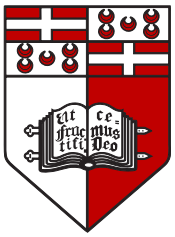
Towards Modular Monitoring for Concurrent Systems

Duncan Paul Attard

Supervisor: Dr. Adrian Fancalanza



Faculty of ICT

University of Malta

September 2016

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

Towards Modular Monitoring for Concurrent Systems

submitted is my work, except where acknowledged and referenced.

Duncan Paul Attard

September 28, 2016

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Adrian Francalanza, for his unwavering guidance, enthusiasm and support offered throughout the entire duration of this work, as well as for instilling in me a greater appreciation of computer science. His insightful comments and discussions incited me to widen my perspective from various angles. I would also like acknowledge that the work presented in Chapter 3 was developed together with Dr. Adrian Francalanza.

I would also like to thank the academic and non-academic staff members of the Faculty of ICT who were always supportive and helpful during this past year of studies within the faculty.

A special thanks goes also to my parents Raymond and Gertrude, and my siblings, Daphne and Björn for being there in difficult times throughout this year. I would also like to thank my closest friends, especially Mandy, for their unconditional support and infinite patience.

Finally, I cannot forget to mention my friends, Annalizz, John and Kevin, who each in their own way, made this journey fun and unforgettable!

Abstract

Concurrent systems typically consist of multiple components or processes that are designed with the intention of working together. Commonly, the correctness of such systems is perceived from a global perspective, and runtime monitors consider the overall functionality of a system rather than focusing on individual components. This work studies localisation as an alternative means of runtime monitoring in concurrent scenarios to mitigate scalability and performance issues that arise when a global monitoring approach is employed. It centres on the benefits that can be gained if the task of monitoring component-based systems is approached from a modular stance. Within this context, the study takes the form of a comparative analysis that rigorously assesses the benefits of local monitoring, and the ways these outweigh those of its global counterpart.

The study presents an implementation of a runtime monitoring tool that automatically synthesises concurrent monitors from formulae specified in mHML, a monitorable subset of the branching-time logic μHML. The synthesis algorithm is compositional w.r.t. to the structure of formula, thereby naturally mapping correctness specifications into concurrent Erlang actors that monitor a running system with minimal instrumentation efforts.

Through this tool, local and global monitoring are evaluated *vis-à-vis* a number of assessment criteria that target the qualitative and quantitative aspects of each approach. To ensure a thorough and holistic examination of the effectiveness of both monitoring approaches, two different experiment setups are considered. The first setup gauges the applicability of local monitoring by studying two forms of component-based architectures, in order to identify in which case the application of local monitoring yields the most benefits. In the second setup, local and global monitoring are used to runtime verify a third-party application, with the intent of assessing the behaviour of each in a real-world use case.

The results obtained from these evaluations demonstrate that one stands to gain when local monitoring is employed in component-based scenarios, and that, in general, it outperforms global monitoring. Furthermore, the results clearly show that local monitoring can be used to great effect when applied to isolated target system components. In cases involving components that communicate between themselves, local monitoring may still yield benefits, though these are harder to interpret and quantify. The applicability of these results is not limited solely to the host technology used to conduct the experiments, nor is it bound to the specification formalism used. Rather, the results should be interpreted from an implementation-agnostic viewpoint, making them applicable to any concurrent scenario where components can be delineated into subsystems that can be individually traced.

Contents

List of Figures

List of Tables

Conventions

Textual content and illustrations within this manuscript adopt these conventions.

Text

Emphasised text denotes important notions or key concepts;

Italic text denotes variables;

Small Capitals identifies process names;

Sans Serif text refers to existing tools and third-party software artefacts;

`Teletype` text identifies language keywords or source code snippets;

Bold Sans Serif denotes special operators in mathematical notation;

"Quoted" italic text symbolises textual descriptions of runtime properties.

Illustrations

⟶ (process communication arrows) represent uni-directional communication between processes;

⋯› (spawn arrows) denote process instantiation actions;

›⟶ (trace message flow arrows) denote a pairing between a tracer and the process it traces, or a pairing between a tracer and its associated monitor;

❶ identifies specific numbered steps in a protocol description;

denotes processes or a subsystem composed of processes;

Trc identifies tracer processes;

denotes newly spawned processes or subsystem boundaries;

highlights subjects that merit the reader's attention.

1. Introduction

Concurrency refers to software systems whose functionality is expressed in terms of multiple components or processes specifically designed to work simultaneously with each other [35]. In recent years, concurrent solutions have become increasingly commonplace and are nowadays preferred over *monolithic* architectures. This is owed to the rigidity the latter type of systems exhibit, where attempts at addressing scalability concerns usually lead to notoriously complex and often, inadequate solutions. In contrast, a *concurrency-oriented* [5] development approach tackles the scalability problem from a software design perspective, thereby ensuring that such systems can easily avail themselves of multi-processor and multi-core platforms which are prevalent today.

The execution of concurrent systems is typically characterised by a high degree of non-determinism — the upshot of interleaving threads or processes that give rise to a vast number of possible execution paths, making the verification of these types of systems an onerous task. Runtime Verification (RV) is a *lightweight* verification technique that makes it an appealing choice when it comes to verifying concurrent systems, because the scalability issues associated with other traditional verification techniques are entirely circumvented.

The manner with which RV is applied in practice is largely driven by the design of the system being considered. For example, monolithic architectures usually render the task of expressing properties on different system components rather cumbersome,

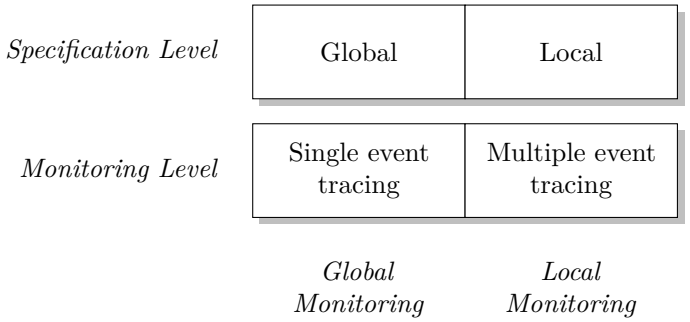


Figure 1.1: The classification levels of a generic RV setting.

whereas modularised setups afford the RV approach a granular view. Instances of the latter may, in general, be described in terms of the two-dimensional matrix in Figure 1.1, consisting of two conceptual levels of classification:

Specification Level Focuses on how correctness properties can be expressed in terms of the system behaviour considered. *Global properties* span the entire system, and usually regard multiple variables or execution trace events. Conversely, *local properties* concentrate only on a fraction of the system behaviour and consider a select subset of components.

Monitoring Level Supports global or local property specifications by defining what tracing mechanisms are made available by the RV framework. In *single event tracing*, trace events are funnelled into a central stream which is then read and processed by a singleton monitor. In *multiple event tracing*, independent monitors can opt to consume portions of the main execution trace concurrently by subscribing to separate *subtraces*.

The work presented in this dissertation studies RV through these two classification levels. In particular, it explores how modular monitoring can be applied within the context of concurrent systems, providing the means by which local properties declared on subparts of the target system can be *clearly specified* and *efficiently monitored*.

1.1 Problem Synopsis and Motivation

This work distinguishes between two runtime monitoring approaches, and defines them according to the classification levels depicted in Figure 1.1:

Global monitoring Determines the correctness of global properties within the context of a modular (concurrent) system supporting *single* event traces. Concurrency requires that global properties account for the possible interleaving of the execution trace events being considered.

Local monitoring Concentrates on system components, and is defined within the context of a modular (concurrent) system supporting *multiple* event tracing. Properties consider only portions of the global trace (*i.e.,* subtraces) containing events that are directly relevant. As these properties are mutually independent, the corresponding synthesised monitors function in complete isolation.

In a concurrent scenario, a global monitoring approach tends to be the cause of rigidity and scalability complaints, both when initially specifying and later, when monitoring for correctness properties. Execution interleaving makes global property specification hard, cumbersome, and the results of such efforts end up in runtime monitors that are brittle and sensitive even to *localised* changes in the system. All of this adds up to unnecessarily complex and inefficient runtime monitors.

Consider the system Sys composed of two simple processes A and B, arranged in parallel as follows:

$$\text{Sys} \stackrel{\text{def}}{=} (\text{A} \mid \text{B}) \qquad \text{A} \stackrel{\text{def}}{=} a + e \qquad \text{B} \stackrel{\text{def}}{=} b + e \tag{1.1}$$

Components A and B execute independently of each other, and can perform a choice (denoted by $+$) between actions a and e in case of component A, b and e in case of component B. Once a choice is made by *both* A and B, Sys completes its execution and terminates (denoted by $\mathbf{0}$). All possible execution paths available to

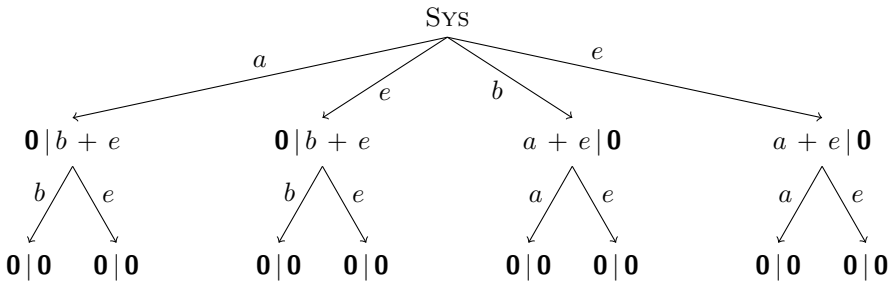


Figure 1.2: The possible execution paths for Sys $\stackrel{\text{def}}{=}$ (A | B).

Sys are depicted in Figure 1.2. For instance, the leftmost branch shows that Sys affords the transition Sys $\xrightarrow{a}$ $(\mathbf{0} \mid b + e)$, and from $(\mathbf{0} \mid b + e)$, it can either perform the transitions $\xrightarrow{b}$ or $\xrightarrow{e}$, both of which lead to the terminated system $(\mathbf{0} \mid \mathbf{0})$.

Example 1.1.1. The property stating that *"neither component A performs action 'a', nor component B action 'b'"* can be expressed using the following *global* safety Hennessy-Milner Logic (HML) [21] formula:

$$\varphi_g = [a]\,\mathbf{ff} \wedge [b]\,\mathbf{ff} \wedge [e]\,([a]\,\mathbf{ff} \wedge [b]\,\mathbf{ff}) \tag{1.2}$$

where $[\alpha]\,\mathbf{ff}$ describes processes that *cannot* afford action α, *i.e.,* if action α is matched, then $\mathbf{ff}$ flags it as invalid. All possible (two-action) execution traces for Sys, are matched in the following way. Subformulae $[a]\,\mathbf{ff}$ and $[b]\,\mathbf{ff}$ match all traces prefixed by actions 'a' and 'b' (*i.e.,* 'ab', 'ae', 'ba', 'be'). The interleaving of action 'e' is accounted for by subformula $[e]\,([a]\,\mathbf{ff} \wedge [b]\,\mathbf{ff})$, which first matches traces prefixed by 'e' (*i.e.,* 'ea' and 'eb'), and then matches the second action 'a' or 'b' using $([a]\,\mathbf{ff} \wedge [b]\,\mathbf{ff})$. In all instances, a violation is flagged immediately due to $\mathbf{ff}$ when actions 'a' or 'b' are encountered in *any* position inside one of the execution traces for Sys (see Figure 1.2). ■

Suppose that now, Sys was amended to include a third concurrent process: C $\stackrel{\text{def}}{=}$ $c + e$, while the property in Example 1.1.1 is extended to prohibit 'c' transitions from the new component C. The global specification φ_g must be updated to handle the additional interleaving action 'c' introduces, resulting in a formula that

considers 34 different (three-action) trace combinations. Such a scenario highlights the fact that as the number of processes (or actions per process) increases, specifying global properties becomes quickly prohibitive. In view of this, it could prove to be beneficial if the overall target system correctness is perceived as a collection of local *subproperties*, rather than as a single global property.

Example 1.1.2. The same requirement expressed by formula (1.2) can be also specified in terms of two *local* HML formulae, one which monitors for action 'a' from process A, and one, for action 'b' from B:

$$\varphi_a = [a]\,\mathbf{ff} \qquad\qquad \varphi_b = [b]\,\mathbf{ff} \tag{1.3}$$

In contrast to the global property φ_g from Example 1.1.1, φ_a and φ_b are much simpler and clearer to work with. It is also evident that in this case, execution interleaving ceases to be an issue, because each property needs only to consider the actions of its own process. Furthermore, as new components are added, existing properties need not necessarily be refactored: introducing process C, requires *only* that formula $\varphi_c = [c]\,\mathbf{ff}$ be added alongside the ones in (1.3). ■

1.2 Aims and Objectives

This dissertation aims to study the benefits that can be reaped if local monitoring is applied to a purely *concurrent, online* setting, and crucially, how this compares in relation to a global monitoring approach. Additionally, the study also concentrates on how these two types of monitoring strategies can be tackled from an asynchronous and non-invasive standpoint.

1.2.1 Assessment Criteria

In order to appropriately measure and assess these findings, the five criteria listed below have been identified. Criteria 1 to 4 shall be evaluated through qualitative

argumentation; Criterion 5 will be gauged using quantitative experiments.

1. **Understandability**: Focuses on the ease with which specification formulae can be concisely written, and whether expressing them requires the user to account for additional system functionality or behaviour that is not directly relevant to the property at hand;

2. **Maintainability**: Determines if extensions in the specification scripts are easy to address, and whether these would need to be refactored, even if other unrelated parts of the system are modified;

3. **Expressivity**: Investigates if certain specific cases make it possible to express a larger number of properties, and whether specifying them locally or globally affects the difficulty with which these can be monitored for at runtime;

4. **Fault Tolerance**: Establishes whether monitors are resilient to failures that can occur in one or more system components;

5. **Performance**: Measures the impact monitors have on the target system's memory consumption, CPU utilisation, and responsiveness, and how well these compare in relation to measurements obtained from an unmonitored version of the target system.

As this work focuses exclusively on concurrent systems, it proved convenient to employ a programming language that *inherently* supports the basic notions of concurrency. Erlang suitably fulfils this requirement, as it embraces the *actor model* [12] which makes it easy and clear to reason about and analyse the interactions that take place between processes. Furthermore, Erlang conveniently provides a per-process native tracing functionality that makes it possible to observe systems at fine levels of granularity.

1.2.2 Objectives

The primary objectives this work sets out to tackle are thus:

1. *Implementing a prototype tool that supports the specification of safety and co-safety properties.* This provides a sufficiently rich specification logic that can be used to express negative and positive properties. The tool should serve as a practical foundational means that enables one to study and compare the attributes of local and global monitoring;

2. *Studying the effectiveness of local monitoring when applied to static systems. Static systems* are characterised by the absence of *dynamism*, where the size of the system in terms of its components is fixed and remains constant during runtime (*cf.* Objective 4). The investigation should result in the correct design and implementation of a local monitoring algorithm that can be *modularly* applied to components within the system under scrutiny. Furthermore, the implementation ought to be substantiated by a preliminary evaluation of local monitoring that explores the manner in which this technique can be effectively applied to different system configurations. To put the benefits that are attributed to localisation into a clearer context, local monitoring should be compared against its global counterpart, based on the assessment criteria listed in Section 1.2.1;

3. *Studying the effects of local monitoring on an industry-level third-party application.* Aside from revisiting the qualitative results obtained in Objective 2, the investigation should also serve as means to reassess in particular, the performance of local monitoring when applied to real-world uses. This investigation plays a crucial part in reaffirming the conclusions drawn from the preliminary evaluation conducted in Objective 2;

4. *Moving towards a local monitoring algorithm that dynamically monitors individual target system components. Local dynamic monitoring* can be applied to systems whose size in terms of their components *scales* (at runtime) in relation to the current computational demands. The quintessential examples of such dynamic systems include web servers and message queuing middleware.

1.3 Document Outline

The content in this manuscript is organised into the following chapters:

- **Chapter 2** provides the reader with the basic concepts and background knowledge required to understand the material that is presented in subsequent chapters. It introduces the main ideas of RV and monitoring, followed by an overview of mHML, a runtime monitorable subset of the branching-time logic μHML that can be used to specify correctness properties over *programs*. This chapter concludes with a brief discussion of the main Erlang concepts upon which this work is based;

- An implementation of a prototype tool that automatically synthesises concurrent runtime monitors from specifications written in mHML is tackled next in **Chapter 3**. Said implementation is based on extensions and refinements of the results from a previous theoretical work in [21]. The chapter also explores the engineering aspects of the synthesis procedure to show how concurrent monitors can be naturally mapped to Erlang actors that monitor a running system with minimal instrumentation efforts;

- **Chapter 4** extends the preliminary prototype tool implementation in Chapter 3 to a local monitoring scenario, where isolated monitor instances can separately and *independently* target different system components with fixed (*i.e.,* static) configurations. The challenges faced when adapting the tool to a localised setup are discussed at length, followed by an investigation that identifies *ideal* scenarios where this type of monitoring can be applied. Furthermore, the effectiveness of local monitoring is qualitatively and quantitatively evaluated in relation to that of global monitoring w.r.t. to the five assessment criteria listed in Section 1.2. These evaluations are conducted over two systems with different component configurations; this helps to determine optimal scenarios where using either monitoring approach is the most advantageous;

- **Chapter 5** presents a detailed case study on the use of local and global monitoring on an industry-level socket acceptor pool for TCP protocols called Ranch [25]. The basics of Ranch and Erlang OTP are first introduced. These provide the necessary insight required to identify a particular *static* subsystem within Ranch that lends itself well to local monitoring. A suitable experiment setup design is afterwards discussed, together with a number of precautions that were taken in order to ensure the least possible amount of measurement errors within the collected results. The system is subjected to the same qualitative and quantitative evaluation as that from Chapter 4, the better to gauge the effectiveness of local and global monitoring when applied to real-world third-party applications;

- **Chapter 6**, builds on the work developed in Chapter 4. It recasts the problem of (static) local monitoring seen so far as a special case of dynamic monitoring, and moves towards a holistic approach that can handle both types of local monitoring. An overview of the general idea of dynamic local monitoring is presented first. This is followed by an analysis of the implementation issues that one might need to address when adapting a dynamic local monitoring algorithm to a concurrent actor-based platform such as Erlang. The chapter concludes by very briefly discussing a proof-of-concept Erlang implementation of the dynamic local monitoring algorithm given in this chapter;

- The conclusion in **Chapter 7** highlights the achievements obtained in this work, in accordance to the original project's aims and objectives, as stated in Section 1.2. Subsequently, a number of avenues for future work are identified. Finally, a number of related works in the area are discussed and compared to the one developed in this dissertation, focusing mainly on the decentralised, asynchronous and concurrent aspects of this study.

2. Background

This chapter presents an overview of the foundational concepts on which the work developed in this dissertation is based. It should serve as guide that very quickly takes the reader through the most relevant areas required to understand and appreciate the material that follows this chapter. These topics are covered:

- A short discussion of RV, monitors and the different ways from which their implementation can be approached;
- Reactive systems and how these can be formally modelled using labelled transition systems;
- A brief account of formal property specification, followed by a presentation of the branching-time logic μHML and its monitorable subset mHML;
- An overview of the Erlang language and its native tracing mechanism.

2.1 Runtime Verification

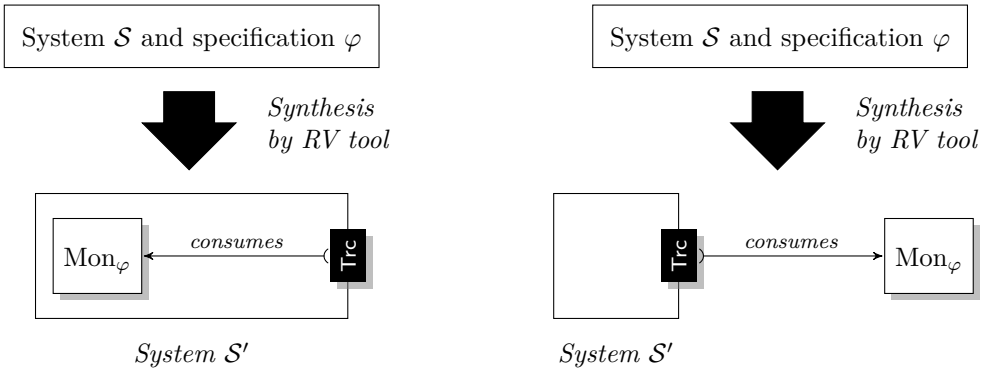
Verification encompasses the approaches employed in showing that a system violates or satisfies its expected behaviour. Commonly used verification techniques include model checking, runtime verification and testing. Model checking, an automated, static verification technique applicable to finite state systems [14], describes the

problem of whether for an abstract model of the system, *all* its possible executions satisfy some correctness specification. This contrasts with Runtime Verification (RV), wherein the analysis of correctness specifications is performed *incrementally* on partial runtime executions, up to the current execution point [20, 27]. Due to this lightweight approach to verification, RV can be employed in complex settings (*e.g.* concurrent systems), since the analysis is conducted solely on the execution path taken by the system at *runtime*. Static verification techniques, fail to scale in these kinds of scenarios, because the size of the system state space grows exponentially w.r.t. the number of state variables [15]. Despite its advantages, RV has limited expressivity and cannot be used to verify arbitrary specifications such as (general) liveness properties [28].

When a system is run, it generates a (possibly infinite) sequence of events, known as a *trace*. These events are the result of internal or external system behaviours, such as message exchanges between processes, function invocations, and the like. An *execution*, *i.e.,* a *finite prefix* of an infinite trace, is consumed by a software device known as a *monitor* which checks whether the execution satisfies some correctness property specification, yielding a *verdict* accordingly. Verdicts denote monitoring outcomes, and are assigned values taken from some truth-domain [20, 27]. For instance, rudimentary approaches can adopt a boolean domain (*e.g.* $\{\mathsf{false}, \mathsf{true}\}$) to denote property violations and satisfactions, while others such as [8, 21] may employ a three-valued system to represent also *inconclusive* verdicts.

2.1.1 Monitors

In RV, a monitor for some correctness property is typically synthesised automatically from high-level specifications that finitely describe these properties, using formal logics [8, 9, 21] or other formalisms such as regular expressions [22] or automata [7, 17, 19, 34]. This process, depicted in Figure 2.1, yields an instrumented version of the original system, together with a monitor which reads and evaluates the execution trace sent to it by the *tracer*. While the tracer mechanism *may* need to be



(a) Tracer and monitor instrumented into the system.

(b) Tracer instrumented into the system, with monitor laying outside.

Figure 2.1: Monitor synthesis from a property specification φ.

instrumented into the system in order to elicit trace events it sends to the monitor, the latter can live both within [19, 29, 34] (Figure 2.1a) and without [6, 17, 23] (Figure 2.1b) the monitored system. A monitor that is integrated into the system *shares* its computational resources and induces performance overheads, whereas a monitor living outside the system can use its own computational resources to limit the overall performance impact. Internal system monitors incur no communication cost, contrary to externalised monitors, although the latter make it possible to distribute monitors if needed.

Monitoring Approaches

RV distinguishes between two kinds of monitoring approaches that are classified according to the *timeliness* with which execution traces are processed[1]. *Online monitoring* actively processes events from the trace incrementally, and yields its verdicts while the system is running; contrarily, *offline monitoring* processes pre-recorded traces, typically after the system has finished executing [27, 34]. Online monitors work with the monitored system in one of two ways. In *online synchronous monitoring*, system and monitor execute in lock-step, *i.e.,* the system is paused until

[1]It is worth mentioning that some literature differs in its definition of online and offline monitoring, and treats online monitoring as *synchronous* or *active* monitoring, and offline monitoring as *asynchronous* or *passive* monitoring [8, 22, 27]. This view is not adopted in this work, however.

an acknowledgement is received back from the monitor in response to a trace event sent earlier (*e.g.* the work in [10, 19]). In *online asynchronous monitoring*, trace events generated by the system are buffered and processed by the monitor *eventually*, thereby decoupling the two (*e.g.* the work in [6, 17, 23]). Synchronous monitoring enables the timely detection of property violations, as opposed to the asynchronous flavour, although the latter usually offers better overall performance as it does not slow the system while monitoring. Online monitoring makes it also possible for the monitor to *react* in response to property detections by administering actions to the system if required. The *efficiency* of online monitoring is paramount, as this can adversely affect the monitored system or even alter its functional behaviour, *e.g.* slowness due to inefficient monitors might cause the system to violate time-dependent properties that would not have been violated in the unmonitored system. A monitoring tool that induces considerable levels of performance overheads is typically infeasible to use in practical scenarios.

2.2 Modelling Reactive Systems

Reactive systems can be described as a collection of one or more processes that respond to *external* events or stimuli. Rather than terminating upon producing a final output, these systems are characterised by continuous communication with their environment, and are often perceived as black box entities whose state is gauged only through the *observation* of visible (external) actions.

Labelled Transition Systems (LTSes) can be used to model reactive systems as *process execution graphs* [2]. A LTS is comprised of the triple $\langle \textsc{Sys}, (\textsc{Act} \cup \{\tau\}), \longrightarrow \rangle$ consisting of a set of states or processes Sys, a set of actions Act together with the distinguished silent action τ, where $\tau \notin \textsc{Act}$ and $\mu \in \textsc{Act} \cup \{\tau\}$, and finally, a ternary transition relation $\longrightarrow \subseteq (\textsc{Sys} \times (\textsc{Act} \cup \{\tau\}) \times \textsc{Sys})$. The existence of a set of *visible* actions $\alpha, \beta \in \textsc{Act}$, and a set of *recursion* variables $x, y, z \in \textsc{Vars}$ is assumed.

Syntax

$$p, q, r \in \textsc{Sys} ::= \mathbf{nil} \quad \text{(inaction)} \quad | \quad \alpha.p \quad \text{(prefixing)} \quad | \quad p + q \quad \text{(choice)}$$
$$| \quad \mathbf{rec}\, x.p \quad \text{(recursion)} \quad | \quad x \quad \text{(recursive variable)}$$

Dynamic behaviour

$$\textsc{Act}\ \frac{}{\alpha.p \xrightarrow{\alpha} p} \qquad \textsc{SelL}\ \frac{p \xrightarrow{\mu} p'}{p + q \xrightarrow{\mu} p'} \qquad \textsc{Rec}\ \frac{}{\mathbf{rec}\, x.p \xrightarrow{\tau} p[{}^{\mathbf{rec}\, x.p}/_{x}]}$$

Figure 2.2: A model for describing reactive systems (adapted from [21]).

Process states in the LTS can be expressed using a fragment of Milner's Calculus of Communication Systems (CCS) [30] syntax shown in Figure 2.2. A process may be inactive, denoted by **nil**, prefixed by α where the process participates in action α and behaves as p thereafter, or perform an external choice and depending on its interaction, reduces itself to either p' or q' (the symmetric rule SelR is omitted). The term $\mathbf{rec}\, x.p$ enables the recursive definition of processes, and acts as a binder for the variable x in p. All recursive variables are assumed to be guarded.

The notation $p \xrightarrow{\mu} p'$ is used in lieu of (p, μ, p'), while $p \xrightarrow{\mu}\!\!\!\!\!\!/\ \ $ is written iff $p \xrightarrow{\mu} p'$ for no process p'. In addition, $p \Longrightarrow p'$ denotes $p(\xrightarrow{\tau})^* p'$, whereas $p \overset{\mu}{\Longrightarrow} p'$, is written in place of $p \Longrightarrow \cdot \xrightarrow{\mu} \cdot \Longrightarrow p'$; p' is termed the μ-derivative of p [31]. Also, $t, u \in \textsc{Act}^*$ range over sequences of visible actions, where $p \overset{\alpha_1}{\Longrightarrow} \ldots \overset{\alpha_n}{\Longrightarrow} p_n$ is written as $p \overset{t}{\Longrightarrow} p_n$, t being the sequence of actions $\alpha_1, \ldots, \alpha_n$ (see [2, 30]).

Example 2.2.1. A simple server which receives a request and sends a response back to the client, or terminates once its request limit count is reached, can be modelled using the CCS process p.

$$p = \mathbf{rec}\, x.(\mathtt{req}.\mathtt{resp}.x + \mathtt{lim}.\mathbf{nil}) \qquad q = \mathbf{rec}\, x.\mathtt{req}.(\mathtt{resp}.x + \mathtt{resp}.\mathtt{resp}.x)$$

A second server that non-deterministically sends duplicate responses to the client is modelled by the process q (see Figure 2.3). ■

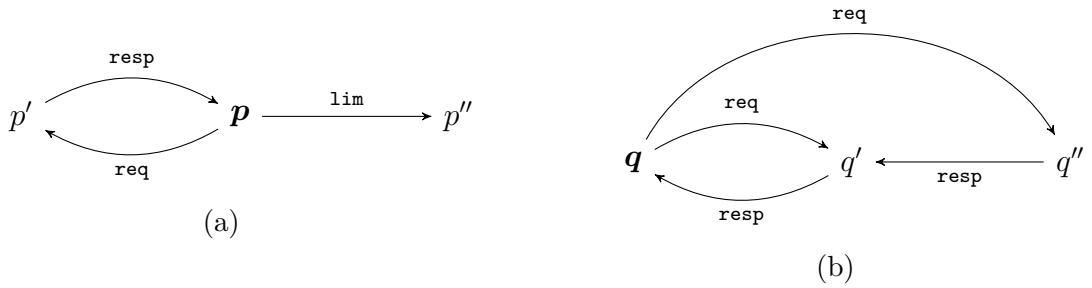


Figure 2.3: The LTSes describing two different servers p and q.

2.3 Specifying Correctness Properties

Correctness properties define the behaviour to which the executing system should adhere. (Branching-time) properties conceptually represent sets of trees that correspond to the behaviour processes are allowed to exhibit. By reading events *incrementally* from the trace, a monitor determines whether the list of events *read so far* constitute a path in the tree described by the property, or otherwise; paths in the tree denote property *satisfactions*, whereas path not in the tree denote *violations*.

This process can be perceived as one that checks whether the *prefix* of a (possibly infinite) trace constitutes an existing or non-existing path in the property tree. *Safety* properties, identified by a prefix that constitutes a non-existing path in the tree, stipulate that *bad* things do not happen, whereas dually, *co-safety* properties, identified by a prefix that constitutes an existing path in the tree, demand that *good* things do happen *eventually* [4, 8, 26].

