $\varsigma$

45 def ForwdRtd$_\circ$($\varsigma$,$r$,$p_\mathrm{M}$)
46   $k \leftarrow r$.emb
47   if ($k$.type = dtc)
48     $\varsigma \leftarrow$ ForwdDtc($\varsigma$,$r$,$p_\mathrm{M}$)
49   else if $k$.type = evt
50     $\varsigma \leftarrow$ ForwdEvt($\varsigma$,$r$)
51   return $\varsigma$

52 def ForwdDtc($\varsigma$,$r$,$p_\mathrm{M}$)
53   $c \leftarrow r$.emb
54   if (($p_\mathrm{T} \leftarrow \varsigma.\Pi(c.\mathrm{tgt})$) $\neq \bot$)
55     Forwd($r$,$p_\mathrm{T}$)
56     $\varsigma.\Pi \leftarrow \varsigma.\Pi \setminus \{\langle c.\mathrm{tgt}, p_\mathrm{T}\rangle\}$ # Clear c.tgt route
57     TryGC($\varsigma$,$p_\mathrm{M}$)
58   return $\varsigma$

Expect: $\varsigma.\Pi(r.\mathrm{emb.src}) \neq \bot$
59 def ForwdEvt($\varsigma$,$r$)
60   $e \leftarrow r$.emb
61   if (($p_\mathrm{T} \leftarrow \varsigma.\Pi(e.\mathrm{src})$) $\neq \bot$)
62     Forwd($r$,$p_\mathrm{M}$)
       # Route for e.tgt goes via the tracer of e.src
63     if ($e$.act = $\rightarrowtriangle$)
64       $\varsigma.\Pi \leftarrow \varsigma.\Pi \cup \{\langle e.\mathrm{tgt}, p_\mathrm{T}\rangle\}$
65   return $\varsigma$
```

Listing 3. *Tracer loop that handles direct (∘) trace events, message routing and forwarding*

forwarded to neighbouring tracers p'_T. Figure 5.4b shows the routing maps of T_P and T_Q. T_P adds $\mathrm{q_s} \mapsto \mathrm{q_T}$, step ⑬, after processing $\langle \mathrm{evt}, \rightarrowtriangle, \mathrm{p_s}, \mathrm{q_s}, g_Q \rangle$ from the message buffer in step ⑩ and instrumenting tracer T_Q with Q in step ⑪; an instance of case $\mathrm{C_K}$. The function Instrument in listing 2 details this on line 5, where the mapping $e.\mathrm{tgt} \mapsto p'_\mathrm{T}$ (with $e.\mathrm{tgt} = p'_\mathrm{S}$) is added to Π, following the creation of tracer p'_T. Step ⑳ of figure 5.4b is an instance of case $\mathrm{C_F}$: T_P adds $\mathrm{r_s} \mapsto \mathrm{q_T}$ after processing $\langle \mathrm{evt}, \rightarrowtriangle, \mathrm{q_s}, \mathrm{r_s}, g_R \rangle$ for R in step ⑱. Crucially, T_P *does not* instrument a new tracer, but delegates this task to T_Q by forwarding the fork event in question. Lines 21 and 64 in listing 3 (and later line 21 in listing 4) are manifestations of this, where the mapping $e.\mathrm{tgt} \mapsto p'_\mathrm{T}$ is added after the fork event e is routed to the next tracer p'_T.

Note that in figure 5.4b, the mappings inside T_P point to tracer T_Q, and the mapping in T_Q points to



(a) *T_Q observes event ∗ before T_P routes ?*

(b) *T_Q processes priority events routed by T_P first*

Figure 5.5. *Trace event order preservation using priority (●) and direct (○) tracer modes (monitors omitted)*

T_R. This arises from cases C_K and C_F, where every tracer in the choreography can *only* forward events to *adjacent* tracers. For instance, the events that R might exhibit and that are collected by T_P must be forwarded twice to reach the intended tracer T_R—from tracer T_P to T_Q, and from T_Q to T_R. The routing map entries of neighbouring tracers form a connected directed acyclic graph (DAG), ensuring that every trace event message is eventually delivered to its correct destination. Our algorithm implements hop-by-hop routing using the operations Route and Forwd (see appendix A). Route creates a *wrapper* message, r, with type rtd, denoting a routing message or command, and embeds the message to be routed. Tracers then process routing messages by (i) either extracting the embedded message through the field r.emb, *e.g.* line 53 in ForwdDtc, or (ii) forwarding it to the next tracer using Forwd, *e.g.* line 55 in ForwdDtc.

5.2.4 Trace Event Routing with Priorty

Hop-by-hop routing does not guarantee that tracers receive events in an order that reflects the correct SuS execution. This reordering can arise when a tracer collects trace events of a SuS component *and simultaneously* receives routed events concerning this component from other tracers. Figure 5.5a gives a different interleaving to the execution of figure 5.4b to showcase the deleterious effect this race condition has on the runtime analysis when events are reordered for T_Q. In step ⑫, T_Q takes the place of T_P and continues tracing process Q, collecting the event ∗ in step ⑮; this *happens before* T_Q receives the routed event ? concerning Q in step ⑰ of figure 5.5a. When T_Q analyses trace events from its message buffer in the order it receives them, as in step ⑱, it violates the temporal event ordering determined in figure 5.2b of example 5.1. A naïve handling of ∗ followed by ? would *erroneously* mean that Q receives messages after it terminates, contradicting definition 5.1.

```
1  def LOOP$_\bullet$($\varsigma$,$p_M$)
2    forever do
3      $r \leftarrow$ next rtd message from buffer $\kappa$
4      $k \leftarrow r$.emb
5      if ($k$.type = evt)
6        $\varsigma \leftarrow$ HANDLEEVENT$_\bullet$($\varsigma$,$r$,$p_M$)
7      else if $k$.type = dtc
         # dtc routed back from router
8        $\varsigma \leftarrow$ HANDLEDTC($\varsigma$,$r$,$p_M$)

9  def HANDLEEVENT$_\bullet$($\varsigma$,$r$,$p_M$)
10   $e \leftarrow r$.emb
11   if ($e$.act = $\rightarrowtriangle$)
12     $\varsigma \leftarrow$ HANDLEFORK$_\bullet$($\varsigma$,$r$,$p_M$)
13   else if $e$.act = $*$
14     $\varsigma \leftarrow$ HANDLEEXIT$_\bullet$($\varsigma$,$r$,$p_M$)
15   else if $e$.act $\in$ {!,?}
16     HANDLECOMM$_\bullet$($\varsigma$,$r$,$p_M$)

17 def HANDLEFORK$_\bullet$($\varsigma$,$r$,$p_M$)
18   $e \leftarrow r$.emb
19   if (($p_T \leftarrow \varsigma.\Pi(e.\text{src})$) $\neq \bot$)
20     FORWD($r$,$p_T$)
21     $\varsigma.\Pi \leftarrow \varsigma.\Pi \cup \{\langle e.\text{tgt}, p_T \rangle\}$
22   else
23     ANALYSEACT($e$,$p_M$) # Analyse event
24     $p'_T \leftarrow r$.rtr
25     $\varsigma \leftarrow$ INSTRUMENT$_\bullet$($\varsigma$,$e$,$p'_T$)
26   return $\varsigma$

27 def HANDLEEXIT$_\bullet$($\varsigma$,$r$,$p_M$)
28   $e \leftarrow r$.emb
29   if (($p_T \leftarrow \varsigma.\Pi(e.\text{src})$) $\neq \bot$)
30     FORWD($r$,$p_T$)
31   else
32     ANALYSEACT($e$,$p_M$) # Analyse event
33     $\varsigma.\Gamma \leftarrow \varsigma.\Gamma \setminus \{\langle e.\text{src}, \bullet \rangle\}$ # Remove dead e.src
34     TRYGC($\varsigma$,$p_M$)
35   return $\varsigma$

36 def HANDLECOMM$_\bullet$($\varsigma$,$r$,$p_M$)
37   $e \leftarrow r$.emb
38   if (($p_T \leftarrow \varsigma.\Pi(e.\text{src})$) $\neq \bot$)
39     FORWD($r$,$p_T$)
40   else
41     ANALYSEACT($e$,$p_M$) # Analyse event

