3

A Runtime Monitoring Tool for Actor-Based Systems

Duncan Paul Attard[1], Ian Cassar[1], Adrian Francalanza[1], Luca Aceto[2] and Anna Ingólfsdóttir[2]

[1]Department of Computer Science, Faculty of ICT, University of Malta, Malta
[2]School of Computer Science, Reykjavík University, Iceland

Abstract

This chapter discusses detectEr, an experimental runtime monitoring tool that can be used to formally verify concurrent systems developed in Erlang. Formal correctness properties in detectEr are expressed using a monitorable subset of Hennessy-Milner Logic with recursion, and synthesised into actor-based runtime monitors. Our exposition focusses on how the specification logic is enriched and extended with pattern-matching and conditional constructs which allow monitors to be adept at processing the data obtained dynamically from the system's execution trace. The tool leverages the native tracing functionality provided by the Erlang language platform so as to produce asynchronous monitors that can be instrumented to run alongside the system with minimal effort. To demonstrate how detectEr can be used in practice, this material also provides a hands-on guide that is especially aimed at users wishing to use our tool to monitor Erlang applications.

3.1 Introduction

Concurrency [30] refers to software systems whose functionality is expressed in terms of multiple *components* or processes that are specifically designed to work simultaneously with each other. In recent years, a *concurrency-oriented* [3] approach to software development has become increasingly commonplace, and is greatly favoured over monolithic-style approaches. This is, in part, owed to the rigidity that the latter types of architectures are synonymous with, where attempts at addressing scalability concerns usually lead to notoriously complex and often, inadequate solutions. Instead, concurrency recasts the notion of system design in a way that makes it possible to avail oneself of the multi-processor and multi-core platforms that are prevalent nowadays.

Formally ensuring the correctness of concurrent systems is an arduous, albeit necessary, task, especially since the interactions between fine-grained computational components can easily harbour subtle software bugs. Despite several success stories in their application to real-life applications, static verification techniques such as Model Checking (MC) scale poorly in concurrent scenarios, particularly because the system state space that needs to be *exhaustively* verified grows exponentially with respect to the size of the system [13, 14] – this is on account of the considerable number of possible execution paths that result from process interleaving. Moreover, situations often arise whereby verification cannot be performed statically (*i.e., pre-deployment*), as certain application components might not always be available for inspection before the system starts executing (*e.g.* in systems where functional components such as add-ons are downloaded and installed dynamically at runtime). There are also cases where the internal workings of a component (*e.g.* source code or execution graph) are not accessible and need to be treated as a black box. In these cases, Runtime Verification (RV) presents an appealing compromise towards ensuring the correctness of component-based applications. It is a *lightweight* verification technique that analyses the current runtime execution path of the system under scrutiny by considering partial executions incrementally, up to the current execution point [17, 26]. Its nature inherently circumvents the scalability issues attributed to MC and provides a means for post-deployment verification. Despite these advantages, RV has limited expressiveness and cannot be used to verify arbitrary specifications such as (general) liveness properties [27].

This chapter discusses the implementation of a prototype RV tool called detectEr, that targets concurrent, component-based applications written in Erlang. The presented material aspires to introduce this tool from a pragmatic standpoint, and thus omits technical details that may be abstruse to users of the tool. Interested readers should consult previous work [4, 19, 21] for details regarding the monitor synthesis and runtime behaviour of the monitoring tool.

The content that follows is organised into three sections. Section 3.2 gives a concise overview of the ideas behind RV and monitoring; this is followed by a review of mHML, the logic used for specifying correctness properties in our tool. Although this section helps to make the presentation self-contained, it may be safely skipped by readers familiar with the subject or merely interested in using the tool. Section 3.3 revisits the logic mHML from Section 3.2, and examines how it was adapted to address the practical requirements of users wishing to define correctness properties for Erlang concurrent programs. It also very briefly touches on the compilation process that transforms mHML specification scripts into executable runtime monitors. The final section takes the form of a hands-on tutorial that guides readers through the basic steps that need to be performed in order to instrument an Erlang application with runtime monitors using the tool.