2.3.1 The logic μHML

μHML is a branching-time logic that can be used to specify correctness properties on the execution graphs of processes [1, 2]. It assumes a countable set of logical variables $X, Y \in \text{LVAR}$, thereby allowing formulae to recursively express least and largest fixpoints using $\mathbf{min}\, X.\varphi$ and $\mathbf{max}\, X.\varphi$ respectively. These constructs in turn bind free instances of the variable X in φ, where the notions of open and closed formulae, as well as equality up to α-conversion apply. In addition to the standard

Syntax

$$
\begin{array}{rlll}
\varphi, \phi \in \mu\text{HML} ::= & \mathbf{ff} \quad \text{(falsity)} & \mid & \mathbf{tt} \quad \text{(truth)} \\
\mid & \varphi \wedge \phi \quad \text{(concjuntion)} & \mid & \varphi \vee \phi \quad \text{(disjunction)} \\
\mid & [\alpha]\,\varphi \quad \text{(necessity)} & \mid & \langle\alpha\rangle\,\varphi \quad \text{(possibility)} \\
\mid & \mathbf{max}\, X.\varphi \quad \text{(max. fixpoint)} & \mid & \mathbf{min}\, X.\varphi \quad \text{(min. fixpoint)} \\
\mid & X \quad \text{(recursive variable)} & &
\end{array}
$$

Semantics

$$
\begin{array}{lll}
[\![\mathbf{ff}, \rho]\!] & \stackrel{\text{def}}{=} \emptyset & [\![\mathbf{tt}, \rho]\!] \stackrel{\text{def}}{=} \textsc{Sys} \\
[\![\varphi \wedge \phi, \rho]\!] & \stackrel{\text{def}}{=} [\![\varphi, \rho]\!] \cap [\![\phi, \rho]\!] & [\![\varphi \vee \phi, \rho]\!] \stackrel{\text{def}}{=} [\![\varphi, \rho]\!] \cup [\![\phi, \rho]\!] \\
[\![[\alpha]\,\varphi, \rho]\!] & \stackrel{\text{def}}{=} \left\{ p \mid \forall p'.\, p \stackrel{\alpha}{\Longrightarrow} p' \text{ implies } p' \in [\![\varphi, \rho]\!] \right\} & [\![\langle\alpha\rangle\,\varphi, \rho]\!] \stackrel{\text{def}}{=} \left\{ p \mid \exists p'.\, p \stackrel{\alpha}{\Longrightarrow} p' \text{ and } p' \in [\![\varphi, \rho]\!] \right\} \\
[\![\mathbf{max}\, X.\varphi, \rho]\!] & \stackrel{\text{def}}{=} \bigcup \{ S \mid S \subseteq [\![\varphi, \rho[X \mapsto S]]\!] \} & [\![\mathbf{min}\, X.\varphi, \rho]\!] \stackrel{\text{def}}{=} \bigcap \{ S \mid [\![\varphi, \rho[X \mapsto S]]\!] \subseteq S \} \\
[\![X, \rho]\!] & \stackrel{\text{def}}{=} \rho(X) &
\end{array}
$$

Figure 2.4: The syntax and semantics of μHML.

constructs for truth, falsity, conjunction and disjunction, the syntax in Figure 2.4 includes the necessity and possibility modalities.

The semantics of the logic is defined in terms of the function mapping μHML formulae φ to the set of LTS states $S \subseteq \textsc{Sys}$ satisfying them. Figure 2.4 defines the semantics for both open and closed formulae, and uses a map $\rho \in \textsc{LVar} \rightharpoonup 2^{\textsc{Sys}}$ from variables to sets of processes, thereby permitting an inductive definition on the structure of the formula φ. The formula **tt** is satisfied by all processes, while **ff** satisfied by none; conjunctions and disjunctions bear the standard set-theoretic meaning of intersection and union. Possibility formulae $\langle\alpha\rangle\,\varphi$ refer to processes that ought to have *at least* one of their α-derivates satisfy φ, whereas necessity $[\alpha]\,\varphi$ formulae describe processes that require all (possibly none) of their α-derivates to satisfy φ.

The recursive formulae $\mathbf{min}\, X.\varphi$ and $\mathbf{max}\, X.\varphi$ are satisfied by the least and largest set of processes satisfying φ [37]. Intuitively, minimal fixpoints refer to solutions where formulae are assumed to hold false unless proven true, and dually, maximal fixpoints are associated with solutions where formulae are assumed to hold true unless proven false. The semantics of recursive variables X w.r.t. an

environment instance ρ is given by the mapping of X in ρ, *i.e.,* the set of processes associated with X. *Closed* formulae (*i.e.,* formulae containing no free variables) are interpreted independently of the environment ρ, and the shorthand $[\![\varphi]\!]$ is used to denote $[\![\varphi, \rho]\!]$, *i.e.,* the set of processes in Sys that satisfy φ. In view of this, a process p satisfies some closed formula φ whenever $p \in [\![\varphi]\!]$, and conversely, violates φ if $p \notin [\![\varphi]\!]$.

Example 2.3.1. The μHML formula $\langle\alpha\rangle\,\mathbf{tt}$ describes processes that afford action α, while $[\alpha]\,\mathbf{ff}$, processes that do not afford action α.

$$\varphi_1 = \mathbf{max}\, X.\Big([\mathtt{req}]\,([\mathtt{resp}]\,X \wedge [\mathtt{resp}]\,[\mathtt{resp}]\,\mathbf{ff})\Big)$$
$$\varphi_2 = \mathbf{min}\, X.(\langle\mathtt{req}\rangle\,\langle\mathtt{resp}\rangle\,X \vee \langle\mathtt{lim}\rangle\,\mathbf{tt})$$

The *safety* formula φ_1 denotes a property describing processes that cannot issue duplicate responses in answer to client requests; process q from Example 2.2.1 violates φ_1 via the trace $(\mathtt{req}.\mathtt{resp})^+.\mathtt{resp}$. The *co-safety* formula φ_2 on the other hand, requires that processes, after a number (possibly zero) of request and response interactions, reach a service limit; process p in Example 2.2.1 satisfies φ_2 through the trace $(\mathtt{req}.\mathtt{resp})^*.\mathtt{lim}$. ■

2.3.2 Monitoring μHML

Section 2.1 established that RV monitors analyse single execution traces, whereas other techniques, such as model checking are afforded a view of the entire program execution graph. Despite this restricted view of the system, RV can be still effectively applied in cases where correctness properties describe *individual* executions [1, 23], because a *single witness* execution trace suffices to determine (at runtime) whether the property requirements have been met (see [21] for more details).

Intuitively, formulae of the form $\langle\alpha\rangle\,\varphi$ state that it is *possible* for some process to perform action α and thereby, satisfy property φ. Given this requirement, finding *one positive witness* execution trace is all that is needed to conclude whether a

satisfaction is possible. Dually, formulae of the form $[\alpha]\,\varphi$ state that *all* α-actions performed by some process will satisfy property φ. To detect a violation of this requirement, finding *one negative witness* execution trace is all that is needed to show that property φ is infringed.

Example 2.3.2. The simple co-safety formula φ_3 below describes the possibility of *"a process affording a* `lim`*-action"*:

$$\varphi_3 = \langle \texttt{lim} \rangle\, \textbf{tt}$$

Process p in Figure 2.3a, can, amongst others, produce the following two execution traces, chosen for elucidative purposes:

$$t_1 = \texttt{lim}.\textbf{nil} \qquad\qquad t_2 = \texttt{rec}.\texttt{resp}$$

The first trace t_1 clearly satisfies formula φ_3, whereas when looking at the second trace t_2, one cannot determine whether φ_3 is actually satisfied, simply because the trace does not contain *enough information* to allow the monitor to make such a judgement. Runtime monitors may choose to label the latter monitoring outcome as being *inconclusive*. ■

Single trace access *does* however limit the applicability of RV in cases involving correctness properties describing *complete* or *branching* executions. Consequently, not all properties turn out to be *runtime monitorable*, as in these cases, monitors cannot yield a verdict at *runtime* based on the observation of a *single* execution trace.

Monitorable Logic Syntax

$$\psi \in \text{mHML} \stackrel{\text{def}}{=} \text{sHML} \cup \text{cHML} \text{ where:}$$

$$\begin{array}{llllllll} \theta, \vartheta \in \text{sHML} ::= & \mathbf{tt} & \mid \mathbf{ff} & \mid \theta \wedge \vartheta & \mid [\alpha]\,\theta & \mid \mathbf{max}\, X.\theta & \mid X \\ \pi, \varpi \in \text{cHML} ::= & \mathbf{tt} & \mid \mathbf{ff} & \mid \pi \vee \varpi & \mid \langle\alpha\rangle\,\pi & \mid \mathbf{min}\, X.\pi & \mid X \end{array}$$

Figure 2.5: The syntax of mHML.

Example 2.3.3. The formula φ_4 states the requirement that *"whenever a* ***req****-action is observed, at least one* ***resp****-action follows"*:

$$\varphi_4 = [\texttt{req}]\,\langle\texttt{resp}\rangle\,\mathbf{tt}$$

Interpreting formula φ_4 over the LTS for process q from Figure 2.3b immediately establishes that φ_4 holds. Such a fact cannot be however gleaned by investigating the execution traces for process q *individually*, as done in Example 2.3.2, because *no* single trace will contain enough evidence to enable the monitor to come to a decisive verdict at *runtime*. This stems from the fact that φ_4 talks about multiple execution paths, as opposed to φ_3, that considers just one execution path. Since the monitor's visibility is limited to a single runtime execution, properties describing multiple executions are in general, not monitorable, and in these cases, a verdict can be given only if the process execution graph is considered in its entirety (see [21] for details). ■

The work in [21] explores the monitorability limits of μHML, identifies a syntactic logical subset called mHML, and shows it to be monitorable and maximally-expressive w.r.t. the constraints of runtime monitoring. Its syntax, given in Figure 2.5, consists of two syntactic classes, Safety HML (sHML) describing *invariant* properties, and Co-Safety HML (cHML), describing properties that hold *eventually* after a *finite* number of events. Formulae φ_1 and φ_2 from Example 2.3.1 are instances of sHML and cHML specifications respectively.

2.4 Erlang

Erlang is a general-purpose, concurrent, functional programming language that facilitates the development of fault-tolerant and distributed systems [5, 12, 24]. It is also considered a soft real-time platform thanks to its lightweight process management, per-process garbage collection and pre-emptive scheduling algorithm.

Erlang adopts the actor model for concurrency, which it uses as the primary means to structure applications. An *actor* is a concurrency primitive that represents a processing entity sharing no mutable memory with other actors. It communicates with its environment exclusively via messages, and changes its internal state based on messages received from other actors. The actor model adopts the philosophy that *everything* is an actor, and encourages finely-grained computational models, which in turn, exhibit high degrees of robustness and distribution [12]. Highly granular designs are made possible since Erlang offers inherent language-level support for working with processes, and tasks like process creation and scheduling are managed at the Virtual Machine (VM)-level, making process handling very efficient.

Interprocess communication in Erlang works via asynchronous message passing. Each process owns a message queue, known as a *mailbox*, to which messages from other processes can be sent in a *non-blocking*, *fire-and-forget* fashion. The recipient process may then at any time *selectively* consume messages from its mailbox using Erlang's `receive` construct. Messages are comprised of any Erlang data type, including integers, floats, atoms, functions, binaries, *etc.*.

To manage process failures, Erlang supports a *linking* mechanism that makes it possible to associate processes together. When a failure in some process or group of processes occurs, linked processes are immediately notified, and depending on how these notifications are handled, processes can be restarted, killed or ignored. This process linking scheme not only makes Erlang's fault handling very robust, but also constitutes the basis over which highly resilient applications are built.

Additionally, Erlang provides a framework of software components (known as OTP), that exposes common programming patterns and utility libraries, making

it possible to develop concurrent applications in an easy and standardised manner. More details on this topic can be found in Section 5.1.1.

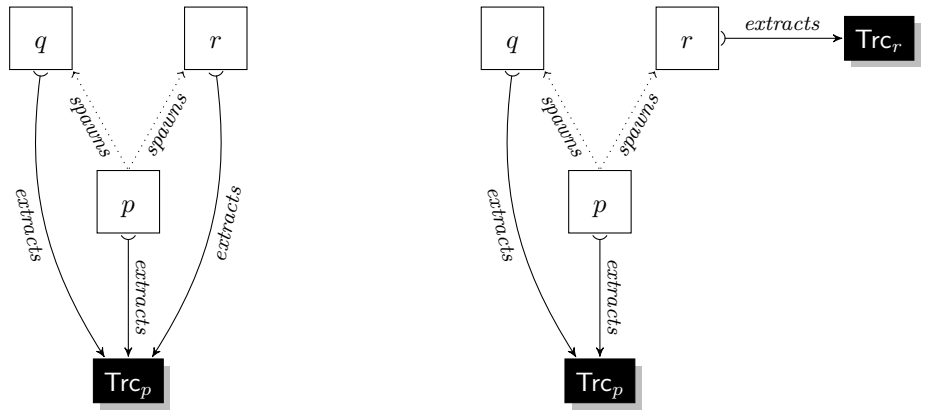
2.4.1 Tracing

Tracing is an Erlang VM mechanism that offers a powerful and flexible means of process observation, making it possible to understand how the system behaves *without modifying* its source code or instrumenting it special ways (*e.g.* unlike other approaches such as Aspect Oriented Programming (AOP) techniques) [5, 12]. Its flexibility stems from the fact that it can be *selectively* applied on specific processes as required, thereby fine tuning the tracing effort to the desired level of granularity. Since Erlang tracing does not rely on instrumentation, it can be switched on and off while the system is executing [24].

Tracing is a low-level platform API exposed through two Built-In Functions (BIFs). The `erlang:trace/3`[2] BIF defines the set of system processes to be traced, by either specifying a Process Identifier (PID) to target one specific process directly, or using other flags such as `all` and `new` to target all or newly spawned processes respectively. It also designates a special *tracer* process to which all trace messages are directed. Conventionally, invoking the `erlang:trace/3` BIF sets the caller as the tracer process, although this behaviour can be changed if required. Tracers cannot trace themselves, and only one tracer is permitted per process. Trace messages take the form of a tagged tuple `{trace, ...}`, and like any other message, are deposited asynchronously inside the tracer's mailbox from where these may be later retrieved using `receive`. Tracing options available to `erlang:trace/3` make it possible to specify the trace events of interest, *e.g.* function calls, message sends and receives, garbage collection triggers, process spawn events, *etc.*.

In addition to the aforementioned tracing options, the `erlang:trace/3` BIF also supports a special `set_on_spawn` flag that allows newly spawned processes to

[2]Erlang functions are grouped into modules. Each function within the module need not be uniquely named, as long as its number of parameters (also known as the *arity*) is different from others with the same name. Functions can be identified using the triple: `mod_name:fun_name/arity`.



(a) Trace messages from all of p, q and r are directed towards the same global tracer Trc_p.

(b) The trace message flow is split and directed to tracer Trc_p for processes p and q, and to Trc_r for process r.

Figure 2.6: Attaching tracers to processes.

inherit their parent process' trace flags. It also directs trace events originating due to these child processes to the same tracer attached to the parent process. Figure 2.6a shows the process-tracer configuration that would have been attained if the *traced* parent process p spawned its children q and r *after* the `set_on_spawn` flag has been defined for tracer Trc_p. As illustrated, all trace messages produced on account of processes p, q and r are directed towards the same tracer Trc_p for process p. If one requires to direct a portion of this *global* trace to a separate tracer, say, for process r, this needs to be *unsubscribed* from tracer Trc_p first, and then, *resubscribed* to another tracer Trc_r intentionally created to handle process r (Figure 2.6b). This subscription-resubscription procedure is required since only one tracer is permitted per process.

While the `erlang:trace/3` BIF specifies *what* processes should be traced, the `erlang:trace_pattern/2` BIF defines *how* this is to be achieved. It can be thought of as a filtering mechanism that determines which trace messages are sent to the tracer process and which are withheld. For instance, while `erlang:trace/3` can be used to specify that some process needs to be traced for function call events, `erlang:trace_pattern/2` can be applied as a refinement to pinpoint one particular function call with a certain combination of argument values.

Tracing serves as the basis for a number of utilities including Erlang's text-based tracing facility `dbg`, and trace tool builder `ttb` [5]. A number of RV tools such as those in [17, 23] and the one developed in Chapter 3 also employ tracing to achieve lightweight and asynchronous trace event extraction. These tackle tracing from a low-level aspect, and therefore use the two BIFs discussed above instead of relying on some high level API such as `dbg`. The reader should note that the tracing mechanism explained in this section differs from the one discussed in Section 2.1.1 in that it does *not* require instrumentation.

2.5 Conclusion

This chapter provided the necessary background required to understand the underlying concepts featured in this work. It introduced the basic notions of RV, together with monitors and the different approaches that can be adopted whilst implementing them. Reactive systems were tackled next, together with a calculus that makes it possible to model them in a formal manner; this served as a foundation upon which the logic μHML and its monitorable subset mHML could be adequately discussed. The topic of Erlang and its native tracing functionality were finally presented with the intent of giving fair insight into how the process of trace event extraction can be achieved without instrumenting the monitored system. As shall be seen in the coming chapters, Erlang's language-level tracing facility provides the underpinning mechanism that enables the development of a global monitoring tool discussed in Chapter 3, and its adaptation to a local scenario in Chapters 4 and 6.

3. A Tool for the Monitorable Subset of μHML*

In the preceding chapter, verification was introduced as a technique that can be used to determine whether a system adheres to its expected specified behaviour. Particular focus was given on how this could be achieved at *runtime* using monitors that continually observe the system as it executes. This chapter discusses the implementation of a RV tool that analyses the correctness of concurrent programs developed in Erlang. It builds on the results in [21], and specifies a synthesis procedure that generates *correct* monitor descriptions from formulae written in mHML. The tool investigates the implementability of this synthesis procedure, instantiating it to generate executable monitors for a specific general-purpose concurrent programming language. This chapter covers the following:

- Extensions to the monitor syntax and semantics in [21], together with a refinement of the synthesis that supports a practical implementation of a mechanism that produces concurrent runtime monitors;
- A discussion on the challenges faced when adapting the refined synthesis function for implementing a tool that targets the Erlang platform.

*The material presented in this chapter has been published in [6].

3.1 Monitor Synthesis

The behaviour of monitors, like that of processes, can be described using a process calculus (see Section 2.2), as shown in Figure 3.1, where $\alpha.m \xrightarrow{\alpha} m$ denotes a monitor in state $\alpha.m$ analysing event α and transitioning to state m. Verdicts v, persistent states that do not change when events are analysed, model the *irrevocability* of monitor verdicts, as per the rule MVER. Recursion is permitted by virtue of MREC. Action τ represents internal transitions, whereas μ ranges over α and τ.

The monitor semantics in Figure 3.1 prohibit the use of the choice construct (see Example 2.2.1), as this allows monitors to behave non-deterministically given an event μ (see Appendix A for details). This problem stems from a limitation in the choice construct semantics as used in [21], $m + n$, which *forces* a selection between submonitor m or n upon the receipt of an event, depending on whether $m \xrightarrow{\mu} m'$ or $n \xrightarrow{\mu} n'$. Said shortcoming is addressed by replacing external choice constructs with a parallel monitor composition construct, $m \times n$ that allows *both* submonitors to process the event without excluding one another.

The semantics of this combinator, also given in Figure 3.1, is defined by the bot-

Syntax

$$\begin{array}{llllllll} m, n \in \text{MON} ::= & v & | & \alpha.m & | & m \times n & | & \textbf{rec}\, x.m \quad | \quad x \\ v, u \in \text{VERD} ::= & \textbf{no} & | & \textbf{yes} & | & \textbf{end} & & \end{array}$$

Dynamic behaviour

$$\text{MACT}\ \frac{}{\alpha.m \xrightarrow{\alpha} m} \qquad \text{MREC}\ \frac{}{\textbf{rec}\, x.m \xrightarrow{\tau} m[{}^{\textbf{rec}\, x.m}/_{x}]} \qquad \text{MVER}\ \frac{}{v \xrightarrow{\alpha} v}$$

$$\text{MPAR}\ \frac{m \xrightarrow{\alpha} m' \quad n \xrightarrow{\alpha} n'}{m \times n \xrightarrow{\alpha} m' \times n'} \qquad \text{MPARL}\ \frac{m \xrightarrow{\alpha} m' \quad n \not\xrightarrow{\alpha} \quad n \not\xrightarrow{\tau}}{m \times n \xrightarrow{\alpha} m'}$$

$$\text{MPARSR}\ \frac{n \xrightarrow{\tau} n'}{m \times n \xrightarrow{\tau} m \times n'} \qquad \text{MPARVL}\ \frac{}{v \times n \xrightarrow{\tau} v}$$

Figure 3.1: The syntax and dynamics of monitors (adapted and extended from [21]).

tom four rules (the symmetric cases mParR, mParSL and mParVR are omitted). Rule mPar states that both monitors proceed in lockstep if they can process the *same* action. The next rule mParL states that if only one monitor can process the action and the other is stuck (*i.e.,* it can neither analyse action μ, nor transition internally using τ), then the able monitor transitions while terminating the stuck monitor; otherwise, the monitor is allowed to transition *silently* by mParSR. Lastly, rule mParVL terminates parallel monitors once a verdict is reached.

The synthesis function $[\![-]\!]$, that maps mHML formulae to monitors is given in Figure 3.2. It constitutes a refinement upon the original function from [21] in that it accommodates the parallel monitor composition construct $\times$ (see Appendix A for details). This instantiation follows closely the procedure described in [21], thereby giving high assurances that the generated executable monitors are indeed correct. Although the function covers both sHML and cHML, the syntactic constraints of mHML mean that synthesis for a formula ψ uses at most the first row (*i.e.,* the logical constructs common to sHML and cHML) and then, either the first column (in the case of sHML) or the second column (in case of cHML). It is worth noting that the monitor synthesis function is compositional w.r.t. the structure of the formula, *e.g.* the monitor for $\psi_1 \wedge \psi_2$ is defined in terms of the submonitors for subformulae ψ_1 and ψ_2. One should also highlight the fact that conditional cases used in the synthesis of necessity and possibility formulae, conjunctions, disjunctions,

$$[\![\mathbf{ff}]\!] \stackrel{\text{def}}{=} \mathbf{no} \qquad [\![\mathbf{tt}]\!] \stackrel{\text{def}}{=} \mathbf{yes} \qquad [\![X]\!] \stackrel{\text{def}}{=} x$$

$$[\![[\alpha]\,\psi]\!] \stackrel{\text{def}}{=} \begin{cases} \alpha.[\![\psi]\!] & \text{if } [\![\psi]\!] \neq \mathbf{yes} \\ \mathbf{yes} & \text{otherwise} \end{cases} \qquad [\![\langle\alpha\rangle\,\psi]\!] \stackrel{\text{def}}{=} \begin{cases} \alpha.[\![\psi]\!] & \text{if } [\![\psi]\!] \neq \mathbf{no} \\ \mathbf{no} & \text{otherwise} \end{cases}$$

$$[\![\psi_1 \wedge \psi_2]\!] \stackrel{\text{def}}{=} \begin{cases} [\![\psi_1]\!] & \text{if } [\![\psi_2]\!] = \mathbf{yes} \\ [\![\psi_2]\!] & \text{if } [\![\psi_1]\!] = \mathbf{yes} \\ [\![\psi_1]\!] \times [\![\psi_2]\!] & \text{otherwise} \end{cases} \qquad [\![\psi_1 \vee \psi_2]\!] \stackrel{\text{def}}{=} \begin{cases} [\![\psi_1]\!] & \text{if } [\![\psi_2]\!] = \mathbf{no} \\ [\![\psi_2]\!] & \text{if } [\![\psi_1]\!] = \mathbf{no} \\ [\![\psi_1]\!] \times [\![\psi_2]\!] & \text{otherwise} \end{cases}$$

$$[\![\mathbf{max}\, X.\psi]\!] \stackrel{\text{def}}{=} \begin{cases} \mathbf{rec}\, x.[\![\psi]\!] & \text{if } [\![\psi]\!] \neq \mathbf{yes} \\ \mathbf{yes} & \text{otherwise} \end{cases} \qquad [\![\mathbf{min}\, X.\psi]\!] \stackrel{\text{def}}{=} \begin{cases} \mathbf{rec}\, x.[\![\psi]\!] & \text{if } [\![\psi]\!] \neq \mathbf{no} \\ \mathbf{no} & \text{otherwise} \end{cases}$$

Figure 3.2: The monitor synthesis function (adapted and refined from [21]).

and maximal and minimal fixpoints are necessary to handle logically equivalent formulae and generate correct monitors (see [21] for details).

Example 3.1.1. The sHML formula φ_1 from Example 2.3.1 (restated below) describes the property stating that *"after any sequence of requests and responses, a request is never followed by two consecutive responses"*. The synthesis function in Figure 3.2 translates φ_1 to the monitor process m_1.

$$\varphi_1 = \mathbf{max}\, X.\Big([\mathtt{req}]\,([\mathtt{resp}]\,X \wedge [\mathtt{resp}]\,[\mathtt{resp}]\,\mathbf{ff})\Big)$$
$$m_1 = \mathbf{rec}\, x.\Big(\mathtt{req}.(\mathtt{resp}.x \;\times\; \mathtt{resp}.\mathtt{resp}.\mathbf{no})\Big)$$

The cHML formula φ_5 describes a property where *"after a (finite) sequence of requests and responses, the system reaches a service limit* `lim`*"*. The subformula $\mathbf{min}\, Y.\mathbf{ff} \vee \langle \mathtt{lim} \rangle\, \mathbf{ff}$ is semantically equivalent to $\mathbf{ff}$; accordingly the side conditions in Figure 3.2 take this into consideration when synthesising monitor m_5.

$$\varphi_5 = \mathbf{min}\, X.\Big(\langle \mathtt{req} \rangle\, \langle \mathtt{resp} \rangle\, X \vee \langle \mathtt{lim} \rangle\, \mathbf{tt} \vee (\mathbf{min}\, Y.\mathbf{ff} \vee \langle \mathtt{lim} \rangle\, \mathbf{ff})\Big)$$
$$m_5 = \mathbf{rec}\, x.(\mathtt{req}.\mathtt{resp}.x \;\times\; \mathtt{lim}.\mathbf{yes})$$

■

The reader's attention should be drawn to the fact that while the synthesis employs both acceptance and rejection verdicts, it generates *uni-verdict* monitors that only produce acceptances or rejections, *never both*; [21] shows that this is essential for monitor correctness.

3.2 Implementation

The RV tool implemented in this work analyses the correctness of concurrent programs developed in Erlang. It exploits the compositional structure of the synthesis shown in Figure 3.2, so as to enable it to produce *concurrent* monitors wherein (sub)monitors autonomously analyse individual parts of the global specification

formula while still guaranteeing the correctness of the overall monitoring process. The discussion that follows shows how these concurrent components can be naturally mapped to Erlang actors [5, 12] that monitor a running system with minimal instrumentation efforts.

3.2.1 Pattern Matching

Actions, in the form of Erlang trace events, consist of two types: (i) outputs $i\,\mathbf{!}\,d$ and inputs $i\,\mathbf{?}\,d$, where i corresponds to actor PIDs , and d denotes the data payload associated with the action in the form of Erlang data values (*e.g.* PID, lists, tuples, atoms, *etc.*), and (ii) termination events denoted by $i\,\mathbf{stp}\,r$ that occur on account of some process i terminating with reason r (*e.g.* `killed`, `normal`, *etc.*). Specifications, defined as instantiations of mHML terms, make use of *action patterns* which possess the same structure as that of the aforementioned actions, but may also employ variables (alphanumeric identifiers starting with an upper-case letter) in place of values; these are then bound to values when pattern-matched to actions at runtime.

Example 3.2.1. A simple client-server setup consists of a *successor* server *Srv* that adds 1 to any numeric payload it receives from clients *Clt*. One safety property that ensures that clients *do not* receive the same value *Num* sent by them to the server, can be expressed using the following sHML formula:

$$\varphi_6 = [Srv\,\mathbf{?}\,\{\texttt{req},\,Clt,\,Num\}]\,[Clt\,\mathbf{!}\,\{\texttt{resp},\,Num\}]\,\mathbf{ff}$$

where *Srv*, *Clt* and *Num* correspond to Erlang variables that bind at runtime. The formula matches trace executions where the server receives a request with payload *Num* from some client, and replies with the same value of *Num*. For instance, an *input* trace event that binds *Srv* to PID `<9.0.3>`, *Clt* to PID `<1.1.6>` and *Num* to `19` yields the runtime binding $[\texttt{<9.0.3>}\ \mathbf{?}\ \{\texttt{req,<1.1.6>,19}\}]\,[\texttt{<1.1.6>}\ \mathbf{!}\ \{\texttt{resp,19}\}]\,\mathbf{ff}$ that will match a subsequent *output* trace event *only if* the client with PID `<1.1.6>` receives a value of `19` in its response payload. ■

Action patterns require the synthesis of a slightly more general form of monitors (*i.e.,* from the ones in Figure 3.1) with the following behaviour: if a pattern e matches a trace event action α, thereby binding a variable list to values from α (denoted by σ), the monitor evolves to the continuation m, substituting the variables in m for the values bound by pattern e (denoted by $m\sigma$); otherwise it transitions to the terminated process **end**.

$$\frac{\textbf{match}(e,\, \alpha) = \sigma}{e.m \xrightarrow{\alpha} m\sigma} \qquad \frac{\textbf{match}(e,\, \alpha) = \bot}{e.m \xrightarrow{\alpha} \textbf{end}}$$

Pattern matching is advantageous because it makes it possible to specify dynamic properties that reason on data values that can only be known at runtime. There might be cases where the subject process i must be exclusively targeted. For instance, the safety property in Example 3.2.1 is allowed to bind the variable *Srv* to values obtained from *any* trace event that can be successfully pattern-matched to $[Srv \texttt{ ? } \{\texttt{req, } Clt\texttt{, } Num\}]$. Preceding the process variable i with the pre-binding operator `@` (*e.g.* `@`*Srv*) instructs the monitor to bind i to its runtime PID *before* any events from the trace are processed; this permits trace matching to be performed against *pre-populated* data values that are fixed throughout the monitoring process. For this facility to be applicable however, processes must be registered with Erlang's process registry (*i.e.,* be *named* processes) in order to make it possible for the monitor to dynamically infer PIDs at runtime based on process names.

3.2.2 Asynchronous Monitors

mHML formulae are parsed and synthesised into Erlang code, following closely the synthesis function discussed in Section 3.1. In particular, the inherent concurrency features offered by Erlang, together with the modular structure of the synthesis are exploited so as to translate submonitors into independent concurrent *actors* that execute in *parallel*. An important deviation from the semantics of parallel composition specified in Section 3.1 is that actors execute *asynchronously* to one

another. For instance, one submonitor may be analysing the second action event, whereas another may forge ahead to a stage where it is analysing the fourth event. In order to ensure that submonitors have access to the same trace events, the (parent) monitor to which the submonitors are attached *forks* (*i.e.,* replicates and forwards) individual trace events to its children. The moment a verdict is reached by any submonitor actor, all others are terminated, and said verdict is used to declare the final monitoring outcome. This alternative semantics still corresponds to the one given in Section 3.1 for three main reasons: (i) monitors are uni-verdict, and there is no risk that one verdict is reached before another, thereby invalidating or contradicting it; (ii) processing is local to each submonitor and independent of the processing carried out by other submonitors; (iii) verdicts are *irrevocable* and monitors can terminate once an outcome is reached, safe in the knowledge that verdicts, once announced, cannot change.