Expect: $r.\text{emb.iss} = \text{self}() \vee \varsigma.\Pi(r.\text{emb.tgt}) \neq \bot$
42 def HANDLEDTC($\varsigma$,$r$,$p_M$)
43   $c \leftarrow r$.emb
44   if (($p_T \leftarrow \varsigma.\Pi(c.\text{tgt})$) $\neq \bot$)
45     FORWD($r$,$p_T$)
46   else
47     $\varsigma.\Gamma \leftarrow \varsigma.\Gamma \setminus \{\langle c.\text{tgt}, \bullet \rangle\}$
48     $\varsigma.\Gamma \leftarrow \varsigma.\Gamma \cup \{\langle c.\text{tgt}, \circ \rangle\}$
49     $\gamma = \{\langle p_S, d \rangle \mid \langle p_S, d \rangle \in \varsigma.\Gamma, d = \bullet\}$
50     if ($\gamma = \emptyset$) # All processes in $\Gamma$ are detached
51       LOOP$_\circ$($\varsigma$,$p_M$) # Switch tracer to $\circ$ mode
52   return $\varsigma$
```

Listing 4. *Tracer loop that handles priority ($\bullet$) trace events and message forwarding*

Tracers circumvent this issue by *prioritising* the processing of routed event messages. This captures the invariant that routed events temporally precede all other events that are to be analysed by the tracer. A tracer operates on two levels, *priority* mode and *direct* mode, respectively denoted by ● and ○ in our algorithm. Figure 5.5b shows that when in priority mode, T_Q dequeues and handles the routed events ? and $\rightarrowtriangle$ (labelled by ●) first; ? is analysed in step ㉓, whereas $\rightarrowtriangle$ results in the instrumentation of tracer T_R in step ㉕ of figure 5.5b. Note that T_Q can still receive trace events directly from process Q while this handling of events underway. However, the *direct trace events* from Q are only considered once T_Q transitions to direct mode. Newly-instrumented tracers default to *priority* mode to process routed events first (see line 7 in listing 5 of appendix A).

LOOP$_\bullet$ in listing 4 shows the logic that prioritises the processing of routed events dequeued on line 3 and handled on line 6. The operations HANDLEFORK, HANDLEEXIT, and HANDLECOMM in LOOP$_\circ$ and LOOP$_\bullet$ in listings 3 and 4 handle trace events differently. In direct mode, a tracer can (i) analyse trace events, (ii) forward the events that have been routed its way to neighbouring tracers, or (iii) start routing events that it directly collects when these need to be handled by other tracers. By contrast, tracers in priority mode only handle routed trace events according to (i) and (ii), *e.g.* the branching statement on lines 19 to 25 in listing 4, and *no* routing is performed.

5.2.5 Detaching Tracers

A tracer in *priority* mode coordinates with the router tracer associated with a particular system process that it traces to determine when all of the process trace events have been routed to it. Each tracer keeps a record of the processes it traces in the *traced-component map*, $\Gamma : \textsc{Pid}_{\text{S}} \rightharpoonup \{\circ, \bullet\}$. Entries to Γ are added when the tracer starts collecting events for a process (lines 7 and 16 in listing 2) and removed when processes terminate (lines 31 in listing 3 and 33 in listing 4). Coordination with the router is effected by a tracer in priority mode for *every* process in Γ, before the tracer can safely transition to direct mode and start operating on the events it collects directly. The tracer issues a special detach command message, c, with type dtc, to notify the router tracer that it is now responsible for tracing a particular system process. The dtc command contains the PID of the tracer issuing the request and the PID of the system process to be detached from the router tracer. These are read respectively via the fields c.iss and c.tgt. A tracer marks a process as detached by updating its mapping $c.\text{tgt} \mapsto \bullet$ in Γ to $c.\text{tgt} \mapsto \circ$ (see lines 47 and 48 in listing 4).

Figure 5.5b shows T_Q in priority mode sending command $\langle \text{dtc}, \text{q}_{\text{T}}, \text{q}_{\text{S}} \rangle$ for Q, step ⑬, after it starts tracing this process in step ⑫. This transaction is implemented by Detach on line 15 in listing 2 (see appendix A). The dtc command issued by T_Q is deposited in the message buffer of (router tracer) T_P after the events ? and ⇾. T_P processes the contents of its message buffer sequentially in steps ⑩, ⑰, ⑲, ⑳ and ㉘, and forwards ? and ⇾ to T_Q, steps ⑱ and ㉑. It also routes the dtc command *back* to the issuer tracer T_Q, step ㉙. T_Q eventually handles the events forwarded by T_P in the correct order, as stipulated by figure 5.2b (steps ㉓ and ㉔). It then handles dtc in step ㉚, marking process Q as detached. This update on the traced-component map Γ of T_Q is performed by HandleDtc in listing 4 on lines 47 and 48. A tracer transitions to direct mode once *all* the processes in its Γ are marked as detached; see lines 49 and 50 in listing 4. For the case of T_Q in figure 5.5b, this transition takes place in step ㉛ when the single process Q that it traces is detached. Finally, T_Q handles event $*$ in the correct order in step ㉜ (as opposed to step ⑱ in figure 5.5a).

A detach command $\langle \text{dtc}, p_{\text{T}}, p_{\text{S}} \rangle$ that is directed to some tracer p_{T} by a router tracer may perform multiple hops before it reaches p_{T}. Every tracer *en route* to p_{T} purges the mapping for p_{S} from its routing map Π once it forwards dtc to the neighbouring tracer. This clean-up logic is performed by RouteDtc and ForwdDtc in listing 3. Figure 5.5b does not illustrate this flow. However, we remark that after receiving dtc, T_P would remove from Π the mapping $\text{q}_{\text{S}} \mapsto \text{q}_{\text{T}}$, calling RouteDtc to route back the detach command $\langle \text{dtc}, \text{q}_{\text{T}}, \text{q}_{\text{S}} \rangle$ it receives from T_Q. Similarly, T_P removes $\text{r}_{\text{S}} \mapsto \text{q}_{\text{T}}$ for R once it handles $\langle \text{dtc}, \text{r}_{\text{T}}, \text{r}_{\text{S}} \rangle$ from T_R. When T_Q receives the routed detach command $\langle \text{rtd}, \text{p}_{\text{T}}, \langle \text{dtc}, \text{r}_{\text{T}}, \text{r}_{\text{S}} \rangle \rangle$ from T_P, it removes $\text{r}_{\text{S}} \mapsto \text{r}_{\text{T}}$ from Π and forwards it, in turn, to T_R.

5.2.6 Selective Instrumentation

To monitor multiple processes as one component, rather than having a dedicated monitor for each as in example 5.1, our algorithm uses the *instrumentation map* discussed in section 4.4. The signature g, carried as part of the fork trace event e, can be retrieved using the field e.sig; see table 5.1. Listing 2 shows the instrumentation operations Instrument that apply Φ to e.sig (lines 3 and 11) to check whether a process is eligible for instrumentation. When $\Phi(e.\text{sig}) = \bot$, no instrumentation is effected, and the tracer becomes automatically shared by the new process e.tgt, as per assumptions A_3 and A_5.

5.2.7 Garbage Collection

Our outline instrumentation can shrink the tracer choreography by discarding unneeded tracers. Apart from determining whether a tracer can be terminated based on flagged monitoring verdicts (refer to introductory part of section 5.2), the algorithm checks that both the routing Π *and* traced-component Γ maps of the tracer are empty. A tracer purges process references from Γ when handling exit trace events via $\textsc{HandleExit}_\circ$ and $\textsc{HandleExit}_\bullet$ (listings 3 and 4). When $\Gamma = \emptyset$ and a tracer has no processes to analyse, it could still be required to forward events to neighbouring tracers, *i.e.,* $\Pi \neq \emptyset$. Therefore, the garbage collection check, $\textsc{TryGC}$, is performed each time mappings from Π or Γ are removed; see lines 32, 43 and 57 in listing 3, and line 34 in listing 4.

5.3 Correctness Validation

The decentralised outline algorithm of section 5.1 is assessed in two stages. First, we confirm its *implementability* by instantiating the core logic of listings 2 to 4 to Erlang, which is tailored for the demands of reactive systems (see section 1.2). Our development follows a test-driven approach [38] to ensure that the tracer logic is implemented correctly. Second, we validate the correctness of our implementation by augmenting the logic given in listings 2 to 4 with runtime checks that guarantee a number of invariants [22] w.r.t. message routing between tracers.

5.3.1 Implementability

Our implementation of decentralised outline instrumentation maps the tracer processes to Erlang actors, where the logic detailed in listings 2 to 4 is directly translatable to Erlang code. We implement the routing (Π), instrumentation (Φ), and traced-component (Γ) maps that represent the tracer state ς as Erlang maps for efficient access. The tracer mailbox coincides with the message buffer κ of section 5.1.1 and figure 3.3 used for asynchronous communication. Every tracer obtains events from components of the SuS by leveraging the native tracing infrastructure exposed by the EVM [57] that deposits event messages inside the mailbox of the calling tracer. The EVM tracing complies with assumptions A_3 and A_4, *i.e.,* a system process can be traced by at most one tracer, although one tracer may trace multiple processes. To meet assumption A_5, we configure the EVM tracing with the `set_on_spawn` [57] flag that instructs the infrastructure to atomically set newly-created child processes to use the tracer of their parent, thereby preventing trace event loss. In addition, we use the `send`, `receive`, and `procs` tracing flags that inform the EVM to only emit trace event messages for `send`, `receive`, `spawn` (*i.e.,* fork) and `exit` process actions. One advantage of the EVM is that it can natively trace any program that is compiled to BEAM, making our instrumentation algorithm accessible to languages that produce this type of intermediate object code, *e.g.* Clojerl [92], Elixir [142]. For instance, our implementation has been used to verify parts of the RAFT [190] consensus algorithm written in Elixir [162]. The implementation we give covers both the externalised and internalised analysis variants of figure 5.1[2].

[2] The full source code can be found on the GitHub repository: `https://github.com/duncanatt/detecter`.

5.3.2 Invariant Implementation

One salient aspect that our algorithm addresses is that of reporting SuS trace events to the analysis component in a *reliable* manner; this is demanded by constraint C_7. The invariants listed below ensure the correct handling of events by tracers. Together with the core logic of listings 2 to 4, these enable us to reason about general properties the tracer choreography should observe. For instance, our algorithm guarantees that 'every trace event that is routed between tracers eventually reaches the intended tracer', that 'the monitor choreography grows dynamically', and that 'redundant tracers are always garbage collected'. We implement these invariant checks in the form of assertions. The invariants below make use of the following two notions introduced earlier:

- *direct trace event* (recalled from section 5.2.4): an event that is not routed but collected straight from a system process via the tracing infrastructure.
- *router tracer* (recalled from section 5.2.3): a tracer that receives the trace events of a system process that are meant to be handled by a *another* tracer.

Tracer choreography invariants Ensure that the dynamic trace event routing topology between tracers always maintains a DAG.

I_1 A tracer has a corresponding analyser.

I_2 The root tracer has *no* router tracers.

I_3 A tracer never terminates unless its routing map, Π, *and* traced-component map, Γ, are empty.

I_4 A tracer never adds a process that already exists in its traced-component map Γ.

I_5 A tracer never removes a non-existing process from its traced-component map Γ.

I_6 A tracer acts on a ⇾ event by adding the process to its traced-component map Γ. Depends on invariant I_4.

I_7 A tracer acts on an ∗ event by removing the process from its traced-component map Γ. Depends on invariant I_5.

I_8 A tracer never adds a route that already exists in its routing map Π.

I_9 A tracer never removes a non-existing route from its routing map Π.

I_{10} A tracer acts on a ⇾ event by adding a route to its routing map Π. Depends on invariant I_8.

I_{11} A router tracer that routes a ⇾ event adds a route to its routing map Π. Depends on invariant I_8.

I_{12} A tracer that forwards a ⇾ event adds a route to its routing map Π. Depends on invariant I_8.

I_{13} A router tracer that routes a dtc command removes a route from its routing map Π. Depends on invariant I_9.

I_{14} A tracer that forwards a dtc command removes a route from its routing map Π. Depends on invariant I_9.

Message routing invariants Ensure that trace events are reported to analysers per definition 5.1, and depend on the guarantees given by invariants I_1 to I_{14}

I_{15} A tracer never routes *or* forwards a message unless a route exists in its routing map Π. Depends on invariants I_{10} to I_{12}.

I_{16} A tracer in ● mode prioritises routing messages until it switches to ○ mode.

I_{17} A tracer in ● mode transitions to ○ mode only when all of the processes in its traced-component map Γ are marked as ○ *or* Γ is empty.

I_{18} The total amount of dtc commands a tracer issues is equal to the sum of the number of processes in its traced-component map Γ *and* the number of terminated processes for the tracer. Depends on invariants I_6 and I_7.

I_{19} A tracer in ○ mode acts on a direct event by analysing *or* routing it. Depends on invariants I_1 and I_{15}.

I_{20} A tracer in ○ mode acts on a routed event by forwarding it. Depends on invariant I_{15}. Analysing a routed trace event in ○ mode implies that the tracer dequeued a priority event, violating invariant I_{16}.

I_{21} A tracer in ○ mode acts on a routed dtc command by forwarding it. Depends on invariants I_{14} and I_{15}. Handling a routed command in ○ mode implies that the tracer dequeued a priority command, violating invariant I_{16}.

I_{22} A tracer in ● mode acts on a routed event by analysing *or* forwarding it, *i.e.,* it *never* routes events. Only tracers in ○ mode can route events, and these events are direct events. Routing in ● mode implies that the tracer dequeued a non-priority event, violating invariant I_{16}.

I_{23} A tracer in ● mode acts on a routed dtc command by handling *or* forwarding it, *i.e.,* it never routes commands. Depends on invariants I_{14} and I_{15}. Only (router) tracers in ○ mode can route commands, and these are received directly from the tracers wishing to detach system processes from the router. Routing in ● mode implies that the tracer dequeued a non-priority command, violating invariant I_{16}.

I_{24} A router tracer that receives a dtc command must route it. Depends on invariants I_{13} and I_{15}. If routing is not possible, the command was issued by mistake.

We implement a suite of unit tests that exhaustively operate on the invariants listed above. These tests ascertain that race conditions are correctly handled by the tracer choreography while it simultaneously analyses trace events. Other tests validate the elasticity aspect of our algorithm in terms of the dynamic instrumentation of tracers and corresponding garbage collection. To drive these tests, we built a harness that can load and replay pre-scripted interleaving scenarios for various systems, such as the one of example 5.1. The harness adheres to assumptions A_3 to A_5 to emulate the native EVM tracing infrastructure. Our comprehensive suite of scenarios is specifically designed to exercise the core logic in listings 2 to 4 and induce edge-case behaviour.

We also use the invariants above in large-scale general tests that delegate the generation of interleaved executions directly to the EVM. Our aim is twofold: (i) we instrument independent monitors to track random groupings of processes, which implicitly controls the size of the traced-component map Γ, and (ii) the interleaving of processes induced by the EVM schedulers dictate how the routing map Π of each monitor evolves over time. This induces dynamic arrangements in the monitor choreography DAG and provides us with high assurances that the algorithm of listings 2 to 4 and its translation to Erlang code is correct. We accomplish (i) by overloading our instrumentation map definition of section 4.4, $\Phi(g)$, to admit a value, $\Pr(\mathit{instr})$, that controls the probability that a function signature g requires a monitor to be instrumented. This overload, $\Phi_{\Pr(\mathit{instr})}(g)$, is modelled on Bernoulli trial [191]. It returns a monitor m for g whenever $X \leq \Pr(\mathit{instr})$, *i.e.,* the Bernoulli trial succeeds, or $\bot$ otherwise; X is drawn from a uniform distribution on the real interval $[0,1]$. Gradually increasing the value of the parameter $\Pr(\mathit{instr})$ enables us to monitor a SuS

- centrally, via a singleton monitor, *i.e.,* $\Pr(\mathit{instr}) = 0$,
- in a fully-decentralised fashion with one monitor per process, such as example 5.1, *i.e.,* $\Pr(\mathit{instr}) = 1$, or
- as randomised groups of processes with independent monitors for each group, *i.e.,* $0 < \Pr(\mathit{instr}) < 1$.

For these tests, we employ the benchmarking framework described in the next chapter, using the same high loads as in chapter 7 (*e.g.* ≈ 40M trace events). The scalability and efficiency facets of our implemented algorithm are extensively treated in the latter chapter.

5.4 Discussion

This chapter proposes a first decentralised outline instrumentation approach for monitoring reactive systems. Section 5.2 details a concrete algorithm, describing how the instrumentation of a component-based SuS is attained in a scalable fashion by relying exclusively on trace events exhibited by the running system. Our reactive design sets itself apart from the state of the art in these aspects. It:

- asynchronously instruments the SuS without modifying it to minimise interference (*responsive*),
- delineates the SuS and monitor components to allow for independent failure (*resilient*),
- does not assume a fixed number of SuS components, but scales accordingly (*elastic*), and
- reorganises the monitor choreography dynamically in response to SuS trace events (*message-driven*).

The algorithm leverages the tracing concepts commonly provided by tracing infrastructures, which makes it applicable in cases where inlining cannot be used. This flexibility comes at the expense of introducing asynchrony between the SuS and monitor components, complicating our RV set-up. Our exposition in section 5.2 identifies the intricacies that the algorithm addresses in order to guarantee that trace events of the SuS are reported *and* analysed correctly (listings 2 to 4). We express our algorithm in terms of general software engineering concepts (*e.g.* encapsulated component states, separation of the routing and analysis concerns) to facilitate its adoption to a variety of settings and technologies. The algorithm presented is evaluated in two respects. First, section 5.3 confirms the *implementability* of choreographed outline instrumentation. It describes how our general algorithm of listings 2 to 4 can be naturally mapped to a tool implementation in a mainstream concurrent language. We augment this with an account of the principled approach employed to ensure the *correct* translation of our algorithm to code. Second, the claims on the reactive characteristics of our algorithm and its implementation are corroborated further via the empirical evaluation of chapter 7.

Our solution adopts a principle similar to the black-box-style of monitoring used by APM tools that are geared towards maintaining large-scale decentralised software. APMs operate externally to the SuS, similar to our approach. They are used extensively to identify and diagnose performance problems such as bottlenecks and hotspots; they presently have an edge on static analysis tools for critical path analysis [226] and unearthing performance anti-patterns [213, 214]. The methods proposed in section 5.2 are general enough to be applied—at least in part—to APM tools in order to make them more decentralised. Although our algorithm is implemented in Erlang, we argue that it is still sufficiently general to be instantiated to other language frameworks (*e.g.* Elixir, Akka for Scala [189], Thespian [194] for Python [173]) that follow constraints C_1 to C_4 and assumptions A_1 to A_5. In particular, it can be used by RV tools that target other platforms, such as the JVM.

Hyperlogics [66] have recently emerged as an expressive formalism for describing complex properties about decentralised systems (*e.g.* non-interference, non-inference, *etc.*). Broadly, these logics can specify conditions across distinct traces, where quantifications range over potentially *infinite* trace domains. One branch in this line of study is the verification of such properties at runtime (*e.g.* [44, 106, 12]). Although we are unaware of any attempts at runtime verifying such properties using outline instrumentation, the inherent dynamicity required to analyse an unbounded number of traces would certainly make our instrumentation method applicable in this setting. Our approach from section 5.2 already disentangles the instrumentation from the analysis, thus providing a platform for plugging new analyses that implement monitoring for hyperproperties.

5.4.1 Related Work

There are other bodies of work that address decentralised monitoring besides the ones already discussed (see also section 1.1.2). The majority of these studies instrument monitors via inlining. For instance, Sen et al. [210] study decentralised monitors that are attached to different threads to extract and analyse trace events internally; see figure 5.1b. In their earlier work, Sen et al. [208] investigate the use of decentralised monitors on distributed SuS components, focussing on the communication efficiency between monitors. Another line of research by Scheffel and Schmitz [203] uses the same instrumentation approach as [210, 208], but employs a past-time three-valued temporal logic in contrast to the two-valued logic used in the former studies. Efficient communication is also the focus of Mostafa and Bonakdarpour [180]. In their setting, the SuS consists of distributed asynchronous processes that interact via message-passing over reliable channels. Similar to our case, their monitoring algorithm does not rely on a global notion of timing (constraint C_1), nor does it tackle aspects of failure (assumptions A_1 and A_2). The work by Basin et al. [31] is one of the few that considers distributed system monitoring where components and network links may fail. While their algorithm does not employ a global clock, it is based on the timed asynchronous model for distributed systems [75] that assumes highly-synchronised physical clocks across nodes. In a different spirit, [45, 110] address the problem of crashing monitors; this is something that we presently do not address, although our decentralised set-up enables us to fail partially (see section 5.3).

Other efforts for decentralised monitoring, such as [138, 87, 148, 207], weave the SuS with code instructions that extract trace events and delegate their analysis to independent processes—this mirrors our externalised event analysis variant of figure 5.1b. While these approaches are occasionally classified as outline [100], they do not treat the SuS as a black box, making them prone to the shortcomings of inlining discussed in section 2.1.4. Crucially, the aforecited works assume a *static* system arrangement, which spares them the challenges of dealing with the dynamic reconfiguration of outline tracers and reordering of tracer events.

Tools such as [185, 219] target the Erlang ecosystem. In Neykova and Yoshida [185], the authors propose a method that statically analyses the program communication flow that is specified in terms of a multiparty protocol. Monitors attached to system processes then check that the messages received coincide with the projected local type (similar to the analysis conducted by our monitors), and in the case of failure, the associated processes are restarted. The authors show that their recovery algorithm induces less communication overhead and improves upon the static process structure recovery mechanisms offered by the Erlang/OTP platform. Similarly, Attard and Francalanza [219] focus on decentralised outline monitoring in a concurrent setting, but assume a static SuS. By contrast to Neykova and Yoshida [185], they leverage the native tracing infrastructure offered by the EVM, as done in other tools such as [113, 21, 222, 51, 71] for centralised monitoring set-ups.

Schneider et al. [205] follow a different approach to the ones mentioned thus far to achieve independent monitors. Unlike our setting that concentrates on local properties (see section 1.2), the authors tackle the general monitoring case where slicing can lead to event duplication that, in turn, inflates runtime overhead. The set-up proposed by the authors is external to the SuS and extends their prior work [32] that targets scalable offline monitoring. It adapts database hash-based partitioning techniques to the monitoring setting, in order to alleviate the overhead induced by slicing. These techniques are implemented in an automatic data slicer that runs on Apache Flink, where trace event streams are obtained via log files or TCP sockets. They can achieve scalability by using data parallelisation to treat monitoring algorithms as a black box, running them on different segments of the trace. The monitoring algorithm of listing 1 that we attach to tracers is an instantiation of this approach. One aspect that distinguishes our setting from that of Schneider et al. [205] is that the event source they use is sequential, whereas ours becomes concurrent when tracers invoke the operation Preempt to partition the trace. Our trace event routing detailed in section 5.2 ensures that trace events are reported to the correct monitors, despite the reordering that may arise from these partitions. We note that the runtime overhead in *op. cit.* is less detrimental to the SuS since their RV set-up is deployed externally, which is not possible in our case. It is worth mentioning that for their evaluation, Schneider et al. [205] develop a tool to emulate online monitoring scenarios by replaying them from a file; this approach is analogous to the one we use when evaluating our algorithm in section 5.3.2.

6 Reactive Runtime Monitoring Benchmarking

Instrumenting a SuS with monitors induces inevitable runtime overhead that should be kept minimal since this impacts the applicability of monitoring tools [95, 100]. While the worst-case complexity bounds for monitor-induced overheads can be calculated via standard methods (see, *e.g.* [154, 44, 7, 114]), benchmarking is, by far, the preferred method for assessing these overheads [25, 119]. One reason for this is that benchmarks tend to better represent the overhead observed in practice [123, 49]. Benchmarking also provides a *common platform* for gauging workloads, making it possible to *compare* different monitoring tools, or rerun experiments to *reproduce* and *confirm* existing results.

This chapter presents a benchmarking framework for evaluating runtime monitoring tools written for reactive component systems. The framework we describe generates synthetic system models following the master-worker paradigm [202]. This architecture is pervasive in both distributed (*e.g.* Big Data frameworks, render farms) and concurrent (*e.g.* web servers, thread pools) system settings [217, 121, 77, 227], which justifies our aim in building a benchmarking tool targeting this paradigm. We:

- detail the design of a configurable benchmarking tool that emulates various master-worker models under commonly-observed load profiles and gathers relevant metrics that give a *multi-faceted* view of runtime overhead, Section 6.1;
- demonstrate that our synthetic benchmarks can be tuned to approximate the *realistic behaviour* of web server traffic with high degrees of fidelity and repeatability, Section 6.4;
- present a case study that (i) shows how the load profiles and parametrisability of benchmarks can produce edge cases that can be measured through our performance metrics to asses runtime monitoring tools in a *comprehensive* manner, and (ii) confirms that the results from (i) coincide with those obtained via a *real-world* use case using OTS software, Section 6.5.

6.1 A Configurable Benchmark Design

Our benchmarking tool addresses the limitations discussed in section 1.1.3. The set-up scales to accommodate high loads and emulates a range of system models that can be subjected to various load profiles that are typically observed in practice. It collects three core metrics to give a comprehensive view of runtime overhead that captures the operation of reactive components, namely the

(i) mean *response time*, measured in milliseconds (ms), that captures how the reactiveness of the SuS is affected when monitors are introduced,

(ii) mean *memory consumption*, recorded in GB, that gauges the impact monitors have on the SuS, and

(iii) mean *scheduler utilisation*, as a percentage of the total available processing capacity, that shows how well the monitors under evaluation maximise its use.

While the mean *execution duration*, measured in seconds (s), is the least relevant metric (see section 1.1.3), we track it in our experiments to indicate to readers the amount of time that monitors require to complete their runtime analysis. Henceforth, we use the shortened metric name (*e.g.* response time instead of mean response time, *etc.*) for the sake of brevity.

Our tool considers master-worker architectures, where one central process, called the *master*, creates and allocates tasks to *worker* processes [202]. Workers process tasks concurrently and relay the result to the master when ready; the latter then combines these results to yield the final result. Each worker is an *abstraction* of sets of cooperating processes that can be treated as a single unit. We focus on reactive architectures that execute on a single node, although our design adheres to the three criteria that facilitate its extension to a distributed setting. Specifically, master and worker components: (i) share neither a common clock, (ii) nor memory, and (iii) communicate exclusively via asynchronous messages. Our model assumes that communication is reliable and components do not fail (see section 1.2)[1]. Table 6.1 on page 70 summarises the benchmark parameters that are described next in sections 6.1.1 to 6.1.3 and 6.1.5.

6.1.1 Load Generation

Load on the system is induced by the master when it creates worker processes and allocates *tasks*. The total number of workers in one benchmark run can be set via the parameter n. Tasks are allocated to worker processes by the master and consist of one or more *work requests* that a worker receives, handles, and transmits back. A worker terminates its execution when all of its allocated work requests have been processed *and* acknowledged by the master. The number of work requests that can be batched in a task is controlled by the parameter w; the *actual* batch size per worker is then drawn randomly from a normal distribution with mean $\mu = w$ and standard deviation $\sigma = \mu \times 0.02$. This induces a modicum of variability in the amount of work requests exchanged between the master and worker processes. The master and workers communicate asynchronously: an allocated work request is delivered to the incoming *task queue* of a worker process where it is eventually handled. Work responses issued by a worker are queued and processed similarly on the master.

6.1.2 Load Configuration

We consider three load profiles (see figure 6.5 for examples) that determine how the creation of workers is distributed along the load timeline, specified by the parameter t. The timeline is modelled as a sequence of *discrete logical time units* that represent instants at which a new set of workers is created by the master. *Steady* loads replicate executions where a system operates under stable conditions. These are modelled on a homogeneous Poisson distribution with *rate* λ, specifying the mean number of workers that are created at each time instant along the load timeline with duration $t = \lceil n/\lambda \rceil$. *Pulse* loads emulate settings where a system experiences gradually increasing load peaks. The Pulse load shape is parametrised by t and the *spread*, s, that determines how slowly or sharply the system load increases as it approaches its maximum peak, halfway along t. Pulses are modelled on a normal distribution with $\mu = t/2$ and $\sigma = s$. *Burst* loads capture scenarios where a system is stressed due to load spikes; these are based on a log-normal distribution with $\mu = \ln(m^2/\sqrt{p^2+m^2})$ and $\sigma = \sqrt{\ln(1+p^2/m^2)}$, where $m = t/2$, and parameter

[1] This coincides with our process model introduced in section 5.1 that fulfils constraints C_1 to C_4 and assumptions A_1 and A_2.

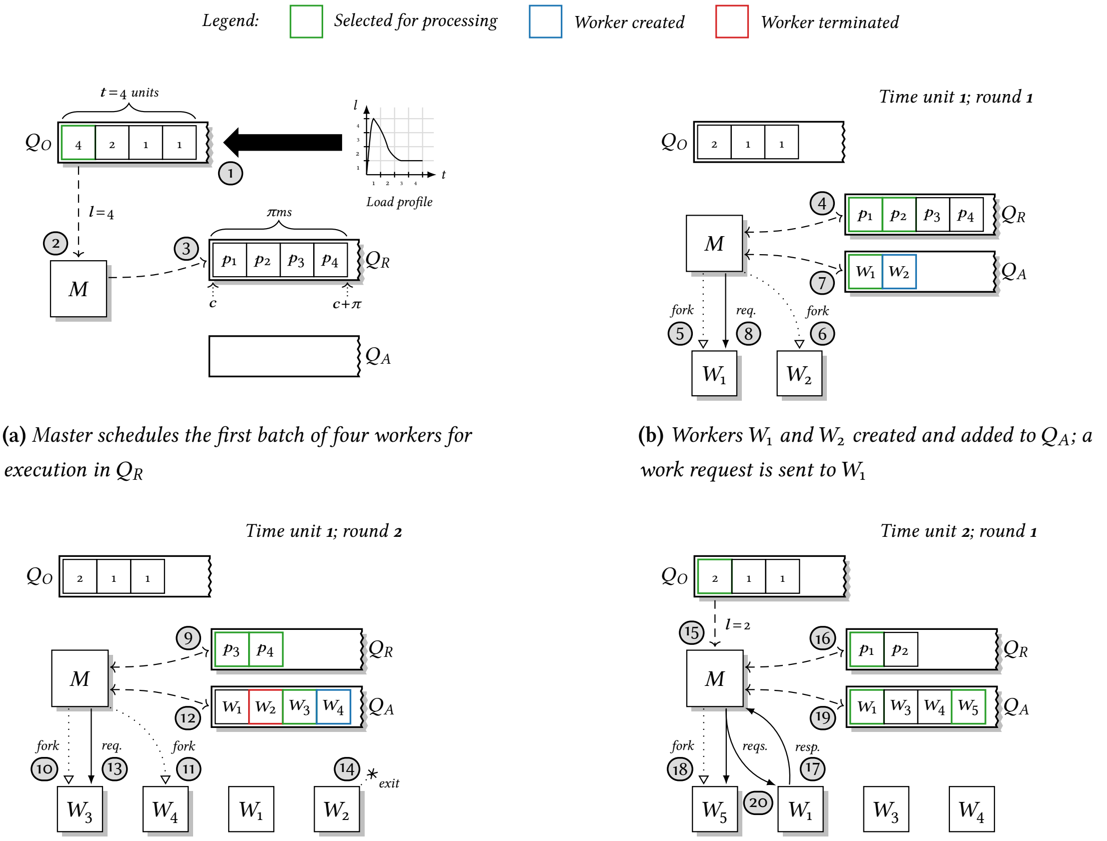
p is the *pinch* controlling the concentration of the initial load burst.

6.1.3 Wall-Clock Time

A load profile created for some logical timeline t is put into effect by the master process when the system starts running. The master *does not* create the worker processes that are set to execute in a particular time unit all at once, since this naïve strategy risks saturating the system, deceivingly increasing the load. In following this strategy, the system may become overloaded not because the mean request rate is high, but because the created workers overwhelm the master when they send their requests simultaneously. We address this issue by introducing the notion of *concrete time* that maps one discrete time unit in t to wall clock time *period*, π. The parameter π is given in ms, and defaults to 1000 ms.

6.1.4 Worker Scheduling

The master process employs a scheduling scheme to distribute the creation of workers uniformly across the period π. It makes use of three queues: the *Order* queue, *Ready* queue, and *Await* queue, denoted by Q_O, Q_R, and Q_A respectively. Q_O is initially populated with the load profile, step ① in figure 6.1a. A load profile consists of an array, $l_1, l_2, \ldots, l_t$, with t elements—each corresponding to a discrete time instant



(a) *Master schedules the first batch of four workers for execution in Q_R*

(b) *Workers W_1 and W_2 created and added to Q_A; a work request is sent to W_1*

(c) *Workers W_3 and W_4 created and added to Q_A; worker W_2 completes its execution*

(d) *Q_R becomes empty; master schedules the next batch of two workers*

Figure 6.1. *Master M scheduling worker processes W_j and allocating work requests*