3.2 Background

An executing system results in the generation of a (possibly infinite) sequence of events known as a *trace*. These events are the upshot of internal or external system behaviours, such as message exchanges between processes or function invocations. An *execution*, *i.e.,* a *finite prefix* of an infinite trace, is consumed and processed by a software entity known as a *monitor*, tasked with the job of checking whether the execution provides enough evidence so as to determine whether a property is satisfied or violated. *Correctness specifications (properties)* serve to unambiguously describe the behaviour to which the executing system should adhere to. *Verdicts* denote monitoring outcomes and are assumed to be *definite* and non-retractable (*i.e.,* once given, cannot change). These typically consist of judgements relating to property violations and satisfactions, but may also include *inconclusive* verdicts for when the exhibited execution trace does not permit any definite judgement in relation to the property being monitored for [4, 6, 17, 19, 26]. A RV monitor for some correctness property is typically synthesised automatically from a high-level specification that finitely describes the property. Property specifications are given in terms of formal logics [4, 6, 7, 19] or other formalisms such as regular expressions [20] or automata [5, 15, 29]. Figure 3.1 depicts a correctness specification (denoted by φ) that is translated into an executable monitor, Monitor_{φ}, and instrumented with the running system. Trace events are sequentially analysed by the monitor whenever these are generated by the system through the instrumentation mechanism. Once the monitor reaches a verdict, it typically stops executing.

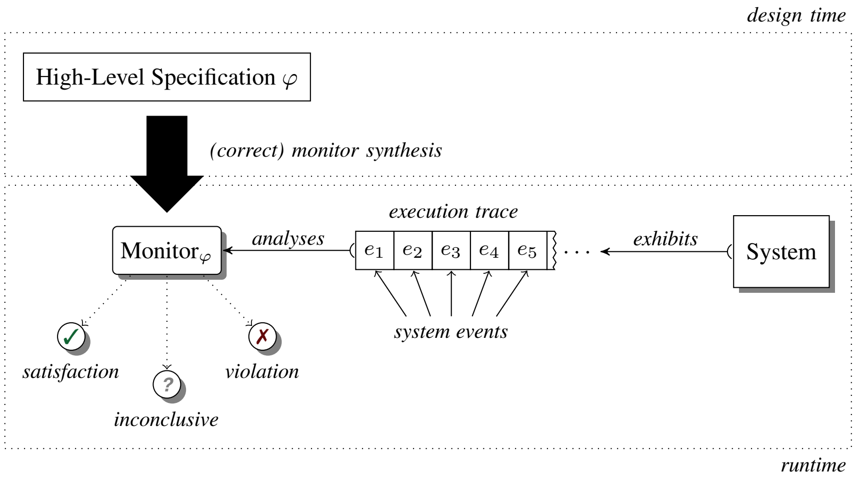


Figure 3.1 Runtime monitor synthesis and operational set-up.

3.2.1 Runtime Monitoring Criteria

Monitor synthesis, *i.e.,* the translation procedure from specifications to monitors and the associated system instrumentation, should ideally provide some *guarantees* of correctness. This covers both aspects that relate to how monitor verdicts correspond to the semantics of the property being monitored for (*e.g.* a monitor trace *rejection* should correspond to the system *violating* the property being monitored for), as well as requirements that the monitors instrumented with the executing system under scrutiny do *not* introduce fresh bugs *themselves* (consult our previous work [9, 18, 19, 21] for a detailed rendition on the subject). Equally important is the *efficiency* with which monitors execute, as this can adversely affect the monitored system or even alter its functional behaviour (*e.g.* slowdown 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 set-up that induces considerable levels of performance overhead may be deemed too costly to be feasibly used in practice.

3.2.2 A Branching-Time Logic for Specifying Correctness Properties

Specification logics can be categorised into two classes. *Linear-time* logics [6, 13, 26] treat time as having one possible future, and regard the behaviour of a system under observation in terms of execution traces or paths. On the other hand, *branching-time* logics [1, 13] make it possible to perceive time instances as potentially having more than one future, thereby giving rise to a *tree* of possible execution paths that may be (non-deterministically) taken by the executing system at runtime.

μHML [1, 25] is a branching-time logic that can be used to specify correctness properties over *Labelled Transition Systems* (LTSs) — graphs modelling the possible behaviours that can be exhibited by executing processes (see Figure 3.3 for a depiction of two LTSs). A LTS consists of a set of system states $p, q \in \textsc{Sys}$, a set of actions $\alpha \in \textsc{Act}$, and finally, a ternary transition relation between states labelled by actions, $p \xrightarrow{\alpha} q$. When $p \xrightarrow{\alpha} q$ for no process q, the notation $p \overset{\alpha}{\not\rightarrow}$ is used. Additionally, $p \Longrightarrow q$ denotes $p(\xrightarrow{\tau})^{*}q$, whereas $p \overset{\alpha}{\Longrightarrow} q$, is written in place of $p \Longrightarrow \cdot \xrightarrow{\alpha} \cdot \Longrightarrow q$. Actions labelled by τ are used to denote unobservable (silent) actions that are performed by the system internally.