Monitor recursion unfolding, similar to the work in [23], constitutes another minor departure from the semantics in Section 3.1, as the implementation uses a process environment that maps recursion variables to monitor terms. Erlang code for monitor $\mathbf{rec}\, x.m$ is evaluated by running the code corresponding to the (potentially open) term m (where x is free in m) in an environment with the map $x \mapsto m$.

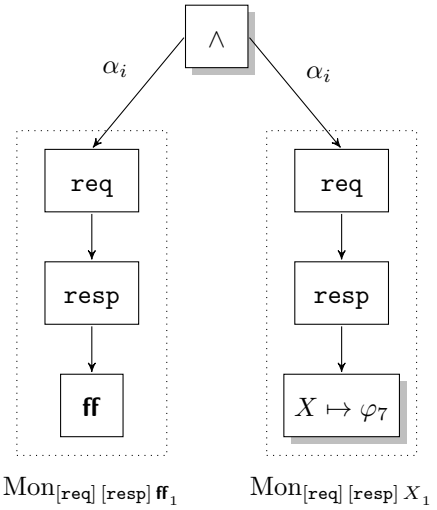
Example 3.2.2. A monitor resulting from the synthesis of formula φ_6 is capable of observing a single client-server interaction before it terminates. To be able to handle continuous monitoring, φ_6 can be reformulated into the following *recursive* (sHML) formula:

$$\varphi_7 = \mathbf{max}\, X.([\mathit{Srv}\,\mathbf{?}\,\{\mathtt{req}, \mathit{Clt}, \mathit{Num}\}]\,[\mathit{Clt}\,\mathbf{!}\,\{\mathtt{resp}, \mathit{Num}\}]\,\mathbf{ff} \wedge [\mathit{Srv}\,\mathbf{?}\,\{\mathtt{req}, \mathit{Clt}, \mathit{Num}\}]\,[\mathit{Clt}\,\mathbf{!}\,\{\mathtt{resp}, \mathit{Succ}\}]\,X)$$

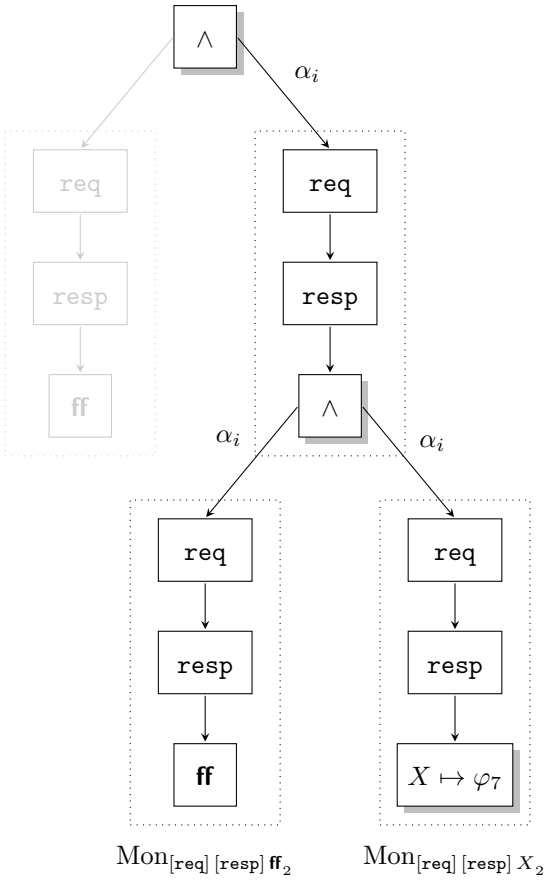
The concurrent compositional monitor resulting from φ_7 is shown in Figure 3.3a. It consists of three processes: the "conjunction monitor" (corresponding to $\wedge$), and its two submonitor actors, each of which monitors independently for subfor-

mulae $[Srv\ \texttt{?}\ \{\texttt{req}, Clt, Num\}]\,[Clt\ \texttt{!}\ \{\texttt{resp}, Num\}]\,\mathbf{ff}$ (*i.e.,* the verdict branch) and $[Srv\ \texttt{?}\ \{\texttt{req}, Clt, Num\}]\,[Clt\ \texttt{!}\ \{\texttt{resp}, Succ\}]\,X$ (*i.e.,* the recursive branch) respectively by observing trace events α_i forked by their parent. The verdict branch incrementally matches events that lead to a violation (resp. satisfaction); the recursive branch matches non-violation (resp. non-satisfaction) events that permit the monitor structure to unfold *lazily*.

Unfolding works by virtue of the *guarded* recursion variable X acting as a *placeholder* for the monitor it maps to in the process environment. When the recursive variable is evaluated, the monitor unfolds and spawns a new copy of itself: this consists of a submonitor arrangement that is *linked* to its parent for error



(a) The monitor tree prior to unfolding.



(b) The monitor tree after the first recursive unfolding takes place.

Figure 3.3: The recursive unfolding of compositional monitors.

handling purposes (see process linking in Section 2.4). This unfolding procedure is illustrated in Figure 3.3b, where the left greyed-out branch denotes a terminated (top-level) submonitor that did not match its expected execution trace pattern. The reader should note that recursion steadily diminishes the monitor's efficiency every time an unfolding takes place, due to the number of extra processes created for each newly cloned monitor instance. ■

3.2.3 Monitor Compilation

Figure 3.4 outlines the compilation steps required to transform a formula script file (`script.hml`) into a corresponding Erlang source code monitor implementation (`monitor.erl`). To adhere to the compositional synthesis of Section 3.1, the tool had to overcome an obstacle attributed to pattern bindings. Specifically, in formulae such as $[e]\,\psi$ or $\langle e\rangle\,\psi$, subformula ψ may contain free (value) variables bound by the pattern e. For instance, in $[\mathit{Srv}\ \texttt{?}\ \{\texttt{req},\ \mathit{Clt},\ \ldots\}]\,[\mathit{Clt}\ \texttt{!}\ \{\texttt{resp},\ \ldots\}]\,\textbf{ff}$, the Erlang monitor code for the subformula $[\mathit{Clt}\ \texttt{!}\ \{\texttt{resp},\ \ldots\}]\,\textbf{ff}$ would contain the free variable *Clt* bound by the preceding pattern $[\mathit{Srv}\ \texttt{?}\ \{\texttt{req},\ \mathit{Clt},\ \ldots\}]$ (refer to Example 3.2.1). Since Erlang does *not* support dynamic scoping [12], the synthesis cannot simply generate open functions whose free variables are then bound dynamically at the program location where the function is used. To circumvent this issue, the synthesis generates an *uninterpreted* source code string composed using the `util:format()` string manipulation function. Compilation is then handled

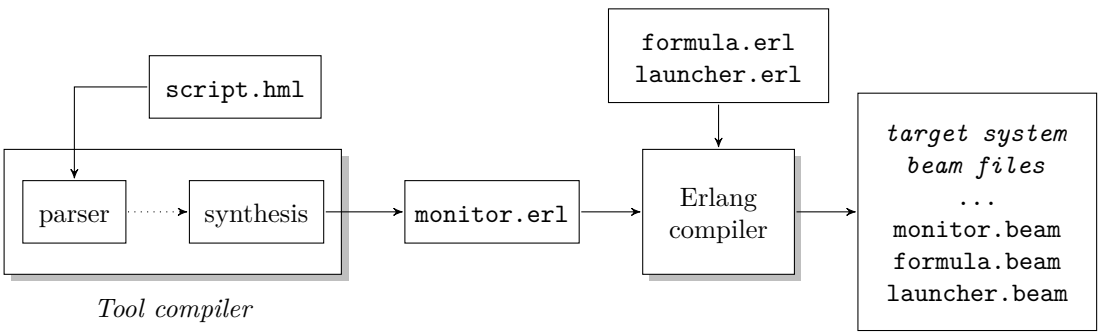


Figure 3.4: The monitor synthesis process pipeline.

normally (using the static scoping of the *completed* monitor source code) via the standard Erlang compiler.

The tool is organised into two main modules. The synthesis process shown in Figure 3.4 is carried out by the function `synth` in module `compiler.erl`. This relies on generic monitor constructs implemented as function macros inside the module `formula.erl`. Table 3.1 outlines the mapping for two of these constructs (the rest can be found in Appendix B). Parallel composition is encoded by spawning two parallel actors (lines 3 - 4) followed by forking trace events to these actors for independent processing (line 5). Action prefixing for pattern e is encoded by generating a pattern-and-continuation specific function `ActMatcher` that takes a trace event `Act`, pattern-matches it with the translation of pattern e (line 10) and executes the continuation monitor returned by `ActMatcher` in case of a successful match (line 11). One should note that the execution of a monitor *always* takes a map environment `Env` as argument.

The function `synth` in module `compiler.erl` consumes the formula parse-tree (encoded as Erlang tuples), generates the Erlang source code string of the respective monitor and writes it to `monitor.erl`. Table 3.2 outlines the tight correspondence between this compilation and the synthesis function from Section 3.1. To encode

Monitor construct	`formula.erl` module code
$[\![\psi_1]\!] \times [\![\psi_2]\!]$	1 `mon_par(Psi1, Psi2) ->` 2 `  fun(Env) ->` 3 `  Pid1 = spawn_link(fun() -> Psi1(Env) end),` 4 `  Pid2 = spawn_link(fun() -> Psi2(Env) end),` 5 `  fork(Pid1, Pid2)` 6 `end.`
$e.[\![\psi]\!]$	7 `mon_cnt(ActMatcher) ->` 8 `  fun(Env) ->` 9 `  receive Act ->` 10 `  Psi = ActMatcher(Act),` 11 `  Psi(Env)` 12 `  end` 13 `  end.`

Table 3.1: The monitor constructs and their corresponding Erlang code.

the branching cases of the synthesis function, the compilation returns a *tuple* where the first element is a *tag* ranging over `yes`, `no` and `any`, and the second element, the monitor source code string. The correspondence is evident for $[\![\psi_1 \wedge \psi_2]\!]$, where the code on lines 6 - 7 performs the necessary string processing and calls the function `mon_par` presented in Table 3.1. For formula $[\![[e]\,\psi]\!]$, the translation inserts directly the function corresponding to `ActMatcher` (lines 14 - 16) alluded to in Table 3.1 — this is passed as an argument to `mon_cnt` from `formula.erl` (line 17), thereby addressing the aforementioned limitation associated with open functions and dynamic scoping. Pattern `Pat` is extracted from the parse tree (line 9), while the continuation monitor source code string `Mon` is synthesised from the subtree of `Psi` (line 10). See Appendix D.0.4 for an example.

The tool instruments the generated monitors to run with the system in asynchronous fashion, using the native tracing functionality provided by the Erlang VM. Erlang trace BIFs (see Section 2.4.1) instruct the VM to report events of

Synthesis function subcase	`compiler.erl` module function
$[\![\psi_1 \wedge \psi_2]\!] \stackrel{\text{def}}{=}$ $\begin{cases} [\![\psi_1]\!] & \text{if } [\![\psi_2]\!] = \textbf{yes} \\ [\![\psi_2]\!] & \text{if } [\![\psi_1]\!] = \textbf{yes} \\ [\![\psi_1]\!] \times [\![\psi_2]\!] & \text{otherwise} \end{cases}$	1 `synth({and_op, Psi1, Psi2}) ->` 2 `  case {synth(Psi1), synth(Psi2)} of` 3 `  {{Tag, Mon}, {yes, _}} -> {Tag, Mon};` 4 `  {{yes, _}, {Tag, Mon}} -> {Tag, Mon};` 5 `  {{Tag1, Mon1}, {Tag2, Mon2}} ->` 6 `  {any, util:format("mon_par(~s,~s)",` 7 `  [Mon1, Mon2])}` 8 `  end;`
$[\![[\alpha]\,\psi]\!] \stackrel{\text{def}}{=}$ $\begin{cases} \alpha.[\![\psi]\!] & \text{if } [\![\psi]\!] \neq \textbf{yes} \\ \textbf{yes} & \text{otherwise} \end{cases}$	9 `synth({nec, Pat, Psi}) ->` 10 `  case synth(Psi) of` 11 `  {yes, _} -> {yes, "mon_tt()"};` 12 `  {Tag, Mon} ->` 13 `  Fun = util:format(` 14 `  "fun(Act) -> case Act of ~s -> ~s;` 15 `  _ -> mon_end() end end",` 16 `  [pat_to_str(Pat), Mon]),` 17 `  {any, util:format("mon_cnt(~s)", [Fun])}` 18 `  end;`

Table 3.2: The monitor synthesis function cases and their corresponding compiler functions.

interest from the system execution to a *tracer* actor executing in parallel; this in turn forwards said events to the monitor (also executing in parallel). Crucially, this type of instrumentation requires *no changes to the monitor source code* (or the target system binaries) increasing confidence of its correctness. In the tool, compiled monitor files together with their dependencies (*e.g.* `formula.erl`) are placed alongside other system binary files. Instrumentation is then handled by a third module, `launcher.erl`, tasked with the responsibility of launching the system and corresponding monitors in tandem.

3.3 Conclusion

This chapter discussed the implementation of a prototype tool that synthesises and instruments asynchronous monitors from specifications written in mHML, a monitorable subset of the logic μHML. A number of refinements were made to the synthesis function from [21] in order to adapt the procedure to generate concurrent and asynchronous runtime monitors for Erlang. These refinements also deal with practical (*e.g.* using Erlang native tracing) and flexibility (*e.g.* pattern-matching) considerations that make the tool applicable in realistic settings. Despite these adaptations, the resulting implementation corresponds tightly to the correct monitor synthesis specification described in [21], thereby giving high assurances that the executable monitors generated by the tool are also correct.

4. Local Monitoring

The prototype implementation presented in the previous chapter tackles monitoring from a global standpoint. Chapter 1 outlined the shortcomings that accompany global monitoring in a concurrent setting, and alluded to the benefits that can be reaped if local monitoring is used instead. This chapter extends the concepts behind the tool developed in Chapter 3, and investigates how employing a local, component-oriented approach makes it possible to consider the system from a granular level, thus focusing the monitoring effort on individual system components. The following topics are covered:

- The general idea behind local monitoring;
- The challenges faced when implementing a *generic* monitoring tool that supports local and global monitoring;
- An investigation of how local monitoring can be applied in practice on two different *static* system setups;
- A qualitative comparison of local and global monitoring on the basis of *understandability*, *maintainability*, *expressivity*, and *fault tolerance*;
- A quantitative investigation of the *performance* benefits attributed to local monitoring when compared against its global counterpart.

The chapter concludes by restating the five assessment criteria from Section 1.2.1 and presenting the final findings achieved by the study in the context of these criteria.

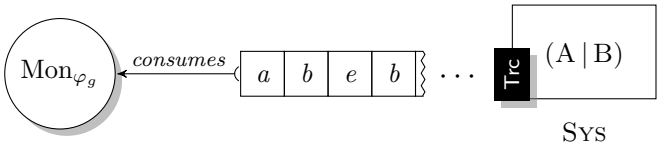
4.1 An Overview of Local Monitoring

Local monitoring is, in its very essence an instantiation of global monitoring that is applied *selectively* to different system components. Selective application focuses the monitoring effort locally, and critically, distinguishes it from global monitoring in that monitors neither have access, nor are interested in processing the global trace. Rather, locally deployed monitors read and process the subset of relevant trace events by subscribing to a *subtrace* for a single system component. This, coupled with the fact that local monitors do not communicate between themselves, lowers data overheads since monitors perform only rudimentary data handling.

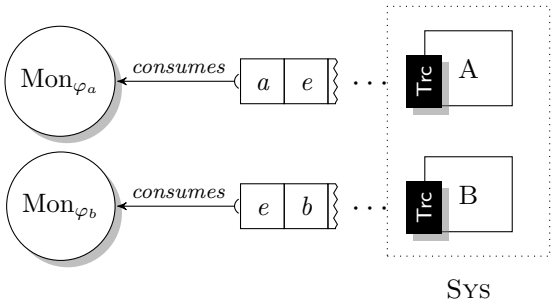
A local approach aids in simplifying the specification of global correctness properties, while in certain specific settings, it might also make it possible to handle particular properties that would be difficult to tackle using a global approach (see Section 4.4.3). Recall the global formula φ_g from Example 1.1.1 and φ_a and φ_b from Example 1.1.2, restated below:

$$\varphi_g = [a]\,\mathbf{ff} \wedge [b]\,\mathbf{ff} \wedge [e]\,([a]\,\mathbf{ff} \wedge [b]\,\mathbf{ff}) \qquad \varphi_a = [a]\,\mathbf{ff} \qquad \varphi_b = [b]\,\mathbf{ff}$$

From a specification aspect, formula φ_g needs to consider all the possible action interleavings resulting from the execution of processes A and B — a disadvantage not incurred by formulae φ_a and φ_b. Localisation is also practical from a synthesis perspective. Large formulae like φ_g are synthesised into monitors composed of an equally large number of processes. This requires events from the global trace to be copied and forwarded to every process within the monitor composition (refer to Section 3.2.2 for details). Handling large numbers of processes in this manner requires considerable amounts of memory and processing power.



(a) Global monitor for formula φ_g consuming action events from the central trace.



(b) Local monitors for formulae φ_a and φ_b consuming action events from individual subtraces.

Figure 4.1: Global and local monitor configurations.

The synthesis process for the three aforementioned formulae φ_g, φ_a and φ_b results in the monitor configurations shown in Figure 4.1. The case for formula φ_g produces a global monitor that processes action events 'a', 'b' and 'e' from the global trace, whereas the case for formulae φ_a and φ_b yields two *local* monitors that *independently* process action events from the subtraces for processes A and B respectively. By virtue of its locally assigned tracer, each local monitor automatically gets access to a subtrace that is *free* from events generated by other system components (*e.g.* the monitor for φ_a never gets action events 'b').

The local monitoring setup in Figure 4.1b can be attained either statically or dynamically. In static scenarios, the number of components to be monitored is known, and the local monitoring configuration can be set-up *before* the target system is loaded. In dynamic scenarios, the system typically scales, and one must consider more flexible ways with which monitors can be created and implanted into the running system. As this work focuses mainly on the effectiveness of local monitoring rather than on how these are actually set up and configured, a static

approach to monitoring is favoured in this chapter as a convenient means of studying local monitoring. Chapter 6 presents an algorithm whereby local monitors are instantiated dynamically at runtime.

4.2 Implementability

A local monitoring solution that concentrates on *static systems* can be implemented in the form of a RV framework, based on the description outlined above. This implementation can, very conveniently, not distinguish between local and global monitoring and consider the latter as a degenerate case of local monitoring with a single component. In order to *correctly* achieve a monitored system, the implementation ensures that tracer processes (shown as Trc in Figure 4.1) are started and registered *before* the target system fully loads. Neglecting to do so can possibly lead to a loss of trace events. This is a direct consequence of asynchronous monitoring, where the system or its subcomponents may start up and continue ahead while the tracers are still not subscribed to their respective subtraces. The problem is mitigated by opting for a *synchronised* loading mechanism which guarantees that any target system component that requires monitoring remains blocked until its tracer is loaded and set up. When this is established, the blocked component is permitted to proceed, and eventually, trace events elicited by the tracer are deposited into the monitor's mailbox in the *same* order as received by the tracer.

Figure 4.2 illustrates the start up procedure when applied to the example system $\textsc{Sys} \stackrel{\text{def}}{=} (\mathrm{A} \mid \mathrm{B} \mid \mathrm{C})$ from Chapter 1. Assuming *only* components A and B are *locally* monitored, Sys is loaded according to the following protocol:

❶ The load specification (`load_spec`) is processed and used to determine the start up order of all of the components listed in it. In this example, the load specification states that components A and B are to be monitored locally. For component A, the monitor synthesised from formula φ_a is used, whereas for B, the monitor from formula φ_b is used. Component C is *not* monitored;

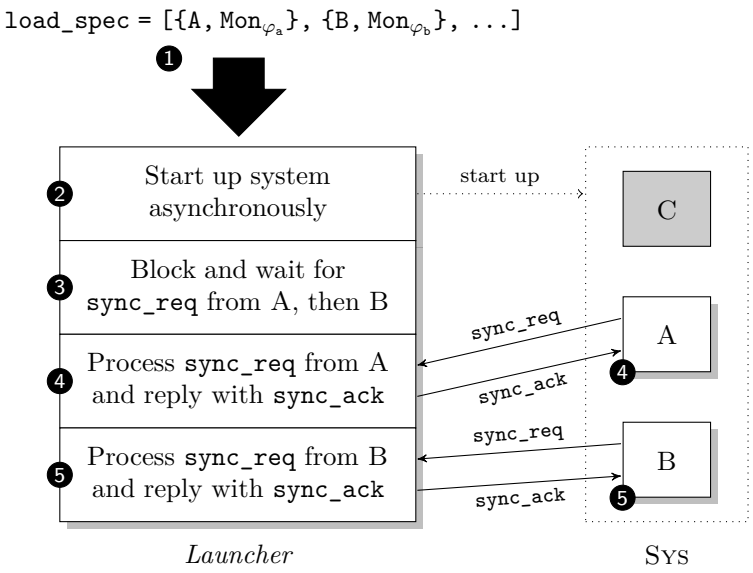


Figure 4.2: Setting up local monitors for components A and B.

❷ All of the components in the target system are *asynchronously* started. Those designated for local monitoring block, and send `sync_req` requests with their PID to the launcher component. Unmonitored components (*i.e.,* C in Figure 4.2) remain unaffected by this procedure and are loaded normally. As the system is fully asynchronous, the launcher makes no assumptions on the order of receipt of the `sync_req` messages;

❸ The launcher blocks its execution and waits for incoming `sync_req` messages, as per loading order specified in `load_spec`. In the scenario depicted in Figure 4.2, the launcher waits for the synchronisation request from component A, followed by the one from B. If perchance, these are received in reverse order, the launcher still ensures that component A is unblocked first, followed by B;

❹ The `sync_req` message from component A is processed and the associated tracer instantiated. If successful, a `sync_ack` acknowledgement is sent back to A, which is now unblocked and continues with its normal execution;

❺ The same happens in the case of component B, resulting in the tracer-monitor arrangement shown in Figure 4.1b.

There are three points that merit the reader's attention. First, the launcher abides by the component start up sequence given in the load specification `load_spec`, *immaterial* of the order in which `sync_req` requests are received. This is made possible through Erlang's *selective message reception* [5], where only messages matching a specific pattern are retrieved from the process mailbox. If no message is pattern-matched, the receiver process simply blocks until a message with a matching pattern is deposited into its mailbox. To illustrate, if say, the message `sync_req` for component B is received first, this is put aside, and unless the message for component A is received, the launcher remains blocked. However, once the message expected from component A is received it is promptly processed, followed by the one stored earlier from component B.

Second, per-component blocking is made possible thanks to source code instructions *instrumented* inside the monitored components. At runtime, these instructions communicate with the launcher process and block the caller (*e.g.* A and B in this discussion) until an acknowledgement message is received in response. This blocking procedure does *not* distinguish between local and global monitors, as it only depends on the number of instrumented system components and the corresponding loading order of each, as specified in `load_spec`. For the case of local monitoring, multiple instrumentations need to be performed, one for each monitored component. Global monitoring requires only a single instrumentation in the top-level system component.

Third, even though the launcher loads the target system components sequentially and synchronously, no linear dependency is assumed between these. This is owed to the simple fact that communication between different concurrent components takes place only via the exchange of *asynchronous* messages, the order of which may be arbitrary, and therefore, *unknown* to the launcher. Any messages that a blocked component cannot presently process are not lost, but queued inside the component's mailbox. These can be subsequently retrieved once the component is eventually unblocked. As a result, the functionality of the monitored system is

never disturbed, but may, in cases, experience a slightly slow start up depending on the number of monitored components that require unblocking.

4.3 The Applicability of Local Monitoring

Local monitoring is difficult to apply in cases where the target system does not lend itself well to being broken down into multiple parts (*e.g.* monolithic systems). Rather, as seen in Section 4.1, local monitoring is more suitably applied to systems that can be easily decomposed into components. The degree of effectiveness of this localisation however, depends on the architectural setup of these components and particularly, on whether said components share some form of interaction or lack thereof.

To illustrate, consider the token issuing system (henceforth referred to as TIS) in Figure 4.3 consisting of three processes: the front-end, and the back-end comprised of two components that share no form of interaction. The hash server is responsible for producing random hash strings, whereas the time server returns the current time packed into an Erlang tuple of the form: `{{yyyy,mm,dd},{hh,mm,ss}}`. To issue one-time tokens to clients connecting over TCP, the front-end combines a random hash string and time value which it obtains by interacting with the back-end servers. Despite its size, the TIS captures the essence of what a typical component-based system looks like, following the setup of a simplistic micro-service architecture. Interprocess communication within the TIS happens according to the protocol defined below:

❶ A new TCP request is accepted by the TIS front-end process and the connected client is put on hold;

❷ *Simultaneous* requests are sent to the hash and time servers in order to speed up the client's service time. The front-end then blocks until replies are received back from *both* back-end servers. Hash server requests take the form of the tagged tuple: `{hash,` *Front-end PID*`}`; time server requests, the

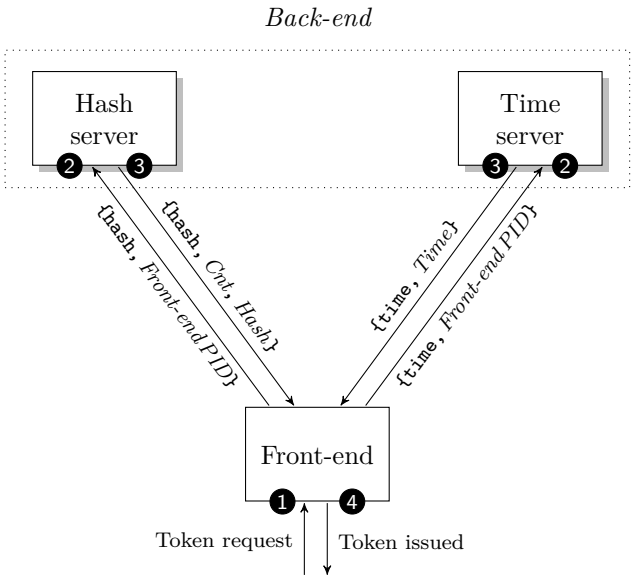


Figure 4.3: The TIS architecture with isolated back-end components.

form: `{time,` *Front-end PID*`}`. No assumption is made on the order in which the hash and time servers handle their respective requests;

❸ Both back-end servers eventually respond back to the TIS front-end. A payload of `{hash,` *Cnt*, *Hash*`}` is returned by the hash server, and of `{time,` *Time*`}` by the time server;

❹ Upon receipt of these messages, the front-end resumes its execution and issues a one-time client token based on the received hash and time values.

As can be gleaned by studying Figure 4.3, if one were to attach a local monitor to each back-end component, this would result in the *combined* (total) processing of four trace events that arise as an outcome of the communication taking place (labelled ❷ and ❸) between the front and back-end components.

Consider now a slightly modified version of the TIS where the random hash code is computed using a seed that is based on a time-stamp obtained from the time server (Figure 4.4). If the same *local* monitoring strategy is applied on this alternate version of the TIS (*i.e.,* a local monitor attached to each back-end component), the combined trace event processing incurred is remarkably higher due to the *added*

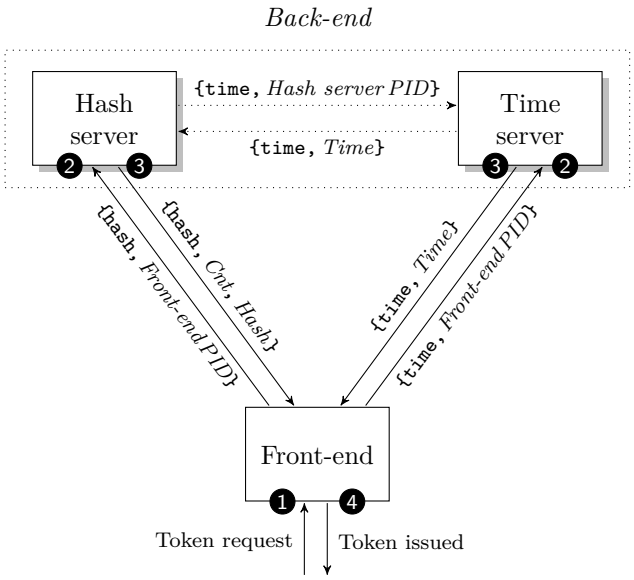


Figure 4.4: The TIS architecture with interacting back-end components.

communication between the back-end components. This stems from the fact that *each* local monitor must now consider four trace events — two more resulting from the mutual interaction between the hash server and time server, in addition to the previous two (*i.e.,* ❷ and ❸) due to the front-end-back-end interaction. At this point, the reader ought to note that the trace events resulting from the shared interaction between the back-end servers are processed *twice*, once by each local monitor. The effectiveness of local monitoring continues to degenerate further as the number of interactions between the back-end components increases, up to the point where this starts to become difficult to manage.

The above discussion points to the fact that local monitoring is expected to be *mostly* effective when applied to scenarios where components preclude all forms of interaction with other components, as is the case in Figure 4.3. Should this not be the case, localisation *may* still be a viable option, but its effectiveness *will* be considerably lower when compared to the former scenario.

The efficacy of local monitoring warrants further study, particularly when compared to that of global monitoring, as this sheds light on the *applicability* of both techniques, and whether one outperforms the other in certain situations. The inves-

tigation presented next in Sections 4.4 and 4.5 conducts this comparative study in a practical fashion *vis-à-vis* the two aforementioned architectures of TIS in Figure 4.3 and Figure 4.4, on the basis of the five assessment criteria listed in Section 1.2.1. Focus is placed on the relationship that exists between the two monitoring approaches, as this makes it possible to *qualify* the benefits that may be gained if local monitoring is used instead of global monitoring. It helps also to better *assess* the earlier claim that local monitoring is mostly effective when applied to scenarios where components share no means of common interaction.

4.4 A Qualitative Study

The first part of this study investigates local and global monitoring from a purely *qualitative* aspect, and concentrates on how properties on Erlang systems can be specified using the formalism introduced in Chapter 2. Property specifications are judged in terms of how well these can be written and understood, maintained and extended. The study also asserts whether failures in system components adversely affect the overall monitors' capability to function properly. To fully appreciate the content that is presented next, the reader is encouraged to refer to Figures 4.3 and 4.4 while reading through. Protocol interaction numbers (*e.g.* ❶) are used to put formulae into the context of the TIS protocol flow outlined earlier in Section 4.3.

4.4.1 Understandability

As seen in Section 4.3, the complexity of local correctness formulae depends *also* on the way in which system components are set up. Local formulae targeted towards non-communicating components are the easiest to specify, because of the limited number of interactions that need to be considered. Communicating components are harder to describe, as formulae must account for *all* the possible interactions that occur between components. These scenarios can be studied by consulting the two versions of the TIS architectures shown in Figures 4.3 and 4.4.