in t—where the value l_i of every element indicates the number of workers to be created at that instant. Workers, $W_1, W_2, \ldots, W_n$, are scheduled and created in *rounds*, as follows. The master picks the first element from Q_{O} to compute the upcoming schedule, step ②, that starts at the *current* time, c, and finishes at $c+\pi$. A series of l_i time points, $p_1, p_2, \ldots, p_{l_i}$, in the schedule period π are *cumulatively* calculated by drawing the next p_k from a normal distribution with $\mu = \lceil \pi / l_i \rceil$ and $\sigma = \mu \times 0.1$. Each time point stipulates a moment in *wall-clock* time when a new worker W_j is to be created; this set of time points is *monotonic* and constitutes the Ready queue, Q_{R}, step ③. The master checks Q_{R}, step ④ in figure 6.1b, and creates the workers whose time point p_k is smaller than or equal to the current wall-clock time[2], steps ⑤ and ⑥ in figure 6.1b. The time point p_k of a newly-created worker is removed from Q_{O}, and a corresponding entry for the worker W_j is appended to the Await queue Q_{A}; this is shown in step ⑦ for W_1 and W_2. Workers in Q_{A} are now ready to receive work requests from the master process, *e.g.* step ⑧. Q_{A} is traversed by the master at this stage so that work requests can be allocated to existing workers. The master continues processing queue Q_{R} in subsequent rounds, creating workers, issuing work requests, and updating Q_{R} and Q_{A} accordingly, as shown in steps ⑨ to ⑬ in figure 6.1c. At any point, the master can receive responses, *e.g.* step ⑰ in figure 6.1d; these are *buffered* inside the incoming task queue of the master process and handled once the scheduling and work allocation phases are complete. A *fresh* batch of workers from Q_{O} is scheduled by the master whenever Q_{R} becomes empty, step ⑮, and the described procedure is repeated. The master stops scheduling workers when all the entries in Q_{O} are processed. It then transitions to *work-only* mode, where it continues allocating work requests and handling incoming responses from workers.

6.1.5 System Responsiveness

Systems generally respond to load with differing rates, due to the computational complexity of the task at hand, IO, or slowdown when the system itself becomes gradually loaded. We simulate these phenomena using the parameters $\Pr(send)$ and $\Pr(recv)$. The master *interleaves* the processing of work requests to allocate them uniformly among the various workers: $\Pr(send)$ and $\Pr(recv)$ bias this behaviour. Concretely, $\Pr(send)$ controls the probability that a work request is sent by the master to a worker, whereas $\Pr(recv)$ determines the probability that a work response received by the master is processed. Sending and receiving is *turn-based* and modelled on a Bernoulli trial [191]. The master picks a worker W_j from Q_{A} and sends *at least* one work request when $X \leq \Pr(send)$, *i.e.,* the Bernoulli trial succeeds; X is drawn from a uniform distribution on the interval $[0,1]$. Further requests to the *same* worker are allocated following this scheme (steps ⑧, ⑬ and ⑳ in figure 6.1) and the entry for W_j in Q_{A} is updated accordingly with the number of work requests remaining. When $X > \Pr(send)$, *i.e.,* the Bernoulli trial fails, the worker misses its turn, and the next worker in Q_{A} is picked. The master also queries its incoming task queue to determine whether a response can be processed. It dequeues one response when $X \leq \Pr(recv)$, and the attempt is repeated for the next response in the queue until $X > \Pr(recv)$. The master signals workers to terminate once it acknowledges all of their work responses (*e.g.* step ⑭). Due to the load imbalance that may occur when the master becomes overloaded with work responses relayed by workers [202], dequeuing is attempted $|Q_{\text{A}}|$ times. This encourages an even load distribution in the system as the number of workers *fluctuates* at runtime.

[2] We assume that the platform scheduling the master and worker processes is *fair*.

Parameter	Description
Master-Worker Model	
n	Total number of worker processes
w	Number of work requests batched in a task
t	Load timeline (not specified for Steady loads)
π	Wall clock time period
Load Profile	
λ	Steady *rate*
s	Pulse *spread*
p	Burst *pinch*
System Reactiveness	
$\Pr(send)$	Probability that the master issues a work request
$\Pr(recv)$	Probability that the master dequeues a work response

Table 6.1. *Load profile and system reactiveness configuration parameters for benchmarks*

6.2 Implementability

We instantiate the set-up of section 6.1 in Erlang. Our implementation maps the master and worker processes to actors, where workers are forked by the master via the Erlang BIF `spawn()`; in Akka and Thespian `ActorContext.spawn()` and `Actor.createActor()` can be respectively used to the same end. The work request queues for both master and worker processes coincide with actor mailboxes. We abstract the task computation and model work requests as Erlang messages. Workers emulate no delay, but respond instantly to work requests once these have been processed; delay in the system can be induced via parameters $\Pr(send)$ and $\Pr(recv)$ introduced in section 6.1.5. To maximise efficiency, the Order, Ready, and Await queues used by our scheduling scheme are maintained *locally* within the master. The master process keeps track of other details, such as the total number of work requests sent and received to determine when the system should stop executing. For the purposes of experiment taking, we extend the parameters of table 6.1 with a *seed* parameter, r, to fix the Erlang pseudorandom number generator to output reproducible number sequences.

6.3 Measurement Collection

The measurement of application performance is closely linked with the functionality offered by the platform on which benchmarks execute, and one typically leverages native operations to maintain low overhead levels. Our implementation relies on the BIFs provided by Erlang to gather the metrics identified in section 6.1 (response time, memory consumption, and scheduler utilisation). These are collected centrally via a designated process, called the *Collector*, that samples the runtime to obtain periodic snapshots of the execution environment (see figure 6.2). We use global sampling and avoid

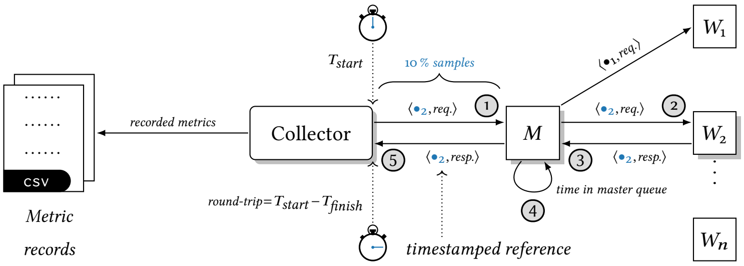


Figure 6.2. *Collector tracking the round-trip time for work requests and responses*

tracking the resource usage per process to minimise any potential perturbations that may be induced by our measurement taking. This is crucial in high-concurrency settings where components tend to be very sensitive to latency [127]. Our sampling frequency is set to 500 ms. This figure was determined empirically, whereby the measurements gathered are neither too coarse, nor excessively fine-grained such that the sampling itself affects the runtime. Every sampled snapshot combines the aforementioned metrics and formats them as records that are written asynchronously to disk to minimise IO delays.

The memory and scheduler readings are gathered via the EVM. We record the scheduler utilisation, rather than the CPU used by the EVM since the latter keeps scheduler threads momentarily spinning to avoid going to sleep and impacting latency [132]. The overall system responsiveness is reflected in the mean response time metric. To track this value, the Collector exposes a hook that the master uses to obtain *unique timestamps*, step ① in figure 6.2. These are embedded in every work request message the master issues to workers. Each timestamp enables the Collector to track the time taken for a specific message to travel from the master to a worker and back, *including* the time it spends in the mailbox of the master until dequeued, *i.e.,* the round-trip in steps ② to ⑤. To efficiently compute the response time, the Collector samples the total number of messages exchanged between the master and workers and calculates the running mean using the algorithm by Welford [224].

6.4 Benchmark Expressiveness and Coverage

We tune the synthetic system models generated by our benchmarking tool implementation via a series of empirical experiments to evaluate it in several ways. Section 6.4.2 discusses sanity checks for its measurement collection mechanisms and section 6.4.3 assesses the repeatability of the results obtained from synthetic system model executions. Sections 6.4.4 and 6.4.5 provide evidence that the tool is sufficiently expressive to cover a number of execution profiles that emulate realistic scenarios. In particular, we establish a set of benchmark configuration parameter values to create experiment set-ups whose behaviour approximates that of web server systems typically found in practice.

6.4.1 Experiment Set-up

An *experiment* consists of ten benchmarks. Each experiment is performed by running the benchmarked set-up with increasing loads, applied in steps of $n/10$, where n is the total number of worker processes (see table 6.1). Every benchmark is executed on a fresh instance of the EVM to ensure that the runtime environment is uninfluenced by previous runs. All experiments in this chapter are conducted on an Intel

Core i7 M620 64-bit machine with 8GB of memory, running Ubuntu 18.04 LTS and Erlang/OTP 22.2.1.

The parameters of the benchmarking tool can be configured to model a range of master-worker scenarios. However, not all of these configurations yield meaningful system models in practice. For example, setting $\Pr(send) = 0$ does not enable the master to allocate work requests to workers; with $\Pr(send) = 1$, the work allocation is enacted sequentially, defeating the purpose of a concurrent master-worker system. The objective is thus, to tune the benchmarking tool to generate different models of the master-worker set-up and find valid parameter values that enable our experiments to adequately approximate the behaviour of realistic web server systems. Our experiments are fixed with $n = 500\text{k}$ workers and $w = 100$ work requests per worker. This configuration generates $\approx n \times w \times (\text{work requests and responses}) = 100\text{M}$ message exchanges between the master and worker processes. We initially set $\Pr(send) = \Pr(recv) = 0.9$ and focus on Steady loads (*i.e.,* Poisson process) since these can be replicated using industry-strength load testing tools such as Tsung [186], Gatling [74] and JMeter [109]. Figure 6.5 (left) shows the load applied at each benchmark run, *e.g.* on the tenth run, the benchmark creates $\approx 5\text{k}$ workers/s. In all experiments, the total loading time is set to $t = 100\,\text{s}$.

6.4.2 Measurement Precision

A series of trials were conducted to select the appropriate sampling window size for measuring the response time. This step is crucial, as it directly affects the capability of the benchmark to scale in terms of its number of worker processes and work requests while remaining responsive. The sampling frequency described in section 6.3 (see also figure 6.2) was calibrated by taking various window sizes over numerous runs for different load profiles ranging from $\approx 10\text{k}$ to $\approx 1\text{M}$ workers. These results were compared to the actual mean calculated on all the work request and response messages exchanged between master and workers. Window sizes close to 10 % yielded the best results ($\approx \pm 1.4\%$ discrepancy from the actual response time). Smaller window sizes produced excessive discrepancy; larger sizes induced noticeably higher system overhead. The precision of our measured samples, including the memory consumption and scheduler utilisation figures was cross-checked against readings obtained from the Erlang Observer tool [57] to confirm that these coincide.

6.4.3 Result Repeatability

Data variability affects the repeatability of experiments [103] and plays a role when determining the number of repeated readings, m, required before the data measured is deemed sufficiently representative. Choosing the lowest m is crucial when experiment runs are time-consuming. The coefficient of variation (CV) [81], *i.e.,* the ratio of the standard deviation to the mean, $\text{CV} = \sigma/\bar{x}$, can be used to establish the value of m empirically, as follows. Initially, the CV_m for one batch of experiments for some number of repetitions m is calculated. The result is then compared to the $\text{CV}_{m'}$ for the next batch of repetitions $m' = m + b$, where b is the batch increment. When the difference between successive CV metrics, m' and m, is sufficiently small (for some ϵ), the value of m is selected, otherwise, the described procedure is repeated with m'. Crucially, the condition $\text{CV}_{m'} - \text{CV}_m < \epsilon$ must hold for *all* the variables measured in the experiment before m can be fixed. For the results presented next, the CV values have been calculated manually. The mechanism that determines the CV automatically is left for future work.

We minimise the data variability between experiments by seeding the Erlang pseudorandom number generator (parameter r in section 6.2) with a constant value. Fixing the seed typically requires fewer repeated runs before the metrics of interest—response time, memory consumption, and scheduler utilisation—converge to an acceptable CV. We conduct experiments set with $m \in \{3,6,9\}$ repetitions to determine the least m that meets this condition. We obtained the CV values of 0.52 %, 0.15 %, and 0.17 % for the response time, memory consumption, and scheduler utilisation respectively using three repeated runs with threshold $\epsilon \approx 0.04\%$ against $m = 3$. Since these figures are sufficiently low, we adopt the number of repetitions $m = 3$ for all experiment runs in the sequel. Note that fixing the seed still permits our models to exhibit a degree of variability that stems from the inherent interleaved execution of components due to process scheduling.

6.4.4 Response Time Tuning

The responsiveness of master-worker systems correlates with the time each worker spends idle, which, in turn, affects the capacity of the system to handle workloads. For instance, the less frequently the master assigns tasks (*i.e.,* low throughput), the larger the portion of idle workers and the shorter the response time (*i.e.,* low latency). As this aspect can influence the results obtained when assessing runtime overhead, we use the parameters $\Pr(send)$ and $\Pr(recv)$ to regulate the speed with which the system reacts to load (refer to section 6.1.5). We illustrate how these parameters affect the overall performance of master-worker models set up with $\Pr(send) = \Pr(recv) \in \{0.1, 0.5, 0.9\}$. Figure 6.3 shows the results, where each performance metric (*e.g.* memory consumption, y-axis) is plotted against the total number of workers for ten benchmarks, starting at 50k up to 500k (x-axis). Our charts also plot the execution duration for reference.

With $\Pr(send) = \Pr(recv) = 0.1$, the system has the lowest response time out of the three configurations (bottom left), as indicated by the gradual linear increase of the plot. This confirms the fact that smaller loads enable worker processes to rapidly handle incoming work requests. As expected, this prolongs the execution duration, when compared to that of the system set with $\Pr(send) = \Pr(recv) \in \{0.5, 0.9\}$ (bottom right). The effect of idle workers can be gleaned from the relatively lower scheduler utilisation as well (top left). Idling increases the consumption of memory (top right) since the worker processes created by the master typically are kept alive for longer periods. By contrast, the plots set with $\Pr(send) = \Pr(recv) \in \{0.5, 0.9\}$ exhibit markedly lower gradients in the memory consumption and execution duration charts; corresponding linear slopes for these two settings can be observed in the response time chart. This indicates that values between 0.5 and 0.9 yield system models that (i) consume tolerable amounts of memory, (ii) execute to completion in a reasonable amount of time, and (iii) maintain a decent response time. Master-worker architectures are typically employed in high throughput, low latency settings, and using values smaller than 0.5 goes against this principle. In what follows, we opt for $\Pr(send) = \Pr(recv) = 0.9$ due to the negligible differences in the response time and execution duration between $\Pr(send) = \Pr(recv) = 0.5$ and $\Pr(send) = \Pr(recv) = 0.9$, but reasonably low memory consumption achieved using the latter setting.

6.4.5 Veracity of the Synthetic Models

Our benchmarks can be configured to closely model *realistic* web server traffic where the request intervals observed at the server are known to follow a Poisson process [126, 168, 144]. The probability

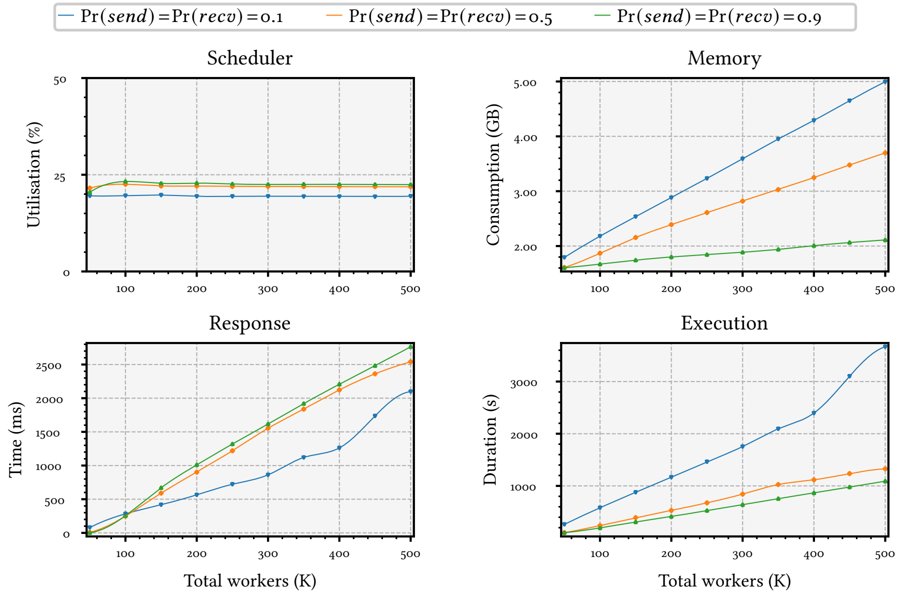


Figure 6.3. *System reactiveness benchmarks modelled by* Pr(*send*) *and* Pr(*recv*)

distribution of the response time of web application requests is generally right-skewed and approximates log-normal [126, 64] or Erlang distributions [144]. We conduct three experiments using Steady loads fixed with $n = 20\text{k}$ for $\Pr(send) = \Pr(recv) \in \{0.1, 0.5, 0.9\}$ to establish whether the response time in our system set-ups follows the aforementioned distributions. Our results, summarised in figure 6.4, are obtained by estimating the parameters for a set of candidate probability distributions (*e.g.* normal, log-normal, gamma, *etc.*) using maximum likelihood estimation [200] on the response time obtained from *each* experiment. We then perform goodness-of-fit tests on these parametrised distributions using the Kolmogorov-Smirnov test, selecting the most appropriate response time fit for each of the three experiments. The fitted distributions in figure 6.4 indicate that the response time of our system models concurs with the findings reported in [126, 64, 144]. This makes a strong case in favour of our benchmarking tool striking a balance between the *realism* of benchmarks based on OTS programs and the *controllability* offered by synthetic benchmarking. Lastly, we point out that figure 6.4 matches the observations made in figure 6.3, which show an increase in the response time as the system throughput increases. This is evident in the histogram peaks that grow shorter as $\Pr(send) = \Pr(recv)$ progresses

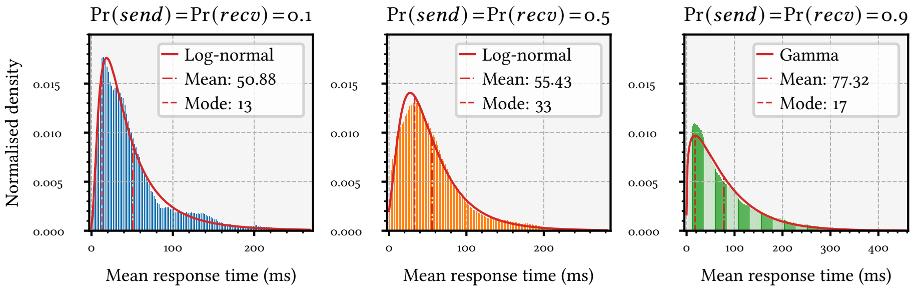


Figure 6.4. *Fitted probability distributions on response time for Steady loads for* 20k *workers*

from 0.1 to 0.9.

6.4.6 Load Profile Models

Our benchmarking tool implementation can generate the load profiles introduced in section 6.1.2, enabling us to gauge the behaviour of monitored systems under varying forms of strain. These loads make it possible to mock specific system scenarios that exercise different aspects of the monitoring tool being considered. For example, a benchmark configured with load bursts could uncover buffer overflows in a particular monitoring tool implementation that only arise under stress, when the length of the trace event processing queue exceeds some preset length. Figure 6.5 shows the distribution of Steady, Pulse, and Burst load that the master induces it creates worker processes with $n = 500\text{k}$.

6.5 Benchmark Validation

We demonstrate how our benchmarking tool can be used to assess the runtime overhead comprehensively via a concurrent RV case study. By controlling the benchmark parameters and subjecting the system to specific workloads, we show that our multi-faceted view of overhead reveals nuances in the observed runtime behaviour, benefiting the interpretation of empirical results. We further assess the veracity of these synthetic benchmarks against the overhead measured from a use case that is set up with industry-strength OTS software.

6.5.1 Runtime Monitoring Set-up

Our experiments use the implementation of the monitoring inlining tool discussed in section 4.5. The monitor code instructions that the tool injects share the process space of components of the SuS, which induces minimal runtime overhead. This enables us to scale benchmarks to considerably high loads, even on our modest experiment set-up of section 6.4.1.

We perform two sets of experiments. For the experiments of section 6.5.2 that focus on the synthetic master-worker models generated by our benchmarking tool, we use properties that ensure the correct operation of worker processes, along with properties that certify the validity of the tasks that workers receive from the master. Readers are directed to appendix B.1 for details about these properties. Section 6.5.3 considers the Cowboy web server introduced in section 4.6. The client request delegation that Cowboy performs to Ranch *protocol handlers* follows closely our master-worker set-up of section 6.1,

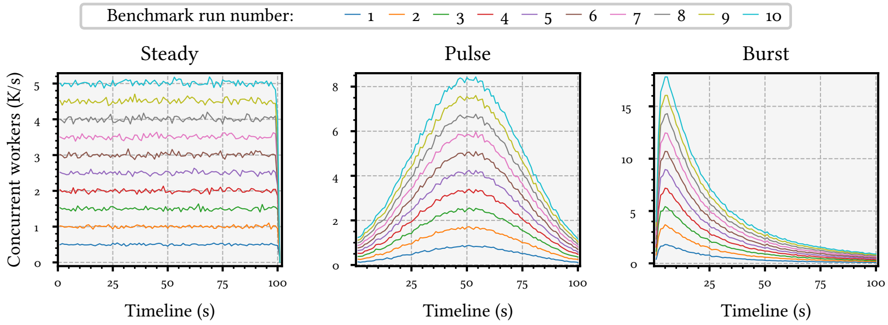
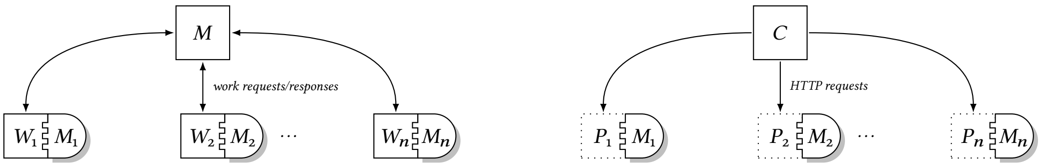


Figure 6.5. *Steady, Pulse and Burst load distributions of* 500 k *workers for* 100 s



(a) *Monitoring worker processes $W_1, W_2, \ldots, W_n$*

(b) *Monitoring Cowboy-Ranch protocol handlers $P_1, P_2, \ldots, P_n$*

Figure 6.6. *Master-worker and Cowboy-Ranch benchmarks instrumented with inline local monitors*

which abstracts minutiae such as TCP connection management and HTTP protocol parsing. We monitor fragments of the Cowboy-Ranch communication protocol used to handle client requests, the particulars of which is found in appendix B.2 together with descriptions of the properties used. All properties selected for these tests are parametric w.r.t. system components (refer to section 1.2) to yield monitors that (i) do not interact and can reach verdicts independently, and (ii) loop continually to exert the maximum runtime overhead possible. Figure 6.6 depicts the two instrumented set-ups described. In figure 6.6a, workers are weaved with the monitor code synthesised from the properties in appendix B.1; figure 6.6b shows the instrumented Cowboy-Ranch protocol handlers with monitors corresponding to properties from appendix B.2. During the course of benchmark runs, monitors communicate their verdicts to a central coordinating process that tracks the expected number of verdicts to determine when a run can be shut down without loss of messages.

6.5.2 Synthetic Benchmarks

Our first set of benchmarks use *mild* loads with $n = 20\text{k}$ and *high* loads $n = 500\text{k}$; $\Pr(\mathit{send}) = \Pr(\mathit{recv})$ is fixed at 0.9 as in section 6.4.4. These configurations generate $\approx n \times w \times$(work requests and responses)=4M and 100M messages respectively to produce 8M and 200M analysable trace events per run. We use a total loading time of $t = 100\,\text{s}$ in our experiments, and perform three experiment repetitions under the Steady, Pulse, and Burst load profiles. Figure 6.5 depicts the number of workers instantiated by the master at each benchmark run for the mentioned loads. The results are summarised in figures 6.7 and 6.8. Every chart in these figures plots the particular performance metric (*e.g.* memory consumption, y-axis) against the number of worker processes (x-axis). Since inlining prevents us from delineating the system and monitor-induced runtime overhead, we follow the standard practice in the literature (*e.g.* [219, 113, 61, 52, 163, 184, 183]) and include *baseline* plots, *i.e.,* the unmonitored system, to compare the relative overhead between our different monitoring set-ups.

Mild loads Figure 6.7 illustrates the plots for the system set with n=20k. These loads are similar to those employed by the state-of-the-art frameworks used to evaluate component-based runtime monitoring, *e.g.* [203, 219, 39, 87, 185], although ours are slightly higher. We remark that none of the benchmarks used in these works consider different load profiles: they either model load on a Poisson process, or fail to specify the kind of load applied. In figure 6.7, the execution duration chart (bottom right) shows that, regardless of the load profile used, the running time of each experiment is comparable to the baseline. Under this mild load, the execution duration alone fails to convey a detailed enough view of runtime overhead, although our benchmarks provide broad coverage in terms of the Steady, Pulse, and Burst load profiles. This trend is mirrored in the scheduler utilisation plot (top left), where both baseline and monitored systems induce a constant load of ≈ 17.5%. On this account, we deem these results to be *inconclusive*. By contrast, our three load profiles induce different overhead for the response time (bottom left), and, to a lesser extent, the memory consumption plots (top right). Specifically, when the system is subjected to a Burst load, it exhibits a surge in the response time for the baseline and monitored system alike at a load of ≈ 16k workers. While this is not reflected in the consumption of memory, the Burst plots do exhibit a larger—albeit linear—rate of increase in memory when compared to their Steady and Pulse counterparts. The latter two plots once again show analogous trends, indicating that both Steady and Pulse loads exact similar memory requirements and exhibit comparable responsiveness under the respectable load of 20k workers. Crucially, the data plots in figure 6.7 *do not* enable us to confidently extrapolate our results. The edge case in the response time chart for Burst plots raises the question of whether the surge in the trend observed at ≈ 16k remains consistent when the number of workers goes beyond 20k. Similarly, although for a different reason, the execution duration plots do not allow us to distinguish between the overhead induced by monitors for different loads at such a (small) scale. This arises due to the perturbations introduced by the underlying OS (*e.g.* scheduling other processes, IO, *etc.*) that affect the sensitive time-keeping of the benchmark metrics.

High loads We increase the load to $n = 500$k workers to determine whether our benchmark set-up can show how the monitored system performs under stress. The response time chart in figure 6.8 indicates that for Burst loads (bottom left), the overhead induced by monitors grows *linearly* in the number of workers. This conflicts with the results in figure 6.7, and supports our claim of section 1.1.3 that the inability of benchmarks to scale makes it hard to extrapolate to general conclusions or identify potential trends. For instance, the evidence in figure 6.7 can easily mislead one to deduce that the RV tool under scrutiny scales poorly under Burst loads of mild and larger sizes. By subjecting the system to high loads,

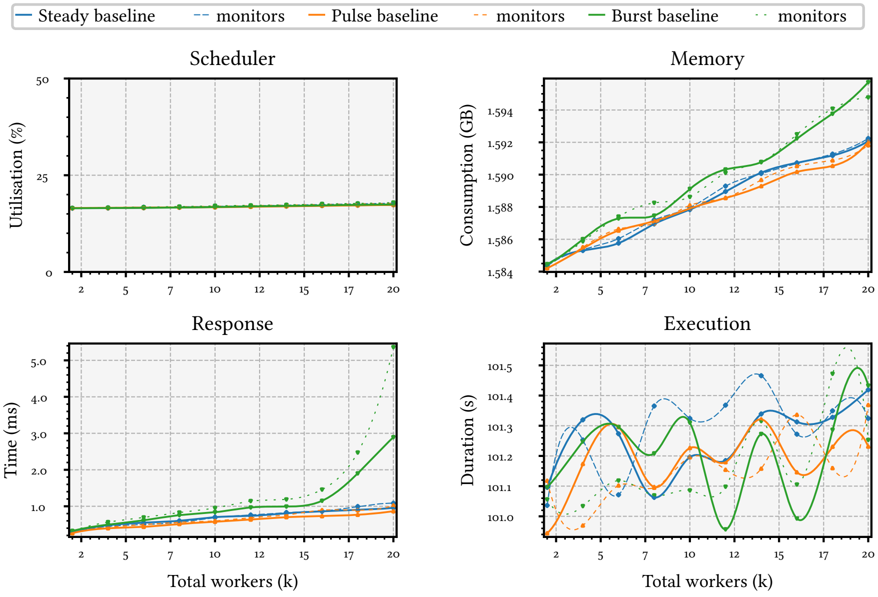


Figure 6.7. *Mean runtime overhead for master and worker processes (20 k workers)*

we also expose the dissimilarity between the response time (bottom left) and memory consumption (top right) gradients for the Steady and Pulse plots that appeared to be comparable under the mild loads of 20k workers. Note that, considering the execution duration chart (bottom right of figure 6.8) as the sole indicator of overhead falsely suggests that the monitored system exhibits virtually identical overhead, regardless of the load profile applied. This erroneous observation is, however, refuted by the memory consumption and response time plots that indicate otherwise, stressing the benefit that multiple metrics offer when interpreting overhead.

We extend the argument for a multi-faceted view of runtime overhead to the scheduler utilisation metric in figure 6.8 that reveals a subtle aspect of our concurrent set-up. Specifically, the charts show that while the response time, memory consumption, and execution duration plots grow in the number of worker processes, scheduler utilisation plateaus at ≈ 22.7%. This is partly caused by the master-worker design that becomes susceptible to bottlenecks when the master is overloaded with requests [202]. In addition, the preemptive scheduling of the EVM [57, 132] obliges the master to share the computational resources of the same machine with the rest of the workers. We conjecture that, in a distributed set-up where the master resides on a *dedicated* node, the overall system throughput may be further pushed.

6.5.3 OTS Application Benchmarks

In this second set of benchmarks, we evaluate the overheads induced by our inline monitoring tool under examination using the Cowboy web server and show that the conclusions we draw are *in line* with those reported earlier for our synthetic benchmark results. The experiment is configured to generate load on Cowboy using the popular load testing tool JMeter [109] that issues HTTP requests. JMeter is hosted on a dedicated node that accesses the local network where the experiment-taking machine of section 6.4.1 running Cowboy resides. To emulate the typical behaviour of web clients (*e.g.* browsers) that fetch resources via multiple HTTP requests, our Cowboy application serves files of various sizes

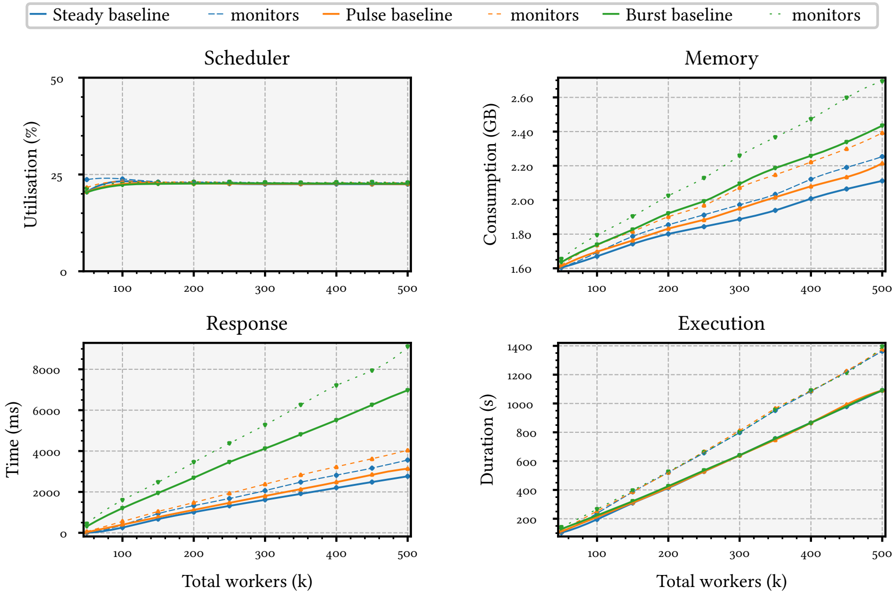


Figure 6.8. *Mean runtime overhead for master and worker processes* (500 k *workers)*

that are randomly accessed by JMeter during the benchmark.

Mild loads Figure 6.9 plots our results for *Steady* loads from figure 6.7, together with the ones obtained from the Cowboy benchmarks; JMeter did not enable us to reproduce the Pulse and Burst load profiles. For the Cowboy benchmarks, we fixed the total number of JMeter request threads to 20k over the span of 100s, where each thread issued 100 HTTP requests. This configuration coincides with parameter settings used in the experiments of figure 6.7. In figure 6.9, the scheduler utilisation, memory consumption, and response time charts (top, bottom left) show conformity between the baseline plots of our synthetic benchmarks and those taken with Cowboy and JMeter. This indicates that, for these metrics, our synthetic system model exhibits *analogous characteristics* to the ones of the OTS system, under the chosen load profile. The argument can be extended to the monitored versions of these systems which follow identical trends. We point out the similarity in the response time gradients of our synthetic and Cowboy benchmarks, even though the latter set of experiments was conducted over a local network. This suggests that, for our single-machine configuration, the synthetic master-worker benchmarks manage to adequately capture local network conditions. The y-axis interval separating the plots of the two experiment set-ups stems from the implementation specifics of Cowboy and our synthetic model. This discrepancy is also attributable to how the runtime metrics are collected, *e.g.* JMeter cannot sample the scheduler utilisation from within the EVM and has to rely on measuring the CPU usage instead. The deviation in the execution duration plots (bottom right) arises for the same reason.

High loads Our efforts to run tests with 500k request threads were stymied by the scalability issues we experienced with Cowboy and JMeter on our experiment set-up of section 6.4.1.

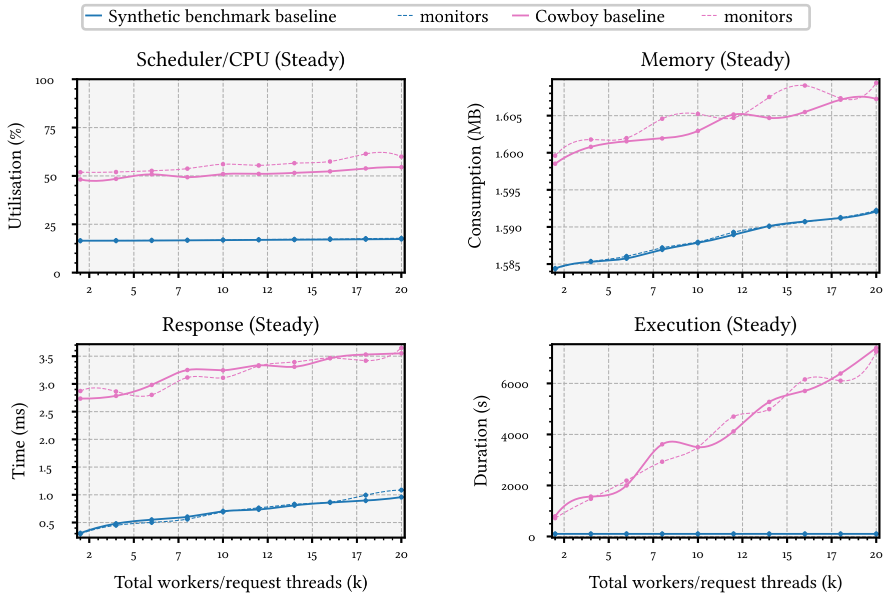


Figure 6.9. *Mean overhead for synthetic and Cowboy benchmarks (20 k threads)*

6.6 Discussion

RV for reactive systems necessitates benchmarking tools that can *scale dynamically* to accommodate considerable load sizes and can provide a *multi-faceted* view of runtime overhead. This chapter presents a benchmarking tool that fulfils these requirements. We demonstrate its implementability in Erlang, arguing that the design is easily instantiatable to other actor frameworks such as Akka and Thespian. Our set-up emulates various system models through configurable parameters and scales to reveal behaviour that emerges only when software is pushed to its limit. The benchmark harness gathers different performance metrics to give a comprehensive perspective on runtime overhead that, to wit, other state-of-the-art tools do not currently offer. Our experiments demonstrate that these metrics benefit the interpretation of empirical measurements: they increase visibility and help uncover insufficiently general, or otherwise, erroneous conclusions. We establish that—despite its synthetic nature—our master-worker model faithfully approximates the response times observed in realistic web server traffic. We also compare the results of our synthetic benchmarks against those obtained using a OTS application use-case to confirm that our tool captures the behaviour of this realistic set-up. It is worth noting that, while the empirical measurements discussed in sections 6.4 and 6.5 depend on our chosen implementation language, the conclusions we draw are transferable to other frameworks, such as Akka and Play [167] that adopt a concurrency model similar to our own.

6.6.1 Related Work

There are other benchmarking tools targeting the JVM besides those mentioned in section 1.1.3. Renaissance [193] employs workloads that leverage the concurrency primitives of the JVM, focussing on the performance of compiler optimisations, similar to DaCapo and ScalaBench. These benchmarks gather metrics that measure software quality and complexity, as opposed to metrics that gauge runtime overhead. Basho Bench [28] is one of the first benchmarking tools available for the Erlang/OTP that was originally implemented to benchmark Riak [29] and has been extended for use with other applications. The tool focusses on capturing throughput and latency metrics. It creates workers to which operations specific to a benchmarking scenario are assigned, *e.g.* issuing HTTP requests. Worker processes can then invoke these operations either by maximising the throughput or at intervals following a Poisson process. Bench also accepts parameters that configure the number of concurrent workers, total benchmark loading time, and randomisation seed, so that tests can be executed in a repeatable fashion. Despite the similarities to our tool in these respects, Bench is similar to other load generation tools like JMeter [109], Tsung [186], and Gatling [74] that assess the performance of APIs (*e.g.* web services, middleware).

By contrast, bencherl [20] assess the scalability of Erlang applications, rather than their performance. This framework combines a suite of synthetic microbenchmarks that measure the Erlang-specific execution behaviour (*e.g.* process spawning, message sending, *etc.*), together with a collection of OTS programs to identify bottlenecks in the EVM. The CRV suite [26] is an initial attempt at standardising the evaluation of RV tools but mainly focusses on RV for monolithic programs written for the JVM. We are unaware of RV-centric benchmarks for reactive systems, such as ours, that are specifically designed to scale dynamically and accommodate high loads that follow realistic patterns.

In Liu et al. [168], the authors propose a queueing model to analyse web server traffic deployed on Apache [161], and develop a distributed benchmarking tool to validate it. Their model coincides with our master-worker set-up and considers loads based on a Poisson process; we also assess other forms of load. A study of message-passing communication on parallel computers is conducted in Grove and Coddington [126]. The authors employ a MPI-based benchmarking tool that measures the probability distributions of communication times between systems loaded with different numbers of processes. This is similar to our approach of sections 6.4.4 and 6.4.5 for synthetic loads. They exclusively focus on MPI, which makes their tool inapplicable to our use case. However, the experiments of section 6.4 that validate our benchmarking tool, and in particular, establish the veracity of the models it generates (*cf.* section 6.4.5), agree with the empirical findings reported by Liu et al. [168] and Grove and Coddington [126].