The μHML syntax, given in Figure 3.2, assumes a countable set of logical variables $X, Y \in \textsc{LVar}$, thereby allowing formulae to recursively express largest and least fixpoints using $\mathbf{max}\,X.\varphi$ and $\mathbf{min}\,X.\varphi$ respectively; these constructs bind free instances of the variable X in φ. In addition to the standard constructs for truth, falsity, conjunction and disjunction, the syntax also 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 3.2 describes 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 system states to enable an inductive definition on the structure of the formula φ. The formula $\mathbf{tt}$ is satisfied by all processes, while $\mathbf{ff}$ is satisfied by none; conjunctions and disjunctions bear the standard set-theoretic meaning of intersection and union. Necessity formulae $[\alpha]\varphi$ state that *for all* system executions producing event α (possibly none), the subsequent system state must then satisfy φ (*i.e.,* $\forall p', p \overset{\alpha}{\Longrightarrow} p'$ implies $p' \in [\![\varphi, \rho]\!]$ must hold). Possibility formulae $\langle\alpha\rangle\varphi$ require the existence of *at least* one system execution with event α whereby the subsequent state then satisfies φ (*i.e.,* $\exists p'$, $p \overset{\alpha}{\Longrightarrow} p'$ and $p' \in [\![\varphi, \rho]\!]$ must hold). The recursive formulae $\mathbf{max}\,X.\varphi$ and $\mathbf{min}\,X.\varphi$ are respectively satisfied by the largest and least set of system states satisfying φ. The semantics of recursive variables X with respect to 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

Syntax

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

Semantics

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

Figure 3.2 The syntax and semantics of μHML.

independently of the environment ρ, and the shorthand $[\![\varphi]\!]$ is used to denote $[\![\varphi, \rho]\!]$, *i.e.,* the set of system states in SYS that satisfy φ. In view of this, we say that a system (state) p satisfies some closed formula φ whenever $p \in [\![\varphi]\!]$, and conversely, that it violates φ whenever $p \notin [\![\varphi]\!]$.

Example 3.2.1. The μHML formula $\langle\alpha\rangle\mathbf{tt}$ describes systems that *can* produce action α, while $[\alpha]\mathbf{ff}$ describes systems that *cannot* produce action α.

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

Formula φ_1 describes a property that prohibits a system from producing duplicate responses in answer to client requests. System p whose LTS is depicted in Figure 3.3a *violates* φ_1 through any trace in the regular language $(\mathtt{req}.\mathtt{resp})^+.\mathtt{resp}$. Formula φ_2 describes systems that *can* reach a service limit after a number (possibly zero) of request and response interactions; system q depicted in Figure 3.3b *satisfies* φ_2 through any trace in the regular language $(\mathtt{req}.\mathtt{resp})^*.\mathtt{lim}$. ■

3.2.3 Monitoring μHML

Despite its limitations (*i.e.,* monitors can only analyse single execution traces), RV can be still effectively applied in cases where correctness properties can be shown to be satisfied (or violated) by analysing a *single* finite execution. As explained previously, the formula $[\alpha]\mathbf{ff}$ states that *all* α-actions performed by a satisfying system state should satisfy property $\mathbf{ff}$ afterwards. Since no system state can satisfy $\mathbf{ff}$, the only way how to satisfy $[\alpha]\mathbf{ff}$ is for a system *not* to perform α. From a RV perspective, for a monitor to detect a violation of this requirement, observing *one negative witness* execution trace that

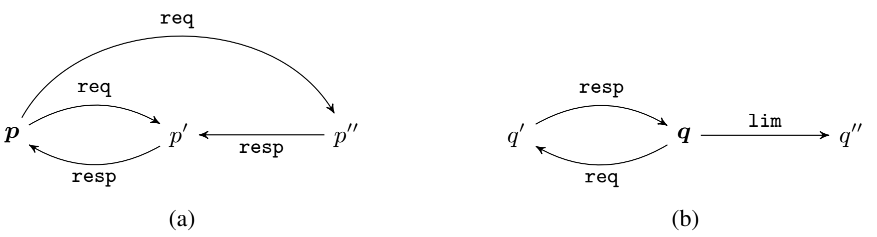


Figure 3.3 The LTSs depicting the behaviour of two servers p and q.

starts with action α suffices to show that property φ is infringed. Dually, when monitoring for the formula $\langle\alpha\rangle\mathbf{tt}$, observing *one positive witness* that starts with action α suffices to show that property φ is satisfied.