Isolated Components

The TIS back-end in Figure 4.3 can be monitored using two local safety properties that target the hash and time servers separately. A property stating that *"the hash server can never return a hash value equal to the empty string"* can be expressed through the local sHML formula:

$$\begin{array}{l}\mathbf{max}(X,\\ \quad [\mathit{HashSrv}\,\mathbf{?}\,\{\mathtt{hash},\,\mathit{ToknSrv}\}]\ \textcircled{2}\ (\\ \quad\quad [\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{hash},\,\mathit{Cnt},\,\mathtt{""}\}]\,\mathbf{ff}\ \textcircled{3}\ \wedge [\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{hash},\,\mathit{Cnt},\,\mathit{Hash}\}]\,X\ \textcircled{3})\\ )\end{array} \tag{4.1}$$

The Erlang variables *HashSrv* and *ToknSrv* in formula (4.1) bind with the hash server and TIS front-end server PIDs respectively; *Cnt* and *Hash* bind to the request count and hash string returned by the hash server. The interaction that formula (4.1) models starts when the hash server receives a request from *ToknSrv* to generate a new hash code ❷. At the point of answering back ❸, the hash server can either (incorrectly) reply with an empty or a non-empty hash string. The former case is detected by $[\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{hash},\,\mathit{Cnt},\,\mathtt{""}\}]\,\mathbf{ff}$ which promptly flags a violation.

A second safety property *"the time server does not crash"* can be formulated using the following sHML specification:

$$\begin{array}{l}\mathbf{max}(X,\\ \quad [\mathit{TimeSrv}\,\mathbf{stp}\,\mathtt{killed}]\,\mathbf{ff}\wedge\\ \quad [\mathit{TimeSrv}\,\mathbf{?}\,\{\mathtt{time},\,\mathit{ToknSrv}\}]\ \textcircled{2}\ ([\mathit{TimeSrv}\,\mathbf{stp}\,\mathtt{killed}]\,\mathbf{ff}\wedge[\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{time},\,\mathit{Time}\}]\,X\ \textcircled{3})\\ )\end{array} \tag{4.2}$$

The time server process *TimeSrv* can crash at three possible instants: (i) before receiving a request, (ii) immediately after it receives the request, or (iii) once a reply is send back to the sender. Crashes due to cases (i) and (iii) are handled by the first necessity $[\mathit{TimeSrv}\,\mathbf{stp}\,\mathtt{killed}]\,\mathbf{ff}$, which flags a violation when the time

server stops with a reason of `killed`. Case (ii) is handled by a second instance of the same necessity in the last line of formula (4.2).

Communicating Components

The second version of TIS with communicating back-end components (Figure 4.4) can be monitored utilising the same two local safety properties mentioned previously. The first property *"the hash server can never return a hash value equal to the empty string"* now results in the following, marginally larger local sHML formula:

$$
\begin{aligned}
&\mathbf{max}(X,\\
&\quad [\mathit{HashSrv}\,\mathbf{?}\,\{\texttt{hash},\,\mathit{ToknSrv}\}]\ \textcircled{2}\\
&\quad [\mathit{TimeSrv}\,\mathbf{!}\,\{\texttt{time},\,\mathit{HashSrv}\}]\,[\mathit{HashSrv}\,\mathbf{?}\,\{\texttt{time},\,\mathit{Time}\}]\,(\\
&\qquad [\mathit{ToknSrv}\,\mathbf{!}\,\{\texttt{hash},\,\mathit{Cnt},\,\texttt{""}\}]\,\mathbf{ff}\ \textcircled{3}\ \wedge\,[\mathit{ToknSrv}\,\mathbf{!}\,\{\texttt{hash},\,\mathit{Cnt},\,\mathit{Hash}\}]\,X\ \textcircled{3})\\
&\quad)
\end{aligned}
\tag{4.3}
$$

The hash server's interaction changes minimally and no interleaving needs to be considered, as the manner in which the hash server acquires the time from the time server is fixed by the back-end interaction protocol. In fact, the formula changes only slightly from the one in (4.1) to cater for this added communication step.

In contrast, the second property *"the time server does not crash"* has to take into consideration the interaction interleaving introduced on account of the execution order of the TIS front-end and hash server components. To illustrate, suppose the time server receives a request from the TIS front-end first. As the time server handles requests *sequentially*, this request is serviced *before* the next one from the hash server is processed. Yet, it could be the case that while the time server is processing the TIS front-end request, the hash server deposits its own into the time server's mailbox — a thing that *cannot* be prevented as all communication happens asynchronously. This does not disrupt the time server's sequential operation, yet, a properly formulated specification *must* account for this possibility as well (*i.e.,* the hash server's request *trace event* is analysed *before* the one that originates due to

the time server sending its response back to the TIS front-end). The same argument applies if the execution order the TIS front-end and hash server is reversed, *i.e.,* the hash server sends its request first, followed by the one from the TIS front-end. All four cases resulting from said interleaved interaction are handled like so:

$$
\begin{array}{ll}
 & \mathbf{max}(X, \\
 & \quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\ \mathbf{ff} \wedge \\
 & \quad [\mathit{TimeSrv}\ \mathbf{?}\ \{\mathtt{time}, \mathit{ToknSrv}\}]\ ❷\ (\\
 & \qquad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\ \mathbf{ff} \wedge \\
\text{Subcase 1} & \qquad [\mathit{ToknSrv}\ \mathbf{!}\ \{\mathtt{time}, \mathit{Time}\}]\ ❸\ (\\
 & \qquad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\ \mathbf{ff} \wedge \\
 & \qquad\quad [\mathit{TimeSrv}\ \mathbf{?}\ \{\mathtt{time}, \mathit{HashSrv}\}]\ (\\
 & \qquad\qquad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\ \mathbf{ff} \wedge [\mathit{HashSrv}\ \mathbf{!}\ \{\mathtt{time}, \mathit{TimeHash}\}]\ X) \\
 & \qquad)\wedge \\
\text{Subcase 2} & \qquad [\mathit{TimeSrv}\ \mathbf{?}\ \{\mathtt{time}, \mathit{HashSrv}\}]\ (\\
 & \qquad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\ \mathbf{ff} \wedge \\
 & \qquad\quad [\mathit{ToknSrv}\ \mathbf{!}\ \{\mathtt{time}, \mathit{Time}\}]\ ❸\ (\\
 & \qquad\qquad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\ \mathbf{ff} \wedge [\mathit{HashSrv}\ \mathbf{!}\ \{\mathtt{time}, \mathit{TimeHash}\}]\ X) \\
 & \qquad) \\
 &)\wedge
\end{array}
\tag{4.4}
$$

$$
\begin{array}{ll}
 & [\mathit{TimeSrv}\ \mathbf{?}\ \{\mathtt{time},\ \mathit{HashSrv}\}]\,(\\
 & \quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge \\
\text{Subcase 3} & \quad [\mathit{HashSrv}\ \mathbf{!}\ \{\mathtt{time},\ \mathit{TimeHash}\}]\,(\\
 & \quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge \\
 & \quad\quad [\mathit{TimeSrv}\ \mathbf{?}\ \{\mathtt{time},\ \mathit{ToknSrv}\}]\ \textcircled{2}\ (\\
 & \quad\quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge [\mathit{ToknSrv}\ \mathbf{!}\ \{\mathtt{time},\ \mathit{Time}\}]\,X\ \textcircled{3}) \\
 & \quad)\wedge \\
\text{Subcase 4} & \quad [\mathit{TimeSrv}\ \mathbf{?}\ \{\mathtt{time},\ \mathit{ToknSrv}\}]\ \textcircled{2}\ (\\
 & \quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge \\
 & \quad\quad [\mathit{HashSrv}\ \mathbf{!}\ \{\mathtt{time},\ \mathit{TimeHash}\}]\,(\\
 & \quad\quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge [\mathit{ToknSrv}\ \mathbf{!}\ \{\mathtt{time},\ \mathit{Time}\}]\,X\ \textcircled{3}) \\
 & \quad) \\
 &) \\
 &)
\end{array}
\tag{4.5}
$$

Subformula (4.4) handles the first two interleavings that arise when the time server receives a request from the TIS front-end initially ❷. In the first subcase, the time server promptly replies back to the TIS front-end ❸, and subsequently services the request that originates from the hash server. In the second subcase, the time server receives a request from the hash server into its mailbox, but replies to the first request made by the TIS front-end ❸. Following this, a response is also sent back to the hash server in reply to the request deposited into the time server's mailbox.

Subformula (4.5) handles the last pair of interleavings that arise when the time server receives a request from the hash server first. In the third subcase, the time server immediately replies back to the hash server, and subsequently receives a request from the TIS front-end ❷, which it answers with a response ❸. In the fourth subcase, the time server receives a request from the TIS front-end into its mailbox ❷, but nevertheless, replies first to the request made by the hash server.

Then, it replies to the TIS front-end ❸ in answer to the previous request deposited into the time server's mailbox.

The reader should note that in the above formulae, the time server *always* issues replies according to the order in which requests were received; this is due to the *sequential* nature of the time server. The subformula $[\mathit{TimeSrv}\ \mathbf{stp}\ \texttt{killed}]\,\mathbf{ff}$ that determines whether the time server has crashed must be interspersed between each trace event, rendering the resulting correctness formula above *considerably* larger. This cannot be avoided because the time server can crash at any point during the lifetime of the TIS back-end.

Localisation makes it possible to treat a large, complex system as pockets of functionality that can be individually described by increasing the level of granularity with which this system is observed. This, in turn simplifies the specification of formulae and in general, makes them easier to *understand* and *maintain*. However, the ease with which local formulae are specified depends also on the nature of the components that are targeted, and as seen, the understandability of local formulae is *reduced* as the communication between system components increases.

4.4.2 Maintainability

As exemplified above, local properties that target interacting components are not easily specified. This effect is exacerbated in global properties, on account that these tend to result in even larger and more complex correctness formulae, because one must consider the overall behaviour of the system under scrutiny. Such an effect does not come as a surprise however, as global properties essentially combine the requirements of the individual local properties.

The two safety properties (4.1) and (4.2) from Section 4.4.1 can be incorporated into a single global property that is used to monitor the entire TIS back-end. To account for the combined behaviour (six interleavings in all) of the hash and time servers in the TIS setup with isolated back-end components (refer to Figure 4.3),

the following global sHML formula needs to be specified:

$$
\begin{array}{ll}
 & \mathbf{max}(X, \\
 & \quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge \\
 & \quad [\mathit{HashSrv}\ \mathbf{?}\ \{\mathtt{hash}, \mathit{TokenSrv}\}]\ \textcircled{2}\ (\\
 & \quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge [\mathit{TokenSrv}\ \mathbf{!}\ \{\mathtt{hash}, \mathit{Cnt}, \mathtt{""}\}]\,\mathbf{ff} \wedge \\
\text{Subcase 1} & \quad\quad [\mathit{TokenSrv}\ \mathbf{!}\ \{\mathtt{hash}, \mathit{Cnt}, \mathit{Hash}\}]\ \textcircled{3}\ (\\
 & \quad\quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge \\
 & \quad\quad\quad [\mathit{TimeSrv}\ \mathbf{?}\ \{\mathtt{time}, \mathit{TokenSrv}\}]\ \textcircled{2}\ (\\
 & \quad\quad\quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge [\mathit{TokenSrv}\ \mathbf{!}\ \{\mathtt{time}, \mathit{Time}\}]\,X\ \textcircled{3}) \\
 & \quad\quad) \wedge \\
\text{Subcases 2, 3} & \quad\quad [\mathit{TimeSrv}\ \mathbf{?}\ \{\mathtt{time}, \mathit{TokenSrv}\}]\ \textcircled{2}\ (\\
 & \quad\quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge [\mathit{TokenSrv}\ \mathbf{!}\ \{\mathtt{hash}, \mathit{Cnt}, \mathtt{""}\}]\,\mathbf{ff} \wedge \\
 & \quad\quad\quad [\mathit{TokenSrv}\ \mathbf{!}\ \{\mathtt{hash}, \mathit{Cnt}, \mathit{Hash}\}]\ \textcircled{3}\ (\\
 & \quad\quad\quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge [\mathit{TokenSrv}\ \mathbf{!}\ \{\mathtt{time}, \mathit{Time}\}]\,X\ \textcircled{3}) \wedge \\
 & \quad\quad\quad [\mathit{TokenSrv}\ \mathbf{!}\ \{\mathtt{time}, \mathit{Time}\}]\ \textcircled{3}\ (\\
 & \quad\quad\quad\quad [\mathit{TimeSrv}\ \mathbf{stp}\ \mathtt{killed}]\,\mathbf{ff} \wedge [\mathit{TokenSrv}\ \mathbf{!}\ \{\mathtt{hash}, \mathit{Cnt}, \mathtt{""}\}]\,\mathbf{ff} \wedge \\
 & \quad\quad\quad\quad [\mathit{TokenSrv}\ \mathbf{!}\ \{\mathtt{hash}, \mathit{Cnt}, \mathit{Hash}\}]\,X\ \textcircled{3}) \\
 & \quad\quad) \\
 & \quad)\wedge
\end{array}
\tag{4.6}
$$

$$
\begin{array}{ll}
 & [\mathit{TimeSrv}\,\mathbf{?}\,\{\mathtt{time},\,\mathit{ToknSrv}\}]\ ❷\ (\\
 & \quad [\mathit{TimeSrv}\,\mathbf{stp}\,\mathtt{killed}]\,\mathbf{ff}\wedge \\
\text{Subcase 4} & \quad [\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{time},\,\mathit{Time}\}]\ ❸\ (\\
 & \quad\quad [\mathit{TimeSrv}\,\mathbf{stp}\,\mathtt{killed}]\,\mathbf{ff}\wedge \\
 & \quad\quad [\mathit{HashSrv}\,\mathbf{?}\,\{\mathtt{hash},\,\mathit{ToknSrv}\}]\ ❷\ (\\
 & \quad\quad\quad [\mathit{TimeSrv}\,\mathbf{stp}\,\mathtt{killed}]\,\mathbf{ff}\wedge[\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{hash},\,\mathit{Cnt},\,\mathtt{""}\}]\,\mathbf{ff}\wedge \\
 & \quad\quad\quad [\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{hash},\,\mathit{Cnt},\,\mathit{Hash}\}]\,X\ ❸) \\
 & \quad)\wedge \\
\text{Subcases 5, 6} & \quad [\mathit{HashSrv}\,\mathbf{?}\,\{\mathtt{hash},\,\mathit{ToknSrv}\}]\ ❷\ (\\
 & \quad\quad [\mathit{TimeSrv}\,\mathbf{stp}\,\mathtt{killed}]\,\mathbf{ff}\wedge \\
 & \quad\quad [\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{time},\,\mathit{Time}\}]\ ❸\ (\\
 & \quad\quad\quad [\mathit{TimeSrv}\,\mathbf{stp}\,\mathtt{killed}]\,\mathbf{ff}\wedge[\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{hash},\,\mathit{Cnt},\,\mathtt{""}\}]\,\mathbf{ff}\wedge \\
 & \quad\quad\quad [\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{hash},\,\mathit{Cnt},\,\mathit{Hash}\}]\,X\ ❸)\wedge \\
 & \quad\quad [\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{hash},\,\mathit{Cnt},\,\mathit{Hash}\}]\ ❸\ (\\
 & \quad\quad\quad [\mathit{TimeSrv}\,\mathbf{stp}\,\mathtt{killed}]\,\mathbf{ff}\wedge[\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{time},\,\mathit{Time}\}]\,X\ ❸)\wedge \\
 & \quad\quad [\mathit{ToknSrv}\,\mathbf{!}\,\{\mathtt{hash},\,\mathit{Cnt},\,\mathtt{""}\}]\,\mathbf{ff} \\
 & \quad\quad) \\
 & \quad) \\
 &)
\end{array}
\tag{4.7}
$$

Subformula (4.6) handles the first three interleavings that arise when the hash server receives a request from the TIS front-end initially ❷. In the first subcase, the hash server immediately replies back to the TIS front-end ❸, followed by the time server receiving a request from the TIS front-end ❷ and also replying back ❸. In the second subcase, the time server receives a request from the TIS front-end ❷ into its mailbox, while in the meantime, the hash server sends its response ❸ in answer to the first request sent to it by the TIS front-end. This is followed by a reply from the time server to the TIS front-end ❸. In the third subcase, the time server also receives a request from the TIS front-end ❷, but this time, it immediately replies ❸

to the TIS front-end. The hash-server then sends its response ❸ in answer to the first request sent to it by the TIS front-end. Subformula (4.7) handles the last three interleavings that occur when the time server receives the first request, and are essentially analogous to the three cases in (4.6).

Comparing the size of the two local formulae (4.1) and (4.2) from Section 4.4.1 with the global formula just discussed makes it evident that global formulae are not just a *mere* summation of the individual local formulae. This stems from the fact that while the mentioned local formulae consider a small number of trace events apiece, the global formula needs to consider the total of these events in *all* possible combinations. Consequently, the size of global formulae increases rapidly, and the interleaving effect of trace events gives rise to repeated parts in the resulting formulae. This effect gets more drastic if the system under observation contains components that interact between themselves. For instance, the global formula for the TIS with *interacting* back-end server components (not shown) needs to consider a total of *ten* interleavings, as opposed to the *six* handled by the global formula in (4.6) and (4.7). While the size of global formulae makes them challenging to write and understand, their *maintenance* is even harder, as even miniscule changes in the behaviour of one process requires a substantial refactoring of the affected formulae. Handling changes in considerably sized global formulae like the one above becomes quickly unmanageable and error-prone due to pockets of repeated logic that must be updated accordingly.

4.4.3 Expressivity

When formulae are specified in a way that accounts for the possible interleaving of executing components, one is effectively imbuing the synthesised monitors with information that establishes how the system is expected to behave at runtime. In certain cases, concurrency gives monitors the possibility of observing a wider range of actions that arise from executing components. For instance, the execution of the

individual components that make up the system SYS $\stackrel{\text{def}}{=}$ (A | B) from Section 1.1 can take place in one of three ways: (i) component A executes before B; (ii) component B executes before A; (iii) both A and B execute simultaneously. Whichever the case, monitors observing SYS will *eventually* get to see both action 'a' from component A, and action 'b' from B. This kind view of the system would *not* have been possible if SYS was forced (perhaps, using internal logic) to choose whether to execute component A *or* component B.

In such cases as the former, a local approach can facilitate the specification of a certain subset of properties that are considered to be non-monitorable according to [21] (*e.g.* $[a]\,\mathbf{ff} \vee [b]\,\mathbf{ff}$). If say, components A and B from SYS are locally monitored using the aforementioned example formula, then the individual verdicts produced by both monitors attached to A and B would need to be *combined* using the $\vee$ operator in order to be able to produce a *global* verdict. In contrast, a global monitoring approach requires monitors *not only* to deal with the different execution interleaving of actions 'a' and 'b' (as seen in Example 1.1.1), but would also necessitate the monitoring algorithm to intelligently handle the forking of action events. This is on account of the fact that according to the monitor semantics adopted by the implementation discussed in Chapter 3, forking events between each of the submonitors for subformulae $[a]\,\mathbf{ff}$ and $[b]\,\mathbf{ff}$ would cause $[b]\,\mathbf{ff}$ to yield inconclusive when action 'a' is forked, and cause $[a]\,\mathbf{ff}$ to yield inconclusive when action 'b' is forked. Physical separation of the submonitors through local monitoring eliminates this problem, which would otherwise need to be handled in a global monitoring scheme, thereby, complicating the monitor artefact produced. In this sense, a local approach not only facilitates the specification of local formulae, but also eases the way with which certain non-monitorable properties can be synthesised into executable monitors.

4.4.4 Fault Tolerance

Localisation is also useful because the physical separation of monitors, can in cases prevent the global monitoring effort to continue in spite of partial system failures. In addition, the lack of communication between monitors removes all kinds of inter-monitor dependencies so that the failure of one monitor does not affect others. Global monitoring is not as resilient, and the failure of a single component could create a domino effect that would prevent the monitored setup from working correctly.

4.5 A Quantitative Study

The second part of the study adopts a *quantitative* approach and evaluates local and global monitoring on the basis of three parameters: the target system's memory consumption, its CPU utilisation, and responsiveness. Correct evaluation of these quantitative attributes is paramount, as runtime overheads ultimately serve as a *litmus test* for determining the feasibility of a RV setup.

To better understand the effects the system architecture has on the monitoring techniques applied, the two configurations of TIS presented in Section 4.3 are studied and the results obtained are compared against those for global monitoring. This serves to substantiate the conclusions drawn from the qualitative argumentation presented in Section 4.4, namely that: (i) localisation offers specification-related benefits when compared to a global monitoring strategy, and (ii) local monitoring is most effective when applied to non-interacting system components.

4.5.1 Data Analysis and Representation

The decision to expose the functionality of TIS over a TCP endpoint made it possible and practical to load test the system using off-the-shelf tools that are not Erlang-specific. For the testing task, the Apache JMeter suite of tools was employed, especially because it provides a useful user interface that facilitates the design of test plans. To streamline and optimise the testing process, a series of

automated benchmarking scripts were developed to ensure that all measurements can be captured *accurately* and *automatically*.

In the material that follows, the term *experiment* is used to refer to collection of benchmarks. A *benchmark* is performed by load testing the target system using a fixed number of concurrent requests n. The value of n per benchmark starts at 200, and is gradually increased to 2000 in steps of 200 (*i.e.,* 200, 400, ..., 2000) for a total of ten (*i.e.,* $2000 \div 200$) benchmarks per experiment. Each benchmark records measurements for the following three parameters: (i) memory consumption in MB, (ii) CPU utilisation as a percentage, and (iii) system response time in milliseconds. Once each benchmark terminates, the collected measurements are *averaged* out so that finally, the experiment consists of three sets of data points (from 200 to 2000), one for each measured parameter.

To ensure that sufficient data is collected, *ten* repetitions of each experiment are performed to obtain metrics for three different system configurations: (i) the unmonitored version of the system (*i.e.,* the baseline), (ii) the system fitted with local monitors, and (iii) the system fitted with global monitors. The results for each set of ten experiments for each system setup are then averaged to obtain *representative results* that are grouped and plotted according to the three performance parameters being investigated, yielding three plots — the memory consumption plot, the CPU utilisation plot, and the system response time plot. System response time plots display the amount of *request failures*. This error figure, given in terms of a percentage, represents the ratio of failed requests that resulted when the load tested system was unable to cope with the number of concurrent requests (*e.g.* TCP connection refusals, timeouts and broken pipes), and is computed by taking the number of failed requests against the total number of concurrent requests performed. To facilitate their interpretation, all the data plots are fitted with a *trend line*.

As with any statistical study, the choice of experiment parameters can greatly affect the outcome of results. The reasons for selecting the values for the *maximum number of concurrent requests per experiment* parameter (*i.e.,* 2000), the *benchmark*

interval parameter (*i.e.,* 200), and the *number of experiment runs per system setup* parameter (*i.e.,* ten) are discussed in considerable depth in Section 5.3.1. Furthermore, the different precautions that were taken in order to ensure the achievement of best results are also discussed in the aforementioned section.

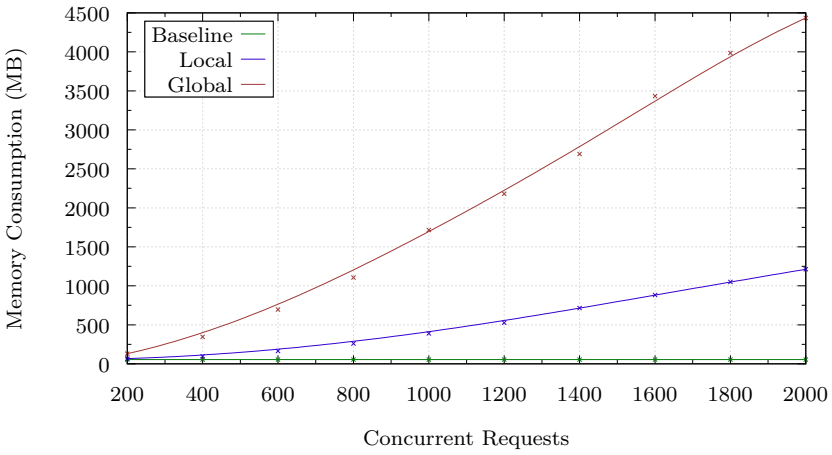
4.5.2 Isolated Components

This TIS architecture from Figure 4.3 is evaluated first. In this configuration, the back-end components share no means of mutual communication. Each of the *local* formulae (4.1) and (4.2) need only to consider the two interactions with the TIS front-end. In contrast, the formula from Section 4.4.2 must contend with the interleaving that arises when the system is perceived from a *global* point of view.

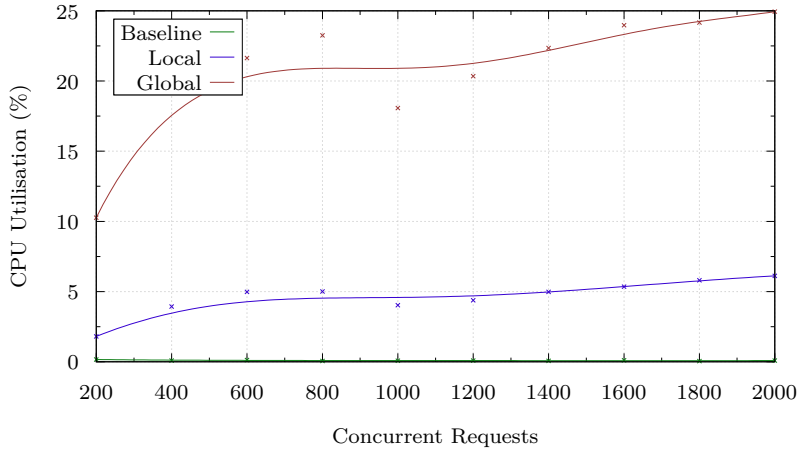
The performance measurements that were obtained for said local and global formulae are shown in Figure 4.5, according to the experiment setup discussed previously. Figure 4.5a shows the effects local and global monitoring have on the overall memory consumption as the number of concurrent requests increases from 200 to 2000. While as expected, the unmonitored system's memory footprint remains constant due to its *static* architecture, the same is not true of the local and global monitored systems. Global monitoring induces the highest overhead, owing to the large number of processes created in the monitor composition as a result of the large global formula. Local monitoring, as expected, has a comparably lower performance cost.

The trace event replication (*i.e.,* forking) that takes place inside global monitors seems to be affecting the system in terms of its CPU utilisation, as can be observed in Figure 4.5b. Local monitors used in this TIS configuration are small (due to the lack of interaction interleaving) and make minimal use of this mechanism, thereby maintaining lower CPU overheads.

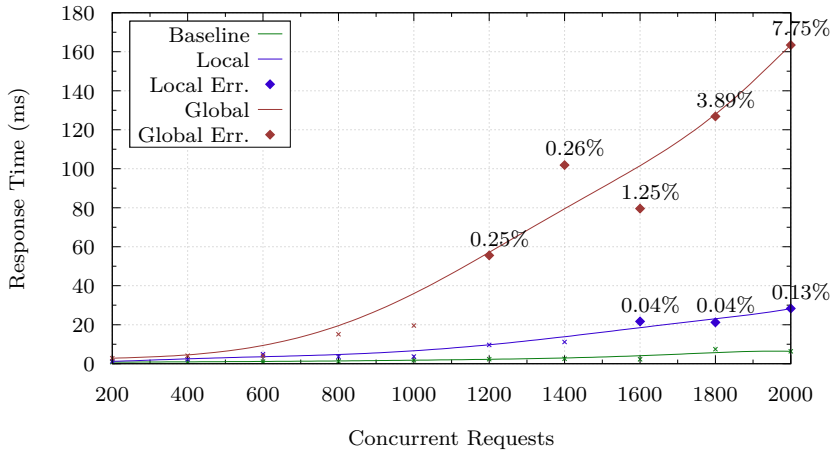
The third plot shows the effects of local and global monitoring on the system's response time: the combined effect of memory consumption and CPU utilisation. A considerable divergence between local and global monitoring is evident, as is the



(a) Concurrent Requests *vs.* Memory Consumption



(b) Concurrent Requests *vs.* CPU Utilisation



(c) Concurrent Requests *vs.* Response Time

Figure 4.5: Performance measurements for the unmonitored system, local and global monitoring (TIS architecture with isolated back-end components).

difference between the number of errors due to failed requests. While for global monitoring, failed request errors initially appear at the point where the number of concurrent requests reaches 1200, local monitoring introduces errors much later in the experiment. Moreover, these are noticeably lower when compared to the ones obtained for global monitoring.

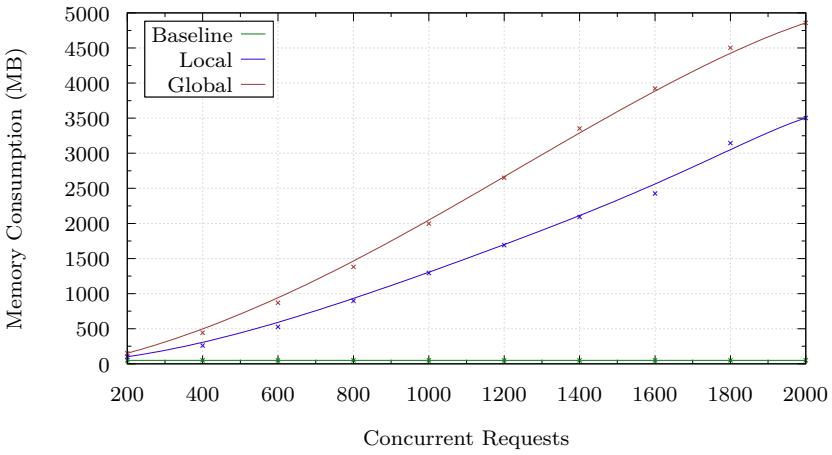
4.5.3 Communicating Components

The second configuration of the TIS with interacting back-end components (see Figure 4.4) is evaluated next. In this setup, the local formulae are modified to accommodate the added communication that takes place between the hash and time servers, as seen in the second part of Section 4.4.1.

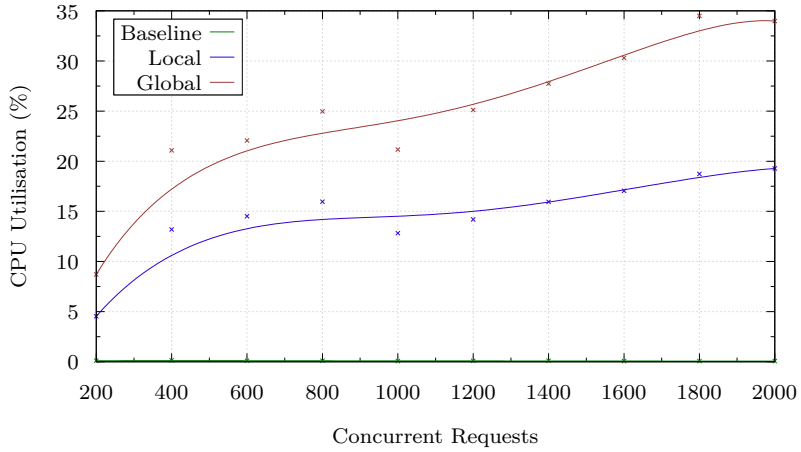
The performance measurements in Figure 4.6 suggest that local monitoring still performs better than its global counterpart, but it does so to a *lesser* degree. Figure 4.6a shows the memory consumption plots for local and global monitoring, where both rates of incline are somewhat close in comparison to those from Figure 4.5a. The two monitoring approaches appear to consume more memory, as one might expect with larger correctness formulae.