7 Evaluating Decentralised Outline Runtime Monitoring

Chapter 1 claims that a decentralised approach to monitoring reactive component systems overcomes the challenges that render its centralised counterpart inadequate. It argues that the runtime monitoring technique itself must be *reactive*, lest it undermines the reactiveness of the SuS. This chapter evaluates the Erlang implementation of our decentralised algorithm given in chapter 5 via a systematic empirical study, demonstrating that it exhibits the characteristics of a reactive system. In particular, it

- effects timely detections with feasible impact on the SuS (*responsive*, sections 7.2.1, 7.2.2 and 7.2.4),
- maximises resource usage but does not crash (*resilient*, sections 7.2.2 to 7.2.4),
- grows and shrinks to accommodate dynamic changes in load (*elastic*, sections 7.2.2 and 7.2.5), and
- reconfigures monitors in reaction to SuS trace events (*message-driven*, sections 7.2.2 and 7.2.5).

We evaluate decentralised and centralised outline monitoring alongside inlining (refer to section 4.5) since it is widely adopted and generally regarded as the most efficient online monitoring technique [91, 90, 25]. This gives us a sound basis against which our results can be compared and generalised. As a by-product of this evaluation, we derive other observations that challenge certain commonly-accepted notions that are not satisfactorily explored in the RV literature cited in section 7.4 (*e.g.* we show that a considerable portion of the runtime monitoring overhead stems from the instrumentation, and that outline monitoring induces overhead comparable to inline monitoring in certain cases).

7.1 Reactive System Monitoring

Our goal is to study decentralised and centralised monitoring under induced edge-case (*e.g.* limited memory) and general-case (*e.g.* typical number of processing elements) scenarios. We judge whether these monitoring approaches scale and optimise the use of available computational resources to determine whether they exhibit reactive behaviour. For this reason, our experiments use two different set-ups:

SU_E *edge-case* scenarios, which reuse the set-up of section 6.4.1 to capture systems with constrained hardware resources, and

SU_G *general-case* scenarios, which use an Intel Core i9 9880H 64-bit machine with 16GB of memory, running macOS 12.3.1 and Erlang/OTP 25.0.3, replicating platforms with modern commodity hardware.

The differences in hardware, OS, and Erlang/OTP versions increase our confidence that the conclusions drawn from this chapter are portable to other settings. To broaden the scope of this investigation and generalise our results, we also consider two archetypal models of reactive systems that:

Set-up	System	Schedulers	Workers n	Work requests w	≈ Messages	≈ Messages/s
SU_E	RS_H	4	100 k	100	20 M	196 k
	RS_L		1 k	10 k	20 M	201 k
SU_G	RS_H	16	500 k	100	100 M	345 k
	RS_L		5 k	10 k	100 M	637 k

Table 7.1. *Experiment configurations and message throughput at maximum Steady loads*

RS_H exhibit *high* degrees of concurrency and perform short-lived tasks. Web server applications instantiate this model, where the server receives numerous HTTP requests from clients and fulfils them by fetching resources or executing commands (*e.g.* Nginx [79]), or

RS_L deal with *lower* concurrency levels and engage in long-running, computationally-intensive tasks. Big data stream processing frameworks are one example (*e.g.* Apache Spark [228]).

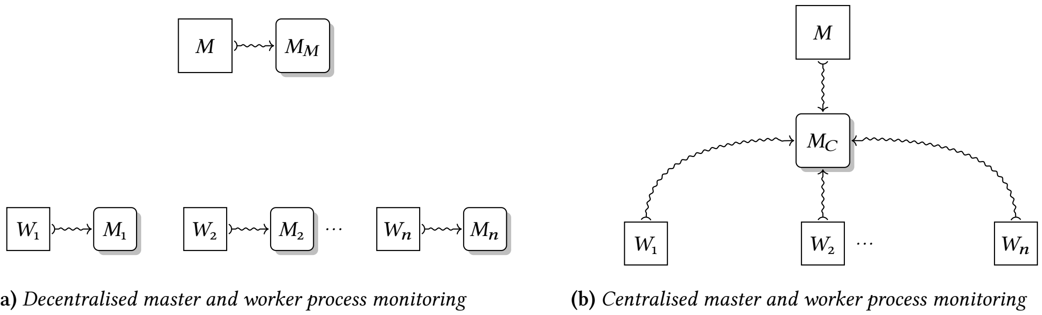
We model these scenarios on set-ups SU_E and SU_G using the benchmarking tool of chapter 6 to show that our decentralised monitoring approach can be feasibly applied to *all* cases.

7.1.1 Experiment Set-Up

Our EVMs on set-ups SU_E and SU_G are configured to use 4 and 16 scheduler threads respectively. The setting for each platform is selected to coincide with the number of logical processors available on the SMP machine [19]. The loads we use to generate our benchmarking models reflect the hardware capacity that SU_E and SU_G afford. For the experiments in sections 7.2.1 to 7.2.3, set-up SU_E is configured for *moderate* loads with $n = 100\text{k}$ workers and $w = 100$ work requests per worker. This model generates $\approx n \times w \times$ (work requests and responses) = 20M message exchanges between the master and worker processes, totalling 20M × (send and receive trace events) = 40M analysable trace events. Set-up SU_G adopts the same *high* load settings of section 6.4.1, *i.e.,* $n = 500\text{k}$ workers, each with $w = 100$ work requests to produce 100M messages and 200M trace events. These load configurations embody the first model of reactive systems, RS_H, with high concurrency, and are used in sections 7.2.4 and 7.2.5.

Section 7.3 uses loads that model the second reactive system, RS_L. The benchmarks on set-up SU_E are configured with $n = 1\text{k}$ and $w = 10\text{k}$ work requests per worker, and SU_G sets $n = 5\text{k}$ and $w = 10\text{k}$. These parameter values roughly yield the same number of trace events as their respective counterparts with moderate (*i.e.,* $n = 100\text{k}$, $w = 100$) and high (*i.e.,* $n = 500\text{k}$, $w = 100$) loads on system RS_H.

In all our experiments, a total loading time of $t = 100\text{s}$ is set. The parameters $\Pr(send)$ and $\Pr(recv)$that control the speed at which the system reacts to load, use the values $\Pr(send) = \Pr(recv) = 0.9$. These generate benchmark models that consume reasonably low memory and emulate realistic response times (refer to section 6.1.5). We subject each benchmark to the three load profiles—Steady, Pulse, and Burst—offered by our benchmarking tool of chapter 6. Each experiment is performed *three* times, based on our CV values calculated according to section 6.4.3. Table 7.1 summarises these experiment configurations and includes the message throughput under maximum *Steady* loads (*i.e.,* 100 k, 500 k, *etc.*) for reference.



(a) *Decentralised master and worker process monitoring* **(b)** *Centralised master and worker process monitoring*

Figure 7.1. *Master-worker benchmarks instrumented with decentralised and centralised outline monitors (internal)*

7.1.2 Runtime Monitoring Set-up

By contrast to the set-up of section 6.5.1, the experiments in this chapter monitor *both* the master and worker processes. Figure 7.1 illustrates the arrangement of decentralised and centralised outline monitors for the case where events are analysed *internally* by tracers (*cf.* figure 5.1b). The system with inline monitors is organised similarly to the one in figure 6.6a. It is worth mentioning that the centralised set-up (figure 7.1b) is obtained by instrumenting the master process only. By virtue of automatic tracer inheritance (assumption A_5), every worker that the master creates gets traced by the monitor at the master, giving rise to the set-up of figure 7.1b. See concluding discussion of section 5.1 on page 50.

7.1.3 Precautions

Our benchmarking tool of chapter 6 focusses on collecting the memory consumption and scheduler utilisation metrics globally to minimise impacting the behaviour of the master-worker models it generates [127]. This measurement-taking strategy prevents us from isolating the operating expense of the monitors from that of the SuS. We, therefore, follow the same approach of section 6.5 and insert the *baseline* system plots for reference in the charts that follow.

Online monitors may introduce runtime overhead biases owed to various specific factors, such as the non-determinism a monitor admits, its size in terms of the number of states, monitor optimisations, persisting trace events, *etc.* As an example, table 7.2 lists the mean time in microseconds (µs) that monitors spend processing events for traces of different lengths. The values in the topmost entry record the time it takes to write an event to file (*e.g.*, for offline monitoring), while the remaining tabulate the average time spent by the monitors synthesised from the properties of appendices B.1 and B.3 to

Event operation	Number of events in trace			
	1 k	10 k	100 k	1 M
Write to file	30.76	33.18	29.59	27.84
Analysis using monitors from formulae φ_{13} to φ_{16}	302.55	304.44	308.99	306.71
Analysis using monitors from formulae φ_{RP} to φ_{CP}	693.46	667.97	715.95	654.96

Table 7.2. *Mean time (µs) taken by monitors to persist or analyse one trace event*

analyse each event. To *objectively* compare the overhead induced in different monitoring set-ups, our benchmarks *simulate* this runtime analysis cost via a configurable delay. We set this analysis cost to a very conservative ≈ 5µs per event to manufacture a *best-case* scenario under which decentralised and, in particular, centralised monitoring can be evaluated. Runtime checking local properties (*i.e.,* ones specified w.r.t. system components) against a global trace can be done efficiently via an approach called parametric trace slicing (PTS) [62, 196], mentioned in section 4.7.1. Recall that PTS partitions the global trace into multiple sub-traces, where each corresponds to the behaviour observed locally at different components. Every sub-trace is then analysed independently of the others by a dedicated local monitor that reaches its verdict based on the events reported thus far. Our centralised monitor implements PTS by demultiplexing the global stream of trace events to different local monitors. It maintains a monitor map that is indexed by the PID of system components to quickly access the associated monitors and analyse events. The central monitor ensures that every local monitor is created when needed and removed when its analysis is completed. This ensures the lowest possible overhead and does not bias our results in favour of decentralised monitoring.

7.2 Monitoring High Concurrency Systems

This section gives a comprehensive view of runtime monitoring that highlights,

(i) the effect overhead has on the SuS as it executes, and
(ii) the average resources monitors consume until their analysis runs to completion

Aspect (i) elucidates how the memory consumption and scheduler utilisation influence the response time that a client might experience in practice (sections 7.2.1 to 7.2.4). Conversely, aspect (ii) reveals whether the monitoring set-up optimally maximises the memory and scheduler capacity provided by the hosting

Experiment	Set-up	Claim and expected outcome
(i) Effect that overhead has on the SuS as it executes		
Instrumentation Overhead	$\mathrm{SU_E}$	Instrumentation induces non-negligible overhead We expect the centralised set-up to induce the highest overhead
Monitoring Overhead	$\mathrm{SU_E}$	Instrumentation *and* runtime analysis add further overhead We expect the centralised set-up to induce the highest overhead
Instrumentation Cost	$\mathrm{SU_E}$	Much of the monitoring overhead arises from instrumentation We expect the overhead gap between the instrumented and monitored set-ups for decentralised monitors to be relatively small
Scaled Set-up	$\mathrm{SU_G}$	Decentralised monitoring leverages the added resource capacity We expect the centralised set-up *not* to scale
(ii) Average resources monitors consume until analysis runs to completion		
Resource Usage	$\mathrm{SU_G}$	Decentralised monitoring is elastic following the load model We expect the centralised set-up to be unaffected by load model

Table 7.3. *Experiments for high concurrency systems ($\mathrm{RS_H}$) investigating overhead, claims, and expected outcomes*

platform and whether monitors can effect timely verdict detections (section 7.2.5). The experiments in this section use set-up SU_E to capture edge-case scenarios with limited resources, and set-up SU_G, capturing general-case scenarios with modern hardware. Both set-ups focus on RS_H, which models high-concurrency systems that execute short-lived tasks.

Our general aims for aspects (i) and (ii) are broken down in table 7.3. It lists claims that we make about experiments, together with the outcomes expected as a result of our interpretation of the corresponding empirical evaluation. Each section named in table 7.3 details the methodology followed in each evaluation and is accompanied by a discussion of the graphed results. We adopt this nomenclature in what follows. The term *instrumentation* is used to mean the 'isolated instrumentation', *i.e.,* without the analysis of runtime monitors, and *monitoring* to mean the 'instrumentation *and* the runtime analysis of monitors'. *Decentralised monitoring* refers to both the inline and outline forms of monitoring.

7.2.1 Instrumentation Overhead

Our first set of experiments isolates the overhead induced on the SuS due to instrumentation, *i.e.,* the cost of tracing system components *and* reporting events to the intended monitors. They show that the

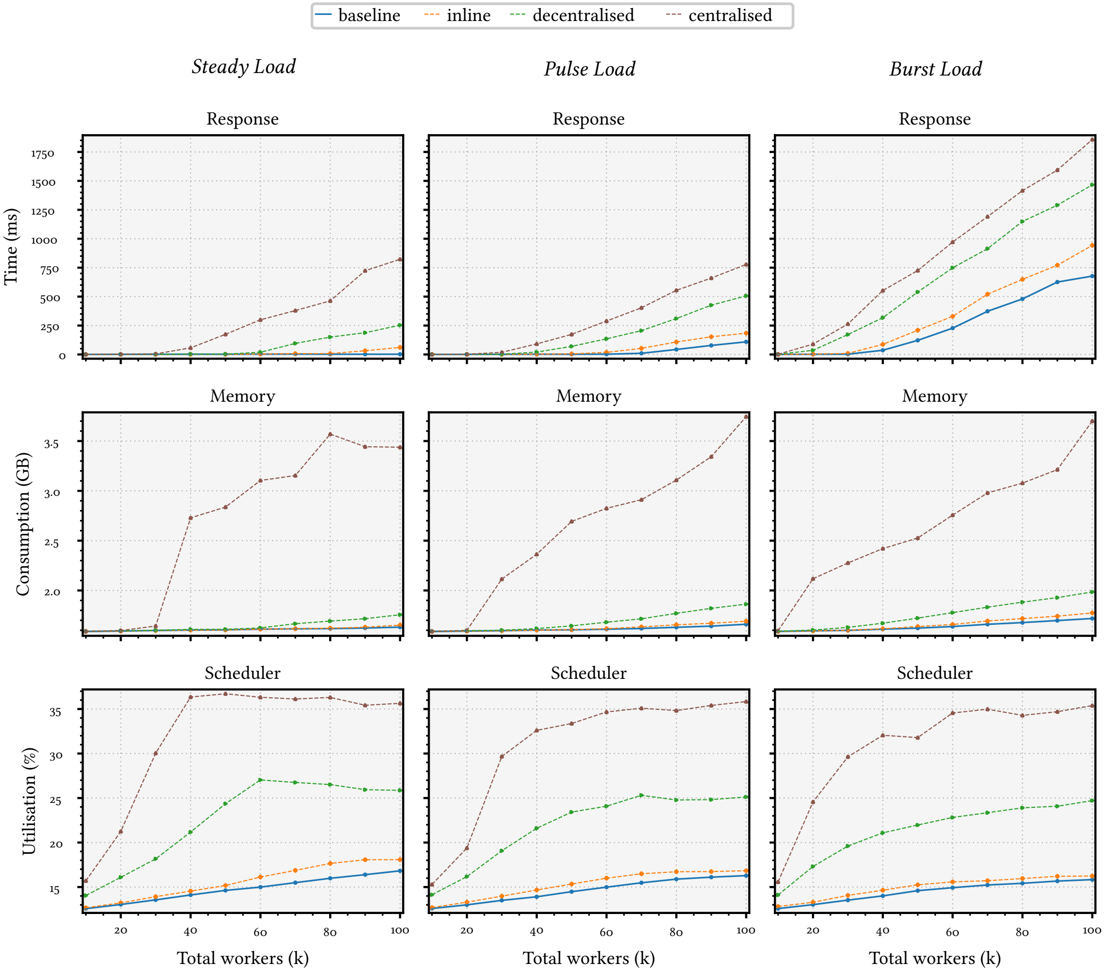


Figure 7.2. *Instrumentation overhead on system under moderate load benchmarks (100k workers)*

instrumentation induces non-negligible overhead, despite the fact that no runtime analysis is conducted by monitors (table 7.3, claim 1). The benchmarks are executed on set-up $\mathrm{SU_E}$, where the master-worker models are run with moderate loads ($n = 100$k, $w = 100$, and 4 scheduler threads). Figure 7.2 shows the results obtained from these benchmarks for the decentralised inline (*inline*), decentralised outline (*decentralised*), and centralised outline (*centralised*) forms of instrumentation.

For the three load profiles, Steady, Pulse, and Burst, figure 7.2 indicates that (i) all types of instrumentation induce overhead that is by no means insignificant, and (ii) that centralised instrumentation carries the larger penalty. Centralised instrumentation occupies more memory due to the backlog that gradually accumulates in the mailbox of the tracer process (*i.e.,* the message buffer κ described in section 5.1.1 on page 50). This build-up is a manifestation of two aspects. Worker processes concurrently deposit trace events into the mailbox of the central tracer. At the same time, the tracer does not manage to consume the events in its mailbox at the same rate at which these are being produced by workers as a result of its *sequential* nature. Evidence of this bottleneck can be gleaned from the scheduler plots which demonstrate high utilisation levels that settle at $\approx 36\%$ for the benchmarks with ≈ 40k workers under Steady load, and ≈ 60k workers under Pulse and Burst load. Considering the scheduler utilisation charts in isolation may suggest that, rather than a bottleneck, centralised instrumentation has the potential to scale since it displays low usage. Its steadily growing memory consumption plots in figure 7.2, however, contradict this hypothesis.

By contrast, our decentralised approach uses considerably fewer resources and yields lower response times throughout the three load profiles of figure 7.2. Readers may notice that the decentralised instrumentation scheduler utilisation plots also plateau slightly in the Steady (≈ 60k workers) and Pulse (≈ 70k workers) load charts. This behaviour is induced by the bottleneck intrinsic to the master-worker paradigm [202] that *throttles* the production of trace events, rather than by the inability of our decentralised approach to scale. One easily supports this assertion by looking at corresponding memory consumption plots that exhibit a gentle rise in the number of worker processes.

7.2.2 Monitoring Overhead

The second set of experiments extends the results of section 7.2.1 by combining the overhead incurred by the analysis performed by the monitors *and* instrumentation, *i.e.,* the full cost of runtime monitoring. We demonstrate that the added cost of runtime analysis induces further growth in the overhead and that centralised monitoring performs poorly as a result (table 7.3, claim 2). Our benchmarks are executed on configuration $\mathrm{SU_E}$ and introduce the ≈ 5µs delay described in section 7.1.3 to stabilise the analysis overhead. Figure 7.3 illustrates the overhead incurred by the monitored master-worker system under the Steady, Pulse, and Burst load models. In addition to the baseline and inline benchmarks, our charts plot the overhead for two variants of decentralised and centralised monitoring (see figure 5.1) that internalise the event analysis within tracers (*internal*), or delegate it to dedicated monitor processes (*external*). These are included to examine whether the benefit of process isolation obtained by separating the tracer and monitor logic justifies the extra overhead induced due to additional concurrency.

Figure 7.3 shows that centralised monitoring exhibits analogous memory consumption and scheduler utilisation patterns to the instrumentation overhead charts of figure 7.2. It reveals that simulating a *best-case* analysis slowdown of ≈ 5µs per event aggravates the overhead to the point of crashing (this is marked by × in figure 7.3). This behaviour is consistent across Steady, Pulse, and Burst loads for both the

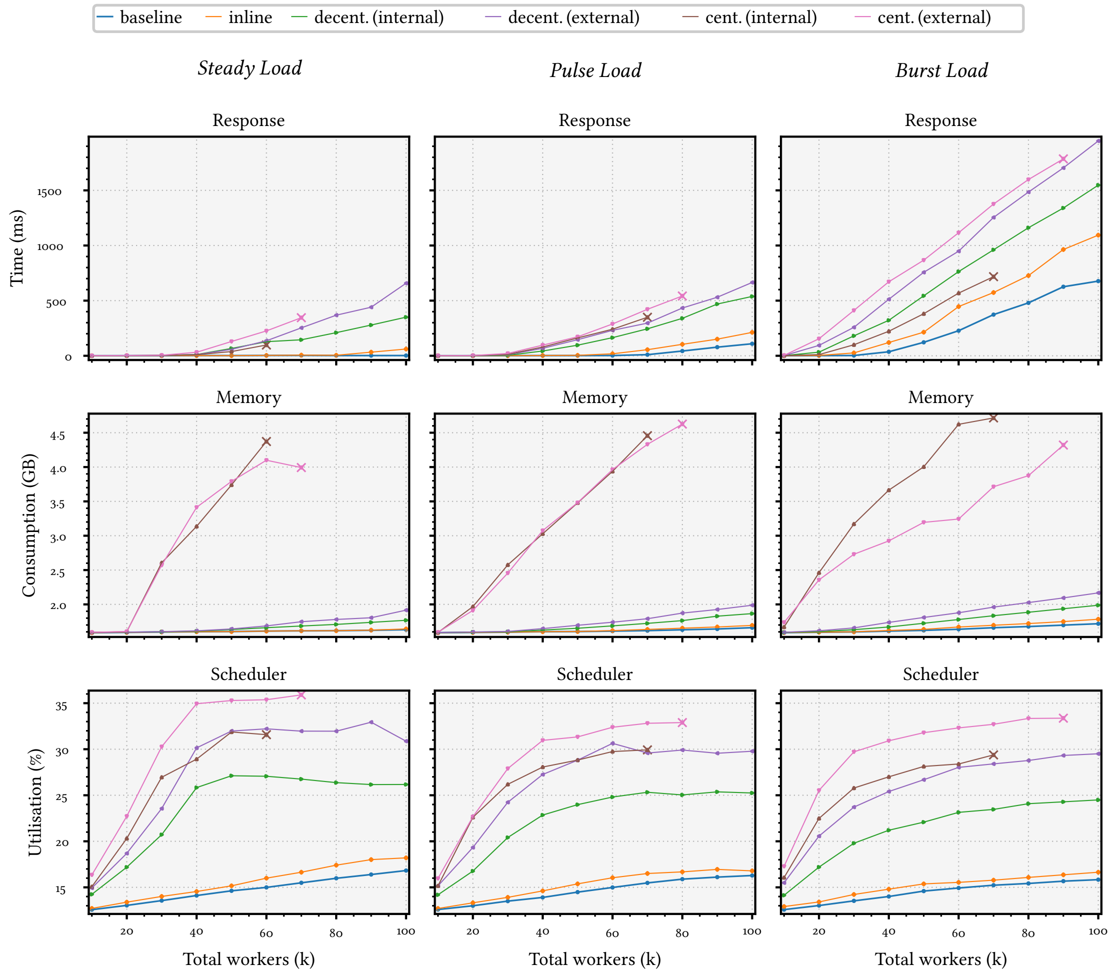


Figure 7.3. *Monitoring overhead on system under moderate load benchmarks* (100k *workers*)

internal and external forms of centralised monitoring. By analysing the crash dumps produced by these benchmarks, we were able to attribute these abrupt terminations to memory exhaustion. The dumps also confirm that the significant amount of memory consumed is due to the central monitor process, which appears to result from the accumulated backlog of trace messages that ultimately leads the EVM to fail. This suggests that centralised monitoring is neither scalable nor resilient.

Decentralised inline and outline monitoring is not afflicted by the analysis slowdown, but rather scales to accommodate this cost. This may be confirmed by cross-referencing the low memory consumption and scheduler utilisation plots of figures 7.2 and 7.3 (refer also to summary in figure C.1). Dissecting these metrics uncovers two important subtleties of decentralisation. First, outline monitors process events quickly (attested by the absence of excessive memory growth) and spend much of their time idle, waiting for trace events (lower scheduler utilisation than centralised monitoring), *i.e.,* they are passive and *message-driven*. Second, the effectiveness of inline monitors should not be judged solely by the low memory and scheduler costs. Inlining entwines the SuS and monitors, and slowdowns in the analysis risk impacting the overall system responsiveness [25, 68].

Figure 7.3 (top) shows that both forms of decentralised monitoring induce latency, yet for crucially different reasons. Our algorithm presented in chapter 5 enables us to deduce that the latency in the case of outline monitoring stems indirectly from the dynamic reconfiguration monitors perform to manage the choreography. In contrast, the effects of inlining are due to the dependency it has on the analysis slowdown. This reasoning follows from the fact that the SuS and monitors execute in lock-step according to the synchronous instrumentation definition of figure 3.2 and our corresponding implementation of section 4.5. We note that other works (*e.g.* [61, 51]) report similar observations. Section 7.3 elaborates further on the slowdowns induced by inlining and shows that increasing the event analysis throughput can deteriorate the response time further.

The latency introduced by decentralised monitoring is decidedly lower than its centralised equivalent (figure 7.3), making decentralisation the better option due to the scalability and resiliency it offers. Figure 7.3 also indicates that our outline approach induces *feasible* response time overhead when judged against inline monitoring. Moreover, in cases that do not warrant strict timely detections, outlining is preferable to inlining as it does not increase the sequentiality (called 'sequentialness' in Armstrong [19]) of the SuS, leaving it more amenable to parallelisation.

Effects of less sequentiality are visible in the plots of figure 7.3—despite the limited parallelism offered by our current configuration, set-up $\mathrm{SU_E}$, with four scheduler threads. Here, the variants of decentralised and centralised outline monitoring that tease apart the instrumentation and trace event analysis (see figure 5.1a) put the scheduler to more use, as opposed to the internalised versions. The decentralised form of externalised monitoring consumes more memory due to the extra monitor processes it creates to delegate the analysis task. By contrast, both variants of the centralised approach consume comparable (Steady and Pulse load) or slightly less (Burst load) amounts of memory since the backlog of trace events occurs *only* on the instrumentation side. This asynchronously forwards events to its corresponding singleton monitor process and helps to relieve some of the pressure build-up on the tracer process. As a result, these two processes handle trace events *concurrently* and seems to be the reason why the externalised analysis variant of centralised monitoring consistently crashes at higher loads in figure 7.3. Our deduction is supported by the crash dumps resulting from these benchmarks.

7.2.3 Instrumentation Cost

Figure 7.4 compares the instrumentation and monitoring overhead of figure 7.2 and figure 7.3 for the two load profile extremities, Steady and Pulse. Readers are pointed to figure C.1 for the plots that include Pulse load. We show that in our experiments, much of the runtime overhead is induced by the instrumentation, rather than by the analysis that monitors conduct (table 7.3, claim 3). In figure 7.4, the centralised approach demonstrates a considerable disparity between the instrumentation (*i.e.,* without runtime analysis) and monitoring (*i.e.,* instrumentation and runtime analysis) overhead for both memory consumption and scheduler utilisation as the load in the number of worker processes increases. This trend is consistent across all load profiles. Evidence of the centralised monitoring bottlenecks are clear in the memory and scheduler values (memory increases but the scheduler plateaus). These values start to grow beyond ≈ 30k and ≈ 20k workers for the Steady and Burst loads respectively. The resulting overhead increase leads our experiments to crash (denoted by a missing bar plot in figure 7.4) at the ≈ 70k workers mark under Steady load and at ≈ 80k under Burst load. Both plots in the figure also demonstrate a degradation in the response time for centralised instrumentation as the load in the number of workers increases, which seems to be a byproduct of the consistently-high demands on the scheduler.

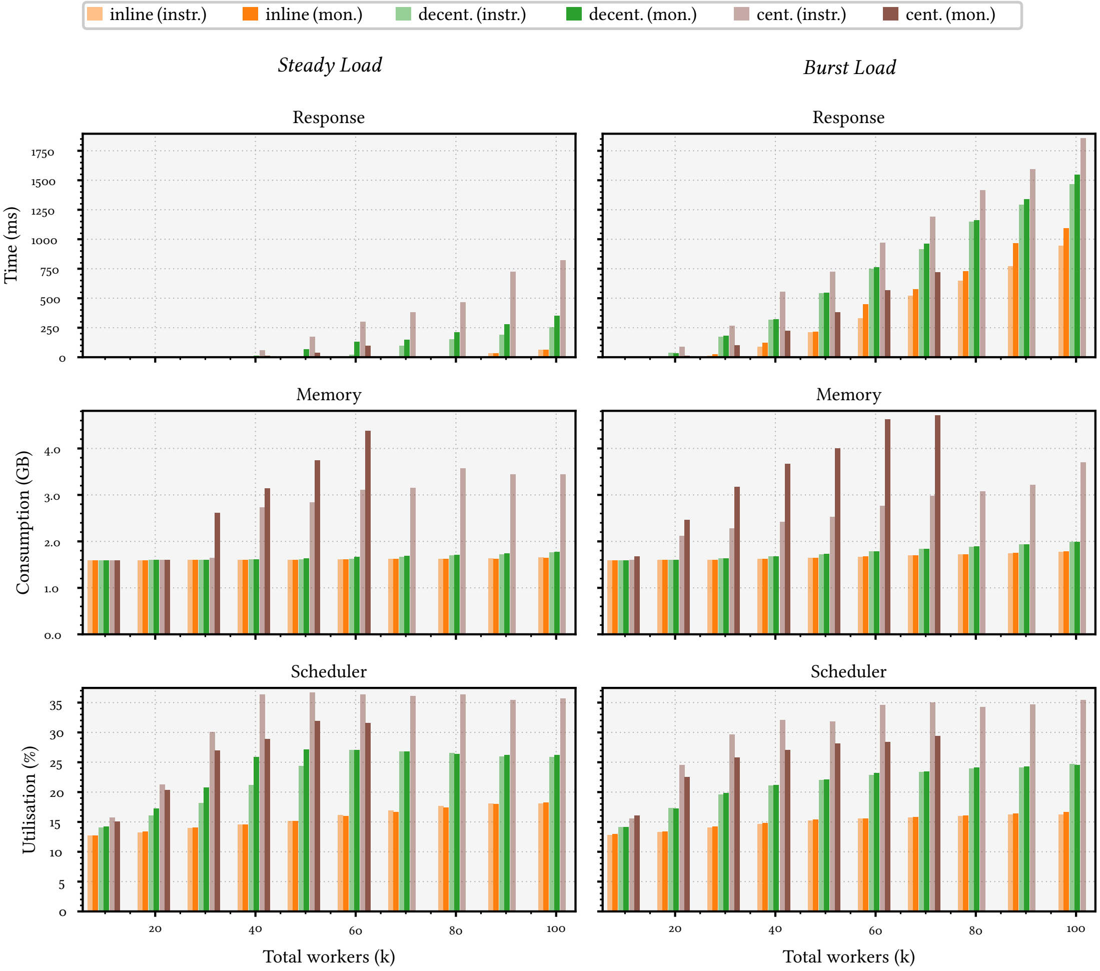


Figure 7.4. *Gap in instrumentation and monitoring overhead on the system under moderate load benchmarks* (100k *workers)*

Decentralised inline and outline instrumentation exhibit comparable overhead measurements to the ones taken with monitors. However, the respective bar plots for inline instrumentation and inline monitoring show a growing pairwise gap in the response time values under Burst load that starts developing at ≈ 80k workers (figure 7.4, top right). Such divergence in the response time readings is arguably smaller in decentralised outline instrumentation and decentralised outline monitoring. Based on this observation and the fact that outline instrumentation decouples the SuS from its monitors, we conjecture that outlining is robust and *absorbs* the additional analysis slowdown. This would enable it to accommodate intricate monitors that runtime check richer correctness properties.

7.2.4 Scaled Set-up

Our benchmarks conducted on SU_E study how decentralised and centralised monitoring behave in edge-case situations where the memory is constrained and the possibility of parallelism is limited. Under these conditions, our findings show that the centralised approach is neither scalable (it utilises the scheduler reasonably, but at the same time, keeps considerable amounts of memory occupied), nor resilient (it exhausts the memory until eventually crashing due to its single point of failure). Decentralised monitoring is not subject to these shortcomings. We transition to the second set-up, $\mathrm{SU_G}$, and scale our experiments to confirm that the aforestated observations are transferable to more *general* cases. In particular, we show that decentralisation yields scalable runtime monitoring that (i) capitalises on the additional memory and processing capacity, and (ii) copes well with high load sizes (table 7.3, claim 4).

Figure 7.5 shows our benchmark results set with $n = 500\text{k}$ workers, $w = 100$ work requests per worker, and a simulated analysis slowdown of $\approx 5\mu\text{s}$ per trace event. The number of scheduler threads on the EVM is increased from 4 to 16. Interested readers can consult figure C.3 which charts the instrumentation and monitoring overhead. Our memory consumption and scheduler utilisation plots of figure 7.5 magnify the bottleneck that adversely affected centralised monitoring in figure 7.3. In the latter benchmarks with 100 k workers, centralised monitoring exhibits higher scheduler utilisation levels (*e.g.* 31.87 % for the internalised analysis variant at 50 k workers under Steady load), by comparison to the plots in figure 7.5 (*e.g.* 4.67 % at an equivalent number of workers and under the same Steady load). The drop in scheduler utilisation stems from two reasons. First, the centralised monitor is limited in its use of computational resources due to its sequentiality (see section 7.2.2). Second, the mean utilisation value is calculated over

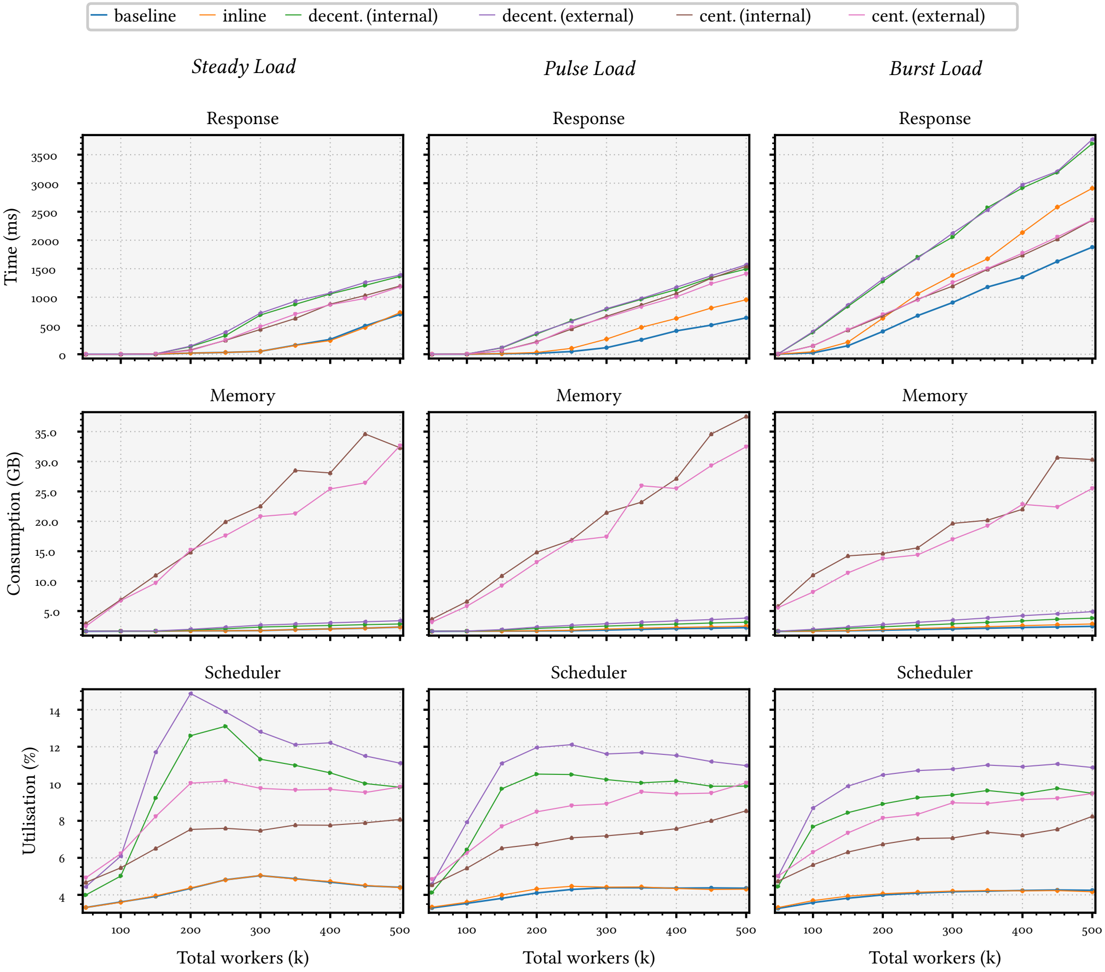


Figure 7.5. *Monitoring overhead on system under high load benchmarks (500k workers)*

16 scheduler threads. On set-up SU_E, this value grows because the EVM schedules processes on a limited number of threads, which concentrates their use; in contrast, processes are spread across more schedulers on set-up SU_G. While the larger number of schedulers on the latter set-up *does* improve the parallelism in our experiments, this processing capability is not exploited to its fullest due to the throttling of tasks the master-worker model is susceptible to. Comparing the scheduler utilisation *baseline* in figures 7.3 and 7.5 corroborates this hypothesis. Nevertheless, the added parallelism gained through the extra schedulers on SU_G instigates the workers to collectively generate *more* trace events than in the previous set-up with 100k workers (*e.g.* the throughput with 100k workers is ≈ 196k messages/s, *vs.* ≈ 345k messages/s in the experiments with 500k workers, table 7.1). The higher message throughput exacerbates the load on the central monitor that is unable to exploit the parallelism offered by set-up SU_G to analyse events. We emphasise that the absence of crashes in these experiments is attributable to the considerable amount of memory set-up SU_G provides, rather than to the ability of the central monitor to manage load. Figure 7.5 demonstrates that the sustained increase in memory consumption by centralised monitors will eventually lead to failure, once the available resources are exhausted.

Decentralised outline monitors benefit from the hardware capacity of set-up SU_G, which manifests as conservative memory consumption and increased scheduler utilisation, supporting our observations in section 7.2.2. The growth in scheduler utilisation follows as a result of the monitor reconfiguration and the routing of trace events effected by our algorithm of chapter 5. As is the case in figure 7.3, the external variant of decentralised outline monitoring (that uses dedicated processes to analyse events) induces slightly higher memory overhead than its internal analogue as a result of the extra processes it creates. Figure 7.5 shows that centralised outline monitoring is also outperformed by inlining, which carries the lowest cost out of the three monitoring approaches considered.

The plots of figure 7.5 exhibit a positive correlation between the scheduler utilisation and the latency induced by decentralised and centralised outline monitoring (*i.e.,* the more the scheduler utilisation increases, the higher the latency). This relationship, equally visible in figure 7.3, is a consequence of our master-worker benchmarks that focus on CPU-intensive tasks (refer to section 6.1.5 on page 69). We assert that the response time of our benchmarks in figure 7.5 degrades since decentralised outline monitors compete for the same pool of scheduler threads in use by worker processes. As a result, workers reside in the run queue [132] for longer periods, which impacts their ability to respond to the master promptly. The singleton monitor employed in the centralised approach adds minimal demands on the EVM schedulers and uses its allotted time slice to keep up with its backlog of trace events. In fact, figure 7.5 shows that organising the instrumentation and runtime analysis into *separate* processes improves the scheduler utilisation of centralised monitoring: this materialises as the small decrease in the memory consumption (middle) and an imperceptible drop in latency (top) across the three load profiles.

Decentralised outline monitoring affects the response time of the SuS, but this comes at the cost of replicating monitors to achieve resilient set-ups that address the SPOF and scalability limitations which make centralised monitoring inept. Besides, decentralised outline monitoring circumvents the issues where inlining cannot be applied (see discussion in section 2.1.4). Figure 7.5 demonstrates that our decentralised approach to monitoring leverages the added hardware capacity and copes with high loads (memory consumption is very gradual). It also induces feasible latency that is adequate in many practical applications such as soft real-time or on-line systems [57], where the response time requirement is often

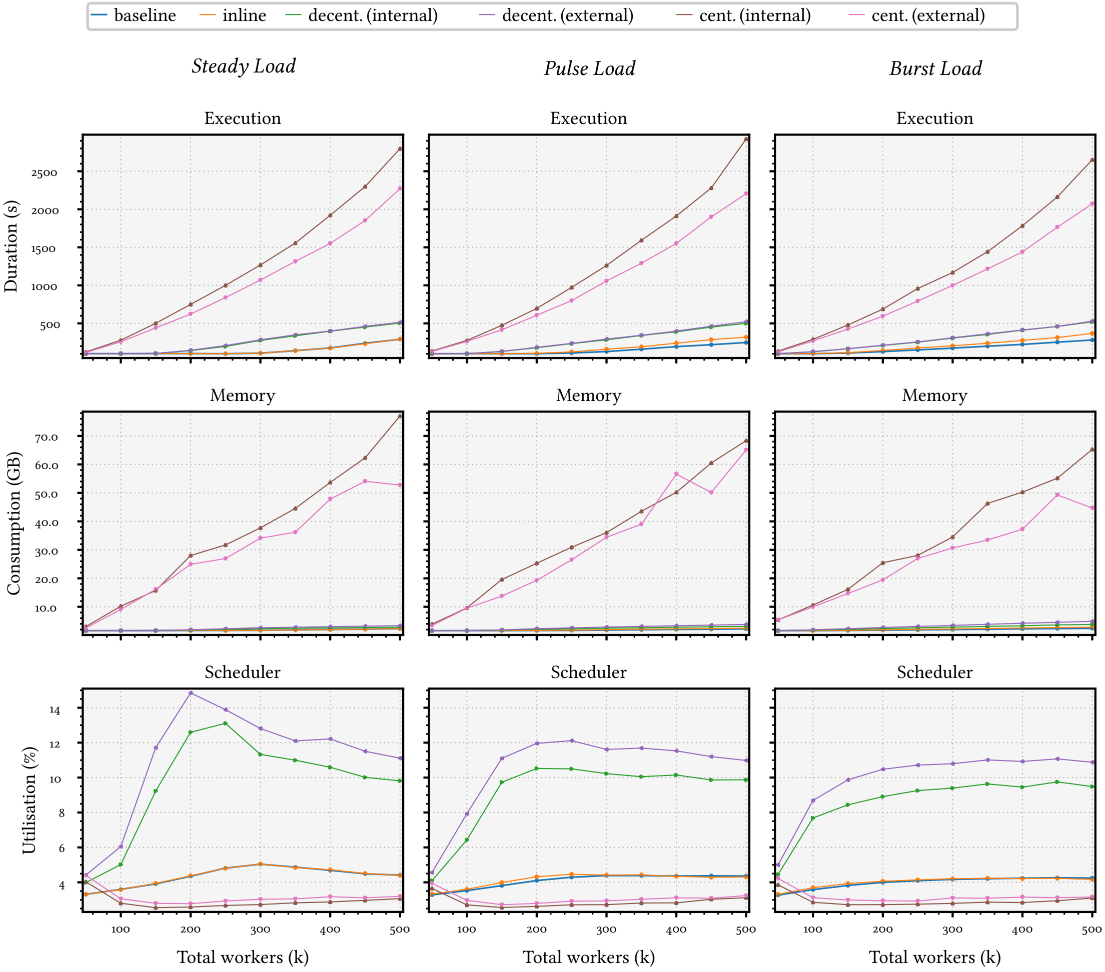


Figure 7.6. *Monitoring overhead for complete experiment runs under high load benchmarks* (500k *workers)*

in the order of seconds [149].

7.2.5 Resource Usage

Sections 7.2.1 to 7.2.4 demonstrate the effects of monitoring overhead on the SuS. Through the mean response time, figures 7.3 and 7.5 capture the *overall* system responsiveness from the point of view of interacting clients, such as end-users or other applications. The memory consumption and scheduler utilisation plots presented in these figures are confined to the time period in which the system runs, thereby giving a truthful depiction of these metrics. This section reinterprets the same metrics collected for the experiments of sections 7.2.2 and 7.2.4. It presents an alternative view that assesses monitoring overhead in its *entirety*—from the time the SuS starts executing until monitors complete their analysis—to investigate whether each monitoring technique puts to optimal use the resources offered by its hosting platform. Through this view, we show that decentralised inline and outline monitoring dynamically adapt to the load applied, *i.e.*, they are elastic, and that centralised monitoring exhibits no such quality (table 7.3, claim 5). The system response time is not relevant to this discussion (it is an attribute of the SuS, not of the monitors), and we replace it by the execution duration metric that records the time taken

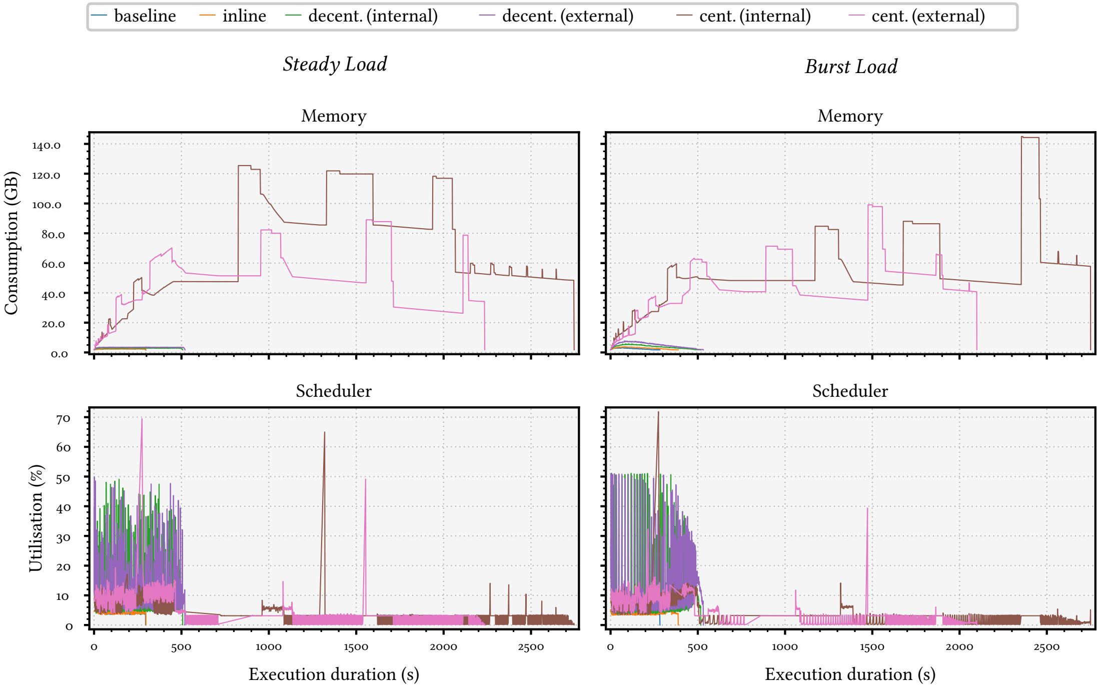


Figure 7.7. *Resource usage for (de)centralised monitoring under high load benchmarks (*500k *workers)*

by experiments to execute to completion. We only consider the results taken on set-up $\mathrm{SU_G}$ with 500k workers processes, since the experiments on $\mathrm{SU_E}$ for centralised monitoring discussed earlier crashed (see section 7.2.2).

The mean metrics calculated over complete experiment runs, depicted in figure 7.6, reaffirm the memory consumption trend for centralised monitoring observed in figure 7.5. One striking difference between these two figures is in the scheduler utilisation, where the plots for the two variants of centralised monitoring (*i.e.,* internal and external) in figure 7.6 dip below the baseline system. This effect results from *skewness* in the mean due to the asymmetry in the distribution of the scheduler utilisation samples collected by our benchmarking tool (refer to section 6.3). Figure 7.7 plots the sampled memory consumption and scheduler utilisation (averaged over the 16 schedulers, y-axis) against the execution duration (x-axis) to capture the resource usage during the course of a *single* experiment run. It underscores the aforementioned lopsidedness in the sampled scheduler utilisation values. This arises because the samples register higher values when the master-worker system and centralised monitor execute concurrently, and lower values once the system terminates but the centralised monitor lingers, processing its backlog of events. The protracted processing of trace events—reflected in figure 7.7 by the 'tail' in the scheduler utilisation plots—also suggests that centralised monitors are susceptible to flagging *late* monitoring verdicts, making them unsuited for cases when timely detections are required. For instance, our benchmark runs for 500 k with centralised monitors (internal) respectively take $\approx 862\%$ and $\approx 843\%$ longer to finish executing than the baseline system under Steady and Burst loads.

Figure 7.6 shows that decentralised outline and inline monitoring take considerably less time to complete their runtime analysis. As an example, our same set-up with decentralised outline monitors (internal) prolongs the execution of experiments by $\approx 73\%$ and $\approx 85\%$ w.r.t. the baseline system under

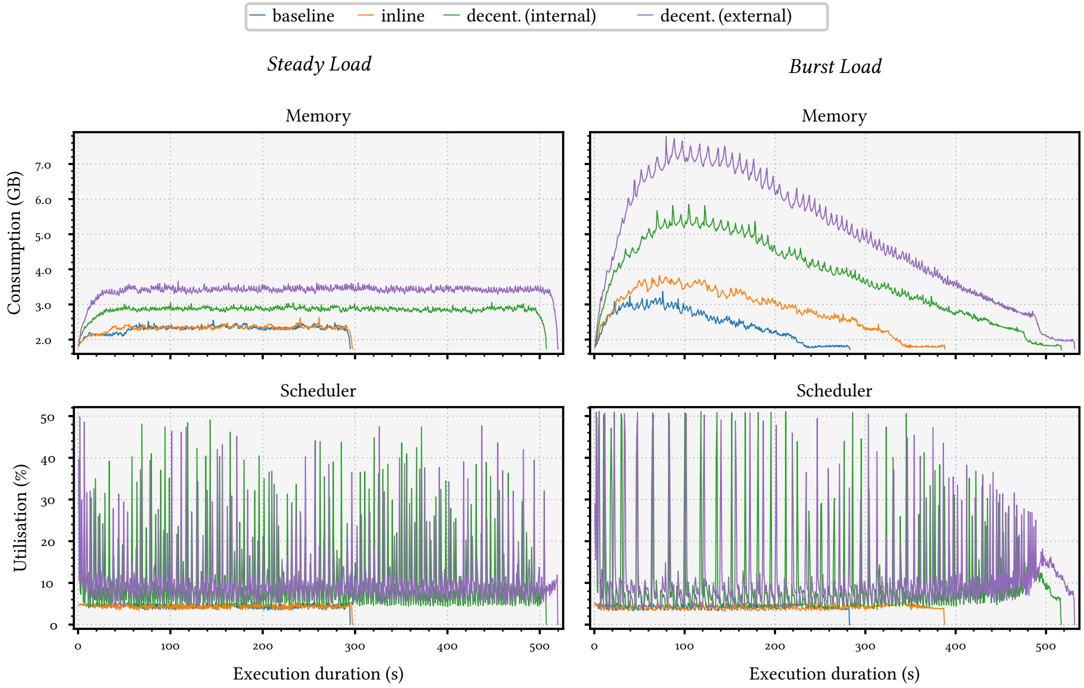


Figure 7.8. *Resource consumption for decentralised monitoring under high load benchmarks* (500k *workers)*

Steady and Burst loads respectively, and ≈ 1% and ≈ 31% for inlined monitors. The memory consumption plots in figure 7.6 (and also figures 7.3 and 7.5) demonstrate the potential of decentralised approaches to scale as the SuS is subjected to increasing load. These figures give the mean memory consumption over the duration of the benchmark executions, which conceals how our decentralised algorithm uses this resource optimally at runtime.