Example 3.2.2. The μHML formula $\varphi_3 = \langle\mathtt{lim}\rangle\mathbf{tt}$ requires that *"a process can perform action* `lim`*"*. System q in Figure 3.3b can exhibit the trace $\mathtt{lim}.\epsilon$ which suffices to show that system q satisfies φ_3. Yet, q may also exhibit other traces, such as those matching $(\mathtt{req}.\mathtt{resp})^*$, that all start with the event `req`. These traces do not provide enough evidence that system q satisfies φ_3. Stated otherwise, the monitor for formula φ_3 can reach an *acceptance* verdict *only* when a trace starting with event `lim` is observed. Otherwise, no verdict relating to the satisfaction or violation of the formula can be reached; in our specific case, the monitors we consider will reach an inconclusive verdict. ■

The availability of a single finite runtime trace *does* however restrict the applicability of RV in cases such as those involving correctness properties describing *infinite* or *branching* executions. In view of this, certain properties expressed using the full expressive power of a branching-time logic such as μHML cannot be monitored for at runtime. The work by Francalanza *et al.* [19] explores the limits of monitorability for μHML, identifies a syntactic logical subset called mHML, and shows it to be monitorable and maximally expressive with respect to the constraints of runtime monitoring. The syntax of mHML, given in Figure 3.4, consists of two syntactic classes, *Safety* HML (sHML), describing *invariant* properties stipulating that bad things do *not* happen, and *Co-Safety* HML (cHML), describing properties that *eventually* hold after a *finite* number of events [2, 6, 23]. Formulae φ_1 and φ_2 from Example 3.2.1 are instances of sHML and cHML specifications respectively.

Monitorable Logic Syntax

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

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

Figure 3.4 The syntax of mHML.

3.3 A Tool for Monitoring Erlang Applications

We briefly review the implementation of our RV tool detectEr that analyses the correctness of concurrent programs developed in Erlang. It builds on the work by Francalanza *et al.* [19] which specifies a synthesis procedure that generates *correct* monitor descriptions from formulae written in mHML. We adapt this synthesis procedure so as to produce *concurrent* monitors in the form of Erlang actors that are instrumented with the running system via the tracing mechanism exposed by the VM of the host language. The synthesis procedure exploits the compositional semantics of mHML formulae to generate a choreography of monitor (actor) components that independently analyse the individual subformulae constituting a global formula, while still guaranteeing the correctness of the overall monitoring process.

In the sequel we refrain from delving into the specifics of how these concurrent monitors are synthesised; readers are encouraged to consult our previous work [4, 21], where the synthesis procedure is discussed at length. Instead, we limit ourselves to a high-level description of the main concepts and technologies required by readers to be able to adequately use the monitoring tool. In particular, we discuss the mechanisms of the host language used by the tool, the adaptations to the specification logic that facilitate the handling of data, and finally, give an overview of the tool's compilation process.

3.3.1 Concurrency-Oriented Development Using Erlang

Erlang is a general-purpose, concurrent programming language suitable for the development of fault-tolerant and distributed systems [3, 12, 22]. It adopts the actor model for concurrency as the primary means for structuring its applications. An *actor* is a concurrency unit of decomposition that represents a processing entity sharing no mutable memory with other actors. It interacts with other actors by sending (asynchronous) messages, and changes its internal state based on the messages received from other actors. In Erlang, actors are implemented as *lightweight* processes that are uniquely identified via their process PID (a number triple). Each process owns a message queue, known as a *mailbox*, to which messages from other processes can be sent in a non-blocking fashion; these can be consumed selectively at a later stage by the recipient process. Messages are comprised of elements of Erlang data types, including integers, floats, atoms, functions, binaries, *etc.*. Since process PIDs are allocated dynamically to newly spawned processes, Erlang provides a mechanism for registering a PID with a fixed alias name. This allows external entities to refer to a specific process statically via the registered name alias [3, 12].