A similar pattern emerges in Figure 4.6b, where the difference in CPU utilisation between local and global monitoring, although obviously evident, is less pronounced than that in Figure 4.5b. To better assess this closer CPU utilisation gap, one does well to consider the differences between the plot in Figure 4.5b and the one in Figure 4.6b. A side-by-side comparison of these plots shows that global monitoring in Figure 4.5b exerts a CPU load of 24.94% at 2000 concurrent requests, whereas in the current TIS setup, a load of 33.98%, *i.e.,* a rise of 9.04%. A similar comparison of *local* monitoring reveals that in Figure 4.5b, local monitoring exerts a CPU load of 6.12% (at 2000 concurrent requests), whereas in Figure 4.6b, this load steps up to 19.28%, *i.e.,* a CPU utilisation increase of 13.16%.

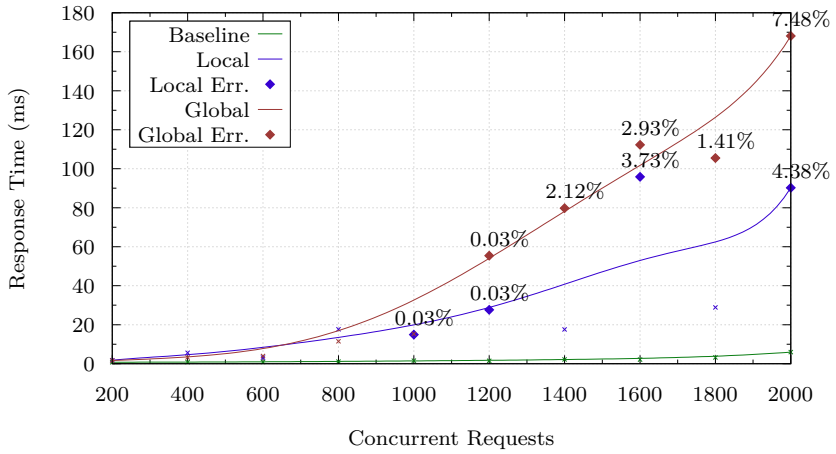
Figure 4.6c exhibits similar trends. The response time difference for local monitoring between the two TIS setups is comparatively *larger* than that for global



(a) Concurrent Requests *vs.* Memory Consumption



(b) Concurrent Requests *vs.* CPU Utilisation



(c) Concurrent Requests *vs.* Response Time

Figure 4.6: Performance measurements for the unmonitored system, local and global monitoring (TIS architecture with interacting back-end components).

monitoring. In fact, the response time impact for global monitoring in Figure 4.5c is of 163.42ms (at 2000 concurrent requests) whereas in Figure 4.6c, this impact is of 168.10ms; the failed request error between both plots is negligible. For local monitoring, Figure 4.5c shows a response time of 28.29ms and a failed request error of only 0.13%, whereas for the current TIS setup with communicating back-end components, a response time of 90.21ms and a failed request error of 4.38% (refer to Figure 4.6c). This amounts to a response time degradation of 61.92ms and an increase of 4.25% in failed request errors for the current TIS setup.

4.5.4 Commentary

The local and global monitoring techniques considered in this dissertation employ the synthesis described in Chapter 3 to generate compositional monitors that unfold recursively. Unfolding *enlarges* the monitor (in terms of its number of composing processes) each time it re-instantiates a fresh copy of itself (refer to discussion in Section 3.1). Consequently, monitors are rendered increasingly inefficient, and while additional processes consume more memory, the forking mechanism requires extra processing in order to be able to replicate and deliver trace events to each of these processes. The adverse effects of this behaviour on both the memory and CPU can be clearly seen in Figures 4.5 and 4.6.

The quantitative results in Sections 4.5.2 and 4.5.3 show that local monitoring offers practical advantages when it is used to monitor component-based systems. Additionally, the results also reveal that while in general, the benefits of localisation are clear in cases where local monitoring is applied to isolated components, these are less evident in scenarios involving communicating components. It is interesting to note that despite the obvious divergence between local and global monitoring, as observed in both TIS configurations, the performance plots for the *unmonitored* system hardly changed. This suggests that alterations to the architecture, however slight, might not only dictate which monitoring strategy is best employed, but also determine the extent of its effectiveness.

4.6 Conclusion

This chapter revisits the five assessment criteria from Section 1.2.1, and summarises the results obtained in this study:

1. **Understandability**: Locally specified formulae focus only on the functionality of isolated components. These are simpler, less error-prone to write, and need not account for all the interleaving actions due to trace events that would otherwise need to be considered if global specifications are used;

2. **Maintainability**: Local specification scripts are more maintainable and thus, easy to extend. Amendments due to new requirements can be safely administered *only* to the affected system components, safe in the knowledge that refactoring local formulae does not impact other unrelated specifications. This is not the case for global specifications, where slight changes in the system require substantial formulae *re-engineering*. Furthermore, the addition of new components can be easily handled using local specifications, but is more difficult to manage if a single global specification is used;

3. **Expressivity**: Concurrency enables monitors to observe a larger portion of the system, making it possible to specify properties considered to be non-monitorable in certain system setups. Localisation facilitates not only the specification process as already seen, but also favours the synthesis of monitors that have a simpler implementation. Utilising monitors that follow closely the synthesis described in Chapter 3 increases the confidence in the monitors' correctness. A global approach makes this difficult to achieve;

4. **Fault Tolerance**: Local monitors can be easily deployed on different system components. Failure in these components, can, in certain cases, be gracefully handled by local monitors, thereby making them resilient to *partial* system failures. Additionally, isolation between local monitors ensures that failed monitors themselves do not affect one another;

5. **Performance**: Global monitoring requires centralised access to the execution trace, whereas local monitors adopt a decentralised approach where each subscribes to a local subtrace. The latter method brings about the following performance benefits: (i) monitoring is cheaper as localised monitors work in complete isolation from each other, (ii) local monitors consume only trace events that are relevant to the formula being evaluated, and (iii) trace events do not flow through a central data collection point and bottlenecks are less likely to occur.

This study demonstrates that local monitoring is *not* a silver bullet that mitigates all the problems associated with global monitoring. While its benefits over the latter are clear when applied to systems with isolated components, applying it to communicating components *may* possibly yield benefits, though these are less straightforward to interpret and quantify.

It would have been interesting to study components that exhibit higher levels of communication. This would have made it possible to explore how quickly local monitoring degrades in proportion to the number of communication interactions between monitored components. Due to the lack of time, this was not attempted. Using the data obtained from the experiments done so far, one can extrapolate the trend lines in Figure 4.6 and come up with an answer for just two interactions. Considering more interactions requires additional experimentation. This would undoubtedly go a long way in demonstrating that any monitoring approach is in general very sensitive to changes in the system architecture it is applied to, and therefore, cannot be studied in a vacuum.

5. Case Study

Chapter 4 studied local monitoring through a number of qualitative arguments and quantitative experiments. These studies were conducted on a small component-based system called TIS *built specifically* for the task of comparing both monitoring approaches. Two different TIS configurations were investigated so as to determine the cases where local monitoring is at its most effective.

Modest systems like the TIS are employed to great effects in *initial studies*, and despite their simplistic functionality, capture the most *essential* and *representative* features of larger and more complex systems. They also make it easy to test different artificial or *corner-case* scenarios without the need to invest in too much development effort. Yet, a generalised study cannot be conducted solely under controlled conditions, and even though promising results were obtained in Chapter 4, a truly holistic approach demands that these be also investigated in a *real-world* setting.

This chapter presents a study of the practical applications of local monitoring by evaluating a third *third-party* industry-level software called Ranch. Since the effectiveness of local monitoring is central to this assessment, a proper analysis of Ranch is made in order to identify areas within the software where local monitoring can be *ideally* applied. The case study aspires to give further evidence of the utility of local monitoring, thereby increasing the level of confidence in the results obtained in Section 4.5. Four main topics are covered in this chapter:

- An overview of the OTP framework on which Ranch is built;

- An analysis of the internal interaction protocol used by Ranch when handling TCP sockets;
- A discussion of the experiment design used to conduct the quantitative case study on Ranch, including the precautions taken to ensure the achievement of best results;
- An evaluation of the performance benefits of local monitoring when applied to three different Ranch configurations.

5.1 A Third-Party Application

Ranch is a socket acceptor pool for TCP protocols developed in Erlang/OTP. It exhibits the inherent qualities of a *dynamic* concurrent system that is able to scale according to the current computational demands. The study presented in this chapter however, concentrates on the *static* aspects of Ranch, and considers only those components whose existence remains fixed throughout the entire application lifetime. To better understand the basic Ranch concepts that follow, a brief overview of the OTP framework is given first.

5.1.1 The Open Telecom Platform

Open Telecom Platform (OTP) is an open source collection of tools and libraries which make it easy to design, develop and deploy concurrent Erlang software systems that exhibit the properties of telecom applications [5, 12, 24]. OTP bases itself on the Erlang infrastructure and provides amongst other things, basic abstraction libraries and common behavioural patterns that relieve the developer of the burden of having to implement recurrent functionality manually.

Out of the most common forms of behavioural patterns offered by OTP, the *supervisor* is perhaps the most fundamental and widely used OTP behaviour. It embodies the notion of *supervision trees* — a hierarchical organisation of processes

wherein supervisors are *only* responsible for spawning, starting and looking after their descendant processes, and in cases of failure, ensure that some corrective action is taken. This depends wholly on the supervisor configuration. For instance, the supervisor may choose to restart the terminated component or kill and restart an entire supervision subtree to avoid inconsistencies that might have arisen due to the terminated component. In addition to *workers*, processes that are located in the extremities of the aforementioned tree arrangement, supervisors can also supervise other supervisors — this is what gives rise to supervision trees, one of the key ingredients used in creating resilient and fault-tolerant applications in Erlang.

A collection of Erlang OTP behaviours, custom modules and resources is typically packaged in an OTP application, which is the standard way of distributing related functionality in the form of libraries. These libraries can either contain normal APIs that support other applications, or full-blown deployable artefacts; Erlang uses the term application to refer to both instances. Applications can themselves depend on other applications, and their dependencies in turn, can do the same, giving rise to a tree of dependencies, as is common in other languages such as Java. Deployable OTP applications conform to the *application* behaviour which requires the presence of a top-level supervisor tasked with starting up and looking after the entire application supervision tree.

5.1.2 The Ranch Architecture

Ranch is packaged as an OTP application whose top-level supervisor, `ranch_sup`, starts the supervision tree depicted in Figure 5.1. It is responsible for overseeing the execution of the `ranch_server` (worker) process and the `ranch_listener_sup` supervisor. The `ranch_server` keeps track of Ranch listeners and associated acceptors. A *listener* represents the collection of *acceptor* processes that wait on a port for incoming connections. Figure 5.1 shows a Ranch configuration with a single listener and two acceptor processes, `acc_1` and `acc_2`, bound to the *same* socket. The `ranch_acceptors_sup` supervisor manages the life cycle of acceptor processes,

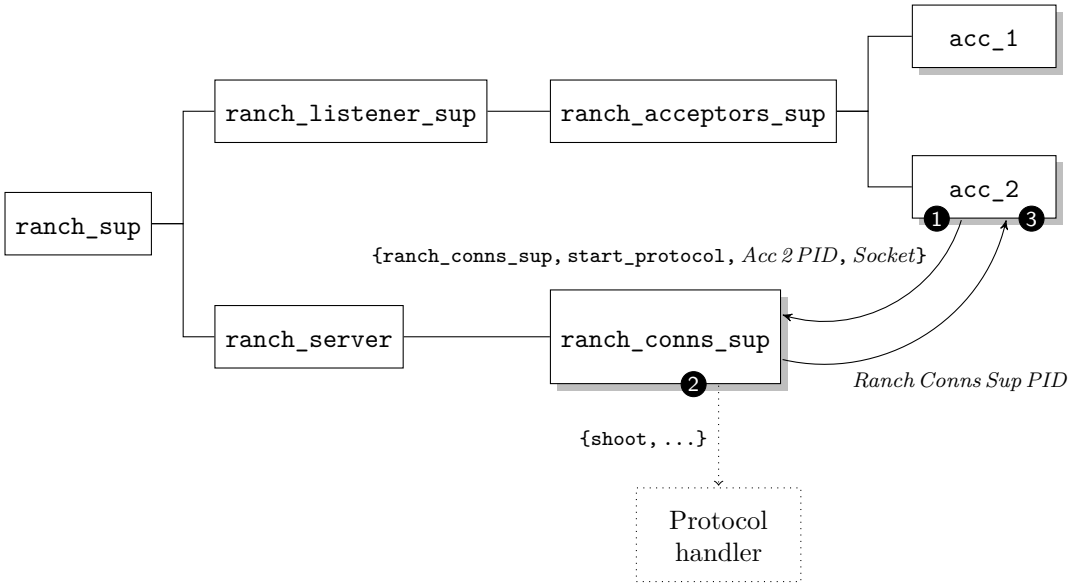


Figure 5.1: The Ranch supervision tree with one listener and two acceptors.

whereas the `ranch_conns_sup` supervisor handles the dynamic instantiation of *protocol handler* processes that attend to TCP connections requests made on Ranch. New connection requests directed to a Ranch socket are handled by any acceptor that happens to be available at the time, as per the protocol below:

❶ The accepting acceptor, `acc_2` in Figure 5.1, sends the new connection details to the `ranch_conns_sup` and immediately *blocks*. These details are packed into a tuple of the form `{ranch_conns_sup, start_protocol,` *Acc 2 PID*`,` *Socket*`}`, where *Acc 2 PID* denotes the PID of the sender acceptor, and *Socket* contains the socket reference number;

❷ The `ranch_conns_sup` supervisor spawns a new protocol handler, transfers the socket ownership to the protocol handler, and hands over to it additional information packed in the tuple `{shoot, ...}`. From this point onwards, the protocol handler engages in direct communication with the socket;

❸ A successful creation of the protocol handler triggers an acknowledgement message that is sent from the `ranch_conns_sup` back to the acceptor. If after creating the protocol handler, the connection limit configured on the listener

is reached, no acknowledgement is sent, leaving the acceptor blocked and preventing it from accepting new connections[1]. Otherwise, on receipt of the acknowledgement, the acceptor unblocks and goes back to listening mode.

The protocol described above encompasses both the dynamic and static aspects of Ranch. In the *dynamic* case, protocol handlers are created on demand, and while bounded by the connection capping configured on the listener, their count can range between zero and this configured maximum. In the *static* case, the number of instantiated acceptor processes is *pre-determined* at start up, and remains fixed throughout the lifetime of the application. At any time instant where the number of concurrent connections exceeds that of available acceptors, these are either queued on the TCP backlog or outright refused if the backlog is full.

5.2 Monitoring for the Ranch Protocol

Ranch's component-based architecture lends itself well to being monitored using a local approach. In particular, the TCP connection handling protocol from Section 5.1.2 shows that acceptor processes do not engage in mutual communication, making these an ideal target on which local monitoring can be applied. A global approach could also be considered as an alternate means of monitoring, although as discussed below, its use turns out to be impractical.

The study that follows investigates the suitability of local monitoring. Furthermore, the investigation is also extended to global monitoring, in order to better compare and understand the benefits attributed to localisation. Both techniques are used to runtime verify a fragment of the Ranch TCP connection handling protocol. This verification focuses on the Ranch *static subsystem* comprised of the `ranch_conns_sup` and one or more acceptor components, as described above (see Figure 5.1). For example, a positive property that monitors for the possibility that

[1]Acceptor processes that have been blocked due to a connection capping are resumed once a sufficient number of protocol handlers have terminated and the total listener connection count goes below the configured maximum.

an acceptor crashes requires that the following holds: *"after an acceptor sends a connection initiation request to the* ***`ranch_conns_sup`*** *supervisor, it either crashes or receives an acknowledgement in reply."* Expressing the property as a local cHML formula for *two* acceptors is easily accomplished:

$$
\begin{array}{l}
\mathbf{min}(X, \\
\quad \langle \mathit{ConnsSup}\ \mathbf{!}\ \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt}, \mathit{Sock}\}\rangle\ ❶\ (\\
\quad\quad \langle \mathit{Acpt}\ \mathbf{stp}\ \texttt{killed}\rangle\ \mathbf{tt} \vee \langle \mathit{Acpt}\ \mathbf{?}\ \mathit{ConnsSup}\rangle\ X\ ❸) \\
)
\end{array}
\tag{5.1}
$$

where the Erlang variable *ConnsSup* binds with the `ranch_conns_sup` supervisor PID, and *Acpt* and *Sock* bind with the acceptor PID and socket reference number respectively. Formula (5.1) specifies the interaction where first, a connection initiation request is sent by the acceptor to `ranch_conns_sup` ❶. Subsequently, the acceptor either crashes instantly, or receives an acknowledgement from `ranch_conns_sup` ❸ in reply to its message.

Specifying the property above using a global approach is not as easily accomplished, because the resulting formula suffers from a size explosion problem due to the number of interleavings that must be considered (see Appendix C for discussion). What is worse, the size and complexity of the global formula worsens the more acceptors are considered. Likewise, monitors synthesised as a result grow increasingly large, up to a point where these become unwieldy to use in practice (refer to Section 5.3).

Conversely, a local approach *retains* all of the qualitative advantages attributed to localisation (refer to concluding remarks in Chapter 4); this is in particular very apparent in the fact that formula (5.1) is agnostic of the number of Ranch acceptors configured. Determining the *performance* impact of local and global monitoring however, requires a deeper analysis of both approaches from a practical aspect. This exploration takes the form of the quantitative investigation that is presented in the next section.

5.3 A Quantitative Evaluation of Ranch

An evaluation of the performance impacts of local and global monitoring on Ranch is presented in this section. It bases itself on a series of experiments specifically designed to test different Ranch configurations. Performance is assessed based on the three parameters already used in Section 4.5, namely, the target system's memory consumption, its CPU utilisation, and its responsiveness. These experiments employ the formulae mentioned in Section 5.2 that target only the *static* part of Ranch.

5.3.1 Experiment Setup

The selection of an appropriate evaluation setup plays a vital role in determining how experiments are conducted, as this has direct and far-reaching effects on the results that are obtained. It is therefore paramount that the design of such a setup incorporates a good choice of tools, makes sensible assumptions and takes the necessary precautions in order to reduce as much as possible the risk of errors in the data collection and analysis phases.

Design Considerations

A holistic approach was taken so that the same tools and automated benchmarking scripts used for evaluating Ranch could also be used to evaluate the two configurations of the TIS from the previous chapter. This made it possible to concentrate the development effort into creating a *unified* experiment setup that caters for all the systems studied in this dissertation. As a result, the launcher implementation from Section 4.2, the Apache JMeter suite of tools, and automated benchmarking scripts from Section 4.5.1 are also used to conduct the evaluation on Ranch in Section 5.3.2.

The benchmarking scripts alluded to in Section 4.5.1 have been developed as a means to *fix the conditions* each experiment[2] run is performed with. This serves the

[2]The same definition given in Section 4.5.1 is reused in this chapter.

purpose of automating the tedious testing procedure, but more importantly makes it also possible to obtain *repeatable measurements* for each of the executed tests.

Single experiment runs are driven by a first script that defines the overall strategy used to benchmark a chosen target system configuration (*e.g.* the locally monitored system). It takes care of (i) starting the Erlang VM for the target system configuration, (ii) executing the JMeter test plan, parametrised by the number of concurrent requests n, (iii) averaging the results obtained by JMeter, and (iv) plotting the averaged data. The number of concurrent requests n for each test plan execution in step (ii) is increased from 200 to 2000 in steps of 200, resulting in *ten benchmarks* for a *single* experiment run. Ten repetitions of the *same experiment* are performed using a second script, and the resulting data is averaged to yield the final plots (see discussion in Section 4.5.1).

The Choice of Experiment Parameters

Choosing parameters is not always straightforward, and generally requires a thorough observation and interpretation of the data. One common method used in statistics involves visually analysing the data plots obtained during experiment trials [32]. Based on this examination, the parameters are tweaked and the process of running experiments is repeated until satisfactory and statistically sensible results are obtained with the chosen settings. These are afterwards plugged into the evaluation setup and used to drive the experiments. The case study adopts this iterative fine-tuning method to determine appropriate values for the three parameters used when evaluating the test setups in Sections 4.5 and 5.3:

1. *maximum number of concurrent requests per experiment* parameter;
2. *benchmark interval* parameter;
3. *number of experiment runs per system setup* parameter.

Selecting the value of Parameter 1 was possible once a fully functional experiment setup had been developed. Its choice is motivated by the need to simulate

benchmarks that approach as much as possible loads typical of a real-world Ranch setup operating under normal conditions. In practice however, the value of this parameter is largely imposed by the *resources available* on the benchmarking machine. A number of exploratory runs were conducted using different Ranch and TIS configurations, so as to determine the maximum possible limit that could be reliably attained without crashing the testing setup. These test runs revealed that while the unmonitored and locally monitored system could be easily pushed beyond a couple of thousand concurrent requests, global monitoring was consistently failing at around the 2050 mark. In view of this upper limit, choosing a value of 2000 concurrent requests was considered reasonable, especially since at values *as low as* 1000, the divergence between local and global monitoring was already *clearly apparent*.

Rounding the number of concurrent requests to 2000 made it also convenient to pick a value for the number of benchmarks that are executed in *one* experiment run, *i.e.,* Parameter 2. This choice also determines the volume of data that is collected during the run, the time it takes to complete, and also the granularity with which data plots are rendered (*i.e.,* the tick mark frequency on the x-axis). An interval of 200 turned out to be a suitably balanced number, because it gives a good visual indication of changes in the data, while still keeping the time taken for an experiment to execute reasonably moderate. Pilot tests conducted using values larger than 200 (*e.g.* 300 or 400) detracted from the experiments' granularity, while smaller values (*e.g.* 100 or 50) did not provide additional statistical insight.

The value of the last parameter, Parameter 3, was chosen on the basis of a simple check that determines whether a sufficient number of experiment repetitions has been performed so that the data collected is *representative* of the *entire* experiment. Such a procedure can be accomplished fairly easy through visual means. It involves plotting the set of data points from each single experiment run onto the same graph, and check whether the trends correspond (*i.e.,* the majority of the plots superimpose). For this experiment design exercise however, this procedure was

carried out using the *numerical method* described next.

To begin with, an initial value of five was chosen for the number of times an experiment run is repeated. This was used to execute five experiment runs for a randomly selected system setup (*e.g.* Ranch with two acceptors and local monitoring). Before going further, one should recall that an experiment consists of ten benchmarks (*i.e.,* 200, 400, ..., 2000), and that each benchmark yields the average values for the three performance parameters: (i) memory consumption, (ii) CPU utilisation, and (iii) system response time.

The average ($\bar{x}$), standard deviation (σ) and coefficient of variation (CV = $\frac{\sigma}{\bar{x}} \times 100$) for each of these three parameters was calculated across *all* five experiments for each benchmark interval. To illustrate, for the first benchmark interval at 200, the $\bar{x}$, σ and CV for performance parameters (i), (ii) and (iii) are computed using the data from each of the five experiment runs; this is then repeated again for the second benchmark interval at 400 to obtain the second set of $\bar{x}$, σ and CV, and so on until the tenth benchmark at 2000 is reached. The result is a set of three (one for each performance property) calculations of $\bar{x}$, σ and CV per benchmark interval, *i.e.,* a total of ten calculated over all of the five individual experiment runs.

In order to measure whether increasing the number of test runs from five to ten yields a better data set, the same procedure was also repeated with ten experiment runs. The CV values for each benchmark interval in the set of ten experiment runs were afterwards *compared* with the corresponding ones obtained with five experiment runs. For instance, the two CV values for ten and five experiment runs were compared for the first benchmark at interval 200, the next pair of CV values compared for the second benchmark at interval 400, and so on. The respective CV values in all benchmark intervals were sufficiently close, indicating that there was *low variability* between the two sets of experiment runs. It also implies that the data obtained with five repetitions is more or less representative of the data obtained with ten repetitions. Furthermore, this similarity also suggests that increasing the number of repetitions beyond ten would only *very slightly* improve the data, and

that using a larger number of repetitions adds *negligible* CV improvements. In the end, a value of ten experiment repetitions was favoured instead of five because the added precision contributed to more robust results[3].

The CV was also useful when performing data *spot checks* on benchmark intervals that were not comparable, *e.g.* 2000 with 200. For instance, $\bar{x}$ and σ for the average response time for 200 concurrent requests cannot be compared to that for 2000 concurrent requests, because the latter will certainly be larger (*e.g.* due to slowness in the system). A comparison based on their CV values ensures that despite the obvious large differences in $\bar{x}$ and σ, their respective ratio is comparable [3].

Precautions

In addition to the standard precautions such as repeated runs of the same experiment, the ones discussed below were additionally taken:

- **Unoptimised Monitors**: As explained in Section 3.2.2, recursive unfolding enlarges monitors in terms of their composing processes. Continuous increase in the number of processes, and in turn, the amounts of memory and CPU resources consumed leads to an eventual *resource exhaustion*, making it difficult to push the limits of the monitored system. This effect was partially mitigated by starting a *fresh* Erlang VM each time a new benchmark was performed;

- **Initial Performance Spikes**: Immediately performing benchmark on freshly loaded systems usually results in the first handful of measurements accounting for the *lazy start* up of the internal VM infrastructure. This is easily spotted in plots that exhibit performance spikes in the initial set of data points. Such

[3] The similarity comparison carried out at different benchmark intervals for ten and five repeated experiments was done on the basis of the CV *alone*. This is because it can be tricky to rely only on $\bar{x}$ and σ, as these quantities may vary between different experiment runs, and choosing an appropriate numerical *range* that deems whether $\bar{x}$ and σ are close or far apart is not always straightforward. However, a comparison using the *ratios* of $\bar{x}$ and σ, *i.e.,* the CV, makes this easy to deal with.

erroneous readings can be minimised or altogether eliminated if a series of *warm-up* requests are performed *before* the actual experiments are started. Doing so ensures that the measurements collected from experiments runs are free from *irregularities* and *representative* of the true behaviour of the system;

- **Performance Measurements Lag**: The asynchronous monitors studied in this work can give rise to instances where the system *outpaces* its monitors. In certain settings, this lag grows significantly large such that the monitors are actively processing trace events long after the system has completed its execution. A correct methodology for collecting *memory consumption* and *CPU utilisation* performance metrics should take this lag into account, and only harvest the required measurements once the monitored system stops executing. Choosing the correct time that is spent waiting before taking the two aforementioned measurements, depends very much on the system being tested and the state of its monitors. For instance, large systems whose monitors unfold repeatedly take longer to execute than small systems with non-recursive monitors. In all of the experiments presented in Section 5.3, the amount of time spent waiting before measurements are taken was selected empirically, based on a number of trials conducted on different Ranch and TIS configurations. One should mention that lagging monitors do *not* affect the system response time performance measurement because this is *instantly* measured. If a response is not returned within the specified time limit, the request times out, and is subsequently logged as an error.

5.3.2 Performance Measurements

The performance impact local and global monitoring exert on Ranch is studied and compared next. To better understand how different Ranch configurations may be affected by local monitoring, two setups were evaluated: (i) the basic Ranch setup in Figure 5.1 with two acceptors, and (ii) one with four acceptors. As done previously

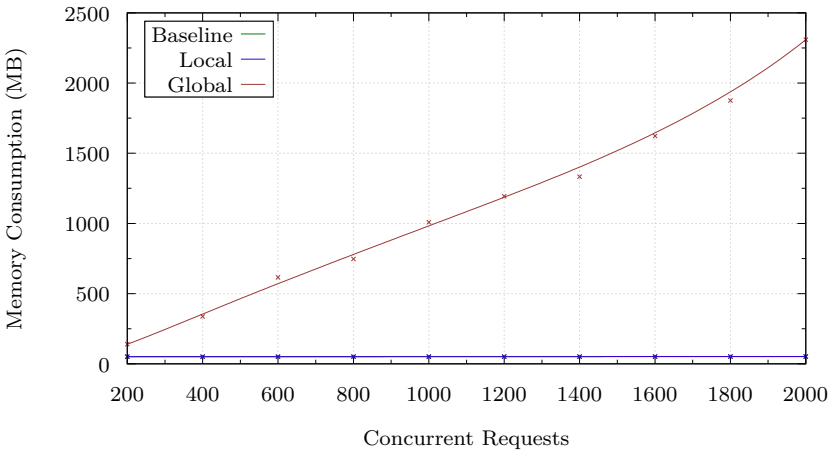
in Section 4.5, the results are compared to the unmonitored version of the system, as this makes it possible to better interpret the data plots obtained from each experiment.

The Ranch setup with two acceptors is used as a starting point of this study. It shows how local and global monitoring behave when applied to the smallest possible configuration that introduces interleaving at the *connection* or front-end level. A small number of acceptors makes Ranch susceptible to *bottlenecks* (*i.e.,* low throughput), and therefore, difficult to push its limit to a point where one can get a clear idea of how the monitored system is affected when subjected to large numbers of concurrent requests. The Ranch documentation [25] recommends that the configured number of acceptors is set to a hundred. This is high enough to always have acceptors ready to handle new connections, and sufficiently low that it does not impact the system's performance. In practice however, studying Ranch configurations with more than four acceptors was problematic because global monitoring *continually* led to resource exhaustion.

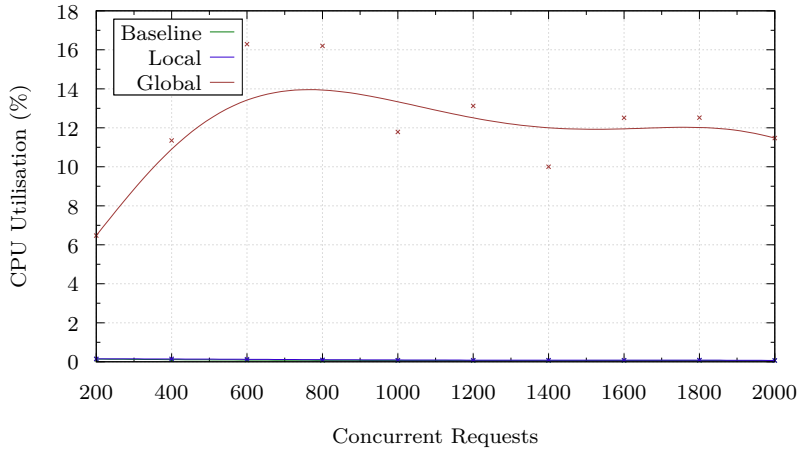
In order to achieve a test configuration that resembles closely those used in real-world settings, Ranch is used in conjunction with a small, fast and modular HTTP server called Cowboy [25]. Cowboy delegates all its TCP socket handling to Ranch so that any connection attempts made on Cowboy are serviced by Ranch. To drive the entire Cowboy-Ranch setup, a simple time service is exposed over a REST end-point so that HTTP requests on Cowboy trigger the Ranch internals as required. All the experiments were conducted on a 3.1 GHz Intel Core i7 processor with 16 GB of memory.