Figure 7.8 replots the decentralised monitoring runs in figure 7.7 to highlight this perspective. The memory consumption patterns in figure 7.8 mirror the profiles of the loads applied (see figure 6.5 for examples), confirming that our decentralised approach grows and shrinks in response to dynamic fluctuations in the load (*cf.* figure C.5 for Steady *vs.* Pulse load). This elasticity results from instrumenting monitors when needed and garbage collecting them when these become redundant to minimise the memory footprint (see section 5.2.7). Centralised monitoring does not exhibit this adaptable behaviour and its use of memory grows steadily, regardless of the load profile applied (figure 7.7 accentuates the substantial difference in memory consumption between decentralised and centralised monitors). Similarly, its scheduler utilisation is largely insensitive to the load profile applied. This occurs despite load profiles dictating different worker creation schemes, which, however, have no effect since the trace events exhibited by workers are always funnelled through a single monitor. In the decentralised approach, the creation and termination of monitors follows that of worker processes. This influences the scheduler utilisation, as figure 7.8 indicates, albeit on a small scale. For the case of Steady load, the utilisation oscillates consistently due to the continual influx of trace events, whereas under Burst load, utilisation is less concentrated and increases slightly towards the end.

Closely inspecting the frequency and amplitude of the scheduler utilisation plots in figures 7.7 and 7.8 corroborates the observation made in section 7.2.2 about decentralised monitoring, namely that, monitors process events quickly and revert to waiting. The prompt handling of trace events by decentralised monitors appears to manifest as peaks in figure 7.8, whereas waiting periods (where monitors are placed on the EVM run queues) are reflected in the regions that show stable scheduler utilisation. Peaks with high amplitude suggest the *simultaneous* use of multiple scheduler threads. The absence of such peaks in the plots of figure 7.7 for centralised monitoring comes from the single-process monitor that is unable to leverage other unoccupied EVM scheduler threads. This is especially evident in the sub ≈ 3.08% scheduler utilisation under both Steady and Burst loads. Figure C.6 depicts the load on the individual 16 EVM schedulers to certify this deduction. It indicates evenly-distributed utilisation across schedulers S_1 to S_{16} for decentralised monitoring (top) under Steady and Burst loads throughout the benchmark run. This makes it consistent with the peaks in the mean scheduler utilisation plot of figure 7.8. By contrast, the load distribution for centralised monitoring in figure C.6 (bottom) becomes concentrated on scheduler S_1 and S_2 once the master-worker system stops executing.

7.3 Monitoring Lower Concurrency Systems

Section 7.2 attests that our decentralised monitoring approach is reactive. At the same time, it preserves the reactive aspect of the SuS by inducing feasible runtime overhead. Centralised monitoring lacks both of these traits. This section considers the second type of reactive architecture, $\mathrm{RS_L}$, which models systems with comparably lower concurrency that focus on long-running computational tasks. We demonstrate that a centralised approach fails to scale in such settings. We also show that decentralised outline monitoring scales even *better* than on system $\mathrm{RS_H}$, and induces overheads *on par* with its inline counterpart.

In these experiments, our master-worker models use *moderate* loads of $n = 1\text{k}$ workers with $w = 10\text{k}$ work requests per worker on set-up $\mathrm{SU_E}$ (edge-case scenarios), and *high* loads with $n = 5\text{k}$ and $w = 10\text{k}$ on $\mathrm{SU_G}$ (general-case scenarios). As before, we set the EVM with 4 scheduler threads on set-up $\mathrm{SU_E}$ and 16 threads on $\mathrm{SU_G}$, keeping the simulated slowdown of ≈ 5μs per trace event. The changes in the benchmark configuration alter the way the execution of our master-worker models unfolds w.r.t. the ones in section 7.2. Concretely, the master instantiates most of its worker processes relatively early in runs and spends the remainder of its execution busy, allocating work requests. This increases the message throughput within the system, *e.g.* table 7.1 shows almost a two-fold growth in throughput for the experiments performed with 5 k workers by comparison to the ones with 500 k in section 7.2. Consequently, our attempts at benchmarking centralised monitors on set-ups $\mathrm{SU_E}$ and $\mathrm{SU_G}$ were consistently hampered by the rapid accumulation of trace events in the backlog of the central monitor that, eventually, exhausts the available memory. For this reason, we only consider the inline and outline (internal variant, figure 5.1b) forms of decentralised monitors in what follows.

Figure 7.9 draws the comparison between our experiments of section 7.2 taken with 500 k workers and the ones taken on set-up $\mathrm{SU_G}$ with 5 k workers under Steady and Burst loads. Since the two experiment set-ups are *incomparable* in their number of processes, figure 7.9 plots the performance metric (*e.g.* memory consumption, y-axis) against the benchmark *iteration number* (x-axis). We recall that each 500 k and 5 k benchmark run generates approximately the same number of message exchanges between the master and worker processes, enabling us to compare the two (*cf.* table 7.1).

The bar plots in figure 7.9 show that decentralised outline monitoring (*outline*) in system $\mathrm{RS_L}$ with 5 k

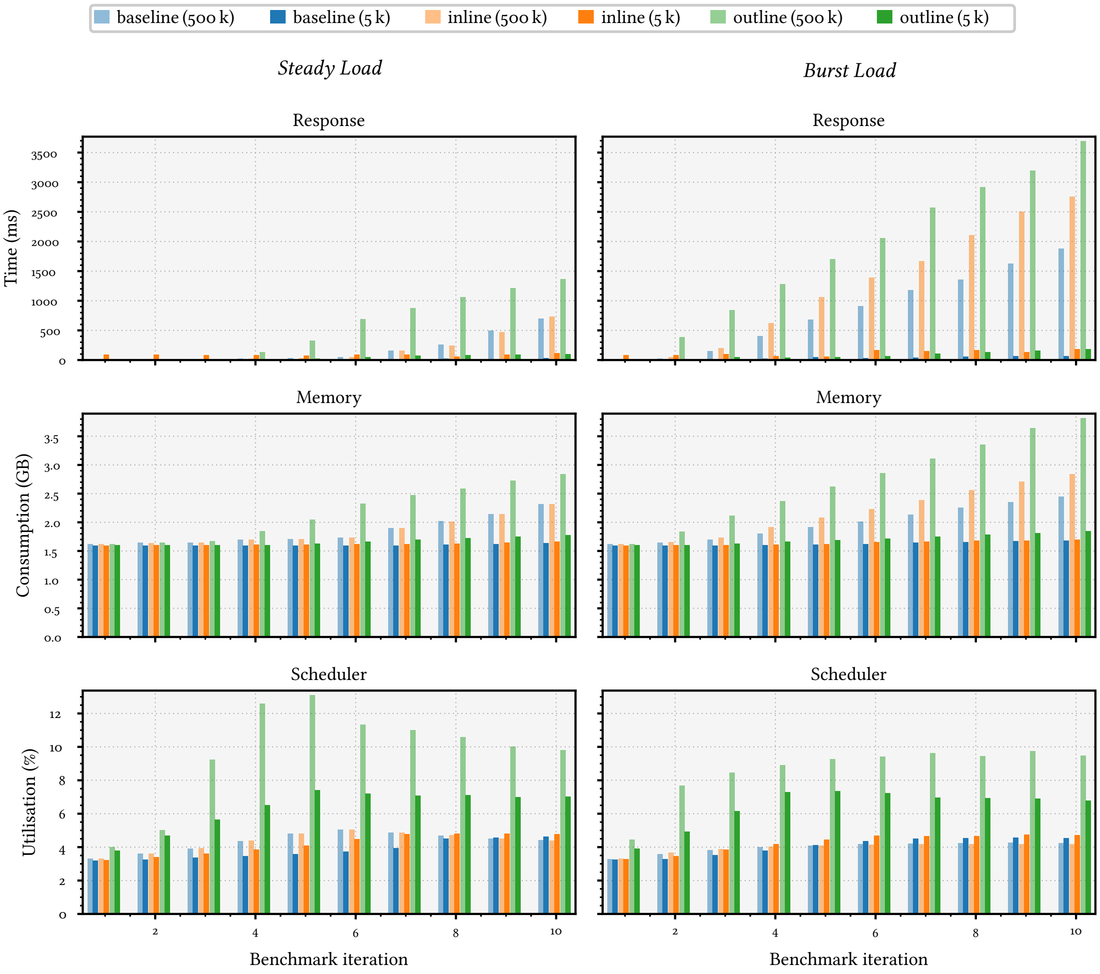


Figure 7.9. *Gap in decentralised monitoring overhead on the system under high load benchmarks* (500k *vs.* 5k *workers)*

workers induces less memory and scheduler overhead, compared to the experiments of system RS_{H} with 500 k workers. This occurs despite the fact that *both* of these configurations generate an approximately equal amount of load in terms of analysable trace event messages (see table 7.1). Table 7.4 estimates these overheads w.r.t. the baseline systems RS_{H} and RS_{L} for the maximum loads at 500 k and 5 k workers respectively. For instance, outline monitors increase the memory overhead by 8 % in our experiments on system RS_{L} *vs.* 23 % on RS_{H} under Steady load, and by 10 % *vs.* 56 % on RS_{L} and RS_{H} respectively under Burst load. The corresponding scheduler plots exhibit analogous trends, with 52 % overhead increase (system RS_{L}) *vs.* 123 % (system RS_{H}) under Steady load, and 50 % (system RS_{L}) *vs.* 123 % (system RS_{H}) under Burst loads. We conclude that this decrease in overhead for outlining on system RS_{L} stems from the lower number of worker processes the master creates, that (i) requires our decentralised algorithm to perform *fewer* reconfigurations to manage the monitor choreography, and (ii) *minimises* the trace event routing performed as a result (refer to section 5.2.3). By contrast to outlining, decentralised inline monitoring (*inline*) registers negligible changes in both memory consumption and scheduler utilisation between our experiment set-ups RS_{L} and RS_{H}. While outline monitoring does not lower the relative response time w.r.t. the baseline set-up on RS_{H}, it *does* induce less latency than inline monitoring on

System	Load	Response time %		Memory consumption %		Scheduler utilisation %	
		Inline	Outline	Inline	Outline	Inline	Outline
RS_H	Steady	4	95	1	23	0	123
	Pulse	50	134	11	41	0	126
	Burst	55	97	16	56	0	123
RS_L	Steady	246	194	1	8	3	52
	Pulse	212	198	0	8	6	57
	Burst	193	190	1	10	4	50

Table 7.4. *Percentage overhead on RS_H (500 k workers) and RS_L (5 k workers) w.r.t. baseline at maximum load*

system RS_L. Table 7.4 reveals that the response time overhead on system RS_H for outline monitoring increases by 95 % and 97 % under Steady and Burst loads respectively, and by 194 % and 190 % on RS_L. By comparison, inline monitoring inflates the response time by 4 % and 55 % under Steady and Burst loads on RS_H, and by 246 % and 193 % on system RS_L. In fact, the *absolute* response time due to inline monitoring is slightly higher than that of outline monitoring on system RS_L (115.80 ms *vs.* 98.40 ms under Steady load and 181.85 ms *vs.* 179.65 ms under Burst load). Figure 7.9 shows that both approaches consume comparable amounts of memory. However, decentralised outline monitoring utilises more of the scheduler than its inline equivalent, owing to the reconfiguration and trace event routing that outline monitors conduct.

Despite the cost paid in terms of scheduler utilisation, our decentralised approach yields marginally lower latency than inline monitoring. We note that the slight degradation in the response time for inline monitoring arises from a combination of the increased trace event throughput and delay in the analysis, which results in frequently 'pausing' worker processes. As remarked in section 7.2.2, this behaviour for inlined monitors could potentially deteriorate further in cases of slower runtime analyses. Decentralised monitoring mitigates this issue by decoupling the instrumentation and analysis tasks. The results of our experiments conducted on set-up SU_E using system RS_H (100 k workers) and system RS_L (1 k workers) are plotted in figure C.2, and are in line with the conclusions drawn above.

7.4 Discussion

Monitoring reactive systems calls for component-based techniques that are reactive, *i.e.,* they are responsive, resilient, elastic, and message-driven. This chapter validates our decentralised outline monitoring algorithm detailed in chapter 5 w.r.t. these four reactive characteristics via a systematic empirical study. We show that the qualitative arguments for decentralised outline monitoring in section 1.1.2 are in line with the quantitative evidence collected in experiments, confirming that our algorithm is, indeed, reactive. In particular, these experiments affirm that the overhead induced by decentralised outline monitoring is *feasible* in practice. Our comprehensive evaluation of sections 7.2 and 7.3 considers (i) different combinations of hardware and software, set up with (ii) two reactive system models that test edge-case and general-case scenarios, under (iii) high loads that go beyond the state of the art in RV, using (iv) realistic load profiles that, to the best of our knowledge, are not considered in the literature. These parameters give us assurance that our conclusions are portable to other platforms, generalisable to various reactive architectures under different load models, and, more importantly, applicable to real-world cases; this is generally not done in other studies *e.g.* [184, 185, 62, 61, 197, 43, 176, 52, 53, 219, 71, 72, 70, 113, 87, 89, 39, 180, 158, 47]. Our evaluation of decentralised outline monitoring is conducted alongside its widely-adopted inline counterpart [91, 90, 25], providing us with a reference point against which our results can be interpreted in a general way. Under these conditions, we also demonstrate that centralised monitoring exhibits none of the attributes of reactive systems due to its inherent analysis bottleneck (*e.g.* Schneider et al. [205] make a similar observation about bottlenecks in their experiments). Moreover, centralised set-ups are prone to failure in scenarios with high-loads such as the ones we used.

Section 7.3 compares decentralised outline and inline monitoring in further detail. It shows that in situations with low to mild concurrency, where system components engage in long-running tasks, outline monitoring performs better than in scenarios involving short-lived tasks (*cf.* section 7.2). In fact, outline monitoring induces comparable memory and response time overhead to that of inline monitors, making it the preferred choice in such cases owing to the other benefits it offers (see section 2.1.4)

We conjecture that outlining also yields low overhead—on par with inlining—in high concurrency settings where the number of system components becomes *stable*, as in section 7.3. In such cases, our decentralised approach should perform well, since it minimises the reconfiguration and message routing that is needed to organise the monitor choreography continually. Since we aim for generality, the results presented in this chapter assume a *worst case* scenario where every component of the SuS is monitored. On this account, we expect decentralised outline monitoring to induce even lower overhead when the number of system components monitored is reasonable (*e.g.* a few hundreds). Both of these assertions warrant further investigation and are left as future work.

7.4.1 Related Work

Our empirical study explores various aspects of runtime monitoring, such as the instrumentation overhead, robustness, and scalability of monitoring approaches, using different metrics to gauge the effect of runtime overhead. While these topics are discussed at different depths by the RV community, our observations in sections 7.2 and 7.3 call into question some of these notions that tend to be occasionally overlooked by, or not satisfactorily tackled in the literature.

Numerous works (*e.g.* [124, 34, 71, 67, 70, 68]) based on inlining do not delineate the instrumentation and runtime analysis aspects. This is common in monolithic settings (see section 2.1.4), where the instrumentation and analysis tasks are coalesced, and the former is often assumed to induce minimal runtime overhead [91, 25]. Consequently, many inlining-based approaches focus on the efficiency of the analysis without considering the instrumentation cost (*e.g.* Falcone et al. [95] attribute the overhead to the analysis aspect alone). This line of reasoning for single-component systems is often ported to the concurrent setting. For instance, [175, 209, 42, 61, 207, 99, 24] propose efficient runtime monitoring algorithms but do not account for, nor quantify the overhead due to collecting trace events. Similarly, [209, 61, 101] inline components with variants of vector clocks to exchange partial information via messaging but overlook the potential memory overhead that may result from the increased size of the message payloads. Section 7.2.1 shows that the overhead due to inlining in component-based settings is non-negligible, which makes the efficiency claims in the cited works unsubstantiated from an instrumentation overhead point of view. Tools such as [53, 51, 219, 47, 113, 229] that *do* quantify the runtime overhead, aggregate the instrumentation and runtime analysis costs, making it difficult to gauge whether potential inefficiencies arise from one or the other. Since the overhead due to the analysis of events depends on different factors (*e.g.* table 7.2), the inability to isolate the respective costs of the instrumentation and analysis limits the interpretability of their results.

The notion of perceived minimal overhead induced by instrumentation is often extended to offline monitoring [100], where events are persisted for subsequent processing. Certain surveys [95, 55] or introductory textbooks [68] either claim that offline monitoring imposes low overhead because the system observation consists 'only' in recording trace events, or are otherwise vague about this overhead [25, 100]. Section 7.2.1 makes a strong case that all forms of instrumentation induce a degree of overhead that is *unavoidable* when observing software systems. In addition, this overhead will be influenced by the technique employed to persist events (*e.g.* file, DB, pub-sub infrastructures [217]) for the case of offline monitoring. We have also shown that the instrumentation overhead depends on the load that the SuS is subjected to, *e.g.*, the difference in overhead between the inline and baseline plots is more evident under Burst load than with Steady load (figure 7.2). Moreover, section 7.2.3 reveals that in our benchmarks, a sizeable portion of the runtime monitoring overhead originates from the instrumentation for the cases of inline and decentralised outline monitoring.

Figures 7.3 and 7.5 show how the performance of our online centralised monitors degrades when a minimal analysis cost is added on top of the instrumentation. Despite this bottleneck-induced issue that leads to crashes in figure 7.3, centralised monitoring is *still* employed by RV tools that target concurrent software. One plausible reason for this is that the empirical evaluation of such RV tools lacks proper benchmarking (*e.g.* [71, 21, 209, 101, 131]), or utilises meager loads that fail to exercise the tool and expose the shortcomings of centralised approaches (*e.g.* [180, 113, 51, 53, 52, 12, 170]). Another potential motive is that centralised *offline* approaches can avoid overloading the central monitor by controlling the rate at which trace events are read from storage and subsequently analysed [99, 101]. In offline mode, this is done under the assurance that, regardless of the speed pre-recorded traces are processed with, no event loss occurs. However, implementing this strategy in online use cases is typically hard in reactive scenarios where system components continually generate streams of trace events directed toward one central monitor. Throttling events in an asynchronous setting, while possible by applying back-pressure [153] to system components, cannot be achieved unless the monitor heavily interferes with the SuS.

Monitoring is a cross-cutting concern [146] that can be encapsulated in own logic unit [95, 59, 68]. Various RV tools such as [70, 60, 52, 221, 13, 197] follow this separation-of-concerns approach where the monitor analysis is kept isolated from the logic of the SuS. Our decentralised outline algorithm extends this notion and separates the execution of the monitor logic from the system by executing monitors as independent processes. This makes the approach less sensitive to slowdowns in the analysis, enabling it to runtime check richer properties whose corresponding monitors could potentially induce varying delays (refer to discussion in section 7.2.3). Online tools using centralised monitoring (*e.g.* [71, 23]) are sensitive to delays in the analysis since these indirectly affect the speed with which events are processed from the central tracing entity. As seen in section 7.2, this increases the consumption of memory, which coupled with the SPOF, could render such tools inapplicable in practice.

RV for single-component systems generally uses the execution slowdown as its principal indicator of runtime overhead (see discussion in section 1.1.3). In reactive settings, this one-dimensional view is inadequate, as the *omitted evidence* could bias the interpretation of empirical results, *e.g.* in consulting only figure 7.6 (top), one would falsely conclude that inlining induces the lowest slowdown without affecting the response time. Despite this, approaches for concurrent RV still base their findings on the execution slowdown (*e.g.* Neykova and Yoshida [185]) or memory consumption (*e.g.* Meredith et al. [176]); [51, 52, 219, 47, 205] are few of the notable exceptions that account for the response time. Others [67, 87, 96] abstract from these metrics, and concentrate instead on the volume of messages that are exchanged between component monitors. While the count of messages exchanged is indicative of efficient communication, it makes it difficult to quantify the overhead in practical terms *e.g.* response time, and memory consumption. The volume of message exchanges is not a metric we track in our benchmarks. Yet, it warrants further consideration, particularly when used alongside our current metrics identified in section 6.1.

8 Conclusion

This thesis investigates how the correctness of reactive systems can be established dynamically at runtime. It considers a lightweight monitoring approach called RV that circumvents the issues connected with traditional pre-deployment verification methods, such as testing and model checking. One major obstacle of RV for reactive systems is in choosing a monitoring technique that does not impinge on the reactive characteristics of the SuS. We hold that this is attainable *only* if the monitoring set-up is itself reactive.

This thesis investigates a novel decentralised outline monitoring approach based on this precept. The approach treats the SuS as a black box: it instruments monitors dynamically and in an asynchronous fashion, which is more attuned to the requirements of reactive architectures. Our development is systematic. We adopt the modular RV practice advocated by Aceto et al. [6, 8], which delineates the semantics of the specification language used to describe the properties that the SuS should comply with, and the semantics of the monitors that check for these property descriptions. The separation of concerns prescribed by the authors gives a principled approach for studying what correct monitors are, and for identifying properties that can be monitored at runtime. This enables the construction of mechanical syntheses procedures that generate correct monitors for monitorable properties. Equally crucial, it permits us to directly map the constituent parts of our formal model to executable code modules, giving us assurances that the correctness results obtained in the theory [6, 8] are preserved in the implementation. Through our study, we make the following contributions.

(i) Build on the theoretical results of Aceto et al. [6] and augment their specification formalism, operational semantics of monitors, and monitor synthesis procedure with predicates to reason on the data carried by trace events. Our extensions make their model amenable to practical use. We implement these extensions and give a technique for instrumenting inline monitors. Additionally, we define an asynchronous instrumentation relation that decouples the operation of the SuS and monitors, in line with the tenets of reactive architectures.

(ii) Devise a decentralised outline monitor instrumentation algorithm that instantiates the asynchronous instrumentation of contribution (i). Our algorithm employs a tracing infrastructure to collect events as the SuS executes and instruments monitors dynamically based on key events observed in the trace. The algorithm accounts for the interleaving of trace events that arise from the asynchronous execution of the SuS and monitors, guaranteeing that the events are reported to monitors in the correct order and without loss.

(iii) Develop a configurable RV benchmarking framework tailored for reactive systems. The framework can generate synthetic SuS models that are shown to reproduce the realistic behaviour of master-worker systems. Our tool collects performance metrics relevant to reactive software, thereby giving a multi-faceted depiction of the overhead induced by monitoring tools. This is conducive to assessing such tools reliably, increasing our confidence in their real-world application.

(iv) Give an extensive evaluation of the overhead induced by our implementation of decentralised outline instrumentation of contribution (ii), using the benchmarking tool developed in (iii). We compare this algorithm against our implementations of inline and centralised outline instrumentation—two popular methods used in the state-of-the-art RV tools. These benchmarks demonstrate that the decentralised approach we propose induces feasible overhead, which for typical cases, is comparable to or outperforms, the inline and centralised approaches. We are unaware of other comprehensive empirical RV studies such as ours that compare decentralised, centralised, and inline monitoring.

These contributions culminated in a suite of tools towards our research goal that:

- demonstrates that the formalisations and methods proposed in contributions (i) and (ii) are implementable in a general-purpose language that targets applications built on the reactive principles;
- debunks the commonly-held belief that decentralised outline instrumentation is necessarily infeasible, showing that it induces acceptable overhead, which in typical cases, is comparable to inlining;
- confirms that centralised monitoring is prone to scalability issues, poor performance, and failure, which makes it generally inapplicable to reactive system settings.

In cases where inlining cannot be performed (see section 2.1.4 for reasons why), a decentralised outline instrumentation approach such as the one we propose is the only viable method to conduct runtime monitoring. Readers may access the source code for the artefacts developed for this thesis here.

8.1 Avenues of Future Research

Our investigation is by no means conclusive; we believe that other research avenues may be followed as a result of our work. The ones suggested below are listed in no particular order.

8.1.1 Parametrised Recursion Variables

Certain properties cannot be expressed in our logic μHML^{D}. Consider an asynchronous server that exhibits the actions con, end, req, and res. The actions con and end respectively demarcate the start and termination of a communication session with our server, whereas req and res denote asynchronous requests and responses. One safety property that this system should observe is that in any communication session (starting with con and terminating with end), all requests are fulfilled. This property describes the language of ω-words in which every finite communication session, the number of observed req actions equals the number of observed res actions. Such a property is not ω-regular.

We propose an extension to the logic that augments the (i) least and greatest fixed point constructs with parametrised variables $x,y \in \text{DV}\textsc{ar}$, and expressions $e,f \in \textsc{Exp}$, *i.e.*, $\mathsf{min}X(x).(\varphi)(e)$ and $\mathsf{max}X(x).(\varphi)(e)$, and (ii) recursion variables with expressions, *i.e.*, $X(e)$. This enables data values to be handed down between successive unfolding of recursive constructs (see also [171, 125]). Via this logic, the aforementioned property can be expressed as the formula below, where the counter y is used to track the number of requests and responses processed by the server.

$$\max X(x).\Big([\mathsf{con}]\max Y(y).\big([\mathsf{req}]\,Y(y{+}1)\wedge[\mathsf{res}]\,Y(y-1)\wedge$$
$$[\mathsf{end}, y=0]\,X(0)\wedge[\mathsf{end}, y\neq 0]\,\mathsf{ff}\big)(x)\Big)(0)$$

We envisage this investigation to replicate the programme of study carried out in [118, 6, 8]. This entails determining possible monitorable logic fragments (*e.g.* safety and co-safety), studying whether the fragments identified can syntactically characterise all the expressible monitorable properties, and devising syntheses procedures that generate monitors from these fragments. The study can be undertaken for both the linear-time and branching-time interpretations of this logic.

8.1.2 Managing the Number of Active Monitor States

Our monitoring algorithm of section 4.3 considers all the possible monitor states, thereby ensuring that monitors are partially-complete (definition 3.3). The operational rules mDisY$_\text{L}$, mDisN$_\text{L}$, mConY$_\text{L}$, and mConN$_\text{L}$ (and their symmetric counterparts) of figure 3.2 are used to terminate redundant monitor states as soon as these are encountered during the runtime analysis. Section 4.3 also argues that emulating the disjunctive and conjunctive parallel composition constructs minimises overhead, by comparison to forking independent component sub-monitors. Monitoring performance may be further optimised by placing a bound on the number of active monitor states that our algorithm manages at runtime. This pragmatic trade-off comes at the expense of sacrificing partial-completeness, which manifests as possibly-missed verdict detections (*e.g.*, the work by Grigore et al. [124]). Monitors that are subject to missed detections may not always be ideal in monolithic settings where applications often consist of a *single* instance. However, reactive architectures can alleviate the effect of missed detections by virtue of replicated components: such a set-up improves the chance that potential detections missed by one monitor may still be reached by other monitor replicas. Note that missed detections still preserve our non-negotiable requirement of sound monitoring, *i.e.,* accept (resp. reject) verdicts that monitors flag imply formulae satisfactions (resp. violations) in the logic.

8.1.3 Component Replication and Monitorable Properties

Component replication opens the possibility of analysing more than one trace of the same component instance and, potentially, monitoring for more properties. For instance, the regular μHML *branching-time* formula, $\varphi_{11} = [\mathsf{a}]\,\mathsf{ff} \vee [\mathsf{b}]\,\mathsf{ff}$ (see section 2.2), is not monitorable in a traditional RV set-up assuming a single execution [6]. Intuitively, this is because observing one trace prefix, say a, that leads to a violation of $[\mathsf{a}]\,\mathsf{ff}$, still requires a second trace to determine whether φ_{11} is violated. However, multiple traces of the same component instance, *e.g.* one trace prefix that starts with a and another starting with b, provide the monitor with sufficient evidence to flag a rejection [4].

The above rudimentary example conceals several challenges. Consider the branching-time formula $\varphi_{12} = [\mathsf{a}]\,([\mathsf{b}]\,\mathsf{ff} \vee [\mathsf{c}]\,\mathsf{ff})$, expressing the requirement that *'after performing the action a, the state that the system reaches can neither perform the action b* nor *c'*. Trace prefixes such as a.b and a.c do not give sufficient information as to whether this property is violated. The reason behind this is that the transitions $p_1 \xrightarrow{\mathsf{a}} p_2 \xrightarrow{\mathsf{b}} p_3 \longrightarrow \cdots$ and $q_1 \xrightarrow{\mathsf{a}} q_2 \xrightarrow{\mathsf{c}} q_3 \longrightarrow \cdots$ (for some p_i, q_j) that give rise to these traces, potentially refer to *unrelated* paths of the component execution graph. When the states $p_1 = q_1$

and $p_2 = q_2$, the traces a.b and a.c share the same initial state p_1 *and* a-derivative state p_2; since p_2 can perform both actions b and c, formula φ_{12} is violated. If $p_2 \neq q_2$, however, φ_{12} is not violated.

Different methods can be explored to address the lack of information in execution traces. One conceivable route is to annotate traces by inlining the monitored component to produce trace events that embed component state metadata. In actor-based paradigms (*e.g.* Erlang, Akka), such a notion of state could consist of a snapshot of all the internal variables that a process mutates over time as a side-effect of the messages it sends and receives. For example, the monitor inlining procedure of section 4.5 can be modified to extend the event payload (*e.g.* lines 4 and 6 in figure 4.5b) to include the values of variables *Tok* and *NextTok*. It is worth noting that the solution we describe may be subject to the limitations of inlining (see section 2.1.4), and implementing a similar procedure with outlining will depend on the flexibility of the tracing infrastructure used.

8.1.4 Failure Injection

Our benchmarking framework of chapter 6 can be naturally extended to accommodate a second widespread software architecture, namely peer-to-peer systems. This gives the tool more scenario coverage and could circumvent the performance bottleneck associated with master-worker set-ups [202]. Another aspect that warrants consideration is the addition of controlled fault injection based on the probability distributions we currently employ to induce load on benchmark models (*i.e.,* Steady, Pulse, and Burst loads). Randtoul and Trinder [195] propose a reliability benchmark for Erlang systems that inject faults in pairs of actor processes that exchange messages. The authors induce failures by forking dedicated 'killer' processes at predetermined intervals to terminate processes, thereby simulating fail-stops [83]. This approach may not be applicable to our case since the creation of 'killer' processes induces additional overhead that can influence the execution of benchmark models, and subsequently, bias the results of empirical experiments. We propose an alternative lightweight design that integrates the termination logic within system processes. Link and communication omission failures [83] are a class of failures whereby work requests that are in transit between components (*e.g.* master and worker) can be dropped, delayed, duplicated, or mutated. This can be implemented by adding proxy logic inside system processes to emulate these failures. Modelling failures enable us to test other facets of runtime monitoring. One metric worth considering is the *detection time*, which measures the time monitors take to reach verdicts in the face of failure. This metric is particularly relevant to a set-up where monitors consider traces from replicated components since it can be used to gauge the efficacy of verdict detection under different probability models and failure severity.

8.1.5 Decentralised Inline and Outline Monitoring

Our decentralised outline monitoring instrumentation leverages the native tracing infrastructure provided by the EVM, making it accessible to any application that executes on the platform (*e.g.* Le Brun et al. [162] use outline monitors to verify properties of an Elixir implementation of the Raft consensus algorithm [190]). Inline instrumentation relies on source-level weaving, and is, therefore, limited to Erlang code. The next stage of development is to revisit inlining and add support for BEAM object code compiled with debugging symbols. Lifting assumption A_1 (*i.e.,* components do not fail-stop or exhibit Byzantine failures) and A_2 (*i.e.,* messaging is reliable) opens up our decentralised approach to distributed settings, introducing various challenges. Chief among these challenges is the capacity of the instrumentation to manage failure. Notable works that can inform this research direction are those by Basin et al. [31], which considers the problem of monitoring distributed systems with failing components and network links, and Bonakdarpour et al. [45] that address failure within monitors themselves, specifically, in the case of fail-stop.

A Further Decentralised Outline Instrumentation Details

Our message routing and forwarding operations described in section 5.2 enable tracers to implement hop-by-hop routing. These operations are given in listing 5. The function self() on line 2 returns the PID of the calling process. Listing 5 includes the Tracer function that is forked in listing 2 to execute the core tracer logic of listings 3 and 4. Detach is used to signal to the router tracer p_T that the system process p_S is being tracer by a new tracer, p'_T. Prior to issuing the message, detach invokes Preempt so that p'_T takes over the tracing of system process p_S. TryGC determines whether a tracer can be safely terminated. For the case of the external analysis variant of figure 5.1a, TryGC also signals the analyser to terminate. The analyser terminates asynchronously so that it can process potential trace events it might still have in its message buffer.

Start in listing 6 launches the SuS and monitoring system in tandem. The operation accepts the code signature g, as the entry point of the SuS, together with the instrumentation map, Φ. As a safeguard that prevents the initial loss of trace events, the SuS is launched in a paused state (line 2) to permit the root tracer to start tracing the top-level system process. Root resumes the system (line 7), and begins its trace inspection in *direct* mode, as shown on line 9.

The tracing mechanism is defined by the operations Trace, Clear, and Preempt listed in listing 7, and are overviewed in section 5.2.1.