The Erlang Virtual Machine (EVM) offers a powerful and flexible *tracing* mechanism that makes it possible to observe process behaviour *without* modifying the system source code through commonly used instrumentation techniques such as Aspect Oriented Programming (AOP) [3, 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. When traced, processes generate *action* messages that are directed by the Erlang runtime to a specially designated *tracer* process. Trace messages assume the form of Erlang *tuples* that describe the nature of trace events (*e.g.* function calls, message sends and receives, garbage collection triggers, *etc.*) and are deposited (like any other message) asynchronously inside the tracer's mailbox. Tracing serves as the basis for a number of utilities, including Erlang's text-based tracing facility dbg, and trace tool builder ttb [3]. Our tool, detectEr, employs this tracing mechanism to achieve *lightweight* trace event extraction for monitoring purposes; refer to the work by Attard *et al.* [4] for further details.

3.3.2 Reasoning about Data

Adapting mHML to be used for specifying the behaviour of Erlang programs adequately requires auxiliary functionality that describes system events carrying *data*; this involves mechanisms for generalising over specific data values and for expressing data dependencies. detectEr assumes a richer set of system events that carry data. Our account focusses on two types of events, namely outputs $i \,!\, d$ and inputs $i \,?\, d$, where i ranges over process 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.*). In addition, our tool enriches the syntax of Figure 3.4 by introducing *pattern-matching* extensions for event actions (see Figure 3.5). Necessity and possibility formulae may contain *event patterns* instead of specific events: these possess the *same structure* of the aforementioned data-carrying events, but may also employ *variables* (Erlang-style alphanumeric identifiers starting with an upper-case letter) in place of values. Variables denote quantifications over data and are dynamically bound to values when they are *pattern-matched* to specific system events at runtime. Event patterns also allow us to express data dependencies across multiple events. Intuitively, whenever a variable is used in a pattern inside a necessity or possibility formula and again in the ensuing guarded subformula, the *first*

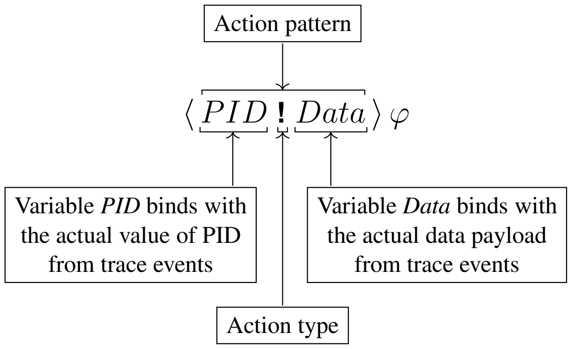

Action pattern
⟨PID ! Data⟩ φ
Variable PID binds with the actual value of PID from trace events
Variable Data binds with the actual data payload from trace events
Action type

Figure 3.5 The anatomy of action patterns for the enriched mHML syntax.

variable instance acts as a binder for subsequent variable uses. The next example illustrates this concept.

Example 3.3.1. The client-server set-up shown in Figure 3.6 consists of a successor server process (with PID `<0.33.0>`) that increments the numeric payloads it receives from requesting clients by 1. Client requests should adhere to the following protocol. A client sends a tuple of the form $\{tag, return\_addr, value\_to\_increment\}$ where the first element is a qualifier tag stating that it is a client request ($tag = \texttt{req}$). The client then awaits for an answer back from the server in the form of a message with format $\{\texttt{resp}, incremented\_value\}$. The server obtains the identity of the client from the client request data *return_addr*, which should carry the PID of the client sending the request (*e.g.* `<0.38.0>` in the case of Figure 3.6). One attempt at verifying the correctness of the executing system is by specifying a safety property stating that

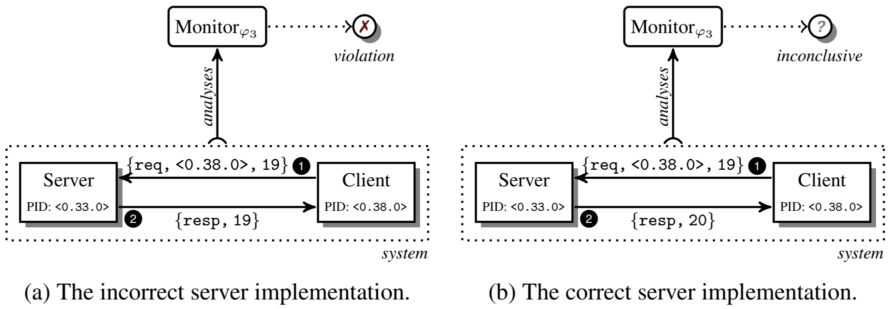


Figure 3.6 Runtime verifying the correctness of a client-server system.