A Ranch Configuration with Two Acceptors

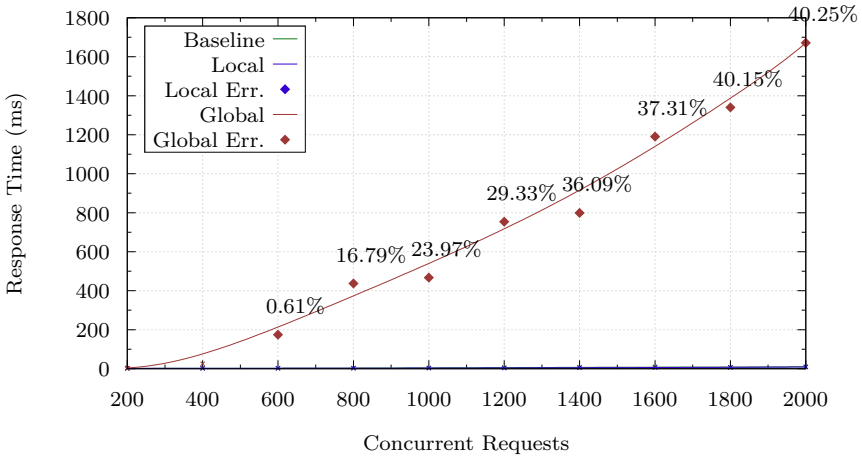
Figure 5.2 shows the performance measurements obtained for the first Ranch configuration with two acceptors. All three plots demonstrate with high consistency that global monitoring induces overheads that are *substantially higher* than those of local monitoring. The effects of the bottleneck created by having two acceptors can



(a) Concurrent Requests *vs.* Memory Consumption



(b) Concurrent Requests *vs.* CPU Utilisation



(c) Concurrent Requests *vs.* Response Time

Figure 5.2: Performance measurements for the unmonitored system, local and global monitoring (Ranch with two acceptors).

be seen if one compares the CPU utilisation and response time plots in Figure 5.2b with those in Figure 5.2c. A close inspection reveals that as the number of errors in the system response time plot increases, CPU utilisation slowly dips. Errors due to failed or timed out connection requests exert a lower load on the CPU because fewer connections need to be handled and less monitor-related processing occurs.

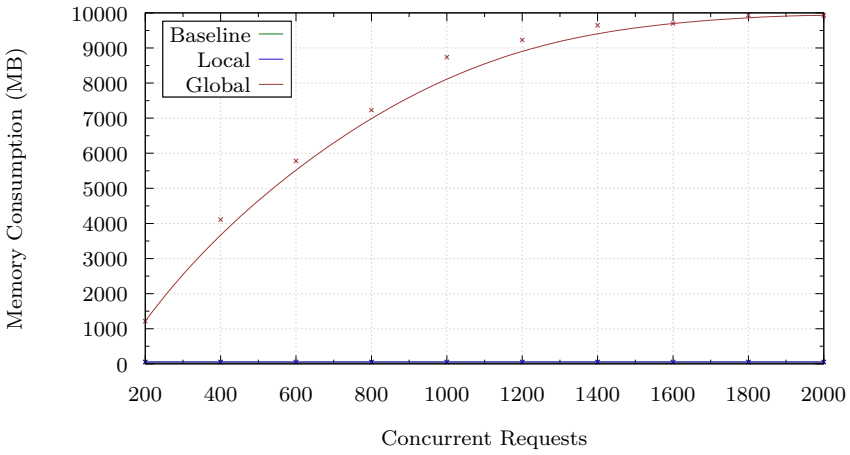
Figure 5.2 makes it hard to discern the relationship that exists between the unmonitored and locally monitored system, as the plot for local monitoring is not clearly displayed. An in-depth discussion on the comparison between the two can be found at the end of this section (see Figure 5.4).

A Ranch Configuration with Four Acceptors

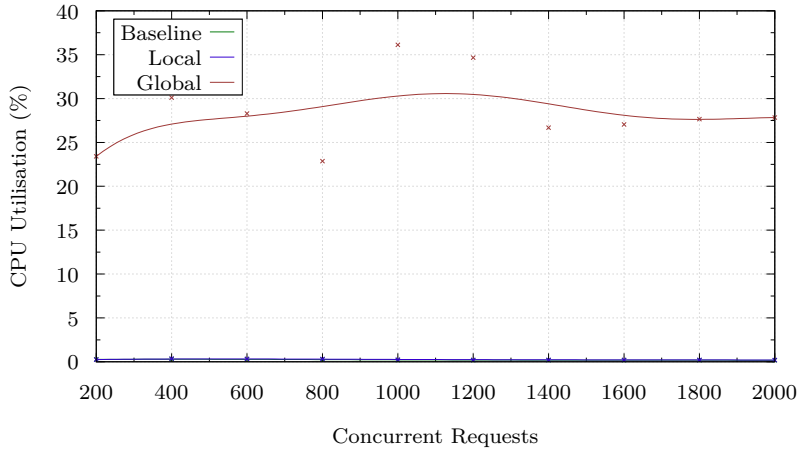
All three performance plots in Figure 5.3 maintain the trend that local monitoring outperforms global monitoring. The plots also show how the updated Ranch configuration with four acceptors affects both monitoring approaches. Increasing the number of accepting processes to four seems to relieve some of the effects created by the bottleneck in the previous configuration. In fact, the consequence of this higher throughput can be clearly seen in the memory consumption and CPU utilisation plots.

The difference in the rate of incline of the memory consumption plot for global monitoring in Figure 5.3a, when compared to the previous one in Figure 5.2a is significant. This is attributed to the higher number of accepted connections, as indicated by the slight decrease of the *overall* request error rate in Figure 5.3c. A reduced number of request failures means that additional connections were handled in comparison to the previous Ranch configuration with two acceptors; these in turn instantiate more monitor processes that require higher computational resources, as attested by Figures 5.3a and 5.3b.

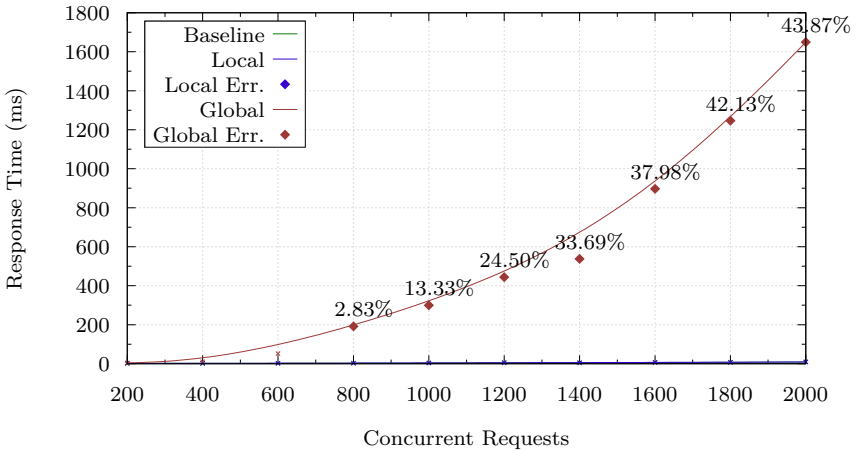
Despite increasing the number of acceptors from two to four, the request error for the global monitoring plots shown in Figure 5.2c and Figure 5.3c hardly varies, even though with additional acceptors, Ranch ought to have been more responsive.



(a) Concurrent Requests *vs.* Memory Consumption



(b) Concurrent Requests *vs.* CPU Utilisation



(c) Concurrent Requests *vs.* Response Time

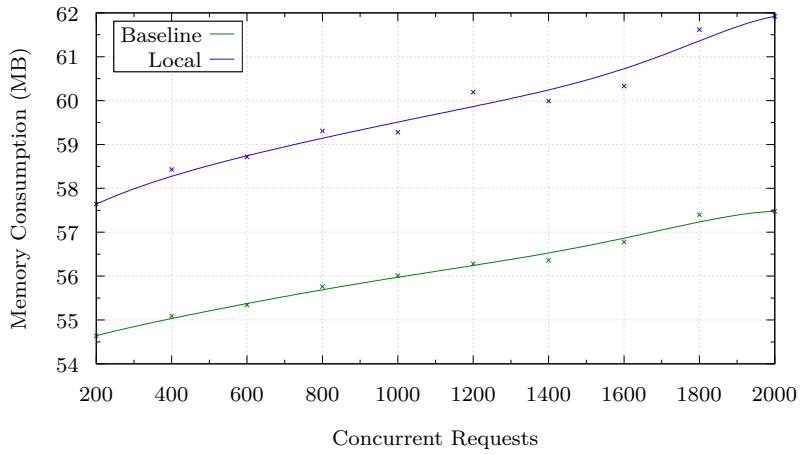
Figure 5.3: Performance measurements for the unmonitored system, local and global monitoring (Ranch with four acceptors).

This can be explained if one considers the sharp increase in memory consumption, as demonstrated by Figure 5.3a. Such a steep trend line can only result from the instantiation of a large amount of monitors, made possible by the higher throughput due to the two extra acceptors. The considerable memory load induced by these monitors slows the system drastically, and may possibly be the cause of the high number of request errors obtained. One may note that the CPU utilisation plot does not dip like the one in Figure 5.2b, suggesting that perhaps, rather than being refused as was the case with two acceptors, requests were taking too long to be serviced, finally timing out. As in the previous plots in Figure 5.2, the relationship between the baseline and local monitoring plots is hardly perceivable. These two are studied and compared next.

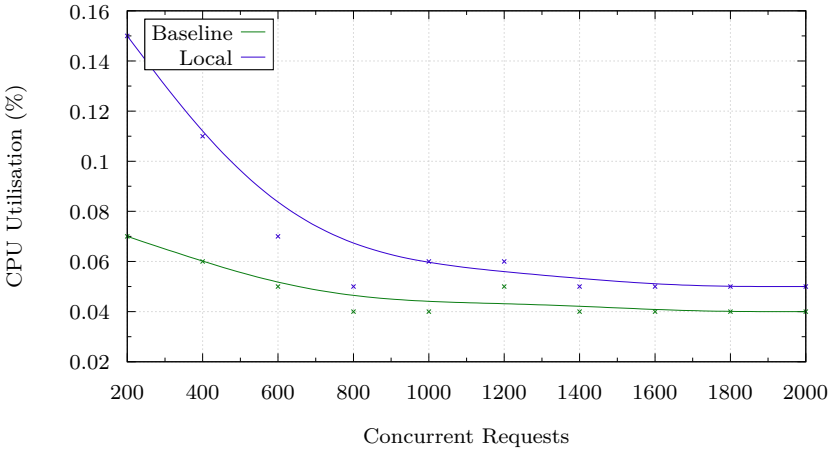
A Realistic Ranch Configuration

In the previous two cases, a proper evaluation of the Cowboy-Ranch setup was stymied by the *high overheads* due to global monitoring. Yet, these served to clearly establish the performance advantages attributed to local monitoring. This next evaluation focuses on the behaviour of local monitoring when applied to a *realistic* Ranch setup configured with the recommended number of one hundred acceptors. Evaluating such a configuration is particularly interesting, since it sheds light on how a locally monitored server might behave under production-level conditions. Moreover, it makes it possible to ascertain whether the behaviour of local monitoring under controlled conditions (see results in Section 4.5) is *comparable* to the one observed in a real-world setting.

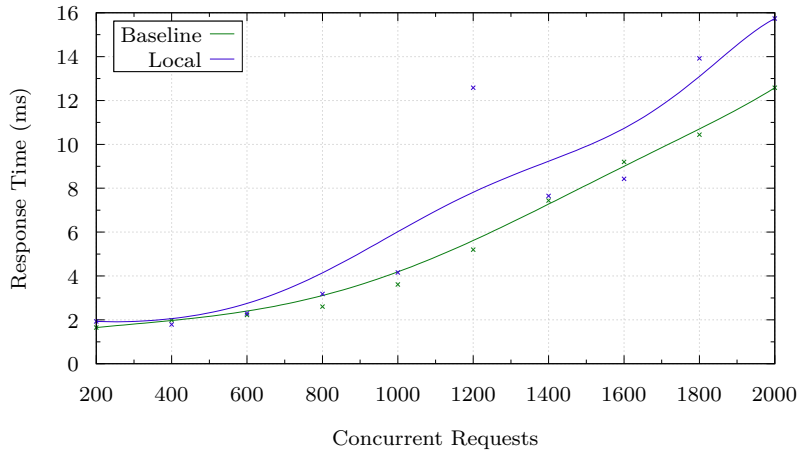
The performance measurement plots of the unmonitored and locally monitored system shown in Figure 5.4 indicate that the overheads induced by local monitoring are reasonably low, and somewhat close to those of the unmonitored system. More importantly, each local monitoring trend line in the three performance plots exhibits an *analogous* rate of change to that of the unmonitored system. This is highly indicative of the possibly of local monitoring inducing the additional performance



(a) Concurrent Requests *vs.* Memory Consumption



(b) Concurrent Requests *vs.* CPU Utilisation



(c) Concurrent Requests *vs.* Response Time

Figure 5.4: Performance measurements for the unmonitored system and local monitoring (Ranch with one hundred acceptors).

overheads in a manner that follows the same resource consumption pattern as the one for unmonitored system.

In such cases, one would be able to *forecast* the additional resource requirements that would be introduced by the use of local monitoring, well ahead of the system's deployment. While much more experimentation and data collection is needed to substantiate this conjecture, an indication of what might happen can be obtained by fitting each of the plots in Figure 5.4 with appropriate equations that can be used to extrapolate the results by extending the x-axis range as required.

5.4 Conclusion

This chapter explored the extent to which local monitoring can be effectively applied in practice to a third party system, specifically designed and configured for *real-world* use cases. It focuses on runtime verifying a fragment of the Ranch acceptor protocol discussed in Section 5.1. The appraisal reassesses the quantitative results achieved earlier in the controlled experiments conducted in Chapter 4. Furthermore, it reaffirms the advantages local monitoring has over its global counterpart. The data plots obtained when evaluating the Ranch configuration with one hundred acceptors particularly demonstrate with a high degree of confidence the viability of possible pragmatic applications of local monitoring. Lastly, the data plots seen so far (see also Chapter 4) strongly suggest that a global monitoring approach is *bound to fail* in practice, due to the substantial amounts of memory it consumes.

6. Towards Dynamic Local Monitoring

At its crudest, dynamic local monitoring describes the problem of creating monitors on demand for some target system component that is instantiated at runtime. This process not only depends on the dynamic behaviour of the system being observed, but more so, on the specifications that determine which system components ought to be monitored. The local monitoring of static components as seen in Chapter 4, can be perceived as a degenerate case of dynamic local monitoring where the monitored components are created once the system starts, and persist throughout its entire lifetime.

This chapter discusses a first preliminary attempt at developing a dynamic local monitoring algorithm for concurrent (actor-based) systems. The following topics are covered:

- The basics and general idea behind dynamic local monitoring;
- A presentation of a generic dynamic local monitoring algorithm accompanied by a discussion on the challenges involved when adapting the algorithm to an actor-based scenario;
- A high level overview of a proof of concept implementation in Erlang.

6.1 An Overview of Dynamic Local Monitoring

Dynamic local monitoring extends the principles of local monitoring discussed earlier in Section 4.1 to a setting where monitored components are created dynamically at runtime. Handling these types of scenarios requires the monitor arrangement to follow closely that of the observed system, so that monitors are created and attached to new components when needed, and discarded when having outlived their use. Practical implementations of dynamic monitoring usually rely on the observation of certain key (*e.g.* object instantiation or process spawning) events from the trace as cues to create monitors on-the-fly.

Many monitoring framework implementations employ a central in-memory storage that permits them to keep track of all the local monitors in existence, *e.g.* the tool in [19]. Typically, this storage takes the form of a map that maintains an association between the object or component unique identifier and the monitor, as this allows the framework to instantly access monitors as required. To achieve efficiency, these data structures are also often appropriately indexed. Depending on the type of algorithm used to process trace events, the framework either maintains the entire monitor instance in the data structure, or simply the present value of the monitor state (see [13, 36] for details). Some implementations also use this data structure to determine when monitors ought to be garbage collected.

Dynamism makes it possible to express certain properties that *cannot* be specified using a static approach, simply because one would not know in advance what target system components are created at runtime. For instance, while it is very easy to specify local properties over the Ranch acceptors from Chapter 5, specifying local properties over the Ranch protocol handlers is not easily achieved. Dynamic monitoring makes it possible to handle both cases in the same manner. With proper garbage collection, dynamic monitoring can also maintain small overall overheads in cases where the size of the monitored system scales down in periods of low activity.

6.2 Implementation Challenges

An implementation of dynamic local monitoring targeting process-oriented scenarios needs to consider multiple aspects that determine how the framework can be developed. Dynamic trace localisation can be tackled using two methods. In an *application-level* approach, the monitoring framework extracts events from the global trace, and based on some routing scheme, *emulates* local traces by forwarding events to individual monitors (see [13, 17, 36]). In a *language framework-level* approach, the monitoring framework relies directly on the native tracing mechanism offered by the language in order to achieve *real* localisation at the framework level. For instance, the local monitoring solution presented earlier in Chapter 4 uses the latter technique.

The algorithm presented in this chapter also uses language framework-level tracing to implement dynamic local monitoring. This approach involves processing the trace, and gradually bifurcating it into smaller subtraces as new monitors are created. Initially, a root tracer t_1 is subscribed to the global trace l_1. This observes the trace for occurrences of process `spawn` events that arise when new processes are created by the target system being monitored. When such an event is encountered, the root tracer instantiates a second child tracer t_2, *unsubscribes* it from the parent trace l_1, and *resubscribes* it to a freshly created local trace l_2 that is *only* associated with the newly spawned target system process. The child tracer t_2 then handles its local trace l_2 in the same manner as done by its parent tracer t_1, such that if a `spawn` trace event is observed on l_2, child tracer t_2 reacts by instantiating a new (grand) child tracer t_3, unsubscribing it from its local trace l_2 and resubscribing it to a freshly created local trace l_3, and so forth. *Progressively* splitting parent traces into smaller child (local) traces as described gives rise to a *tree of tracers*, each with its own local trace that is *independently* managed.

There are two salient points that need to be considered in view of the above algorithm. First, tracers (and correspondingly, monitors) are *not* created for each `spawn` event encountered in the trace, but *only* for `spawn` events that are relevant.

Relevant `spawn` events are those attributed to system processes that are *targeted* by local formulae specifications. Second, unsubscribing child tracers from the parent trace and resubscribing them to their own local trace ensures that each trace event produced by the monitored system appears *once* in any of the existing local traces, *i.e.,* is processed by just one tracer. The Erlang native tracing mechanism prevents duplicate trace messages from ever occurring by allowing only one tracer to be subscribed with a process trace at any point in time.

6.2.1 Trace Event Loss

Section 4.2 discussed that in order to prevent trace event loss, tracers should be subscribed with the Erlang VM *before* the system components that require monitoring are started. A similar manifestation occurs in the dynamic local monitoring algorithm described above. The issue arises at the point where a newly instantiated child tracer is unsubscribed from its parent trace and resubscribed to its own local trace. At the instant when the child tracer is not subscribed with neither the parent trace nor its own local trace, the target system component may forge ahead, and any trace events that could have possibly been elicited from this component are lost because no tracer is set to receive them. This is addressed using one of the two approaches below:

- **Process Suspension**: Suspending the system process being monitored before the newly instantiated child tracer is switched from the parent trace to its own local trace, prevents said process from actioning any new trace events. Paused processes are however still able to receive messages in their mailbox due to the asynchronous communication model adopted by actor-based frameworks. Message deposits raise trace events, even though deposited messages are not consumed by the process. In view of this, process suspension is only partially effective, although its use can reduce the loss of trace events by a significant amount;

- **Atomic Switch Between Traces**: Unsubscribing the child tracer from the parent trace and resubscribing it to its own local trace using a single atomic operation mitigates the possibility of event loss entirely. Yet, this solution depends wholly on whether the language framework being used offers such an API call.

6.2.2 Trace Event Routing

When a child tracer is being instantiated on account of a `spawn` event observed in the trace, the corresponding target system process that instigated the event continues with its execution. Prior to subscribing to its own local trace, all trace events resulting from the executing monitored process are directed to the *parent* tracer's mailbox instead of the child tracer's mailbox (see Section 2.4.1). As soon as the child tracer subscribes to its local trace however, the delivery of trace event messages switches from the parent tracer's mailbox to the child tracer's mailbox. The aftermath of this transaction results in the parent having a portion of the trace events that should have been delivered to the child tracer, whereas the child tracer is now the one *actively receiving* trace events from its local trace. To preserve the temporal order of trace messages and guarantee that the child tracer consumes trace events in the same order as *intended* by the language framework, the following trace event delivery scheme is used:

1. The parent tracer forwards all the trace event messages that were destined for child tracer to the child tracer's mailbox. Forwarded trace event messages are marked with a *high priority* tag in order to inform the child tracer that these are to be processed first;

2. Once all the relevant trace event messages have been forwarded to the child tracer, the parent tracer also forwards a special `stop` marker which denotes the end of the trace event message stream. Going forward, the parent tracer is no longer responsible for delivering trace events to the child tracer;

3. The child tracer consumes all high priority messages in the order these were enqueued in its mailbox. To be able to distinguish between the high priority events forwarded to it by the parent tracer and the trace events being actively deposited into its mailbox (from the local trace), the child tracer employs selective message reception (see discussion in Section 4.2);

4. As soon as the `stop` marker is encountered, the child tracer starts processing its own local trace event messages currently (accumulating) in its mailbox.

Parent tracer processes can instantiate an arbitrary number of child tracer processes. In order to ensure the correct delivery of trace event messages to child tracers, (parent) tracer processes maintain a routing table that is consulted before messages are forwarded. If for some message being handled, an entry is found inside the routing table, said message is forwarded to the *immediate* descendant tracer, otherwise, the message is handled by the parent tracer *itself*. The immediate child tracer performs the same routing operation, and based on the table's contents, either forwards the trace event to its immediate descendant or handles it itself. This indirect forwarding of trace messages from one tracer to the next means that in the worst case scenario, trace messages intended for some tracer situated in the *extremities* of the tracer tree need to go through all of its ancestors. This rarely happens in practice because newly created tracers are almost always instantly subscribed to their local trace, and only minimal forwarding takes place.

It is interesting to note that in spite of the fact that the dynamic local monitoring algorithm described relies on the native functionality offered by the the language framework, an application-level form of trace localisation is also employed. This is required to mitigate the issues that arise from the concurrent execution of processes.

6.2.3 Routing Table Management

The routing table maintained within each tracer underpins the trace event forwarding algorithm described above. It consists of a list of pairs where the key corresponds

to the monitored process PID, and the value, to the tracer process PID. Entries in the table are added when relevant `spawn` events are encountered in the trace being observed, and removed when `stop` messages are *sent by the current (parent) tracer* to some child tracer. This is on account of the (parent) tracer having fulfilled its role as a router for the particular child tracer, which is now capable of extracting trace events from its own local trace. Correct maintenance of the routing table is essential not only because it dictates the manner in which trace event messages are forwarded from one tracer to the next, but also because of its role in managing garbage collection.

6.2.4 Garbage Collection

Tracers can exist in one of two states: active or passive. *Active* tracers serve the dual purpose of (i) routing trace event messages to other child tracers, and simultaneously, (ii) extracting events from the local trace and directing these to their attached monitor. Active tracers are switched *irreversibly* to *passive* mode once their attached monitor terminates (either by crashing or by flagging a detection). Garbage collecting unused tracers depends on their state and whether their internal routing table is empty. Passive tracers with an empty routing table can be safely garbage collected because these serve neither routing nor monitoring purposes. On the other hand, passive tracers with a non-empty routing table still need to fulfil their responsibility as routers to other tracers. However, each time the routing table is swept (*i.e.,* when a `stop` event is sent to some child tracer), the garbage collection mechanism runs to determine whether the tracer ought to be discarded or retained.

The dynamic local monitoring algorithm description given above hardly ever mentions monitors. This is because localisation concerns itself with the correct placement of tracers, these, being the main devices upon which the dynamic and local aspects of monitoring rest. Monitors attached to tracers are merely trace consuming machines, *agnostic* of the manner with which trace events are extracted

from the target system. Rather, their function makes sense only when considered in the broader context of tracers and the way these are implanted into the target system.

6.3 A Preliminary Proof of Concept

A proof of concept implementation of the dynamic local monitoring algorithm described above is tackled next. To adequately adapt the generic algorithm for Erlang use, a number of design choices were made; these are briefly examined below:

- **Monitor Loading**: A load specification scheme similar to the one in Section 4.2 is used to associate local monitors with Erlang function specifications. Erlang function specifications take the form of the tuple `{M, F, A}`, where `M` denotes the module name, `F`, the function name, and `A`, the list of arguments passed to `F`. This is required since `spawn` trace events generated by Erlang contain the `{M, F, A}` tuple used to spawn the processes. Comparing the observed `spawn` event contents against `{M, F, A}` entries contained in the load specification enables the tracer to determine whether the event is relevant. Relevant `spawn` events result in the creation of new child tracers;

- **A Lossy Implementation**: Erlang does not provide an appropriate BIF that permits the atomic trace switch discussed in the previous section to be affected. Instead, the implementation uses process suspension. Suspended processes are effectively blocked from executing, and are therefore prevented from *initiating* trace events intentionally. The only exception are `receive` events, which can still be observed in the trace if some message is delivered to the traced process' mailbox. Though process suspension does not achieve a fully lossless implementation, it appropriately handles all the possible types of Erlang trace events, save for `receive`;

- **Tracing Spawned Processes Automatically**: Monitored processes can themselves spawn other child processes. Depending on the tracing scenario, one may require that these newly spawned child processes are also traced. The implementation achieves this using Erlang's `set_on_spawn` trace flag that permits newly spawned child processes to inherit their parent process' trace flags. This is convenient for two reasons. First it automates the job of manually setting trace flags on each spawned process, and second, trace events initiated by the spawned child processes are still directed to the parent process tracer (see Section 2.4.1). In the dynamic local monitoring described above, parent tracers *depend* on having access to a trace that contains all of the events being generated by a particular subsystem or component (*i.e.,* not necessarily a single process) in order to determine when to split the trace into smaller subtraces.

6.3.1 An Example

The following example illustrates how dynamic local monitoring works when employed to monitor a simple system consisting of four processes: A, B, C and D. Process A acts as system's entry point from which all the other processes are spawned according to the assumed spawn order shown below:

1. Process A spawns B first;
2. Then it spawns process C;
3. Finally, process B spawns D.

Out of these four processes, C is not monitored, while the rest are monitored according to the following load specification:

$$\texttt{load\_spec} = [\{\{M_a, F_a, A_a\}, Mon_a\}, \{\{M_b, F_b, A_b\}, Mon_b\}, \{\{M_d, F_d, A_d\}, Mon_d\}]$$

Figure 6.1 shows the final configuration of the monitored system that is attained after all the processes have been launched as per the order given above. Although

the monitors are not shown in the figure, these are actually attached to their corresponding tracers and started once the latter are *subscribed* to their respective local traces. The configuration also includes the launcher component encountered earlier in Section 4.2; this is responsible for initialising the dynamic local monitoring algorithm, and starting the system as follows:

❶ The launcher sets up the root tracer Trc_l;

❷ Then, it launches process A;

❸ A `spawn` event with $\{\mathtt{M}_a, \mathtt{F}_a, \mathtt{A}_a\}$ is sent to Trc_l;

❹ Trc_l reads the `spawn` event with $\{\mathtt{M}_a, \mathtt{F}_a, \mathtt{A}_a\}$, queries `load_spec`, finds a match and instantiates a new tracer Trc_a for process A;

❺ Process A is paused while Trc_a is unsubscribed from Trc_l's trace, and is resubscribed to its own local trace. A is then resumed. Tracer Trc_l no longer receives trace events for process A;

❻ Process A spawns B;

❼ A `spawn` event with $\{\mathtt{M}_b, \mathtt{F}_b, \mathtt{A}_b\}$ is sent to Trc_a;

❽ Trc_a reads the `spawn` event with $\{\mathtt{M}_b, \mathtt{F}_b, \mathtt{A}_b\}$, queries `load_spec`, finds a match and instantiates a new tracer Trc_b for process B;

❾ Process B is paused while Trc_b is unsubscribed from Trc_a's trace, and is resubscribed to its own local trace. B is then resumed. Tracer Trc_a no longer receives trace events for process B;

❿ Process A spawns C;

⓫ A `spawn` event with $\{\mathtt{M}_c, \mathtt{F}_c, \mathtt{A}_c\}$ is sent to Trc_a. This event is not found inside `load_spec`, and *no tracer is instantiated for* C;

⓬ Process B spawns D;

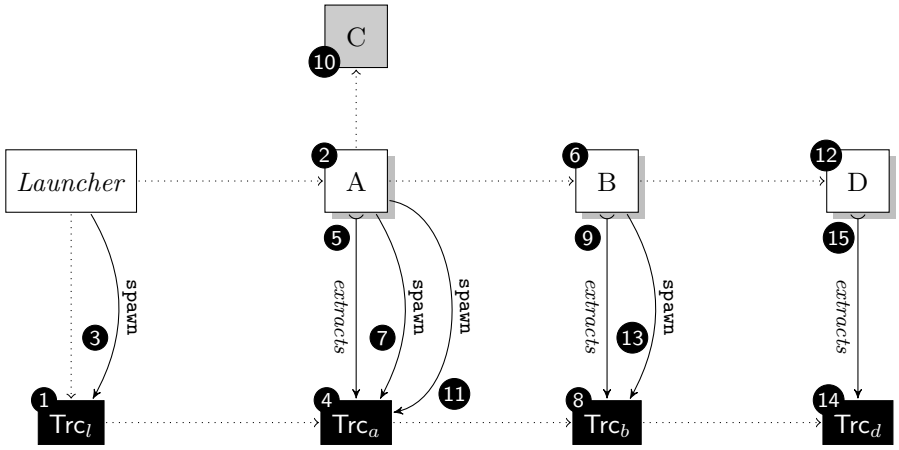


Figure 6.1: The final process configuration of the monitored system after dynamic local monitoring is applied.

13. A `spawn` event with $\{M_d, F_d, A_d\}$ is sent to Trc_b;

14. Trc_b reads the spawn event with $\{M_d, F_d, A_d\}$, queries `load_spec`, finds a match and instantiates a new tracer Trc_d for process D;

15. Process D is paused while Trc_d is unsubscribed from Trc_b's trace, and is subscribed to its own local trace. D is then resumed. Tracer Trc_b no longer receives trace events for process D.