```
Expect: $k.\text{type} = \text{evt} \lor k.\text{type} = \text{dtc}$
1  def Route($k$,$p_T$)
2      $p_T$ ! ⟨rtd,self(),$k$⟩

3  def Tracer($\varsigma$,$m$,$p_S$,$p_T$)
       # New tracer state $\varsigma'$ initialised with an
       # empty routing map $\emptyset$, a copy of the
       # instrumentation map $\varsigma.\Phi$, and the
       # traced-component map is set to the
       # (first) process being traced, $p_S$
4      $\varsigma' \leftarrow \langle \Pi \leftarrow \emptyset, \varsigma.\Phi, \Gamma \leftarrow \{\langle p_S, \bullet \rangle\}\rangle$
5      Detach($p_S$,$p_T$)
6      $p_M \leftarrow$ fork($m$) executable monitor
       # Start in $\bullet$ mode to prioritise routed events
7      Loop$_\bullet$($\varsigma'$,$p_M$)

Expect: $k.\text{type} = \text{rtd}$
8  def Forwd($k$,$p_T$)
9      $p_T$ ! $k$

10 def Detach($p_S$,$p_T$)
11     $p'_T \leftarrow$ self()
12     Preempt($p_S$,$p'_T$)
13     $p_T$ ! ⟨dtc,$p'_T$,$p_S$⟩

14 def TryGC($\varsigma$,$p_M$)
15     if ($\varsigma.\Gamma = \emptyset \land \varsigma.\Pi = \emptyset$)
16         Signal analyser $p_M$ to terminate
17         Terminate tracer
```

Listing 5. *Operations used by the (○) and priority (●) tracer loops*

```
def START($g$,$\Phi$)
  # Pausing allows root tracer to be set
  # up; no initial message loss
  $p_S \leftarrow$ fork($g$) in paused mode
  $p_T \leftarrow$ fork(ROOT($p_S$,$\Phi$))
  return $\langle p_S,p_T \rangle$

def ROOT($p_S$,$\Phi$)
  TRACE($p_S$,self())
  Resume system $p_S$
  $\varsigma \leftarrow \langle \Pi \leftarrow \emptyset, \Phi, \Gamma \leftarrow \{\langle p_S, \circ \rangle\}\rangle$
  # Root tracer has no monitor
  LOOP$_\circ$($\varsigma$,$\bot$)
```

Listing 6. *System starting operation and root tracer*

```
def TRACE($p_S$,$p_T$)
  if ($p_S$ is not traced)
    Set tracer for $p_S$ to $p_T$
    # $p_T$ will trace descendants of $p_S$, $A_5$
    while $p_S$'s tracer is set do
      $s \leftarrow$ next event exhibited by $p_S$
      $e \leftarrow$ encode $s$ as a message
      $p_T$ ! $e$
    end while

Expect: $p_S$'s tracer is set
def CLEAR($p_S$,$p_T$)
  if ($p_S$ is traced)
    Clear tracer $p_T$ from $p_S$
    # $p_T$ still traces descendants of $p_S$, $A_5$
    repeat
      # Wait for $p_S$'s in-transit trace event
      # messages to get delivered to $p_T$, $A_2$
    until trace events of $p_S$ are delivered to $p_T$