"the numeric payload contained in the server's response cannot equal the one sent in the original client request."

This requirement can be expressed as follows:

$$\varphi_3 = [\mathit{Srv}\ \textbf{?}\ \{\texttt{req},\mathit{Clt},\mathit{Num}\}]\,[\mathit{Clt}\ \textbf{!}\ \{\texttt{resp},\mathit{Num}\}]\,\textbf{ff}$$

The two necessity constructs in the sHML formula φ_3 describe a request-response interaction between the client and server processes. The first necessity $[\mathit{Srv}\ \textbf{?}\ \{\texttt{req},\mathit{Clt},\mathit{Num}\}]$ specifies an *input* event data pattern that conforms to the structure of the data sent by the client when initiating its interaction with the server (*i.e.,* the action labelled by ❶ in Figure 3.6); meanwhile, the second necessity $[\mathit{Clt}\ \textbf{!}\ \{\texttt{resp},\mathit{Num}\}]\,\textbf{ff}$ specifies an *output* action data pattern that conforms to the structure of the data sent by the server in reply to the client's request (*i.e.,* action ❷ in Figure 3.6). Formula φ_3 matches events in the execution trace whenever the server *Srv* receives a request with numeric payload *Num* from client *Clt*, and replies back to the same client *Clt* with an unchanged value *Num*. Note the dependency between the patterns in the two necessities: the values matched to the variable *Clt* and *Num* in first pattern are then instantiated in the subsequent necessity pattern.

To illustrate concretely how binding actually works, we can consider how the two different executions of client-server system depicted in Figures 3.6a and 3.6b are monitored at runtime. When the event pattern $\mathit{Srv}\ \textbf{?}\ \{\texttt{req},\mathit{Clt},\mathit{Num}\}$ from the first necessity is matched to the first trace event `<0.33.0>` **?** $\{$`req, <0.38.0>, 19`$\}$ (resulting from the execution of action ❶), the free pattern variables *Srv*, *Clt* and *Num* become *bound* to the runtime values `<0.33.0>`, `<0.38.0>` and `19` respectively. The runtime binding of variables *Srv*, *Clt* and *Num* in turn, also instantiates subsequent (guarded) patterns in the second necessity — this leaves us with the (continuation) residual formula [`<0.38.0>` **!** $\{$`resp, 19`$\}$] **ff** to check for. This closed formula can now match the *second* trace event (due to action ❷), *only if* an *incorrectly* implemented server responds to the initial client request with the *same* numeric payload sent to it, as is the case in Figure 3.6a. This leads to a *violation* detection. Contrastingly, Figure 3.6b shows the case where the server's reply sent back to the client contains the value '20' that does not match the runtime binding for the subformula $[\mathit{Clt}\ \textbf{!}\ \{\texttt{resp},\mathit{Num}\}]\,\textbf{ff}$ of φ_3. After the first pattern-match, *Num* is bound to '19', and this does not match with event $\{$`resp, 20`$\}$ of action ❷ in Figure 3.6b (*Clt* is bound to `<0.38.0>` as before), thus leading to an *inconclusive* verdict. ■

3.3.2.1 Properties with specific PIDs

Since process PIDs are allocated at runtime, there is no direct way for a correctness property to refer to a *specific* process. Nevertheless, the tool still provides an indirect method how to specify this via the process PID registering mechanism offered by the host language. For instance, in the case of formula φ_3 from Example 3.3.1, one could refer to a particular process (instead of *any* arbitrary process that is dynamically bound to variable *Srv* in the pattern $[\mathit{Srv}\ \textbf{?}\ \{\texttt{req}, \mathit{Clt}, \mathit{Num}\}]$) using the notation @*srv* in place of *Srv*. This would then map to the process that is registered with the fixed (atom) name *srv* in the system and, subsequently, the respective event analysis would only match events sent specifically to the process whose PID is registered as *srv*.