The reader should note that although the monitored processes are spawned in the order assumed above, different interleavings can occur in a real execution. For instance, while Trc_l is busy processing the `spawn` event sent to it from process A, the latter may have already spawned processes B and C, in which case, trace event routing is applied (see discussion in Section 6.2) to resolve the issue. This example was chosen for elucidative purposes because it portrays the simplest, most straightforward interleaving scenario.

6.4 Conclusion

Dynamic local monitoring makes it possible to specify properties over systems whose number of components can only be determined at runtime. The generic algorithm

described in this chapter attempts to address this in the context of concurrent actor-based systems. A preliminary proof of concept implementation in Erlang was discussed, followed by a demonstrative example that shows how the algorithm works in practice.

There are a number of points which have not been satisfactorily explored due to time constraints. Chief among them is a detailed study of the performance overheads that are induced when dynamic local monitoring is used, and how these compare with the results obtained in Chapters 4 and 5. It would also be interesting to come up with a number of tests that determine the extent to which the algorithm is lossy w.r.t. `receive` trace events that may be potentially missed when a tracer switches from the parent trace to its local trace. Other design improvements have yet to be considered, whereas the overall implementation may require further refinement.

7. Conclusion

The material presented in this dissertation is motivated by the need to address the scalability, rigidity and performance issues that arise when a global monitoring strategy is employed in concurrent scenarios. This intent takes the form of a thorough study that explores how these shortcomings can be mitigated if a localised monitoring approach is adopted instead. Although this investigation is by no means the first to suggest the use of localisation to monitor the behaviour of component-based systems (see Section 7.2), its applicability and advantages have never been investigated.

This work also centres on the benefits that may be gained if the task of runtime monitoring is approached from a *modular* stance. Particular focus is placed on the level of *effectiveness* with which a localised approach can be employed to monitor various component-based architectures using an *online*, *asynchronous* strategy. The analysis conducted in this report provides a detailed comparison between local and global monitoring, as a means to better understand and assess how the benefits attributed to local monitoring outweigh those of its global counterpart.

The contributions of this work *vis-à-vis* the objectives and assessment criteria presented in Section 1.2 are thus:

1. The implementation of prototype RV tool that synthesises asynchronous local and global monitors from safety and co-safety formulae specified using the formalism in Chapter 3;

2. A study on the *effectiveness* of local monitoring. In particular, the technique (i) makes it possible to write small, well-structured and manageable formulae that promote *understandable* and *maintainable* specification scripts which are less likely to suffer from re-factoring in cases where new components are added to the system under scrutiny, (ii) promotes *fault tolerance* through the synthesis of isolated monitors that help achieve a *functionally segregated* system, (iii) has a *lower performance impact* when compared to global monitoring on the basis of memory consumption, CPU utilisation and system response time. The results also show that local monitoring is *most effectively* applied on components that do not interact with one another;

3. A qualitative and quantitative appraisal of a *real-world* application of local and global monitoring that corroborates the results obtained in Contribution 2;

4. The preliminary design of an algorithm that extends local monitoring to *dynamically* instantiated system components.

The applicability of the material presented in this dissertation is not limited to the host technology where the experiments were conducted, nor is it bound to the specification logic used. Rather, these findings should be regarded from an *implementation-agnostic* perspective, making them applicable to any concurrent scenario where components can be delineated into subsystems that can be *separately* traced. For instance, the conclusions obtained in this study would equally hold if the component-based system under study is developed in Scala and analysed using local monitors synthesised from LTL formulae.

The same benefits gained from localisation should also be enjoyed in a *distributed* setting. Local monitors dispersed over multiple (remote) locations function in isolation, as the extraction of system trace events is conducted locally by each monitor on site. As in the centralised case, local monitors are spared the communication overheads that afflict global monitoring approaches. The advantages enumerated in Contribution 2 above should thus be retained, for the most part.

Before concluding, there are a couple of points that merit the reader's consideration. First, attention should be drawn to the fact that all the quantitative experiments presented in this work employ the monitors introduced in Chapter 3. As explained earlier, continuous formula unfolding causes these monitors to grow increasingly larger — a consequence attributed to the lack of a suitable garbage collection mechanism. Despite the fact that this shortcoming afflicts both local and global monitors, the latter may, in cases, suffer more from this because global specifications tend to be quite large when compared to their local counterparts. While this is best kept in mind when interpreting the results rendered on the plots in Chapters 4 and 5, it in *no* way impinges on the conclusions drawn from the study.

Second, although the case study in Chapter 5 did not consider the dynamic scenario wherein Ranch protocol handlers are created on demand, the conclusions drawn from the study should mostly apply in this setting as well. At their very essence, the static and dynamic flavours of local monitoring address the *logistical* problem of monitor instantiation. In the static case, monitors are attached to a designated set of components that have been identified *beforehand* (see discussion in Section 4.1), whereas in the dynamic case, monitors are instantiated for target system components that are spawned at runtime. Regardless of the instantiation scheme used, once a local monitor is created and subscribed with its local trace, its behaviour is no different from that of other local monitors. However, there is one condition that must be taken into account. While in static cases, monitors are created *before* the target system starts (see Section 4.2), creating monitors dynamically on demand can induce resource overhead spikes that affect the system's performance. Depending on the component organisation within the system under observation and the frequency with which monitors are spawned, this may, occasionally, render the dynamic instantiation of local monitors fractionally slower.

7.1 Future Work

While the contributions presented in this work are in accordance with the original aims and objectives outlined in Section 1.2, there are other aspects worthy of further investigation:

Evaluation of Dynamic Monitoring Ascertaining the effectiveness of the dynamic local monitoring approach from Chapter 6 increases confidence in its practical applicability. While the qualitative results obtained in Section 4.4 apply equally to this dynamic setting, its quantitative aspect must be reassessed in order to determine how dynamically instantiated monitors affect the overall system performance. Two possible evaluation scenarios have been identified. In the first scenario, monitors are instantiated over long-lived components that when created, tend to execute for relatively long periods of time (*e.g.* Ranch acceptors). In the second scenario, monitors are instantiated over short-lived components that are usually spawned to perform minute, specific tasks, after which these are immediately discarded (*e.g.* Ranch protocol handlers). The findings will make it possible to identify specific situations where dynamic local monitoring is advantageous. Moreover, these also serve to determine the degree of efficacy with which the lossy algorithm presented in Chapter 6 performs in practice.

Garbage Collection The dynamic monitoring algorithm already handles garbage collection at the *monitor* level. This can be further optimised if redundant processes within monitor arrangements themselves are also discarded. A previous work in [11] proposes a dynamic, online garbage collection algorithm that *re-configures* the monitor hierarchy in order to eliminate redundant submonitors (refer to Section 3.2.2 for details on the monitor synthesis procedure), and keep its arrangement optimal (*i.e.,* as shallow as possible).

Distribution Localisation, as studied in this work, targets asynchronous scenarios. The tool developed in Chapter 4 relies on non-blocking message passing and

process encapsulation, making monitor interdependencies and data synchronisation issues of little concern. Such an architecture fits naturally within the constraints of distributed implementations, and one conjectures that distribution is an extension of the existing tool. This can be employed as a means to lower the monitoring overheads further.

7.2 Related Work

The prototype implementation from Chapter 3 builds upon detectEr, a RV tool developed as part of the work in [23]. detectEr formalises the notion of monitor correctness and applies this in practice to synthesise concurrent runtime monitors that are correct w.r.t. their specification. It shares a number of common aspects with the tool developed in Chapter 3 — for instance, correctness properties are specified using safety formulae expressed in terms of sHML, uses Erlang as a target language, and performs asynchronous instrumentation. However, it differs in these respects: (i) the tool in Chapter 3 considers a substantially larger syntactic monitorable subset of μHML, making it possible to specify positive properties, (ii) supports action patterns which complicates the modularity of the synthesis process, and (iii) also supports localised monitors, unlike detectEr which is limited to global monitoring.

Trace localisation is approached from a different angle in the work presented in [29]. Instead of relying on the native trace event extraction mechanism provided by the language platform, the authors achieve locality using a technique called *parametric trace slicing* [13, 36]. This method perceives events in the execution trace as being one of two kinds: *propositional* events consisting of simple event names, *e.g.* `open`, or *parametric* events that include one or more associated data values, *e.g.* `open("f1")`. *Parametric properties* are specified in terms of symbolic events that are matched to concrete event instantiations from the trace by binding values in the concrete events to parameters in the symbolic events. In practice, parametric specifications are commonly used, especially in languages where properties need to

be specified over instances of some unit of encapsulation (*e.g.* objects in Java, actors in Erlang). At a conceptual level, slicing works by employing *parametric binding* to chop the parametrised input trace into smaller *propositional* subtraces based on the data values in events. Each subtrace is then consumed by separate, dedicated monitors that process propositional events from these localised traces. For example, the property *"an open file event is followed by a close file event"*[1] specified in terms of the parametric regular expression `(open(`f`).close(`f`))*` slices the parametric trace `open(`"f1"`).open(`"f2"`).close(`"f1"`)` into the following two propositional subtraces: `open.close` and `open` according to the binding of parameter f with the event parameter values "f1" and "f2" respectively. The first subtrace satisfies the property, whereas the second does not. Parametric trace slicing prompts flexibility by effectively decoupling the parameter binding process from property checking, although, using a native tracing mechanism as done in Chapter 4 achieves *true* trace segregation that is managed at a lower abstraction level, namely the language framework-level. One should not confuse slicing with *extraction*: the former routes parametric events from the central trace to subtraces, whereas the latter may choose to hide or withhold certain events from being delivered to the monitor.

JavaMOP [29], a RV tool targeting Java applications, relies on parametric trace slicing to efficiently conduct monitoring on localised traces. It extracts trace events from applications through the use of AspectJ. Specifications in JavaMOP consist of two parts: the part that defines the events, and the part that declares correctness properties over these events. The tool developed in this dissertation distinguishes itself from JavaMOP in the following: (i) it synthesises asynchronous monitors, whereas JavaMOP employs synchronous monitors, (ii) the flavour of asynchronous monitors employed requires minimal instrumentation, as opposed to a fully-fledged AOP approach as in JavaMOP, (iii) concurrency requires correctness specifications to deal with execution interleaving, unlike in JavaMOP, and (iv) while the tool in Chapter 3 is limited to detections only, JavaMOP permits the user to specify optional

[1]The example is adapted from [7].

handling code that is invoked when violations or validations are encountered.

Parametric trace slicing suffers from three shortcomings that limit its expressivity [7]. First, parametric events in the same property cannot be associated with different lists of variables (*e.g.* `(open(`f`).close(`g`))*`), because trace slicing assumes unique parameter bindings when creating subtraces. Second, it also assumes that *all* parameters in an event participate in slicing, thereby fixing their values once these are bound with events from the trace (*e.g.* for the event `write(`f`,` *bytes*`)`, variable f should remain fixed while *bytes* must be permitted to become free once the `write` trace event is processed, as this makes it possible to handle multiple write events with different byte content written to the *same* file). Third, it also implicitly assumes universal quantification, meaning that the trace is sliced *for each* value that becomes bound to parameters in the property.

The work in [7] introduces a specification formalism called Quantified Event Automata (QEA) that addresses these deficiencies. A QEA can be thought of as a pair consisting of a list of quantified variables and an event automaton. An event automation is a FSM instantiation whose transitions are labelled with *parametric events*, and optionally, guards and assignments. Parameters specified inside the automaton's transition labels are *bound* if they also appear in the list of quantified variables in the automaton. Quantified variables indicate that an automation instance is to be created *for each* variable binding, while variables that are not quantified (*i.e., free*) denote placeholders that are *local* to each automation instance. To illustrate, if the file name variable f is quantified in a QEA specified over the parametric trace `write(`"f1"`,` "abc"`).write(`"f1"`,` "def"`)`, a single event automaton is created for the file "f1" and the local (free) variable *bytes* is bound two times, first with "abc", and then with "def'. Guards and assignments can manipulate the values in both quantified and free variables as required.

MarQ [34, 33] is a preliminary implementation that extends parametric trace slicing through the use of QEA. Like JavaMOP, it targets systems developed in Java, and produces synchronous dynamic monitors that are *weaved* into the target appli-

cation using AspectJ. While the trace localisation technique developed in Chapter 4 implicitly handles universal quantification, like MarQ, it can also emulate, to an extent, existential quantification using the pre-binding operator @ from Chapter 3[2]. In addition to the differences between the tool presented in this work and JavaMOP, MarQ *adds* the following: (i) variable assignments in QEA modify mutable variables that are not thread-safe and therefore, require proper synchronisation when used in multithreaded scenarios, and (ii) in addition to online monitoring, MarQ also supports offline monitoring of traces stored in XML and CSV files. At the time of writing, QEA specifications in MarQ must be built manually in Java code using the libraries provided for this purpose. Additionally, monitors produced from the QEA specifications must be manually implanted into the system using hand-coded AspectJ aspects. These shortcomings will be mitigated in future versions of MarQ.

Larva [19] is another MOP-based tool targeting Java applications. In Larva, correctness properties are written using a textual representation of Dynamic Automata with Timers and Events (DATEs)[18], and like JavaMOP, divides specification scripts into two parts: one that defines events, and one that declares properties on these events. Events are defined using fragments of the AspectJ pointcut syntax that targets *join points* in the application. Larva synthesises synchronous monitors which are weaved into the target system using AOP techniques. Apart from global properties, Larva, similar to JavaMOP and MarQ, also supports the specification of local properties through its use of the `foreach` construct. Specifications using `foreach` are parametrised with the *type* of the Java language class that is to be monitored locally, in order to instruct Larva to dynamically instantiate monitors *for each* of the targeted object at runtime. This mechanism makes it possible to write local specifications that permit different monitors to target the same event; contrastingly, in the monitoring approach developed in Chapter 4, events from the local trace can only be read by a single receiver monitor. In addition, Larva also supports monitor

[2]For this to be successful however, the registered name of the Erlang process to be monitored must be known.

communication via global variables and broadcast channels. Besides the mentioned differences, the contrasts drawn earlier when discussing JavaMOP and MarQ also apply to this comparison of Larva and the tool developed in this work.

JavaMOP, MarQ and Larva target object-oriented frameworks, as opposed to process-oriented frameworks, where each component is assigned its own thread of execution. ELarva [17], an Erlang port of Larva, was created with the intent of bridging this gap, as well as to expand the Larva tool set to a process-oriented scenario. It relies on the native tracing mechanism provided by Erlang (see Chapters 2 and 3), rendering the monitoring effort totally asynchronous, while disposing of the instrumentation component that was previously required to elicit trace events. The `foreach` construct semantics are reformulated so that monitors target process instances instead of objects. Unlike the technique developed in Chapter 4, where localisation is achieved by subscribing to subtraces at the Erlang VM-level, ELarva relies on a centrally-managed singleton tracer that extracts trace events and demultiplexes between *all* local monitor instances in order to direct these events as required. Although this scheme handles the problem of dynamic local monitor creation easily, the additional processing induces unnecessary overheads (*e.g.* bottlenecks) that can be altogether avoided if local monitors are *directly* subscribed to subtraces. Furthermore, funnelling trace events through a central process runs the risk of introducing a single point of failure that can adversely affect the entire monitoring endeavour. This contrasts with a truly localised strategy where failures in some monitor or monitored component are contained to the affected subsystem (see conclusion in Chapter 4). Similar to its Java counterpart, ELarva also supports monitor communication, albeit in Erlang, this is given a point-to-point interpretation, in order to reduce the amount of traffic that is generated with broadcast-style messaging used by Larva.

Localisation requires the specifier to perceive the target system as a collection of components, each having its own local behaviour that contributes towards the system's global behaviour. In [9], and later in [16], the authors present an approach

where *global* LTL formulae are monitored locally over subtraces. Their work concentrates on how global specifications can be verified in the absence of a central decision making point that asserts whether the target system's behaviour is violated or validated. To achieve this, the authors propose a *decentralised* algorithm that can effectively rewrite some global LTL formula into smaller subformulae that are then monitored over local traces. Dependencies between distributed local formulae are handled by enabling monitors to communicate partial evaluations to other monitors, thereby *progressively* evaluating individual subformulae until a *global* verdict is reached. Like the decentralised approach developed in this dissertation, efficiency in terms of performance gains is also central in [9]. There are a number of differences that distinguish both approaches however: (i) this work adopts a granular view focusing on different target system components, whereas the authors in [9] perceive the system from a global standpoint, (ii) monitors work in complete isolation, as opposed to the ones in [9] which depend on communication channels to be able to evaluate formulae, (iii) while the presented work focuses on asynchronous monitoring, the technique applied in [9] targets synchronous systems, and (iv) synchronous monitoring is spared the interleaving issues that would make a global specification approach unmanageable in concurrent settings (see Chapter 4). In contrast to the dynamic RV tools discussed above, the one in [9] assumes a static system architecture wherein local monitors are deployed before the verification process commences. This is similar to the approach developed in Chapter 4, which also requires knowledge of the target system components that are to be monitored before the actual monitoring can be performed.

Before concluding, there are a number of general comments that ought to be made in view of the above discussion. The aforementioned tools, together with the one developed in this dissertation, tackle online RV, and present different strategies by which monitoring can be rendered more efficient. As already seen in Section 4.5, efficiency and performance are of the utmost importance in online approaches, as these ultimately determine the tools' usability in practical settings. On the one

hand, achieving localisation at a language framework level (*e.g.* using Erlang's native tracing API) increases efficiency, addresses fault tolerance, and removes the burden of having to manage the specifics of multiple traces using manual means. On the other hand, an application-level approach (*e.g.* parametrised trace slicing) offers portability and the possibility of using the RV tool on platforms that do not offer decentralised tracing facilities. Efficiency also poses challenges when designing dynamic monitoring tools, as these need to take into account how and when monitors are to be created and garbage collected. In this regard, while the task of dynamically monitoring synchronous systems can be handled non-trivially, the asynchronous and non-deterministic nature of concurrent systems makes this problem even harder to address.

A. Refining the Monitor Synthesis

The monitor syntax from [21], together with the synthesis function $(\!|-|\!)$ from mHML to monitors is shown in Figure A.1. A refinement of the synthesis function is required in order to make the implementation of the RV tool in Chapter 3 capable of producing monitors that can correctly handle detections. Specifically, there are cases where the synthesis function in Figure A.1 produces monitors with non-deterministic behaviour.

Example A.0.1. The sHML formula φ_8 describes the property that *"after any sequence of requests and responses, a request is never followed by two consecutive responses"*, *i.e.,* the subformula $[\mathtt{resp}]\,[\mathtt{resp}]\,\mathsf{ff}$. The synthesis function in Figure A.1 translates φ_8 to the monitor process m_8.

$$\varphi_8 = \mathsf{max}\, X.\Big([\mathtt{req}]\,([\mathtt{resp}]\,X \wedge [\mathtt{resp}]\,[\mathtt{resp}]\,\mathsf{ff})\Big)$$
$$m_8 = \mathsf{rec}\, x.\Big(\mathtt{req}.(\mathtt{resp}.x \,+\, \mathtt{resp}.\mathtt{resp}.\mathsf{no})\Big)$$

Monitor m_8 may exhibit the following behaviour:

$$\mathsf{rec}\, x.\Big(\mathtt{req}.(\mathtt{resp}.x \,+\, \mathtt{resp}.\mathtt{resp}.\mathsf{no})\Big) \xrightarrow{\tau} \cdot \xrightarrow{\mathtt{req}} \mathtt{resp}.m_8 \,+\, \mathtt{resq}.\mathtt{resp}.\mathsf{no}$$

at which point, upon analysing action `resp`, it may non-deterministically transition to either m_8 or $\mathtt{resp}.\mathsf{no}$. The latter case can raise a rejection if it receives another

Syntax

$$m, n \in \textsc{Mon} ::= v \mid \alpha.m \mid m + n \mid \mathbf{rec}\, x.m \mid x$$
$$v, u \in \textsc{Verd} ::= \mathbf{no} \mid \mathbf{yes} \mid \mathbf{end}$$

Dynamic behaviour

$$\textsc{mAct}\ \frac{}{\alpha.m \xrightarrow{\alpha} m} \qquad \textsc{mRec}\ \frac{}{\mathbf{rec}\, x.m \xrightarrow{\tau} m[{}^{\mathbf{rec}\, x.m}/_{x}]} \qquad \textsc{mVer}\ \frac{}{v \xrightarrow{\alpha} v}$$

$$\textsc{mSelL}\ \frac{m \xrightarrow{\mu} m'}{m + n \xrightarrow{\mu} m'} \qquad \textsc{mSelR}\ \frac{n \xrightarrow{\mu} n'}{m + n \xrightarrow{\mu} n'}$$

Monitor synthesis

$$(\!|\mathbf{ff}|\!) \stackrel{\text{def}}{=} \mathbf{no} \qquad (\!|\mathbf{tt}|\!) \stackrel{\text{def}}{=} \mathbf{yes} \qquad (\!|X|\!) \stackrel{\text{def}}{=} x$$

$$(\!|[\alpha]\,\psi|\!) \stackrel{\text{def}}{=} \begin{cases} \alpha.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathbf{yes} \\ \mathbf{yes} & \text{otherwise} \end{cases} \qquad (\!|\langle\alpha\rangle\,\psi|\!) \stackrel{\text{def}}{=} \begin{cases} \alpha.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathbf{no} \\ \mathbf{no} & \text{otherwise} \end{cases}$$

$$(\!|\psi_1 \wedge \psi_2|\!) \stackrel{\text{def}}{=} \begin{cases} (\!|\psi_1|\!) & \text{if } (\!|\psi_2|\!) = \mathbf{yes} \\ (\!|\psi_2|\!) & \text{if } (\!|\psi_1|\!) = \mathbf{yes} \\ (\!|\psi_1|\!) + (\!|\psi_2|\!) & \text{otherwise} \end{cases} \qquad (\!|\psi_1 \vee \psi_2|\!) \stackrel{\text{def}}{=} \begin{cases} (\!|\psi_1|\!) & \text{if } (\!|\psi_2|\!) = \mathbf{no} \\ (\!|\psi_2|\!) & \text{if } (\!|\psi_1|\!) = \mathbf{no} \\ (\!|\psi_1|\!) + (\!|\psi_2|\!) & \text{otherwise} \end{cases}$$

$$(\!|\mathbf{max}\, X.\psi|\!) \stackrel{\text{def}}{=} \begin{cases} \mathbf{rec}\, x.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathbf{yes} \\ \mathbf{yes} & \text{otherwise} \end{cases} \qquad (\!|\mathbf{min}\, X.\psi|\!) \stackrel{\text{def}}{=} \begin{cases} \mathbf{rec}\, x.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathbf{no} \\ \mathbf{no} & \text{otherwise} \end{cases}$$

Figure A.1: The monitor syntax and dynamics, and the compositional synthesis function (adapted from [21]).

`resp` event but the former case, *i.e.,* m_8, does *not* — this results in a missed detection. Although this behaviour suffices for the theoretical results required in [21], it is not ideal from a practical standpoint. The problem is on account of a limitation in the choice construct semantics, $m + n$, which forces a selection between submonitor m or n upon receiving an event to analyse. ■

This problem is solved by replacing the external choice constructs with a parallel monitor composition construct, $m \times n$ that permits both m and m to process the event without excluding one another. The semantics of the $\times$ combinator are given in Figure 3.1. For completeness' sake, the symmetric versions of the rules omitted

in Figure 3.1 are given here (*n.b.* MPAR is restated for the reader's convenience).

$$\textsc{mPar}\ \frac{m \xrightarrow{\alpha} m' \quad n \xrightarrow{\alpha} n'}{m \times n \xrightarrow{\alpha} m' \times n'} \qquad \textsc{mParR}\ \frac{m \xrightarrow{\alpha}\!\!\!\!\!\!\not\quad \quad m \xrightarrow{\tau}\!\!\!\!\!\!\not\quad \quad n \xrightarrow{\alpha} n'}{m \times n \xrightarrow{\alpha} n'}$$

$$\textsc{mParSL}\ \frac{m \xrightarrow{\tau} m'}{m \times n \xrightarrow{\tau} m' \times n} \qquad \textsc{mParVR}\ \frac{}{m \times v \xrightarrow{\tau} v}$$

The synthesis function $[\![-]\!]$ shown in Figure 3.2 is defined by structural induction on the structure of the formula. Most cases are identical to those of $(\!|-|\!)$ in Figure A.1 with the exception of the two cases below, substituting the choice construct for the parallel construct:

$$[\![\psi_1 \wedge \psi_2]\!] \stackrel{\text{def}}{=} \begin{cases} [\![\psi_1]\!] & \text{if } [\![\psi_2]\!] = \textbf{yes} \\ [\![\psi_2]\!] & \text{if } [\![\psi_1]\!] = \textbf{yes} \\ [\![\psi_1]\!] \times [\![\psi_2]\!] & \text{otherwise} \end{cases} \qquad [\![\psi_1 \vee \psi_2]\!] \stackrel{\text{def}}{=} \begin{cases} [\![\psi_1]\!] & \text{if } [\![\psi_2]\!] = \textbf{no} \\ [\![\psi_2]\!] & \text{if } [\![\psi_1]\!] = \textbf{no} \\ [\![\psi_1]\!] \times [\![\psi_2]\!] & \text{otherwise} \end{cases}$$

The two monitor synthesis functions correspond in the sense of Theorem A.0.1. In [21], verdicts are associated with logic satisfactions and violations, and thus Theorem A.0.1 suffices to show that the new synthesis is still correct.

Theorem A.0.1. For all $\psi \in \text{mHML}$, $(\!|m|\!) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} v$ iff $[\![m]\!] \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} v$.

Proof. By induction on the structure of ψ. Most cases are immediate because the resp. translations correspond. In the case of $\psi_1 \wedge \psi_2$, where the synthesis yields $(\!|\psi_1|\!) + (\!|\psi_2|\!)$, a verdict is reached only if $(\!|\psi_1|\!) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} v$ or $(\!|\psi_2|\!) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} v$. By I.H. the following is obtained $[\![\psi_1]\!] \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} v$ (or $[\![\psi_2]\!] \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} v$), and is sufficient to show that $[\![\psi_1]\!] \times [\![\psi_2]\!] \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} v$. A dual argument can be constructed for the implication in the opposite direction. □

Example A.0.2. Applying the synthesis function from Figure 3.2 on φ_8 (see Example A.0.1) results in a monitor that exhibits only the following behaviour:

$$\textbf{rec}\, x.(\texttt{req}.(\texttt{resp}.x \times \texttt{resp}.\texttt{resp}.\textbf{no})) \xrightarrow{\tau} \cdot \xrightarrow{\texttt{req}} \texttt{resp}.m_8 \times \texttt{resq}.\texttt{resp}.\textbf{no} \xrightarrow{\texttt{resp}} m_8 \times \texttt{resp}.\textbf{no} \xrightarrow{\texttt{resp}} \textbf{no}$$

Submonitors are now permitted to evolve together without excluding one another

and actions can be processed by independently. Transition $m_8 \times \texttt{resp}.\textbf{no} \xrightarrow{\texttt{resp}} \textbf{no}$ shows that when the action `resp` does not match the one expected by m_8 (*i.e.,* `req`), the submonitor is terminated, while `resp`.**no** is permitted to evolve into **no**, according to mParR above. ■

B. Translation From mHML to Erlang Monitors

The remaining monitor constructs from Table 3.1 introduced in Section 3.2.3 are shown in Table B.1.

Monitor construct	`formula.erl` module code

rec $x.[\![\psi]\!]$	`mon_rec(Var, Psi) ->`
	`  fun(Env) ->`
	`  Psi([{Var, Psi} | Env])`
	`  end.`

x	`mon_var(Var) ->`
	`  fun(Env) ->`
	`  Psi = look_up(Var, Env),`
	`  Psi(Env)`
	`  end.`

no	`mon_no() ->`
	`  fun(_Env) ->`
	`  sup ! ff`
	`  end.`

yes	`mon_yes() ->`
	`  fun(_Env) ->`
	`  sup ! tt`
	`  end.`

end	`mon_end() ->` `  fun(_Env) -> end.`

Table B.1: The monitor constructs and their corresponding Erlang code.

Recursion for the variable x is encoded by a new mapping that is created inside the map environment `Env` in line 3. The variable x itself encodes recursive unfolding in `mon_var`, where a monitor is first retrieved from the map environment using the function `look_up` (line 7), and re-instantiated with the same environment `Env` (line 8). Constructs **no** and **yes** are encoded by two functions that each communicate the monitoring verdict by sending `ff` and `tt` messages respectively to the supervising top-level monitor `sup` (lines 12 and 16). The inconclusive verdict **end** is encoded by an empty function that terminates the monitor gracefully (line 19).

Table B.2 shows the remaining synthesis function subcases from Table 3.2 for handling sHML translations. For formula $[\![\mathbf{max}\, X.\psi]\!]$, the translation performs the string processing required to insert a call to the function `mon_rec` (lines 5 - 6). It passes the recursion variable `Var` obtained from the parse tree (line 1), together with the submonitor source code string `Mon` generated from the parsed subtree of `Psi` (line 2), as arguments. The recursive variable X is translated into a call for the `mon_var` (line 9) function from Table B.1, whereas $[\![\mathbf{ff}]\!]$ and $[\![\mathbf{tt}]\!]$ are translated into calls for `mon_no` (line 11) and `mon_yes` (line 13) respectively. The other cases for cHML (*i.e.,* $[\![\psi_1 \vee \psi_2]\!]$, $[\![\langle\alpha\rangle\, \psi]\!]$ and $[\![\mathbf{min}\, X.\psi]\!]$) are handled in a similar manner.

Synthesis function subcase	`compiler.erl` module function
$[\![\mathbf{max}\, X.\psi]\!] \stackrel{\text{def}}{=}$ $\begin{cases} \mathbf{rec}\, x.[\![\psi]\!] & \text{if } [\![\psi]\!] \neq \mathbf{yes} \\ \mathbf{yes} & \text{otherwise} \end{cases}$	1 `synth({max, Var, Psi}) ->` 2 `  case synth(Psi) of` 3 `  {yes, _} -> {yes, "mon_tt()"};` 4 `  {Tag, Mon} ->` 5 `  {any, util:format("mon_rec(~p,~s)",` 6 `  [Var, Mon])}` 7 `  end;`
$[\![X]\!]$	8 `synth({var, Var}) ->` 9 `  {any, util:format("mon_var(~p)", [Var])};`
$[\![\mathbf{ff}]\!]$	10 `synth({ff}) ->` 11 `  {no, "mon_no()"};`
$[\![\mathbf{tt}]\!]$	12 `synth({tt}) ->` 13 `  {yes, "mon_yes()"};`

Table B.2: The monitor synthesis function cases and their corresponding compiler functions.