def PREEMPT($p_S$,$p_T$)
  $p'_T \leftarrow p_S$'s tracer
  CLEAR($p_S$,$p'_T$)
  TRACE($p_S$,$p_T$)
```

Listing 7. *Abstraction of the operations offered by the tracing infrastructure*

B Case Study: Monitoring Reactive Applications

Our tool implementation supports a succinct pattern notation where atomic values can be *directly* specified in patterns, *e.g.* $*\langle\_,x_2\rangle, x_2 = \texttt{atom}$ may be written as $*\langle\_,\texttt{atom}\rangle$. This notation is employed in the ensuing examples. We elide redundant binders and variables from formulae patterns for succinctness using the 'don't care' pattern _, when necessary.

B.1 Monitoring the Master-Worker Model

The master-worker model used in our benchmarking tool of chapter 6 employs a simple protocol to track the work requests distributed to different workers. Workers are initialised with IDs, which we denote by the placeholder *Id*, which enables the master to track the progress of *tasks* assigned. Each worker task is comprised of a sequence of *work requests* totalling *NumReqs*. Work requests in a task are incrementally numbered with a sequence number, *ReqNum*, where $1 \leq \mathit{ReqNum} \leq \mathit{NumReqs}$, identifying the request submitted to a worker. The master process relies on the request number to determine when a task assigned to a particular worker is completed. Tasks are marked complete when $\mathit{ReqNum} = \mathit{NumReqs}$, at which point, the master sends a termination instruction to the worker. Work requests are uniquely identifiable from all other work requests issued by the master via the triple $\langle \mathit{Id}, \mathit{ReqNum}, \mathit{NumReqs}\rangle$. The work responses relayed by workers to the master are identified in the same manner. The following summarises the different messages exchanged between the master and worker processes:

- $\langle \mathit{Pid}_{\mathrm{M}}, \langle \mathsf{chunk}, \langle \mathit{Id}, \mathit{ReqNum}, \mathit{NumReqs}\rangle\rangle\rangle$: work request message sent by the master process to the worker
- $\langle \mathit{Pid}_{\mathrm{M}}, \langle \mathsf{term}, \langle \mathit{Id}, \mathit{ReqNum}, \mathit{NumReqs}\rangle\rangle\rangle$: termination message sent by the master process to the worker once a task is complete, *i.e.,* $\mathit{ReqNum} = \mathit{NumReqs}$
- $\langle \mathit{Pid}_{\mathrm{W}}, \langle \mathit{chunk}, \langle \mathit{Id}, \mathit{ReqNum}, \mathit{NumReqs}\rangle, \mathsf{ack}\rangle\rangle$: work response message sent by the worker process to the master
- $\langle \mathit{Pid}_{\mathrm{W}}, \langle \mathit{chunk}, \langle \mathit{Id}, \mathit{ReqNum}, \mathit{NumReqs}\rangle, \mathsf{complete}\rangle\rangle$: completion message sent by the worker process to the master when the last work request in a task has been processed, *i.e.,* $\mathit{ReqNum} = \mathit{NumReqs}$

The local properties used in section 6.5.1 to monitor the master-worker models concern the operation of workers, and are specified from their point of view.

Example B.1. Consider the property stating that *'no worker ever crashes'*, specified as the recursive maxHML$^{\mathrm{D}}$ formula:

$$[\leftarrow\langle\_,\_,\_,\_,\_\rangle]\,\mathsf{max}\,X.\big([?\langle\_,\_\rangle]([\,!\,\langle\_,\_,\_\rangle]X \wedge [*\langle\_,\_\rangle]\mathsf{ff}) \wedge [*\langle\_,\_\rangle]\mathsf{ff}\big) \qquad (\varphi_{13})$$

Formula φ_{13} does not make use of the data embedded in work requests issued by the master. It merely matches the shape of the crash event ($*$) that is not allowed to arise once the worker process enters its work request-response handling loop. ■

Example B.2. The property that states that *'the work number is larger than 0'* is written as follows:

$$[\lhd\!\!-\langle\_,\_,\_,\_,\_\rangle]\,\mathsf{max}X.\left(\begin{array}{l}[?\langle\_,\langle\_,\langle\mathsf{chunk},\_,\mathit{ReqNum},\_\rangle\rangle\rangle,\mathit{ReqNum}\geq 1]\,[\,!\,\langle\_,\_,\_\rangle]X\\ \wedge\\ [?\langle\_,\langle\_,\langle\mathsf{chunk},\_,\mathit{ReqNum},\_\rangle\rangle\rangle,\mathit{ReqNum}<1]\,\mathsf{ff}\end{array}\right) \qquad (\varphi_{14})$$

Formula φ_{14} checks the work request sequence number to determine whether it carries a value larger than 0. The second pair of necessities that match the receive event shape and work request payload instantiates the variable *ReqNum* with the value of the work request sequence number. A violation of φ_{14} occurs when $\mathit{ReqNum}<1$, otherwise the formula unfolds after the third necessity $[\,!\,\langle\_,\_,\_\rangle]$ matches a send event. ■

Example B.3. The property stating that *'workers do not receive more requests than expected'* is specified as:

$$\begin{array}{l}[\lhd\!\!-\langle\_,\_,\_,\_,\_\rangle]\\ \mathsf{max}X.\left(\begin{array}{l}[?\langle\_,\langle\_,\langle\mathsf{chunk},\_,\mathit{ReqNum},\mathit{NumReqs}\rangle\rangle\rangle,\mathit{ReqNum}\leq\mathit{NumReqs}]\,[\,!\,\langle\_,\_,\_\rangle]X\\ \wedge\\ [?\langle\_,\langle\_,\langle\mathsf{chunk},\_,\mathit{ReqNum},\mathit{NumReqs}\rangle\rangle\rangle,\mathit{ReqNum}>\mathit{NumReqs}]\,\mathsf{ff}\end{array}\right)\end{array} \qquad (\varphi_{15})$$

Similar to example B.2, formula φ_{15} relies on the current work request sequence number issued by the master process *and* the total number of expected requests. The variable *NumReqs* becomes instantiated with the latter value when a receive trace event, together with its work request payload, matches the second necessity modality. Subsequently, *NumReqs* is compared against *ReqNum* to determine whether the work request sequence number has been exceeded. ■

Example B.4. The property stating that *'workers receive only their responses'* is specified thus:

$$[\lhd\!\!-\langle\_,\_,\_,\_,[\mathit{Id}_1,\_]\rangle]\,\mathsf{max}X.\left(\begin{array}{l}[?\langle\_,\langle\_,\langle\mathsf{chunk},\mathit{Id}_2,\_,\_\rangle\rangle\rangle,\mathit{Id}_1=\mathit{Id}_2]\,[\,!\,\langle\_,\_,\_\rangle]X\\ \wedge\\ [?\langle\_,\langle\_,\langle\mathsf{chunk},\mathit{Id}_2,\_,\_\rangle\rangle\rangle,\mathit{Id}_1\neq\mathit{Id}_2]\,\mathsf{ff}\end{array}\right) \qquad (\varphi_{16})$$

Formula φ_{16} compares the worker ID to detect whether a work request sent by the master was meant for another worker. The very first necessity, $\lhd\!\!-\langle\_,\_,\_,\_,[\mathit{Id}_1,\_]\rangle$, matches the process initialisation event pattern, including the shape of the argument list used to launch worker processes. Worker processes are initialised with two arguments, the first of which is the worker ID assigned by the master; φ_{16} stores this value in the variable Id_1. In the second pair of necessity modalities that match the receive event and the shape of the embedded work request payload, instantiate the variables Id_2. The Boolean constraint $\mathit{Id}_1\neq\mathit{Id}_2$ in the symbolic action of the violating conjunct of φ_{16} ensures that the formula is violated only when the worker does not match with the worker ID carried by the work request. ■

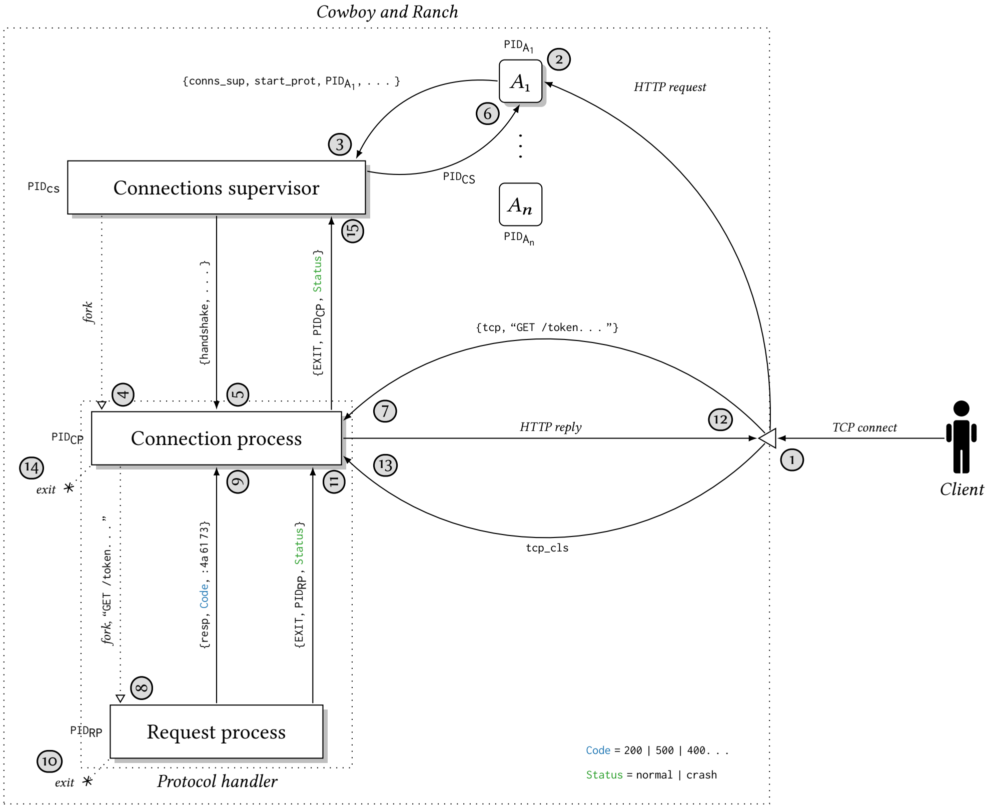


Figure B.1. *The Cowboy and Ranch communication protocol*

B.2 The Cowboy and Ranch Communication Protocol

Figure B.1 describes a fragment of the interaction protocol that Cowboy and Ranch use to service HTTP requests. In this protocol, *acceptors* wait on the socket for incoming client connections, step ①. When a connection is established on the server, the acceptor exchanges the newly-acquired transmission control protocol (TCP) socket information with the *connections supervisor*, as steps ② and ③ indicate. This instruction notifies the connections supervisor that a new client connection needs handling; in turn, the former forks a new *connection process* and delegates this task, steps ④ and ⑤. The acceptor is informed accordingly in step ⑥, where it waits anew for future connections. Henceforth, the connection process has complete ownership of and communicates *directly* with the client socket. Step ⑧ illustrates the point when the connection process forks the *request process*, specifying as argument the HTTP request data it acquires from the socket in step ⑦. Once the request process completes its execution, it issues a reply to its connection process and terminates, steps ⑨ and ⑩. This reply is comprised of the HTTP response code and respective payload that the connection process communicates to the client in step ⑫. A socket closed notification is sent by the Erlang TCP library, step ⑬, whereupon the connection process terminates in step ⑭. Messages {`EXIT`, *Pid*, *Status*} in steps ⑪ and ⑮ result from Erlang *process linking*, and are issued by the EVM when processes terminate [57]. The connection and request process pair is termed the *protocol handler*, where the interaction between the two happens in *lockstep*, *i.e.*, steps ⑧ to ⑬ are sequential.

B.3 Monitoring Cowboy and Ranch

Example B.5. Recall the formula φ_{RP} from section 4.6, stating that *'a request process does not issue HTTP responses with code 500, nor does it crash'*.

$$\mathsf{max}\,X.\left(\begin{array}{l}[\,\mathtt{!}\langle\_,\_,\{\mathtt{resp},\,\mathit{code},\,\ldots\,\}\rangle,\mathit{code}=\mathtt{200}]\,X\wedge\\ [\,\mathtt{!}\langle\_,\_,\{\mathtt{resp},\,\mathit{code},\,\ldots\,\}\rangle,\mathit{code}=\mathtt{500}]\,\mathsf{ff}\wedge\\ [\,\mathtt{*}\langle\_,\mathit{stat}\rangle,\mathit{stat}=\mathtt{crash}]\,\mathsf{ff}\end{array}\right)\tag{φ_{RP}}$$

Its corresponding synthesised monitor, $m_{\varphi_{\text{RP}}}$, consists of a recursion construct whose body is composed of the three sub-monitors m_{200}, m_{500}, and m_{crash} conjuncted in parallel. The monitor m_{200} handles the case when the HTTP response code is 200, unfolding the monitor via the recursion variable X if $\mathit{code}=\mathtt{200}$, or reaches the verdict yes otherwise. Monitor m_{500} flags a rejection verdict no when it analyses a response message containing the response code 500. Analogously, monitor m_{CRASH} flags no when an error event with the status `crash` is detected.

$$m_{\varphi_{\text{rp}}}=\mathsf{rec}\,X.(m_{200}\otimes m_{500}\otimes m_{\text{crash}})\tag{$m_{\varphi_{\text{RP}}}$}$$

$$m_{200}=\begin{cases}(\,\mathtt{!}\langle\_,\_,\{\mathtt{resp},\,\mathit{code},\,\ldots\,\}\rangle,\mathit{code}=\mathtt{200}).X+\\ (\,\mathtt{!}\langle\_,\_,\{\mathtt{resp},\,\mathit{code},\,\ldots\,\}\rangle,\mathit{code}\neq\mathtt{200}).\mathsf{yes}\end{cases}\tag{m_{200}}$$

$$m_{500}=\begin{cases}(\,\mathtt{!}\langle\_,\_,\{\mathtt{resp},\,\mathit{code},\,\ldots\,\}\rangle,\mathit{code}=\mathtt{500}).\mathsf{no}+\\ (\,\mathtt{!}\langle\_,\_,\{\mathtt{resp},\,\mathit{code},\,\ldots\,\}\rangle,\mathit{code}\neq\mathtt{500}).\mathsf{yes}\end{cases}\tag{m_{500}}$$

$$m_{\text{CRASH}}=\begin{cases}(\mathtt{*}\langle\_,\mathit{stat}\rangle,\mathit{stat}=\mathtt{crash}).\mathsf{no}+\\ (\mathtt{*}\langle\_,\mathit{stat}\rangle,\mathit{stat}\neq\mathtt{crash}).\mathsf{yes}\end{cases}\tag{m_{CRASH}}$$

Figure B.2 details how the trace '$\mathtt{!}\langle\mathtt{PID}_{\mathtt{RP}},\mathtt{PID}_{\mathtt{CP}},\{\mathtt{resp},\,\mathtt{500},\,\ldots\,\}\rangle\ldots$' exhibited by a Cowboy request process bearing the PID $\mathtt{PID}_{\mathtt{RP}}$ leads the monitor $m_{\varphi_{\text{RP}}}$ to a violation verdict. Before analysing events, monitor $m_{\varphi_{\text{RP}}}$ unfolds the recursion variable X of sub-monitor m_{200} by transitioning internally via $\textsc{mRec}$ in step ①. The resulting parallel composition of monitors is reduced by applying the rule $\textsc{mPar}$ twice. In sub-derivation $\textcircled{\scriptsize 2.1}$, $\textsc{mPar}$ reduces $\big((\,\mathtt{!}\langle\_,\_,\{\mathtt{resp},\,\mathit{code},\,\ldots\,\}\rangle,\mathit{code}=\mathtt{200}).m_{\text{rp}}+(\,\mathtt{!}\langle\_,\_,\{\mathtt{resp},\,\mathit{code},\,\ldots\,\}\rangle,\mathit{code}\neq\mathtt{200}).\mathsf{yes}\big)\otimes m_{500}$ to the monitor $\mathsf{yes}\otimes\mathsf{no}$, using the respective sub-derivations $\textcircled{\scriptsize 2.1.1}$ and $\textcircled{\scriptsize 2.1.2}$ obtained from $\textsc{mChs}_{\text{R}}$ and $\textsc{mChs}_{\text{L}}$. For example, $\textsc{mChs}_{\text{L}}$ applied to m_{500} reduces the monitor to no when the trace event $\mathtt{!}\langle\mathtt{PID}_{\mathtt{RP}},\mathtt{PID}_{\mathtt{CP}},\{\mathtt{resp},\,\mathtt{500},\,\ldots\,\}\rangle$ is analysed. This follows from rule $\textsc{mAct}$, where $\mathsf{match}\big(\,\mathtt{!}\langle\mathtt{PID}_{\mathtt{RP}},\mathtt{PID}_{\mathtt{CP}},\{\mathtt{resp},\,\mathtt{500},\,\ldots\,\}\rangle,\,\mathtt{!}\langle\_,\_,\{\mathtt{resp},\,\mathit{code},\,\ldots\,\}\rangle\big)$ yields the substitution $[^{\mathtt{500}}/_{\mathit{code}}]$, and the instantiated Boolean constraint, $(\mathit{code}=\mathtt{500})[^{\mathtt{500}}/_{\mathit{code}}]$, is satisfied. The application of $\textsc{mChs}_{\text{R}}$ to monitor m_{CRASH} in sub-derivation $\textcircled{\scriptsize 2.2}$ follows a similar argument. Finally, sub-derivations $\textcircled{\scriptsize 2.1}$ and $\textcircled{\scriptsize 2.2}$ are used as premises to $\textsc{mPar}$, yielding $\mathsf{yes}\otimes\mathsf{no}\otimes\mathsf{yes}$ in ②. The latter monitor is reduced via $\textsc{mConY}_{\text{R}}$and $\textsc{mConY}_{\text{L}}$to reach the violating verdict no.

The remaining examples briefly overview other properties that were used when evaluating Cowboy. Readers should consult the depiction of the protocol of figure B.1 while reading these examples.

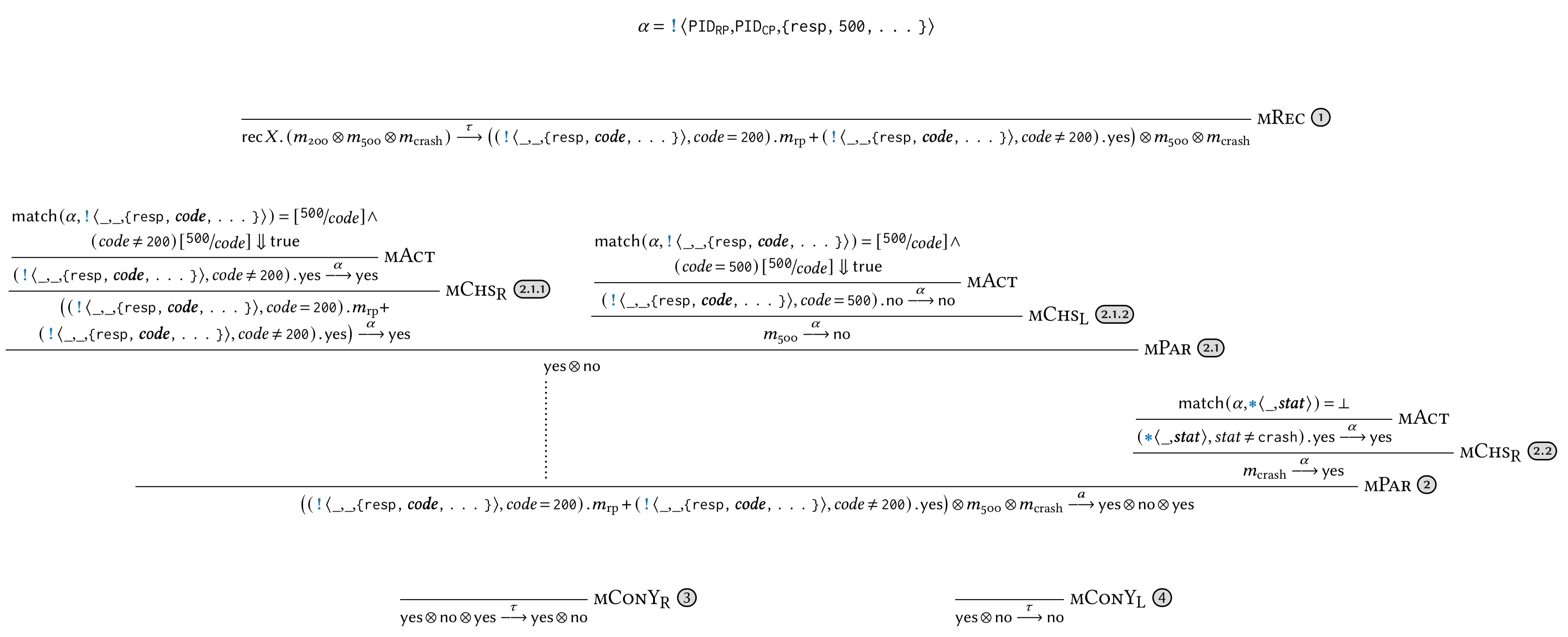

$\alpha = \,!\langle \mathtt{PID_{RP}},\mathtt{PID_{CP}},\{\mathtt{resp}, \mathtt{500}, \ldots\}\rangle$
$\mathrm{rec}\,X.(m_{200} \otimes m_{500} \otimes m_{\mathrm{crash}}) \xrightarrow{\tau} \big((\,!\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle, code = \mathtt{200}).m_{\mathrm{rp}} + (\,!\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle, code \neq \mathtt{200}).\mathsf{yes}\big) \otimes m_{500} \otimes m_{\mathrm{crash}}$
mRec ①
$\mathrm{match}(\alpha, !\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle) = [^{500}/_{code}] \wedge$
$(code \neq \mathtt{200})[^{500}/_{code}] \Downarrow \mathsf{true}$
mAct
$(\,!\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle, code \neq \mathtt{200}).\mathsf{yes} \xrightarrow{\alpha} \mathsf{yes}$
mChsR 2.1.1
$\big((\,!\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle, code = \mathtt{200}).m_{\mathrm{rp}} +$
$(\,!\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle, code \neq \mathtt{200}).\mathsf{yes}\big) \xrightarrow{\alpha} \mathsf{yes}$
$\mathrm{match}(\alpha, !\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle) = [^{500}/_{code}] \wedge$
$(code = \mathtt{500})[^{500}/_{code}] \Downarrow \mathsf{true}$
mAct
$(\,!\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle, code = \mathtt{500}).\mathsf{no} \xrightarrow{\alpha} \mathsf{no}$
mChsL 2.1.2
$m_{500} \xrightarrow{\alpha} \mathsf{no}$
mPar 2.1
yes ⊗ no
$\mathrm{match}(\alpha, *\langle\_,\boldsymbol{stat}\rangle) = \bot$
mAct
$(*\langle\_,\boldsymbol{stat}\rangle, stat \neq \mathsf{crash}).\mathsf{yes} \xrightarrow{\alpha} \mathsf{yes}$
mChsR 2.2
$m_{\mathrm{crash}} \xrightarrow{\alpha} \mathsf{yes}$
mPar ②
$\big((\,!\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle, code = \mathtt{200}).m_{\mathrm{rp}} + (\,!\langle\_,\_,\{\mathtt{resp}, \boldsymbol{code}, \ldots\}\rangle, code \neq \mathtt{200}).\mathsf{yes}\big) \otimes m_{500} \otimes m_{\mathrm{crash}} \xrightarrow{a} \mathsf{yes} \otimes \mathsf{no} \otimes \mathsf{yes}$
mConYR ③
$\mathsf{yes} \otimes \mathsf{no} \otimes \mathsf{yes} \xrightarrow{\tau} \mathsf{yes} \otimes \mathsf{no}$
mConYL ④
$\mathsf{yes} \otimes \mathsf{no} \xrightarrow{\tau} \mathsf{no}$

Figure B.2. *Monitor $m_{\varphi_{RP}}$ justifies how the verdict no is reached along '$!\langle \mathtt{PID_{RP}},\mathtt{PID_{CP}},\{\mathtt{resp}, \mathtt{500}, \ldots\}\rangle \ldots$'*

Example B.6. Formula φ_{ACC} concerns Ranch *acceptor* components that listen to incoming TCP requests.

$$\max X.\left(\begin{array}{l} [\,!\,\langle acc_1, csup_1, \{\texttt{conns\_sup, start\_prot, \_, \_}\}\rangle] \\ \left(\begin{array}{l} [?\langle acc_2, csup_2\rangle, acc_1 = acc_2 \land csup_1 = csup_2]X \land \\ [?\langle acc_2, csup_2\rangle, acc_1 = acc_2 \land csup_1 \neq csup_2]\,\text{ff} \end{array}\right)\end{array}\right) \qquad (\varphi_{\text{ACC}})$$

It states that when a new connection is established, the acceptor, denoted by the binder acc_1, issues the request {conns_sup, . . . } to the connections supervisor process, $csup_1$. The property ensures that the *same* process acknowledges back to the sending acceptor, *i.e.*, $acc_1 = acc_2 \land csup_1 = csup_2$. ■

Example B.7. Formula φ_{CP} specifies the interaction protocol that a Cowboy connection process should follow when servicing a client HTTP request.

$$\max X.\left(\begin{array}{l} [?\langle cprc_1, \{\texttt{handshake, . . . }\}\rangle]\,[?\langle cprc_2, \{\texttt{tcp, } req_1\}\rangle, cprc_1 = cprc_2] \hookleftarrow \\ [\rightarrowtriangle\langle cprc_3, rprc_1, \texttt{req\_prc}, \texttt{start}, req_2\rangle, cprc_2 = cprc_3 \land req_1 = req_2] \hookleftarrow \\ \left(\begin{array}{l} [?\langle cprc_4, \{\texttt{resp, 200, . . . }\}\rangle, cprc_3 = cprc_4] \hookleftarrow \\ \left(\begin{array}{l} [?\langle cprc_5, \{\texttt{EXIT, } rprc_2\texttt{, normal}\}\rangle, cprc_4 = cprc_5 \land rprc_1 = rprc_2] \hookleftarrow \\ [?\langle cprc_6, \texttt{tcp\_cls}\rangle, cprc_5 = cprc_6]X \land \\ [?\langle cprc_5, \{\texttt{EXIT, } rprc_2\texttt{, crash}\}\rangle, cprc_4 = cprc_5 \land rprc_1 = rprc_2]\,\text{ff} \end{array}\right) \land \\ [?\langle cprc_4, \{\texttt{resp, 500, . . . }\}\rangle, cprc_3 = cprc_4]\,\text{ff} \end{array}\right)\end{array}\right) \qquad (\varphi_{\text{CP}})$$

Connection processes interact with the connections supervisor through a handshake before reading the HTTP request directly from the TCP socket (steps ⑤ and ⑦ in figure B.1). This interaction is given by $[?\langle cprc_1, \{\texttt{handshake, . . . }\}\rangle]\ [?\langle cprc_2, \{\texttt{tcp, } req_1\}\rangle, cprc_1 = cprc_2]$ in formula φ_{CP}. The binder $cprc_1$ in the first necessity becomes instantiated with the PID of the connection process, whereas req_1 in the second necessity becomes instantiated with the HTTP request data read from the socket. The third necessity uses the *fork* action pattern $\rightarrowtriangle\langle cprc_3, rprc_1, \texttt{req\_prc}, \texttt{start}, req_2\rangle$. It describes the protocol step where the connection process under analysis forks a request process via the function start in module req_prc, where the argument specified must be the request data acquired from the socket. This constraint is imposed by $req_1 = req_2$. If the fork trace event exhibited by the connection process matches the aforementioned fork action pattern, the binder $rprc_1$ is instantiated with the PID of the newly-forked request process (step ⑧ in figure B.1). The necessity $[?\langle cprc_4, \{\texttt{resp, 200, . . . }\}\rangle, cprc_3 = cprc_4]$ dictates that the connection, $cprc_4$, process receives a HTTP 200 response message from the request process. A violation of φ_{CP} occurs when HTTP 500 is contained in the response message instead, $[?\langle cprc_4, \{\texttt{resp, 500, . . . }\}\rangle, cprc_3 = cprc_4]\,\text{ff}$. We remark that the latter two necessities describing the receive actions w.r.t. HTTP response codes are the counterparts to the send messages of formula φ_{RP}. The final steps of the protocol requires it to wait for the request process $rprc_2$ to terminate its execution normally, {EXIT, $rprc_2$, normal} and afterwards, wait for the TCP socket to close, receiving the message tcp_cls. The formula is however violated when the connection process receives the message {EXIT, $rprc_2$, crash}, informing it that the request process crashed. Note that formula φ_{CP} ensures that *all* the sub-formulae describe the behaviour of the *same* connection process (see figure B.1) by ensuring that $cprc_1 = cprc_2 = cprc_3 = cprc_4 = cprc_5 = cprc_6$. ■

C Auxiliary Data Plots for Benchmarks

C.1 Moderate Loads

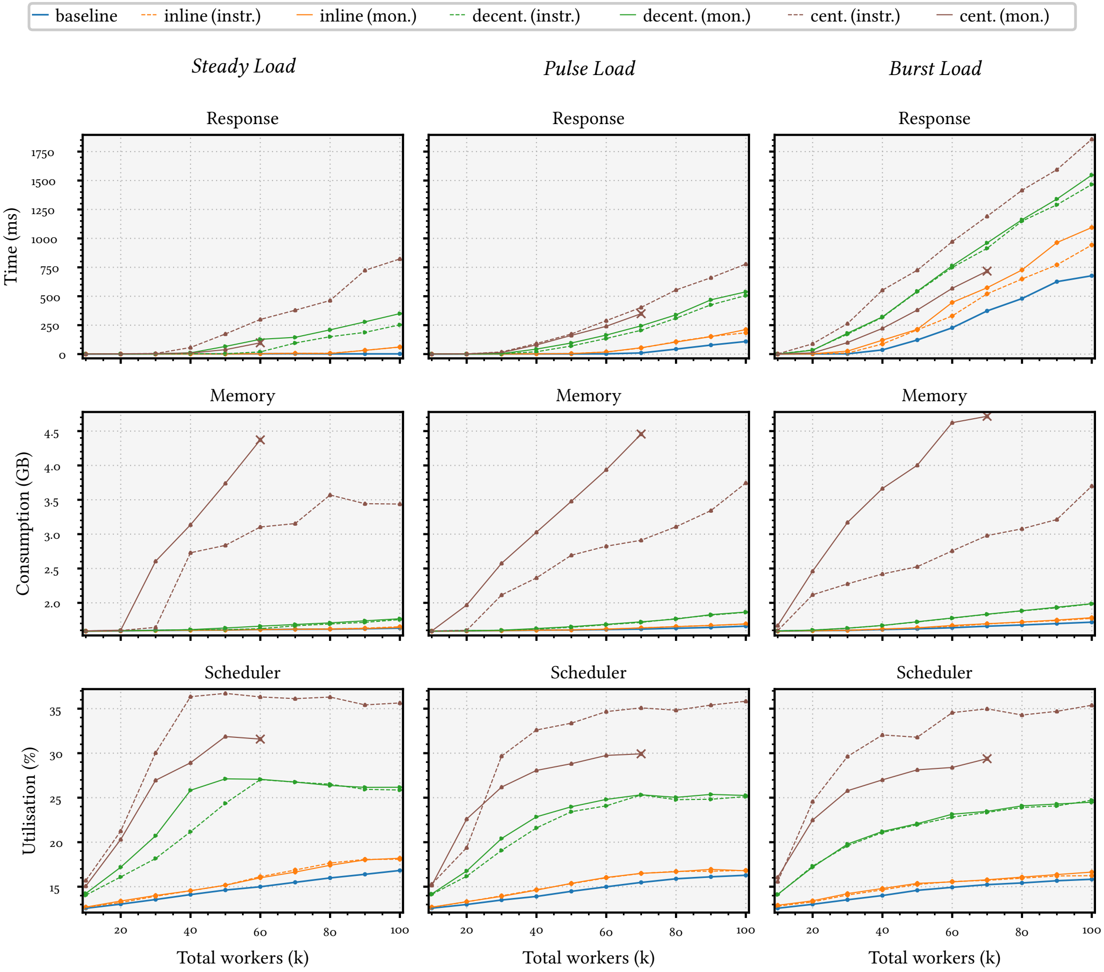


Figure C.1. *Gap in instrumentation and monitoring overhead on the system under moderate load benchmarks* (100k *workers)*

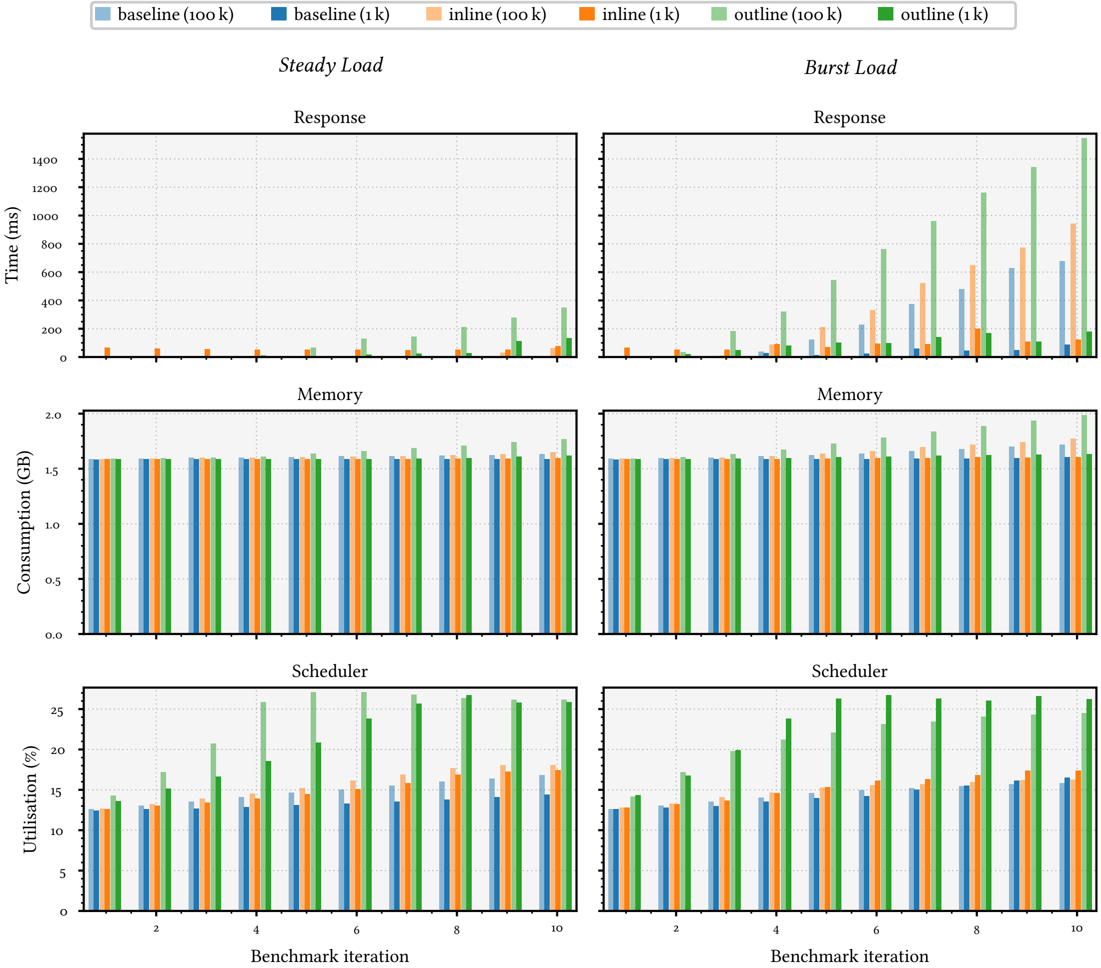


Figure C.2. *Gap in decentralised monitoring overhead on the system under moderate load benchmarks (*100k *vs.* 1k *workers)*

C.2 High Loads

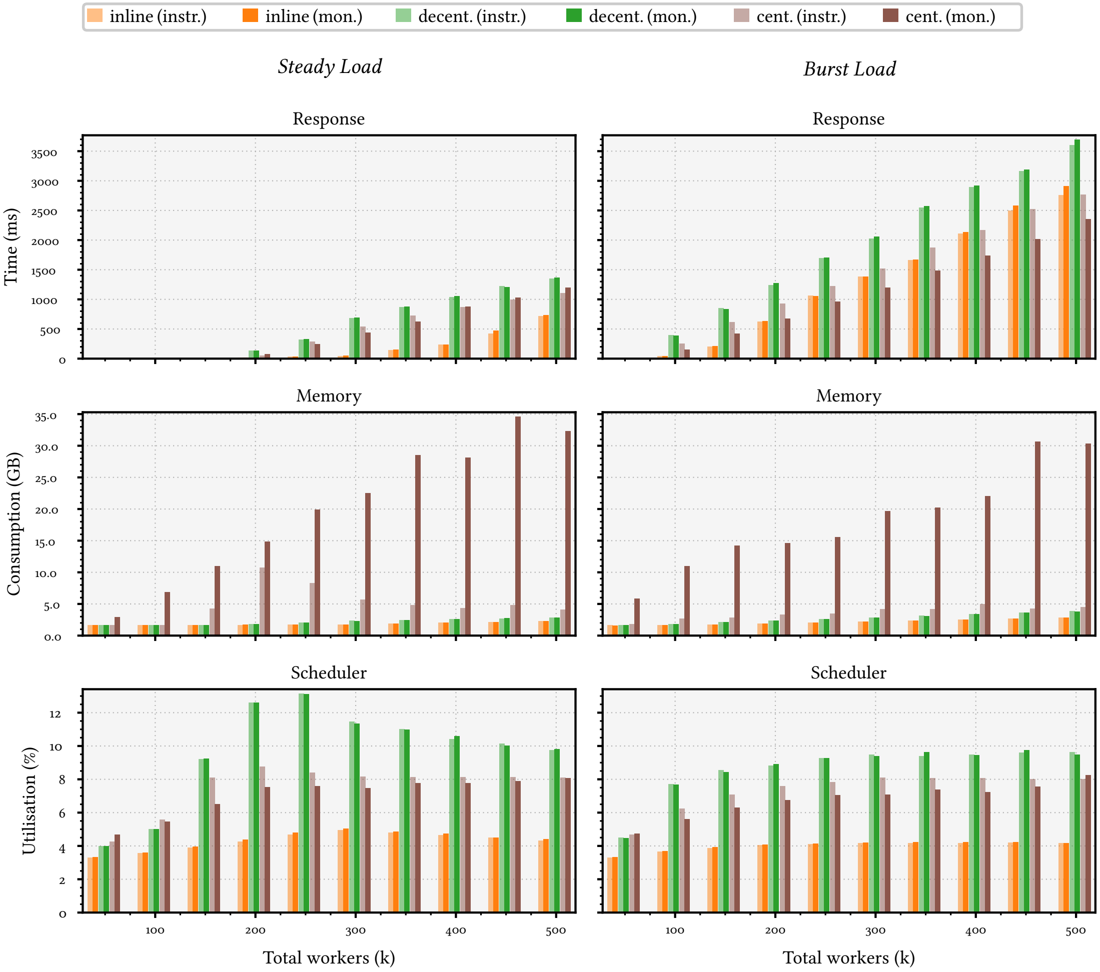


Figure C.3. *Gap in instrumentation and monitoring overhead on the system under high load benchmarks (500k workers)*

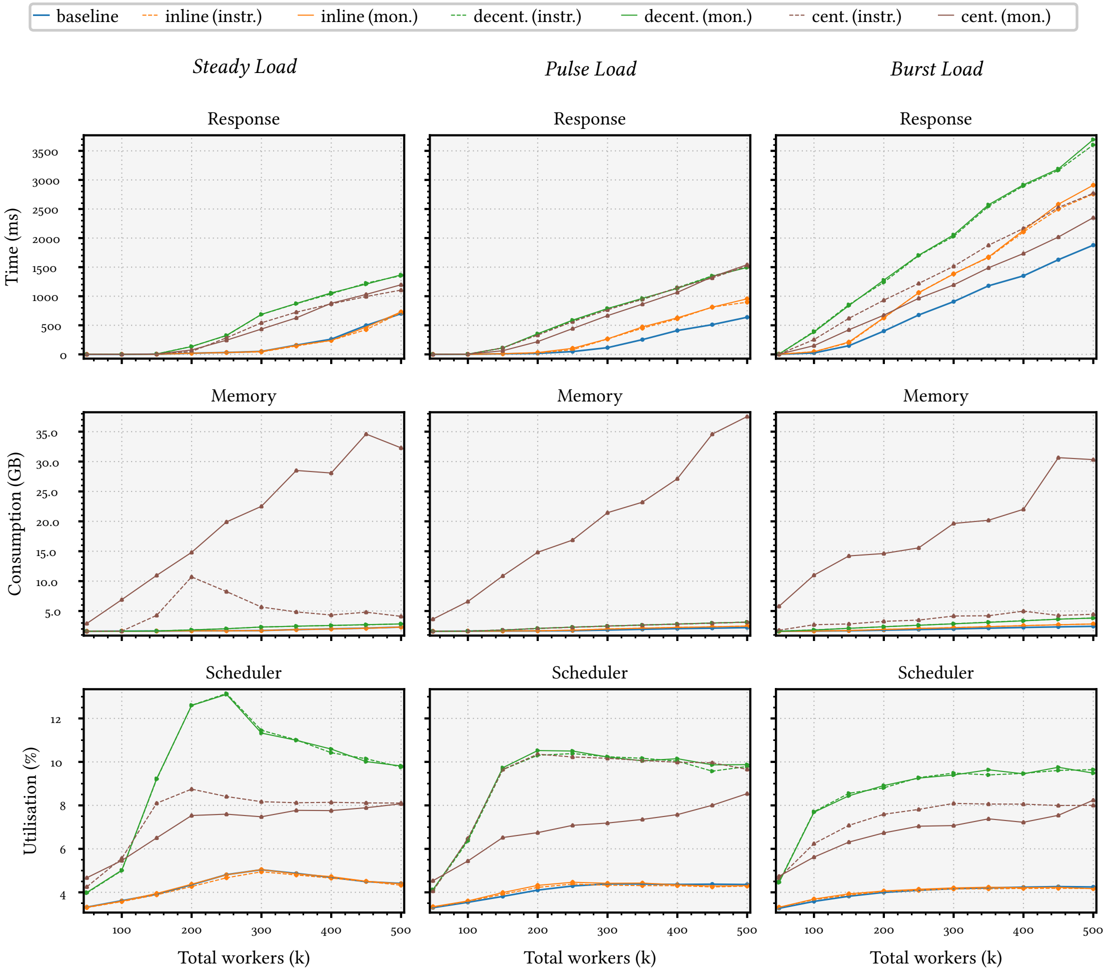


Figure C.4. *Gap in instrumentation and monitoring overhead on the system under high load benchmarks (500k workers)*

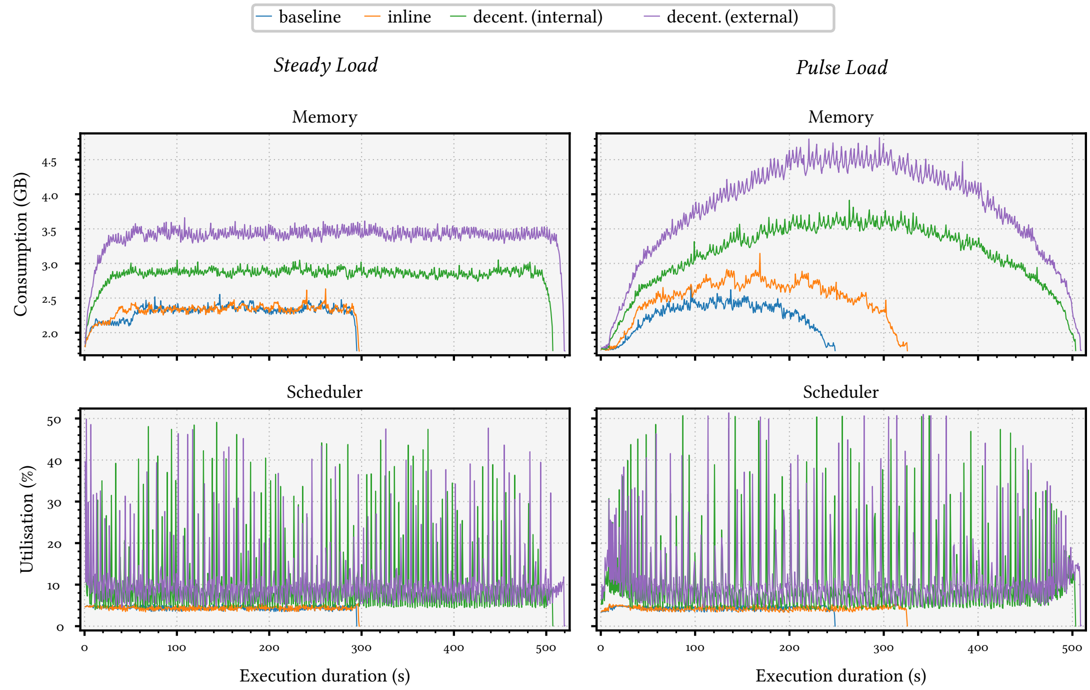


Figure C.5. *Resource consumption for decentralised monitoring under high load benchmarks (500k workers)*

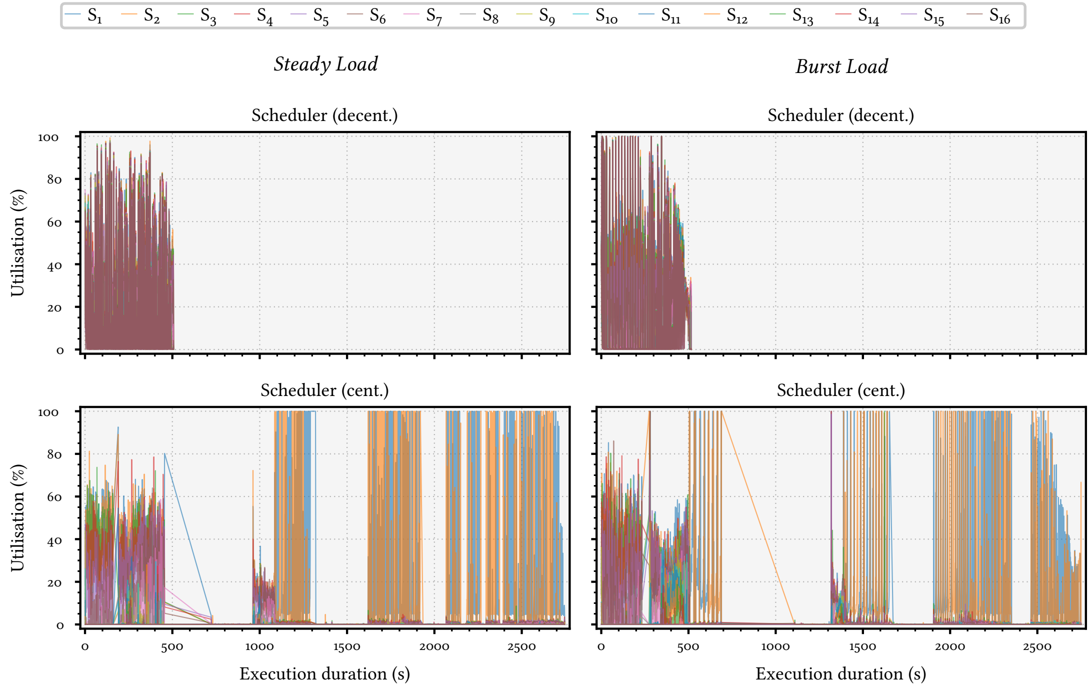


Figure C.6. *Load on scheduler threads for complete experiment runs under high load benchmarks (500k workers)*

D A Summary of the State of the Art

D.1 Concurrent Monitoring

There are a number of works [113, 21, 219, 52, 51, 71, 210, 39] that address RV in a local concurrent setting; others [34, 97] use the term decentralised to refer to synchronous monitoring. A comparison of their various characteristics is provided in table D.1.

D.2 Distributed Monitoring

Previous work for decentralised local monitoring [34] was extended by Colombo and Falcone [67] to a distributed setting while retaining a number of core characteristics such as the decentralised approach, and in particular, the availability of a common clock. Correctness properties over the global system state are specified via LTL_3; these are synthesised into decentralised component sub-monitors that are organised across nodes on a network. The monitor choreography is arranged in the form of a tree, reflecting the compositional structure of formulae, such that each child feeds intermediate results to its

	Decentralised	*Global state*	*Asynchronous*	*Shared memory*	*Message passing*	*Total ordering*	*Dynamic set-up*
Attard and Francalanza [21]	·	✓	✓	·	✓	·	·
Duncan Paul Attard and Francalanza [219]	✓	*	✓	·	✓	·	·
Aceto et al. [13]	✓	·	·	·	✓	✓	✓
Bauer and Falcone [34]	✓	✓	·	·	✓	✓	·
Berkovich et al. [39]	·	✓	·	✓	·	✓	·
Cassar and Francalanza [51]	·	✓	✓	·	✓	·	·
Cassar and Francalanza [52]	·	✓	✓	·	✓	·	·
Colombo et al. [71]	·	·	✓	·	✓	·	✓
Falcone et al. [97]	·	✓	·	·	✓	✓	·
Francalanza and Seychell [113]	·	✓	✓	·	✓	·	·
Sen et al. [210]	✓	✓	✓	✓	·	·	·

Table D.1. *State-of-the-art on concurrent monitoring classified by characteristics (* denotes both)*

parent. System components operate in synchronous rounds, meaning that a unique global trace can be reconstructed by combining multiple sub-traces collected locally by monitors at each component. Monitor judgements are obtained by *rewriting formulae* in a compositional fashion: sub-constituents of a formula are evaluated on events from the trace and progressively simplified by monitors until the formula eventually equates to $\top$ or $\bot$, at which point, the monitoring stops. The authors give a proof of correctness of the monitor synthesis and show that a decentralised monitoring set-up induces substantially lower communication overheads when compared to centralised or migrating monitors. While the monitoring algorithm does not make any assumptions on the delay of messages, it does assume a reliable connection between system components and monitors and also requires the number of system components to remain fixed at runtime.

Basin et al. [31] is one of the few works that consider the problem of monitoring distributed systems with failing components and network links. Despite the absence of a global clock, the monitoring algorithm is based on the *timed asynchronous* model for distributed systems [75] that assumes the availability of highly-synchronised physical clocks across nodes. Correctness properties are specified over the global system state using metric temporal logic (MTL), a logic that allows the specification of real-time properties. Monitors synthesised from MTL formulae are arranged in a choreographed fashion in the form of a directed acyclic graph, following the compositional structure of formulae. A monitor rooted at the graph handles the top-level formula being monitored, while other sub-monitors are responsible for its sub-formulae constituents. During execution, sub-monitors propagate messages to their parents to inform them about verdicts that have been reached for their respective sub-formulae under analysis at that point in time. This enables the root monitor to formulate and eventually report its verdict for the entire formula. Monitors attached to system components collect trace events locally; these are timestamped by the system before being communicated to monitors, thereby enabling the latter to compute the precise delay between events and check whether real-time constraints are met. In addition, events are equipped with a locally-unique sequence number that allows monitors to detect gaps that may arise between subsequent trace events, due to lost or delayed messages and process crashes. We observe that events are totally ordered locally, and even though these may be delivered out-of-order due to the asynchronous communication between monitors, a global ordering of events may still be possible by virtue of the local timestamps. This is in contrast to the *time-free* model [107], where events in a distributed system can only be partially ordered using logical clocks. The authors argue that while the physical time drift that occurs between clocks on different locations might impinge on certain monitoring verdicts, this is often acceptably small, and relying on timestamps from local clocks for monitoring purposes is good enough in practical scenarios. They also show soundness for their algorithm in the presence of failures, and completeness when no failure is assumed, *i.e.,* a monitor eventually reports a verdict for the given specification.

Bonakdarpour et al. [45] address failure within monitors themselves, specifically in the case of fail-stop. They propose a framework for distributed fault-tolerant RV using a multi-valued temporal logic that redefines the semantics of LTL, where the truth values represent a degree of certainty that a formula has been satisfied or violated. Correctness properties are synthesised as choreographed automaton monitors that interact asynchronously using the wait-free read/write shared memory model, which is known to be equivalent to a message-passing model where less than half of the processes can fail-stop [83]. Monitors have a partial view of the global system state and communicate with each other for a fixed number of rounds until a verdict about the global system state is reached. Verdicts are given from a set of possible truth values associated with the property being monitored. The authors show that verdicts collectively provided by monitors can be mapped to one that is computed by a centralised monitor having a full view of the SuS.

RV of shared state concurrency programs has also been studied by Sen et al. [210], where decentralised monitors are attached to different threads to collect and process trace events locally. In an earlier work by the same authors [208], this investigation is conducted in a distributed setting using decentralised monitors that are weaved into the SuS. Correctness properties are expressed in terms of PtDTL, a variant of past-time LTL that is equipped with epistemic operators, allowing formulae specified on the local state of system components to internally refer to the state of other remote components. In this sense, a property about a particular component is interpreted over a projection of the global system state. A PtDTL formula is synthesised into a monitor choreography reflecting its structure; these are attached to different system components in order to collect trace events locally to minimise communication overheads. Monitors in the choreography interact via asynchronous send and receive operations and exchange partial information about the system state that is relevant to the property under consideration. This information takes the form of a *knowledge vector*, a data structure similar to a vector clock [172, 105], that summarises the local state of the system components related to the monitored PtDTL formula. Monitors exchange local copies of their knowledge vector by attaching them to outgoing messages sent by *system* components and update their local knowledge vector state in turn with the most recent information received. A formula is evaluated in a step-wise fashion by cooperating monitors by consulting their local knowledge vector whenever it gets updated until a verdict is eventually reached. The authors focus on the efficiency of the monitoring set-up and argue that the monitoring information piggybacked on messages already being passed between system components does not incur additional overheads. However, this renders the monitoring algorithm incomplete, since monitors only gain knowledge of the system through the existing communication among its components, and in cases where these rarely communicate, the little information exchanged may lead to missed detections. The set-up is also not amenable to scenarios where node or link failure is present, due the to dependency monitors have on the architecture of the SuS.

Scheffel and Schmitz [203] argue that the two-valued semantics of PtDTL is insufficient to enable monitors to distinguish between verdicts relating to safety or fulfilment properties. They adopt an approach similar to Sen et al. [208], but allow correctness properties to be expressed in DTL—an extended version of PtDTL equipped with the three-valued semantics of LTL$_3$. As in Sen et al. [208], correctness properties specified over the local state of system components can, in turn, include sub-properties that reference the state of other remote components through epistemic operators. Monitors disseminate partial information using the notion of knowledge vectors of Sen et al. [208], employing the same mechanism that piggybacks monitoring information on asynchronous messages exchanged between system components, making their algorithm efficient but incomplete.

Minimising communication and memory overhead is also the focus of Mostafa and Bonakdarpour [180]. In this setting, the SuS consists of distributed asynchronous processes that communicate together via message-passing primitives over reliable channels. Correctness specifications given in terms of LTL$_3$ are specified over the global system state: these are synthesised into automaton monitors and composed with system processes. The monitor algorithm does not assume a common global clock and

	Decentralised	*Global state*	*Global clock*	*Asynchronous*	*Shared memory*	*Message passing*	*Total ordering*	*Message loss*	*Failure*	*Dynamic set-up*
Basin et al. [31]	✓	✓	·	✓	·	✓	✓	✓	✓	·
Bonakdarpour et al. [45]	✓	✓	·	✓	✓	·	·	·	✓	·
Colombo and Falcone [67]	✓	✓	✓	·	·	✓	✓	·	·	·
Graf et al. [122]	✓	✓	·	*	·	✓	*	·	·	·
Mostafa and Bonakdarpour [180]	✓	✓	·	✓	·	✓	·	·	·	·
Scheffel and Schmitz [203]	✓	·	·	✓	·	✓	·	·	·	·
Sen et al. [208]	✓	·	·	✓	·	✓	·	·	·	·

Table D.2. *State of the art on distributed monitoring classified by characteristics (* denotes both)*

partially orders the trace events collected locally by monitors using vector clocks. To contend with the non-determinism that arises due to this partial ordering, each automaton in the monitor maintains a number of possible verdicts that are continually updated when new local state information is exchanged between monitors. This spares monitors from having to consider system states that are not relevant to the property under consideration. The algorithm progresses by merging similar monitor states to keep the number of possible verdicts manageable throughout the monitoring process until the final verdict is eventually issued.

Graf et al. [122] adopt a hybrid verification approach that employs model checking to pre-calculate the states of a program that enable violations to be reported by a monitor acting alone. Invariants are specified via knowledge properties [93] over the global system state; these are synthesised into asynchronous decentralised monitors that communicate with each other to obtain additional information about the local state of remote components. When the information computed *a priori* during the model checking phase determines that monitors cannot reach a verdict in isolation, synchronisation ensues to enable them to cooperatively conclude whether the invariant is violated. In this manner, monitors may operate independently and engage in synchronous communication only when necessary, contributing to lower overheads. The pre-calculation step assumes that components within the system are reliable and that their number remains fixed throughout the entire execution.

A summary of the discussed works is given in table D.2. The various monitoring approaches use decentralised monitors to collect and process trace events locally at each component; this tends to better address the communication overhead that arises in centralised approaches, and at the same time, eliminates SPOFs. While works such as Sen et al. [210] and Mostafa and Bonakdarpour [180] do not explicitly focus on failure, their decentralised set-ups may still benefit from a modicum of fault containment when correctness properties target only specific components.

Acronyms

maxHML$^{\mathbf{D}}$ greatest fixed point fragment of μHML with data.

minHML$^{\mathbf{D}}$ least fixed point fragment of μHML with data.

μHML Hennessy-Milner logic with recursion.

μHML$^{\mathbf{D}}$ μHMLwith data.

AOP aspect-oriented programming.

API application programming interface.

APM application performance monitoring.

AST abstract syntax tree.

BEAM Bogdan's Erlang Abstract Machine.

BIF built-in function.

CCS calculus of communicating systems.

CPU central processing unit.

CRV competition on runtime verification.

CTL computation tree logic.

CV coefficient of variation.

DAG directed acyclic graph.

DB database.

DTL distributed temporal logic.

EVM Erlang virtual machine.

FIFO first in first out.

HTTP hypertext transfer protocol.

IO input/output.

IP internet protocol.

JVM Java virtual machine.

LTL linear temporal logic.

LTS labelled transition system.

MPI message passing interface.

MTL metric temporal logic.

OOP object oriented programming.

OS operating system.

OTP open telecom platform.

OTS off-the-shelf.

PD process dictionary.

PID process identifier.

PtDTL past-time distributed temporal logic.

PTS parametric trace slicing.

RE regular expression.

REST representational state transfer.

RV runtime verification.

SMP symmetric multiprocessing.

SPOF single point of failure.

SuS system under scrutiny.

TCP transmission control protocol.

UUID universally unique identifier.

Bibliography

[1] Luca Aceto and Anna Ingólfsdóttir. Testing Hennessy-Milner Logic with Recursion. In *FoSSaCS*, volume 1578 of *LNCS*, pages 41–55, 1999.

[2] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiří Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, 2007.

[3] Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. Monitoring for silent actions. In *FSTTCS*, volume 93 of *LIPIcs*, pages 7:1–7:14, 2017.

[4] Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. A Framework for Parameterized Monitorability. In *FoSSaCS*, volume 10803 of *LNCS*, pages 203–220, 2018.

[5] Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. On Runtime Enforcement via Suppressions. In *CONCUR*, volume 118 of *LIPIcs*, pages 34:1–34:17, 2018.

[6] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in Monitorability: From Branching to Linear Time and Back Again. *PACMPL*, 3: 52:1–52:29, 2019.

[7] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Sævar Örn Kjartansson. Determinizing Monitors for HML with Recursion. *JLAMP*, 111:100515, 2020.

[8] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An Operational Guide to Monitorability with Applications to Regular Properties. *Softw. Syst. Model.*, 20:335–361, 2021.

[9] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. The Best a Monitor Can Do. In *CSL*, volume 183 of *LIPIcs*, pages 7:1–7:23, 2021.

[10] Luca Aceto, **Duncan Paul Attard**, Adrian Francalanza, and Anna Ingólfsdóttir. On Benchmarking for Concurrent Runtime Verification. In *FASE*, volume 12649 of *LNCS*, pages 3–23, 2021.

[11] Luca Aceto, **Duncan Paul Attard**, Adrian Francalanza, and Anna Ingólfsdóttir. A Choreographed Outline Instrumentation Algorithm for Asynchronous Components. Technical report, Reykjavik University, IS, 2021.

[12] Luca Aceto, Antonis Achilleos, Elli Anastasiadi, and Adrian Francalanza. Monitoring Hyperproperties with Circuits. In *FORTE*, volume 13273 of *LNCS*, pages 1–10, 2022.

[13] Luca Aceto, Antonis Achilleos, **Duncan Paul Attard**, Léo Exibard, Adrian Francalanza, and Anna Ingólfsdóttir. A Monitoring Tool for Linear-Time μHML. In *COORDINATION*, volume 13271 of *LNCS*, pages 200–219, 2022.

[14] Luca Aceto, Antonis Achilleos, **Duncan Paul Attard**, Léo Exibard, Adrian Francalanza, and Anna Ingólfsdóttir. A Monitoring Tool for Linear-Time μHML. *Sci. Comput. Program.*, 232:103031, 2024.

[15] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Foundation for Actor Computation. *JFP*, 7:1–72, 1997.

[16] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA*, pages 345–364, 2005.

[17] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Inf. Process. Lett.*, 21:181–185, 1985.

[18] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Spring Joint Computing Conference*, volume 30 of *AFIPS Conference Proceedings*, pages 483–485, 1967.

[19] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[20] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. A Scalability Benchmark Suite for Erlang/OTP. In *Erlang Workshop*, pages 33–42, 2012.

[21] Duncan Paul Attard and Adrian Francalanza. A Monitoring Tool for a Branching-Time Logic. In *RV*, volume 10012 of *LNCS*, pages 473–481, 2016.

[22] Ralph-Johan Back. Invariant Based Programming: Basic Approach and Teaching Experiences. *Formal Aspects Comput.*, 21:227–244, 2009.

[23] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule Systems for Run-time Monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20:675–706, 2010.

[24] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *FM*, volume 7436 of *LNCS*, pages 68–84, 2012.

[25] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to Runtime Verification. In *Lectures on Runtime Verification*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018.

[26] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First International Competition on Runtime Verification: Rules, Benchmarks, Tools, and Final Results of CRV 2014. *STTT*, 21:31–70, 2019.

[27] Ezio Bartocci, Yliès Falcone, and Giles Reger. International Competition on Runtime Verification (CRV). In *TACAS*, volume 11429 of *LNCS*, pages 41–49, 2019.

[28] Basho. Bench, 2017. URL `https://github.com/basho/basho_bench`.

[29] Basho. Riak, 2022. URL `https://github.com/basho/riak`.

[30] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring Metric First-Order Temporal Properties. *J. ACM*, 62:15:1–15:45, 2015.

[31] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. Failure-Aware Runtime Verification of Distributed Systems. In *FSTTCS*, volume 45 of *LIPIcs*, pages 590–603, 2015.

[32] David A. Basin, Germano Caronni, Sarah Ereth, Matús Harvan, Felix Klaedtke, and Heiko Mantel. Scalable Offline Monitoring of Temporal Specifications. *FMSD*, 49:75–108, 2016.

[33] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. Runtime Verification of Temporal Properties over Out-of-Order Data Streams. In *CAV*, volume 10426 of *LNCS*, pages 356–376, 2017.

[34] Andreas Bauer and Yliès Falcone. Decentralised LTL Monitoring. *FMSD*, 48:46–93, 2016.

[35] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL Semantics for Runtime Verification. *J. Log. Comput.*, 20:651–674, 2010.

[36] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20:14:1–14:64, 2011.

[37] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. The Ins and Outs of First-Order Runtime Verification. *FMSD*, 46:286–316, 2015.

[38] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley, 2002.

[39] Shay Berkovich, Borzoo Bonakdarpour, and Sebastian Fischmeister. Runtime Verification with Minimal Intrusion through Parallelism. *FMSD*, 46:317–348, 2015.

[40] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[41] Sebastian Blessing, Kiko Fernandez-Reyes, Albert Mingkun Yang, Sophia Drossopoulou, and Tobias Wrigstad. Run, Actor, Run: Towards Cross-Actor Language Benchmarking. In *AGERE!@SPLASH*, pages 41–50, 2019.

[42] Eric Bodden. The Design and Implementation of Formal Monitoring Techniques. In *OOPSLA Companion*, pages 939–940, 2007.

[43] Eric Bodden, Laurie J. Hendren, Patrick Lam, Ondrej Lhoták, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. *J. Log. Comput.*, 20:707–723, 2010.

[44] Borzoo Bonakdarpour and Bernd Finkbeiner. The Complexity of Monitoring Hyperproperties. In *CSF*, pages 162–174, 2018.

[45] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, David A. Rosenblueth, and Corentin Travers. Decentralized Asynchronous Crash-Resilient Runtime Verification. In *CONCUR*, volume 59 of *LIPIcs*, pages 16:1–16:15, 2016.

[46] Werner Buchholz. A Synthetic Job for Measuring System Performance. *IBM Syst. J.*, 8:309–318, 1969.

[47] Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. On the Monitorability of Session Types, in Theory and Practice. In *ECOOP*, volume 194 of *LIPIcs*, pages 20:1–20:30, 2021.

[48] David R. Butenhof. *Programming with POSIX threads.* Addison-Wesley, 1997.

[49] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing: Principles and Paradigms.* Wiley-Blackwell, 2011.

[50] Bryan Cantrill. Hidden in Plain Sight. *ACM Queue*, 4:26–36, 2006.

[51] Ian Cassar and Adrian Francalanza. On Synchronous and Asynchronous Monitor Instrumentation for Actor-based Systems. In *FOCLASA*, volume 175 of *EPTCS*, pages 54–68, 2014.

[52] Ian Cassar and Adrian Francalanza. On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In *IFM*, volume 9681 of *LNCS*, pages 176–192, 2016.

[53] Ian Cassar, Adrian Francalanza, and Simon Said. Improving Runtime Overheads for detectEr. In *FESCA*, volume 178 of *EPTCS*, pages 1–8, 2015.

[54] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. eAOP: An Aspect Oriented Programming Framework for Erlang. In *Erlang Workshop*, pages 20–30, 2017.

[55] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A Survey of Runtime Monitoring Instrumentation Techniques. In *PrePostiFM*, volume 254 of *EPTCS*, pages 15–28, 2017.

[56] Ian Cassar, Adrian Francalanza, **Duncan Paul Attard**, Luca Aceto, and Anna Ingólfsdóttir. A Suite of Monitoring Tools for Erlang. In *RV-CuBES*, volume 3 of *Kalpa Publications in Computing*, pages 41–47, 2017.

[57] Francesco Cesarini and Simon Thompson. *Erlang Programming: A Concurrent Approach to Software Development.* O'Reilly Media, 2009.

[58] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Characterization of Temporal Property Classes. In *ICALP*, volume 623 of *LNCS*, pages 474–486, 1992.

[59] Feng Chen and Grigore Rosu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and implementation. *Electron. Notes Theor. Comput. Sci.*, 89:108–127, 2003.

[60] Feng Chen and Grigore Rosu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *TACAS*, volume 3440 of *LNCS*, pages 546–550, 2005.

[61] Feng Chen and Grigore Rosu. Mop: An Efficient and Generic Runtime Verification Framework. In *OOPSLA*, pages 569–588, 2007.

[62] Feng Chen and Grigore Rosu. Parametric Trace Slicing and Monitoring. In *TACAS*, volume 5505 of *LNCS*, pages 246–261, 2009.

[63] Feng Chen, Patrick O'Neil Meredith, Dongyun Jin, and Grigore Rosu. Efficient Formalism-Independent Monitoring of Parametric Properties. In *ASE*, pages 383–394, 2009.

[64] David M. Ciemiewicz. What Do You mean? - Revisiting Statistics for Web Response Time Measurements. In *CMG*, pages 385–396, 2001.

[65] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model Checking and the State Explosion Problem. In *LASER Summer School*, volume 7682 of *LNCS*, pages 1–30, 2011.

[66] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. The Hierarchy of Hyperlogics. In *LICS*, pages 1–13, 2019.

[67] Christian Colombo and Yliès Falcone. Organising LTL Monitors over Distributed Systems with a Global Clock. *FMSD*, 49:109–158, 2016.

[68] Christian Colombo and Gordon J. Pace. *Runtime Verification - A Hands-On Approach in Java*. Springer, 2022.

[69] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In *FMICS*, volume 5596 of *LNCS*, pages 135–149, 2008.

[70] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *SEFM*, pages 33–37, 2009.

[71] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. Elarva: A Monitoring Tool for Erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374, 2011.

[72] Christian Colombo, Adrian Francalanza, Ruth Mizzi, and Gordon J. Pace. polyLarva: Runtime Verification with Configurable Resource-Aware Monitoring Boundaries. In *SEFM*, volume 7504 of *LNCS*, pages 218–232, 2012.

[73] Oscar Cornejo, Daniela Briola, Daniela Micucci, and Leonardo Mariani. In the Field Monitoring of Interactive Application. In *ICSE-NIER*, pages 55–58, 2017.

[74] Gatling Corp. Gatling, 2020. URL `https://gatling.io`.

[75] Flaviu Cristian and Christof Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Trans. Parallel Distrib. Syst.*, 10:642–657, 1999.

[76] Markus Dahm. Byte Code Engineering with the BCEL API. Technical report, Java Informationstage 99, 2001.

[77] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51:107–113, 2008.

[78] Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Runtime Monitoring with Union-Find Structures. In *TACAS*, volume 9636 of *LNCS*, pages 868–884, 2016.

[79] Derek DeJonghe. *NGINX Cookbook: Advanced Recipes for High-Performance Load Balancing*. O'Reilly Media, 2020.

[80] Mathieu Desnoyers and Michel Dagenais. The LTTng Tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux. Technical report, École Polytechnique de Montréal, 2006.

[81] Jay L. Devore and Kenneth N. Berk. *Modern Mathematical Statistics with Applications*. Springer, 2012.

[82] Edsger W. Dijkstra. *Chapter I: Notes on Structured Programming*, page 1–82. Academic Press Ltd., 1972.

[83] Jean Dollimore, Tim Kindberg, and George Coulouris. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2005.

[84] Doron Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *CAV*, volume 2725 of *LNCS*, pages 114–117, 2003.

[85] Doron Drusinsky. *Modeling and verification using UML statecharts - a working guide to reactive system design, runtime monitoring and execution-based model checking*. Elsevier, 2006.

[86] Eclipse/IBM. OpenJ9, 2021. URL `https://www.eclipse.org/openj9`.

[87] Antoine El-Hokayem and Yliès Falcone. Monitoring Decentralized Specifications. In *ISSTA*, pages 125–135, 2017.

[88] Antoine El-Hokayem and Yliès Falcone. THEMIS: A Tool for Decentralized Monitoring Algorithms. In *ISSTA*, pages 372–375, 2017.

[89] Antoine El-Hokayem and Yliès Falcone. On the Monitoring of Decentralized Specifications: Semantics, Properties, Analysis, and Simulation. *ACM Trans. Softw. Eng. Methodol.*, 29:1:1–1:57, 2020.

[90] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, US, 2004.

[91] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *NSPW*, pages 87–95, 1999.

[92] Joan Facorro. Clojerl Language, 2021. URL `http://clojerl.org`.

[93] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Vardi. *Reasoning About Knowledge*. MIT Press, 2004.

[94] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14:349–382, 2012.

[95] Yliès Falcone, Klaus Havelund, and Giles Reger. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

[96] Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez. Efficient and Generalized Decentralized Monitoring of Regular Languages. In *FORTE*, volume 8461 of *LNCS*, pages 66–83, 2014.

[97] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime Verification of Component-Based Systems in the BIP Framework with Formally-Proved Sound and Complete Instrumentation. *SoSyM*, 14:173–199, 2015.

[98] Yliès Falcone, Dejan Nickovic, Giles Reger, and Daniel Thoma. Second International Competition on Runtime Verification CRV 2015. In *RV*, volume 9333 of *LNCS*, pages 405–422, 2015.

[99] Yliès Falcone, Hosein Nazarpour, Mohamad Jaber, Marius Bozga, and Saddek Bensalem. Tracing Distributed Component-Based Systems, a Brief Overview. In *RV*, volume 11237 of *LNCS*, pages 417–425, 2018.

[100] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A Taxonomy for Classifying Runtime Verification Tools. *STTT*, 23:255–284, 2021.

[101] Yliès Falcone, Hosein Nazarpour, Saddek Bensalem, and Marius Bozga. Monitoring Distributed Component-Based Systems. In *FACS*, volume 13077 of *LNCS*, pages 153–173, 2021.

[102] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A Stream-Based Specification Language for Network Monitoring. In *RV*, volume 10012 of *LNCS*, pages 152–168, 2016.

[103] Dror G. Feitelson. From Repeatability to Reproducibility and Corroboration. *ACM SIGOPS Oper. Syst. Rev.*, 49:3–11, 2015.

[104] Thomas Ferrère, Thomas A. Henzinger, and N. Ege Saraç. A Theory of Register Monitors. In *LICS*, pages 394–403, 2018.

[105] Colin J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10:56–66, 1988.

[106] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. In *RV*, volume 10548 of *LNCS*, pages 190–207, 2017.

[107] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, 1985.

[108] Philip J. Fleming and John J. Wallace. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Commun. ACM*, 29:218–221, 1986.

[109] Apache Software Foundtation. JMeter, 2020. URL `https://jmeter.apache.org`.

[110] Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. On the Number of Opinions Needed for Fault-Tolerant Run-Time Monitoring in Distributed Systems. In *RV*, volume 8734 of *LNCS*, pages 92–107, 2014.

[111] Adrian Francalanza. Consistently-Detecting Monitors. In *CONCUR*, volume 85 of *LIPIcs*, pages 8:1–8:19, 2017.

[112] Adrian Francalanza. A Theory of Monitors. *Inf. Comput.*, 281:104704, 2021.

[113] Adrian Francalanza and Aldrin Seychell. Synthesising Correct Concurrent Runtime Monitors. *FMSD*, 46:226–261, 2015.

[114] Adrian Francalanza and Jasmine Xuereb. On Implementing Symbolic Controllability. In *COORDINATION*, volume 12134 of *LNCS*, pages 350–369, 2020.

[115] Adrian Francalanza, Andrew Gauci, and Gordon J. Pace. Distributed System Contract Monitoring. *JLAMP*, 82:186–215, 2013.

[116] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. On Verifying Hennessy-Milner Logic with Recursion at Runtime. In *RV*, volume 9333 of *LNCS*, pages 71–86, 2015.

[117] Adrian Francalanza, Luca Aceto, Antonis Achilleos, **Duncan Paul Attard**, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A Foundation for Runtime Monitoring. In *RV*, volume 10548 of *LNCS*, pages 8–29, 2017.

[118] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Monitorability for the Hennessy-Milner Logic with Recursion. *FMSD*, 51:87–116, 2017.

[119] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. Runtime Verification for Decentralised and Distributed Systems. In *Lectures on RV*, volume 10457 of *LNCS*, pages 176–210. Springer, 2018.

[120] Vijay K. Garg. *Elements of Distributed Computing*. Wiley, 2014.

[121] Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach*. CRC, 2014.

[122] Susanne Graf, Doron A. Peled, and Sophie Quinton. Monitoring Distributed Systems Using Knowledge. In *FORTE*, volume 6722 of *LNCS*, pages 183–197, 2011.

[123] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, 1993.

[124] Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. Runtime Verification Based on Register Automata. In *TACAS*, volume 7795 of *LNCS*, pages 260–276, 2013.

[125] Jan Friso Groote and Radu Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In *AMAST*, volume 1548 of *LNCS*, pages 74–90, 1998.

[126] Duncan A. Grove and Paul D. Coddington. Analytical Models of Probability Distributions for MPI Point-to-Point Communication Times on Distributed Memory Parallel Computers. In *ICA3PP*, volume 3719 of *LNCS*, pages 406–415, 2005.

[127] Mark Harman and Peter W. O’Hearn. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *SCAM*, pages 1–23, 2018.

[128] Klaus Havelund and Doron Peled. Runtime Verification: From Propositional to First-Order Temporal Logic. In *RV*, volume 11237 of *LNCS*, pages 90–112, 2018.

[129] Klaus Havelund and Doron Peled. BDDs for Representing Data in Runtime Verification. In *RV*, volume 12399 of *LNCS*, pages 107–128, 2020.

[130] Klaus Havelund and Grigore Rosu. An Overview of the Runtime Verification Tool Java PathExplorer. *FMSD*, 24:189–215, 2004.

[131] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zalinescu. Monitoring Events that Carry Data. In *Lectures on Runtime Verification*, volume 10457 of *LNCS*, pages 61–102. Springer, 2018.

[132] Fred Hebert. *Stuff Goes Bad: Erlang in Anger.* Manning, 2014.

[133] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

[134] Loïc Hoguin. Cowboy, 2020. URL `https://ninenines.eu`.

[135] Loïc Hoguin. Ranch, 2020. URL `https://ninenines.eu`.

[136] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2003.

[137] Shams Mahmood Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE!@SPLASH*, pages 67–80, 2014.

[138] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE*, pages 1427–1430, 2012.

[139] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* CRC, 2020.

[140] Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design: Theory in Practice.* O'Reilly Media, 2007.

[141] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[142] Saša Jurić. *Elixir in Action.* Manning, 2019.

[143] Michael Kaminski and Nissim Francez. Finite-Memory Automata. *Theor. Comput. Sci.*, 134:329–363, 1994.

[144] Bill Kayser. What is the expected distribution of website response times?, 2017. URL `https://blog.newrelic.com/engineering/expected-distributions-website-response-times`.

[145] Robert M. Keller. Formal Verification of Parallel Programs. *Commun. ACM*, 19:371–384, 1976.

[146] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, volume 1241 of *LNCS*, pages 220–242, 1997.

[147] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353, 2001.

[148] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *FMSD*, 24:129–155, 2004.

[149] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications (Real-Time Systems Series).* Springer, 2011.

[150] Dexter Kozen. Results on the Propositional μ-Calculus. In *ICALP*, volume 140 of *LNCS*, pages 348–359, 1982.

[151] Ajay D. Kshemkalyani. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, 2011.

[152] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, 2011.

[153] Roland Kuhn, Brian Hanafee, and Jamie Allen. *Reactive Design Patterns.* Manning, 2016.

[154] Lars Kuhtz and Bernd Finkbeiner. LTL Path Checking is Efficiently Parallelizable. In *ICALP*, volume 5556 of *LNCS*, pages 235–246, 2009.

[155] Orna Kupferman and Moshe Y. Vardi. Model Checking of Safety Properties. *FMSD*, 19:291–314, 2001.

[156] Leslie Lamport. "Sometime" is Sometimes "Not Never" - On the Temporal Logic of Programs. In *POPL*, pages 174–185, 1980.

[157] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982.

[158] Julien Lange and Nobuko Yoshida. Verifying Asynchronous Interactions via Communicating Session Automata. In *CAV*, volume 11561 of *LNCS*, pages 97–117, 2019.

[159] Kim Guldstrand Larsen. Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *TCS*, 72:265–288, 1990.

[160] Jonathan Laurent, Alwyn Goodloe, and Lee Pike. Assuring the Guardians. In *RV*, volume 9333 of *LNCS*, pages 87–101, 2015.

[161] Ben Laurie and Peter Laurie. *Apache: The Definitive Guide.* O'Reilly Media, 2002.

[162] Matthew Alan Le Brun, Duncan Paul Attard, and Adrian Francalanza. Graft: General Purpose RAFT Consensus in Elixir. In *Erlang Workshop*, pages 2–14, 2021.

[163] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *ICPE*, pages 3–14, 2017.

[164] Martin Leucker and Christian Schallhart. A Brief Account of Runtime Verification. *JLAP*, 78: 293–303, 2009.

[165] Bryon C. Lewis and Albert E. Crews. The Evolution of Benchmarking as a Computer Performance Evaluation Technique. *MIS Q.*, 9:7–16, 1985.

[166] Jay Ligatti, Lujo Bauer, and David Walker. Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. *Int. J. Inf. Sec.*, 4:2–16, 2005.

[167] Lightbend. Play Framework, 2020. URL `https://www.playframework.com`.

[168] Zhen Liu, Nicolas Niclausse, and César Jalpa-Villanueva. Traffic Model and Performance Evaluation of Web Servers. *Perform. Evaluation*, 46:77–100, 2001.

[169] Mark Loy, Patrick Niemeyer, and Daniel Leuck. *Learning Java: An Introduction to Real-World Programming with Java*. O'Reilly Media, 2020.

[170] Qingzhou Luo and Grigore Rosu. EnforceMOP: A Runtime Property Enforcement System for Multithreaded Programs. In *ISSTA*, pages 156–166, 2013.

[171] Radu Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In *VMCAI*, volume 98, 1998.

[172] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[173] Eric Matthes. *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*. No Starch Press, 2019.

[174] Deep Medhi and Karthik Ramasamy. Chapter 3 - routing protocols: Framework and principles. In *Network Routing (Second Edition)*, The Morgan Kaufmann Series in Networking, pages 64–113. Morgan Kaufmann, 2018.

[175] Patrick O'Neil Meredith and Grigore Rosu. Efficient Parametric Runtime Verification with Deterministic String Rewriting. In *ASE*, pages 70–80, 2013.

[176] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An Overview of the MOP Runtime Verification Framework. *STTT*, 14:249–289, 2012.

[177] Microsoft. MSDN, 2021. URL `https://msdn.microsoft.com`.

[178] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[179] Dario Della Monica and Adrian Francalanza. Pushing Runtime Verification to the Limit: May Process Semantics Be With Us. In *OVERLAYAI*IA*, volume 2509 of *CEUR Workshop Proceedings*, pages 47–52, 2019.

[180] Menna Mostafa and Borzoo Bonakdarpour. Decentralized Runtime Verification of LTL Specifications in Distributed Systems. In *IPDPS*, pages 494–503, 2015.

[181] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley, 2011.

[182] Samaneh Navabpour, Yogi Joshi, Chun Wah Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. RiTHM: A Tool for Enabling Time-Triggered Runtime Verification for C Programs. In *ESEC/SIGSOFT FSE*, pages 603–606, 2013.

[183] Rumyana Neykova. *Multiparty Session Types for Dynamic Verification of Distributed Systems.* PhD thesis, Imperial College London, UK, 2017.

[184] Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. *LMCS*, 13, 2017.

[185] Rumyana Neykova and Nobuko Yoshida. Let it Recover: Multiparty Protocol-Induced Recovery. In *CC*, pages 98–108, 2017.

[186] Nicolas Niclausse. Tsung, 2017. URL `http://tsung.erlang-projects.org`.

[187] Jakob Nielsen. *Usability Engineering.* Morgan Kaufmann, 1993.

[188] Scott Oaks. *Java Performance: In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond.* CRC, 2020.

[189] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala.* Artima Inc., 2020.

[190] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[191] Athanansios Papoulis. *Probability, Random Variables, and Stochastic Processes.* McGraw Hill, 1991.

[192] Amir Pnueli and Aleksandr Zaks. PSL Model Checking and Run-Time Verification via Testers. In *FM*, volume 4085 of *LNCS*, pages 573–586, 2006.

[193] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *PLDI*, pages 31–47, 2019.

[194] Kevin Quick. Thespian, 2020. URL `http://thespianpy.com`.

[195] Aidan Randtoul and Phil Trinder. A Reliability Benchmark for Actor-Based Server Languages. In *Erlang Workshop*, pages 21–32, 2022.

[196] Giles Reger and David E. Rydeheard. From First-Order Temporal Logic to Parametric Trace Slicing. In *RV*, volume 9333 of *LNCS*, pages 216–232, 2015.

[197] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. MarQ: Monitoring at Runtime with QEA. In *TACAS*, volume 9035 of *LNCS*, pages 596–610, 2015.

[198] Giles Reger, Sylvain Hallé, and Yliès Falcone. Third International Competition on Runtime Verification - CRV 2016. In *RV*, volume 10012 of *LNCS*, pages 21–37, 2016.

[199] Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka in Action.* Manning, 2015.

[200] Richard J. Rossi. *Mathematical Statistics: An Introduction to Likelihood Based Inference.* Wiley, 2018.

[201] Grigore Rosu and Feng Chen. Semantics and Algorithms for Parametric Monitoring. *LMCS*, 8, 2012.

[202] Sartaj Sahni and George L. Vairaktarakis. The Master-Slave Paradigm in Parallel Computer and Industrial Settings. *J. Glob. Optim.*, 9:357–377, 1996.

[203] Torben Scheffel and Malte Schmitz. Three-Valued Asynchronous Distributed Runtime Verification. In *MEMOCODE*, pages 52–61, 2014.

[204] Fred B. Schneider. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, 2000.

[205] Joshua Schneider, David A. Basin, Frederik Brix, Srdan Krstic, and Dmitriy Traytel. Scalable Online First-Order Monitoring. *Int. J. Softw. Tools Technol. Transf.*, 23:185–208, 2021.

[206] Koushik Sen and Grigore Rosu. Generating Optimal Monitors for Extended Regular Expressions. *Electron. Notes Theor. Comput. Sci.*, 89:226–245, 2003.

[207] Koushik Sen, Grigore Rosu, and Gul Agha. Runtime Safety Analysis of Multithreaded Programs. In *ESEC / SIGSOFT FSE*, pages 337–346, 2003.

[208] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *ICSE*, pages 418–427, 2004.

[209] Koushik Sen, Grigore Rosu, and Gul Agha. Online Efficient Predictive Safety Analysis of Multithreaded Programs. *Int. J. Softw. Tools Technol. Transf.*, 8:248–260, 2006.

[210] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Decentralized Runtime Analysis of Multithreaded Applications. In *IPDPS*, 2006.

[211] Steven C. Seow. *Designing and Engineering Time: The Psychology of Time Perception in Software.* Addison-Wesley, 2008.

[212] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. DaCapo con Scala: design and analysis of a Scala benchmark suite for the JVM. In *OOPSLA*, pages 657–676, 2011.

[213] Connie U. Smith and Lloyd G. Williams. Software Performance AntiPatterns; Common Performance Problems and their Solutions. In *CMG*, pages 797–806, 2001.

[214] Connie U. Smith and Lloyd G. Williams. New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In *CMG*, pages 667–674, 2002.

[215] SPEC. SPECjvm2008, 2008. URL `https://www.spec.org/jvm2008`.

[216] Volker Stolz. Temporal Assertions with Parametrized Propositions. *J. Log. Comput.*, 20:743–757, 2010.

[217] Sasu Tarkoma. *Overlay Networks: Toward Information Networking.* Auerbach, 2010.

[218] The Pony Team. Ponylang, 2021. URL `https://tutorial.ponylang.io`.

[219] **Duncan Paul Attard** and Adrian Francalanza. Trace Partitioning and Local Monitoring for Asynchronous Components. In *SEFM*, volume 10469 of *LNCS*, pages 219–235, 2017.

[220] **Duncan Paul Attard**, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Introduction to Runtime Verification. In *Behavioural Types: from Theory to Tools*, Automation, Control and Robotics, pages 49–76. River, 2017.

[221] **Duncan Paul Attard**, Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Better Late than Never or: Verifying Asynchronous Components at Runtime. In *FORTE*, volume 12719 of *LNCS*, pages 207–225, 2021.

[222] Germán Vidal. Computing Race Variants in Message-Passing Concurrent Programming with Selective Receives. In *FORTE*, volume 13273 of *LNCS*, pages 188–207, 2022.

[223] Craig Walls. *Spring in Action.* Manning, 2022.

[224] B. P. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4:419–420, 1962.

[225] Pierre Wolper. Temporal Logic Can be More Expressive. *Inf. Control.*, 56:72–99, 1983.

[226] Cui-Qing Yang and Barton P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *ICDCS*, pages 366–373, 1988.

[227] Jiali Yao, Zhigeng Pan, and Hongxin Zhang. A Distributed Render Farm System for Animation Production. In *ICEC*, volume 5709 of *LNCS*, pages 264–269, 2009.

[228] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.

[229] Teng Zhang, Greg Eakman, Insup Lee, and Oleg Sokolsky. Overhead-Aware Deployment of Runtime Monitors. In *RV*, volume 11757 of *LNCS*, pages 375–381, 2019.