C. Global Monitoring for the Ranch Protocol

Formula (C.1) expresses the positive requirement in Section 5.2, *i.e., "after an acceptor sends a connection initiation request to the* `ranch_conns_sup` *supervisor, it either crashes or receives an acknowledgement in reply"*, for a Ranch configuration with two acceptors using a global cHML formula. As opposed to the local formula in (5.1), the global specification needs to take into account all possible interleavings that arise due to acceptors. Writing the formula by hand, although possible, becomes tedious, error-prone and quickly unmanageable, as more acceptors are added. In an effort to address this issue, all the global formulae used to benchmark Ranch in Chapter 5 were programmatically generated. This approach was especially helpful when testing the monitored system using formulae of different sizes (*e.g.* three, four, five) in order to establish the limits of the setup being used.

To be able to distinguish between the two different acceptors, the `@` pre-binding operator is used in front of the variables for acceptors one and two, as can be seen in (C.1). As explained earlier in Section 3.2, this instructs the monitor to bind these variables to actual PID values *before* any events from the trace are consumed, thereby allowing pattern matching to be performed against pre-populated data values.

$$
\begin{aligned}
&\mathbf{min}(X,\\
&\quad((((\langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt1}, \mathit{Sock1}\}\rangle\\
&\qquad\qquad \langle @\mathit{Acpt1} \mathbin{?} \mathit{ConnsSup}\rangle X\\
&\quad\vee \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt1}, \mathit{Sock1}\}\rangle \langle \mathit{Acpt1}\ \mathbf{stp}\ \texttt{killed}\rangle\ \mathbf{tt})\\
&\quad\vee (\langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt1}, \mathit{Sock1}\}\rangle\\
&\qquad\qquad \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt2}, \mathit{Sock2}\}\rangle X\\
&\quad\vee \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt1}, \mathit{Sock1}\}\rangle\\
&\qquad\qquad \langle @\mathit{Acpt2} \mathbin{?} \mathit{ConnsSup}\rangle X))\\
&\quad\vee ((\langle \mathit{Acpt1} \mathbin{?} \mathit{ConnsSup}\rangle \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt1}, \mathit{Sock1}\}\rangle X\\
&\quad\vee \langle \mathit{Acpt1} \mathbin{?} \mathit{ConnsSup}\rangle \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt2}, \mathit{Sock2}\}\rangle X)\\
&\quad\vee (\langle \mathit{Acpt1} \mathbin{?} \mathit{ConnsSup}\rangle \langle \mathit{Acpt2} \mathbin{?} \mathit{ConnsSup}\rangle X\\
&\quad\vee \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt2}, \mathit{Sock2}\}\rangle\\
&\qquad\qquad \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt1}, \mathit{Sock1}\}\rangle X)))\\
&\quad\vee ((((\langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt2}, \mathit{Sock2}\}\rangle \langle \mathit{Acpt1} \mathbin{?} \mathit{ConnsSup}\rangle X\\
&\quad\vee \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt2}, \mathit{Sock2}\}\rangle \langle \mathit{Acpt2} \mathbin{?} \mathit{ConnsSup}\rangle X)\\
&\quad\vee (\langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup, start\_protocol}, \mathit{Acpt2}, \mathit{Sock2}\}\rangle \langle \mathit{Acpt2}\ \mathbf{stp}\ \texttt{killed}\rangle\ \mathbf{tt}\\
&\quad\vee \langle \mathit{Acpt2} \mathbin{?} \mathit{ConnsSup}\rangle \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt1}, \mathit{Sock1}\}\rangle X))\\
&\quad\vee (\langle \mathit{Acpt2} \mathbin{?} \mathit{ConnsSup}\rangle \langle \mathit{Acpt1} \mathbin{?} \mathit{ConnsSup}\rangle X\\
&\quad\vee \langle \mathit{Acpt2} \mathbin{?} \mathit{ConnsSup}\rangle \langle \mathit{ConnsSup} \mathbin{!} \{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt2}, \mathit{Sock2}\}\rangle X)))\\
&)
\end{aligned}
\tag{C.1}
$$

D. Using the Tool

In this appendix, we go through the steps required to monitor an existing system using the tool in Chapter 3. We start by creating a rudimentary system by borrowing code from the tool distribution itself. Following this, we specify a simple correctness property using sHML, and apply it to the system just created.

D.0.1 Creating the Target System

Since we do not have a test system available for this tutorial, we will quickly create one by copying the `plus_one.erl` server module to serve this purpose. This will enable us to set up a client-server system which suffices to demonstrate runtime monitoring using our tool. Though this example is fairly basic, it embodies the essence of how the tool should be applied; more complex properties follow the same instructions outlined in this tutorial.

The material presented in this appendix assumes that Erlang has been set up correctly. In addition, it also assumes that GNU make is installed on the host system: OSX users can acquire make by installing the XCode Command Line Tools; Windows users can install the MinGW suite of tools. Although Linux is used, the steps below can be replicated on any other operating system.

Setting up the Erlang project

To make the development of Erlang applications straightforward, we have created a generic makefile which we use in this guide. The following make targets are provided:

- `all`: Compiles the Erlang project;
- `clean`: Removes the Erlang `.beam` and temporary files;
- `init`: Creates the standard Erlang project structure;
- `docs`: Compiles the HTML documentation from Erlang source files using EDoc;
- `instrument`: Synthesises and instruments the monitors into the target system, given the HML script, target system binary directory and application entry point.

We start by creating the target application directory which for the sake of this example, we name, `example`:

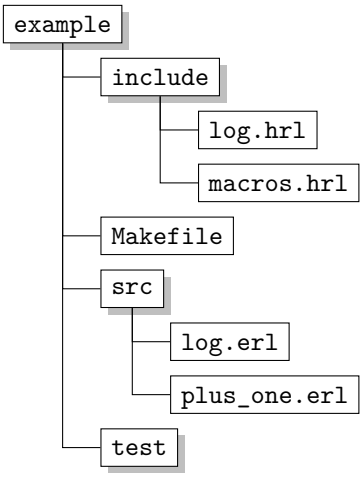
```
duncan@term:/$ mkdir example
```

Navigate into the newly created `example` directory and download the latest version of the aforementioned makefile using wget:

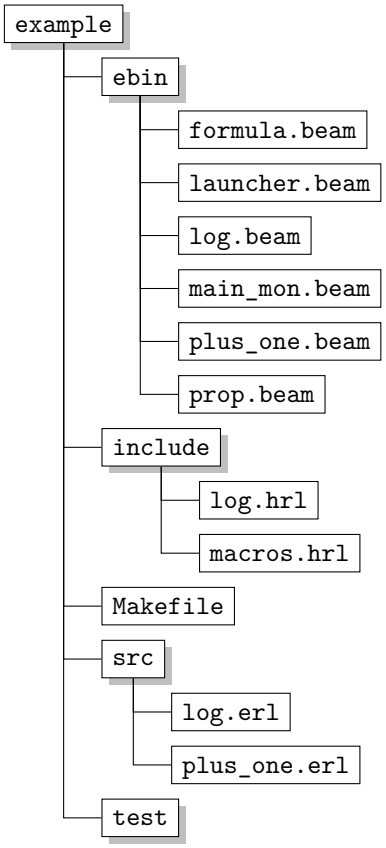
```
duncan@term:/$ cd example
duncan@term:/example$ wget https://bitbucket.org/duncanatt/detecter-lite\
       /raw/detecter-lite-1.0/Makefile
```

Once the makefile is downloaded, we create the standard Erlang directory structure using the `init` target:

```
duncan@term:/example$ make init
duncan@term:/example$ ls -l
drwxrwxr-x 2 duncan duncan 4096 May 15 16:53 include
-rw-rw-r-- 1 duncan duncan 5463 May 15 16:53 Makefile
drwxrwxr-x 2 duncan duncan 4096 May 15 16:53 src
drwxrwxr-x 2 duncan duncan 4096 May 15 16:53 test
```



(a) The `example` project directory tree before compilation.



(b) The `example` project directory tree after compilation and instrumentation.

Instead of writing an Erlang server ourselves, we reuse the `plus_one.erl` module included in the tool's distribution. If you have not yet downloaded it, refer to the instructions provided at https://bitbucket.org/duncanatt/detecter-lite. For simplicity, we assume that the tool is set up in the same directory as our `example` project directory. The `plus_one` server and its dependencies should then be copied into the `src` and `include` directories as shown below; this results in the directory tree in Figure D.1a.

```
duncan@term:/example$ cd src
duncan@term:/example/src$ cp ../../detecter-lite/test/plus_one.erl .
duncan@term:/example/src$ cp ../../detecter-lite/src/mon/log.erl .
duncan@term:/example/src$ cd ../include/
duncan@term:/example/include$ cp ../../detecter-lite/include/* .
```

Once all files are copied in place, the whole project can be built by invoking make:

```
duncan@term:/example/include$ cd ..
duncan@term:/example$ make

Compiling Erlang source file: src/log.erl to ebin/log.beam
Compiling Erlang source file: src/plus_one.erl to ebin/plus_one.beam

>------------------------------<
Build completed successfully!
>------------------------------<
```

Running and Testing the Server

With the build now complete, it is time to launch and test the `plus_one` server. As we have not developed a complete application, but only the server part of it, testing will be conducted using the Erlang shell in place of a full client implementation. The `plus_one` server and shell can be launched from the terminal as follows:

```
1 duncan@term:/example$ erl -pa ebin -eval "plus_one:start(eql)"
2
3 Erlang/OTP 18 [erts-7.2] [source] [smp:4:4] [async-threads:10] [kernel-poll:false]
4 [<0.2.0> - plus_one:22] - Started PLUS ONE server with initial value '0' and mode 'eql'.
5 Eshell V7.2  (abort with ^G)
6 1> _
```

The `plus_one` server has been purposefully started in *equal* mode (using the start up flag `eql`); this simulates incorrect behaviour whereby client requests sent to the server are not incremented but merely echoed back as is. This serves us later when we verify for the safety property in Appendix D.0.2.

For now, confirm that the server started up successfully by ensuring that the `plus_one` start up log (line 4) shows up in the terminal. Once loaded, the server can be tested by submitting requests to it using the Erlang `!` (send) operator (line 7):

```
7  1> plus_one ! {request, self(), 1}.
8
9  [ <0.33.0> - plus_one:41] - Received request with value '1'.
10 [ <0.33.0> - plus_one:46] - Sending response with value '{result,1}', Current cnt '1'.
11 {request,<0.36.0>,1}
12 2> _
```

The request sent to the `plus_one` server identified with the registered process name "plus_one" follows the format: `{request, PID, Number}`, where *PID* is the Erlang Process Identifier of the sender actor (in this case, the Erlang shell), and *Number* is the actual data payload, *i.e.,* the number which the client wishes to increment. Note that Erlang shell commands must terminate with the period symbol, otherwise these will not be processed.

As seen in the above logs, the `plus_one` server receives the number '1' as payload, and replies back with a response of '1' (lines 9 - 10). A correct implementation of the `plus_one` server *ought* to have replied with a value of '2', which corresponds to the client's request being incremented by '1'. To view the server's response from the Erlang shell and verify that an *incorrect* response has been indeed sent back, invoke the `flush()` function to empty the shell's mailbox (line 13).

```
13 2> flush().
14 Shell got {result,1}
15 ok
16 3> _
```

Now that we have confirmed that the server is working (incorrectly) as intended, the Erlang shell can be closed by typing "`q().`" at the terminal. In the next section we explore how the erroneous behaviour of the `plus_one` server can be detected using a recursive safety property specified in sHML.

D.0.2 Instrumenting the Target System

We are now in a position to generate a simple monitor that verifies for the safety property: *"the server's response cannot be equal to the client's request sent to it"*. The monitor synthesised from this property should detect violating behaviour in the `plus_one` server introduced in the preceding section.

Specifying the Safety Property

Properties using our tool are specified in plain text files that are processed by the tool to produce monitors in the form of Erlang code. These, together with their

dependencies, are compiled to Erlang `.beam` files and copied into the target system's binary directory. The compiler also generates a `launcher` module which bootstraps the system together with the synthesised monitor. Once both are executing concurrently, the system proceeds as usual, while the monitor continually observes the system's behaviour expressed in terms of the messages exchanged between it and its environment (in this case, the Erlang shell). Upon detecting a *violation*, the monitor flags it accordingly and terminates.

The safety property above can be specified by opening any plain text editor and pasting the following sHML, saving it as `prop.hml`:

```
max('X',
  [Server ? {request, Client, Request}][Client ! {result, Request}] ff
  &&
  [Server ? {request, Client, Request}][Client ! {result, Result}] 'X')
```

Alternatively, it can be done using the terminal like so:

```
duncan@term:/example$ echo -e "max('X',\n\
  [Server ? {request, Client, Request}][Client ! {result, Request}] ff\n\
  &&\n\
  [Server ? {request, Client, Request}][Client ! {result, Result}] 'X')" > prop.hml
```

Either approach should result in the creation of the HML file `prop.hml` located *in* the `example` directory.

The expression above uses a conjunction (`&&`) construct to state the possible behaviours that are to be expected by the system. The violating behaviour stated by `[`*`Server`*`?{request,`*`Client`*`,`*`Request`*`}][`*`Client`*`!{result,`*`Request`*`}] ff` specifies that a violation ought to be flagged if the server receives a request from *Client* with a numeric payload of *Request*, and sends back to *Client* that very same *Request* value. The recursive (non-violating) behaviour expressed through `[`*`Server`*`?{request,`*`Client`*`,`*`Request`*`}][`*`Client`*`!{result,`*`Result`*`}] 'X'` states that the monitor should recurse if the server receives a request from *Client* with a numeric payload of *Request* and sends back to the same *Client* a different value *Result*.

The term different in this context is taken to mean *any* value, not just the

successor or predecessor of the value in *Request*. This is perfectly acceptable since we are *only* interested in cases where the `plus_one` server sends the *same* value in *Request* back to *Client*. It is important to take note of the differences between the contents of *Request* and *Result* which are attributed to the values to which these variables bind to while the trace event is being processed. Also observe the recursion construct `max('X', ...)`, referenced by variable `X` in the right operand of the conjunction.

Synthesising the Monitor

The monitor corresponding to the script created above is synthesised using the `instrument` target from the application makefile, as shown below:

```
duncan@term:/example$ cd ../detecter-lite
duncan@term:/detecter-lite$ make instrument hml="../example/prop.hml"\
        app-bin-dir="../example/ebin"\
        MFA="{plus_one,start,[eql]}"
```

The command line arguments of `instrument` stand for the following:

- `hml`: The relative or absolute path of the plain text file containing the correctness property to be synthesised;
- `app-bin-dir`: The target application's binary directory base;
- `MFA`: The target application's entry point function in the form of a `{Module, Function, [Arguments]}` tuple, where we specified the `plus_one` module's `start` function passing `eql` as the argument, as done previously in Appendix D.0.1.

The resulting instrumented system results in the project depicted in Figure D.1b. Note that the original target system binaries remain untouched, and the previous `plus_one` server can still be run with no monitoring applied to it.

Running the Monitored System

The instrumented target system can now be run using the `launcher` module generated by the tool as follows:

```
1  duncan@term:/example$ erl -pa ebin -eval "launcher:start()"
2
3  Erlang/OTP 18 [erts-7.2] [smp:4:4] [async-threads:10] [kernel-poll:false]
4  [<0.34.0> - main_mon:38] - Started main monitor for processes/PIDs [].
5  [<0.33.0> - plus_one:22] - Started PLUS ONE server with initial value '0' and mode 'eql'.
6
7  [<0.33.0> - main_mon:24] - System to be monitored started.
8  Eshell V7.2  (abort with ^G)
9  [<0.34.0> - main_mon:62] - Resolved procs [].
10 [<0.40.0> - formula:152] - mon_max adding var 'X' to formula env.
11 [<0.40.0> - formula:91] - mon_and spawned processes '<0.41.0>' and '<0.42.0>'.
12 [<0.34.0> - main_mon:84] - Starting main monitor loop.
13 1> _
```

Different to the logs already seen in the previous execution of the `plus_one` server, we note that now, both the target system under scrutiny, as well as the monitor for it are running in parallel. Observe that the "conjunction monitor" `mon_and` (PID $\langle 0.40.0 \rangle$) has already spawned its two submonitors, as announced by the log in line 11. Like before, the system can now be tested using the same request sent from the Erlang shell (line 14):

```
14 1> plus_one ! {request, self(), 1}.
15
16 [<0.35.0> - plus_one:41] - Received request with value '1'.
17 [<0.41.0> - formula:120] - mon_nec evaluating action:
18 {recv,<0.35.0>,{request,<0.38.0>,1}}.
19 [<0.42.0> - formula:120] - mon_nec evaluating action:
20 {recv,<0.35.0>,{request,<0.38.0>,1}}.
21 [<0.35.0> - plus_one:46] - Sending response with value '{result,1}', Current cnt '1'.
22
23 {request,<0.38.0>,1}
24 [<0.41.0> - formula:120] - mon_nec evaluating action: {send,<0.38.0>,{result,1}}.
25 [<0.42.0> - formula:120] - mon_nec evaluating action: {send,<0.38.0>,{result,1}}.
26 [<0.41.0> - formula:67] - mon_ff matched 'ff' action.
27 [<0.42.0> - formula:180] - mon_var retrieving var 'X' from formula env and recursing.
28 [<0.34.0> - main_mon:113] -
29
30 Main monitor/tracer received 'ff' - *** Violation detected! ***
31
32 2> _
```

As may be gleaned from the logs above, once the trace event for `{request,`

`self(), 1}` is raised by the Erlang tracing mechanism, both left (PID `<0.41.0>`) and right (PID `<0.42.0>`) submonitors immediately acquire it from the top "conjunction monitor" (lines 17 - 19). Next, the `plus_one` server computes the result and sends it back to the Erlang shell; this causes the second trace event to be raised, and likewise, is processed by both submonitors (lines 24 - 25). At this point, note that while the right submonitor tries to unfold the next computation (line 27), the left submonitor flags a violation verdict **ff** (line 26), which is noted by the main monitor. As the existence of a single detection suffices for the main monitor to be able to yield a *global* verdict, the monitor terminates accordingly with **ff** (line 30).

Running the Correct Server

Recall that we intentionally launched the `plus_one` server using the `eql` flag in order to demonstrate how the monitor handles violations. We now re-instrument the server and initialise it with the correct behaviour flag: `lim`, as shown in line 6. Note that the only difference in the instrument command lies only in the MFA tuple that starts the server:

```
duncan@term:/detecter-lite$ make instrument hml="../example/prop.hml"\
        app-bin-dir="../example/ebin"\
        MFA="{plus_one,start,[lim]}"
```

The `plus_one` server should now behave correctly and increment the numeric payloads contained in requests sent to it by the Erlang shell.

```
1  duncan@term:/example$ erl -pa ebin -eval "launcher:start()"
2
3  Erlang/OTP 18 [erts-7.2] [source] [smp:4:4] [async-threads:10] [kernel-poll:false]
4
5  [<0.34.0> - main_mon:38] - Started main monitor for processes/PIDs [].
6  [<0.33.0> - plus_one:22] - Started PLUS ONE server with initial value '0' and mode 'lim'.
7  [<0.33.0> - main_mon:24] - System to be monitored started.
8  Eshell V7.2  (abort with ^G)
9  [<0.34.0> - main_mon:62] - Resolved procs [].
10 [<0.40.0> - formula:152] - mon_max adding var 'X' to formula environment.
11 [<0.40.0> - formula:91] - mon_and spawned processes '<0.41.0>' and '<0.42.0>'.
12 [<0.34.0> - main_mon:84] - Starting main monitor loop.
13 1> _
```

What happens if we try to send the same `{request, self(), 1}` request to the `plus_one` server, as done in line 14?

```
14 1> plus_one ! {request, self(), 1}.
15 [<0.35.0> - plus_one:41] - Received request with value '1'.
16
17 [<0.41.0> - formula:120] - mon_nec evaluating action:
18 {recv,<0.35.0>,{request,<0.38.0>,1}}.
19 [<0.42.0> - formula:120] - mon_nec evaluating action:
20 {recv,<0.35.0>,{request,<0.38.0>,1}}.
21 [<0.35.0> - plus_one:46] - Sending response with value '{result,2}', Current cnt '1'.
22 {request,<0.38.0>,1}
23 [<0.41.0> - formula:120] - mon_nec evaluating action: {send,<0.38.0>,{result,2}}.
24 [<0.42.0> - formula:120] - mon_nec evaluating action: {send,<0.38.0>,{result,2}}.
25 [<0.41.0> - formula:59] - mon_id no match.
26 [<0.42.0> - formula:180] - mon_var retrieving var 'X' from formula env and recursing.
27 [ <0.42.0> - formula:91] - mon_and spawned processes '<0.44.0>' and '<0.45.0>'.
28 2> _
```

Contrary to the previous run, no violations are flagged, despite the fact that the exact same trace events are raised by the Erlang tracing mechanism. The difference lies only in the processing of the last event (*i.e.,* `{result, 2}`) which causes the left submonitor to terminate due to a pattern mismatch (line 25), and the right submonitor to unfold recursively in preparation for the next trace events (line 26).

D.0.3 Co-safety Properties

The monitor synthesised previously from the safety property in Appendix D.0.2, flags a violation whenever the server does not increment the numeric payload in the client's request. We saw that when a correct working server (started with the `lim` flag) was monitored using this same monitor, no violations were flagged.

In this example, we consider a simple co-safety property with which the *positive* behaviour of the `plus_one` server can be ascertained. The `lim` flag used to start the server in Appendix D.0.2 imposes a limit on the number of request-response exchanges, essentially making it a finite server. After this limit is attained, the server accepts no subsequent client requests. We devise the co-safety property *"the server's process limit is finally reached"* to verify for this desired behaviour, and specify it in cHML as follows:

```
min('X',
  /Server ? {request, _, _}\/Client ! {stop, limit_reached}\tt
  ||
  /Server ? {request, _, _}\/Client ! {result, _}\ 'X')
```

Note that since we do not care about the values of bound variables (as opposed to the earlier safety property), the wildcard binder `_` is used in the above specification; although `_` binds with any value, it retains none. As done previously, we re-instrument the `plus_one` server system using the new cHML specification:

```
duncan@term:/detecter-lite$ make instrument hml="../example/prop2.hml"\
        app-bin-dir="../example/ebin"\
        MFA="{plus_one,start,[lim]}"
```

The monitor resulting from the specification file `prop2.hml` is again launched in tandem with the target system like so:

```
1  duncan@term:/example$ erl -pa ebin -eval "launcher:start()"
2
3  Erlang/OTP 18 [erts-7.2] [source] [smp:4:4] [async-threads:10] [kernel-poll:false]
4
5  [<0.34.0> - main_mon:38] - Started main monitor for processes/PIDs [].
6  [<0.33.0> - plus_one:22] - Started PLUS ONE server with initial value '0' and mode 'lim'.
7  [<0.33.0> - main_mon:24] - System to be monitored started.
8  Eshell V7.2  (abort with ^G)
9  [<0.34.0> - main_mon:62] - Resolved procs [].
10 [<0.40.0> - formula:166] - mon_min adding var 'X' to formula env.
11 [<0.40.0> - formula:106] - mon_or spawned processes '<0.41.0>' and '<0.42.0>'.
12 [<0.34.0> - main_mon:84] - Starting main monitor loop.
13 1> _
```

The behaviour of the monitor follows that of the one already seen earlier in Appendix D.0.2: the "disjunction monitor" `mon_or` (PID `<0.40.0>`) spawns its left (PID `<0.41.0>`) and right (PID `<0.42.0>`) submonitors upon starting, in preparation for incoming trace events (line 11). Once a sufficiently high number of client requests (1000 in our example, line 14) are sent, the server reaches its request-response limit of 100, and consequently, the monitor flags the property *satisfaction* accordingly using **tt** (line 31). Note that this time, instead of sending the numeric payload directly, we make use of the `plus_one:request/1` function (line 14).

```
1> lists:foreach(fun(N) -> plus_one:request(N) end, lists:seq(1, 1000)).
...
...
[<0.240.0> - formula:106] - mon_or spawned processes '<0.241.0>' and '<0.242.0>'.

[<0.241.0> - formula:136] - mon_pos evaluating action:
{recv,<0.35.0>,{request,<0.38.0>,101}}.
[<0.242.0> - formula:136] - mon_pos evaluating action:
{recv,<0.35.0>,{request,<0.38.0>,101}}.
[<0.241.0> - formula:136] - mon_pos evaluating action:
{send,<0.38.0>,{stop,limit_reached}}.
[<0.242.0> - formula:136] - mon_pos evaluating action:
{send,<0.38.0>,{stop,limit_reached}}.
[<0.241.0> - formula:76] - mon_tt matched 'tt' action.
[<0.242.0> - formula:59] - mon_id no match.
[17/5/2016 20:03:25, INFO - <0.34.0> - main_mon:110] -

Main monitor/tracer received 'tt' - *** Satisfaction detected! ***

2> _
```

D.0.4 Correct Property Synthesis

We present a final example aimed at showcasing the generation of correct monitors from mHML formulae according to the synthesis function refined in Section 3.1. Consider the sHML formula below:

```
[Server ? {request, Client, Request}][Client ! {result, Request}] ff
&&
[Server ? {request, Client, Request}][Client ! {result, Request}] tt
```

This specifies that the *"the server's response cannot be equal to the client's request sent to it"*, and also that *"the server's response can be equal to the client's request sent to it"*. By virtue of the side conditions of the refined monitor synthesis function in Section 3.1, these cases are appropriately handled and in this particular instance, the right operand of the conjunction `&&` (equating to **tt**) is removed altogether from the generated Erlang monitor, finally resulting in the following:

```
formula:mon_cnt(fun(Act) ->
  case Act of
    {recv, Server, {request, Client, Request}} ->
      formula:mon_cnt(fun(Act1) ->
        case Act1 of
```

```
          {send, Client, {result, Request}} -> formula:mon_no();
          _ -> formula:mon_end()
        end
      end);
    _ -> formula:mon_end()
  end
end)
```

An exhaustive test suite, `compiler_tests.erl` located in the EUnit `tests` directory within the distribution of the tool considers and tests all the possible side conditions handled by the refined synthesis function in Section 3.1. The interested reader is encouraged to explore these tests in order to appreciate the inner workings of the monitor generation process.

This hands-on guide provided the general workflow that can be adopted when specifying properties and instrumenting the corresponding monitors into existing system implementations. Our approach is advantageous for two main reasons:

1. Instrumentation relies only on the application's binary files, and requires no access to the system source code. This stems from the fact that the collection of trace events employs exclusively the native tracing functionality provided by Erlang;

2. The synthesis process places the compiled monitor files and their dependencies alongside the original target system binary files, leaving these untouched. This makes it possible to run both the uninstrumented and instrumented versions of the target system either by invoking it directly or through the `launcher` module respectively.

As seen throughout this appendix, employing a non-intrusive instrumentation mechanism makes the monitoring effort quite lightweight. In addition, the fact that the target system binaries are not modified makes it possible for our tool to be applied to (commercial) software with licenses and/or support agreements that explicitly forbid the modification of binary code.

E. Deliverables

The source code for the RV tool developed in Chapter 3, together with its extensions is available on the CD accompanying this manuscript. It can be accessed by navigating to the `/dev` directory.

A soft copy of this manuscript is also included, and can be accessed by navigating to the `/doc` directory.

References

[1] L. Aceto and A. Ingólfsdóttir. Testing hennessy-milner logic with recursion. In W. Thomas, editor, *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1578 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 1999.

[2] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge Univ. Press, New York, NY, USA, first edition, 2007.

[3] B. L. Agarwal. *Basic Statistics*. Anshan Publishers, first edition, 2012.

[4] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[5] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, first edition, 2007.

[6] D. P. Attard and A. Francalanza. A Monitoring Tool for a Branching-Time Logic. In Y. Falcone and C. Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 473–481. Springer, 2016.

[7] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012.

[8] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.

[9] A. K. Bauer and Y. Falcone. Decentralised LTL monitoring. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.

[10] I. Cassar and A. Francalanza. On Synchronous and Asynchronous Monitor Instrumentation for Actor-based Systems. In *FOCLASA*, volume 175 of *EPTCS*, pages 54–68, 2014.

[11] I. Cassar, A. Francalanza, and S. Said. Improving Runtime Overheads for detectEr. In B. Buhnova, L. Happe, and J. Kofron, editors, *Proceedings 12th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2015, London, United Kingdom, April 12th, 2015.*, volume 178 of *EPTCS*, pages 1–8, 2015.

[12] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, first edition, 2009.

[13] F. Chen and G. Rosu. Parametric trace slicing and monitoring. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2009.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, first edition, 1999.

[15] E. M. Clarke, W. Klieber, M. Novácek, and P. Zuliani. Model Checking and the State Explosion Problem. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011.

[16] C. Colombo and Y. Falcone. Organising LTL monitors over distributed systems with a global clock. In B. Bonakdarpour and S. A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2014.

[17] C. Colombo, A. Francalanza, and R. Gatt. Elarva: A monitoring tool for erlang. In S. Khurshid and K. Sen, editors, *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *Lecture Notes in Computer Science*, pages 370–374. Springer, 2011.

[18] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In D. D. Cofer and A. Fantechi, editors, *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 2008.

[19] C. Colombo, G. J. Pace, and G. Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In D. V. Hung and P. Krishnan, editors, *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 33–37. IEEE Computer Society, 2009.

[20] Y. Falcone, J. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[21] A. Francalanza, L. Aceto, and A. Ingólfsdóttir. On verifying hennessy-milner logic with recursion at runtime. In E. Bartocci and R. Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2015.

[22] A. Francalanza, A. Gauci, and G. J. Pace. Distributed system contract monitoring. *J. Log. Algebr. Program.*, 82(5-7):186–215, 2013.

[23] A. Francalanza and A. Seychell. Synthesising correct concurrent runtime monitors. *Formal Methods in System Design*, 46(3):226–261, 2015.

[24] F. Hebert. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, first edition, 2013.

[25] L. Hoguin. 99s. http://ninenines.eu. Accessed: 2016-08-13.

[26] O. Kupferman. Variations on safety. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2014.

[27] M. Leucker and C. Schallhart. A Brief Account of Runtime Verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[28] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comput. Sci.*, 83(1):91–130, 1991.

[29] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.

[30] R. Milner. *A Calculus of Communicating Systems (Lecture Notes in Computer Science)*. Springer, first edition, 1982.

[31] R. Milner and R. Milner. *Communication and Concurrency (Prentice Hall International Series in Computer Science)*. Prentice Hall PTR, first edition, 1995.

[32] C. O'Neil and R. Schutt. *Doing Data Science: Straight Talk from the Frontline.* O'Reilly Media, first edition, 2013.

[33] G. Reger. *Automata based monitoring and mining of execution traces.* PhD thesis, University of Manchester, UK, 2014.

[34] G. Reger, H. C. Cruz, and D. E. Rydeheard. Marq: Monitoring at runtime with QEA. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 596–610. Springer, 2015.

[35] A. Roscoe. *Theory and Practice of Concurrency.* Prentice Hall, first edition, 1997.

[36] G. Rosu and F. Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1), 2012.

[37] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, jun 1955.