3.3.2.2 Further reasoning about data

Readers might have been wary of the fact that formula φ_3 in Example 3.3.1 only guards against cases where the server merely *echoes* back the same numeric payload sent to it by clients. This only partially addresses the ideal correctness requirements, because it does not capture the full behaviour expected of the successor server in Figure 3.6. Reformulating the property from Example 3.3.1 to read as

> *"the numeric payload contained in the server's response must be equal to the successor of the one sent in the original client request."*

while more specific, requires the monitor to check whether *all* responses issued by the server in reply to client requests do in fact contain the successor of the number enclosed in said requests.

Our logic handles this expressiveness requirement by extending the enriched mHML syntax from this section with conditional constructs and predicates, thus enabling it to perform complex reasoning on data values acquired dynamically through pattern matching. Data predicates[1], together with boolean expressions, are evaluated to values $b \in \{\mathsf{false}, \mathsf{true}\}$. Conditionals, written as **if** b **then** θ **else** ϑ for sHML formulae and **if** b **then** π **else** ϖ for cHML formulae, evaluate to θ and π respectively when b evaluates to true, and to ϑ and ϖ otherwise. The **else** clause may be omitted if not required. Correctness formulae of the latter form are given

[1]Data predicates are assumed to be decidable (*i.e.,* guaranteed to terminate). Our implementation makes use of a restricted subset of Erlang side effect-free functions employed in standard guard expressions (*e.g.* `is_list/1`, `is_number/1`, `is_pid/1`, *etc.*) [12].

an *inconclusive* interpretation whenever the boolean condition inside the **if** clause evaluates to false. Conditional constructs increase the expressiveness of mHML, because they make it possible to formalise properties that are otherwise hard to express using the basic form of the logic. When compiled, conditional formulae are translated into monitors whose runtime analysis branches depending on dynamic decisions made on data obtained at runtime.

Example 3.3.2. The reformulated safety property *"the numeric payload contained in the server's response must be equal to the successor of the one sent in the original client request"* can be specified as follows using the extended sHML syntax:

$$\begin{aligned}\varphi_4 = & [\mathit{Srv}\ \mathbf{?}\ \{\mathtt{req},\mathit{Clt},\mathit{Num}\}]\,[\mathit{Clt}\ \mathbf{!}\ \{\mathtt{resp},\mathit{Succ}\}] \\ & \quad \mathbf{if}(\mathit{Succ} \neq \mathit{Num}+1)\ \mathbf{then\ ff}\end{aligned}$$

Formula φ_4 differs slightly from the one specified in Example 3.3.1. It introduces a new variable *Succ* that binds to the server's return value. This, in turn, enables the conditional construct to determine whether the successor operation is correctly implemented by the server, thus ensuring that φ_4 is violated *only* when the value bound to *Succ* is not the successor of *Num*. An inconclusive verdict is assumed by the formula whenever $(\mathit{Succ} \neq \mathit{Num}+1)$ does not hold, *i.e., Succ is* indeed the successor of *Num*, as in the case of Figure 3.6b. ■

3.3.3 Monitor Compilation

Following closely the synthesis function of [4], our tool is able to parse mHML formulae and generate Erlang code that monitors for the input formulae. The inherent concurrency features offered by Erlang, together with the modular structure of the synthesis function are used to translate formulae into choreographed collections of (sub)monitors. These are expressed as concurrent *processes* that execute independently of one another and analyse different parts of the exhibited system trace (*e.g.* one submonitor may be analysing the second event in an execution trace of length five, whereas another may forge ahead and analyse the fourth event in the trace). In order to ensure that submonitors have access to the same trace events, they are organised as supervision trees [3, 12]: 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 process, all monitoring processes are terminated, and said verdict is used to declare the final monitoring outcome. Interested readers are referred to our previous work [4, 21] for details on how these monitor choreographies are organised.

Figure 3.7 outlines the compilation steps required to transform a formula script file (*e.g.* `script.hml`) into a corresponding Erlang source code implementing the monitor functionality (*e.g.* `monitor.erl`). The tool instruments the synthesised monitors to run asynchronously with the system to be analysed using the native tracing functionality provided by the EVM. Crucially, this type of instrumentation requires *no changes to the monitor source code* (or the target system binaries). In Figure 3.7, the file packaging component of the compiler leaves the system source files unchanged; this increases confidence in the correctness of the resulting monitoring set-up. In addition to the monitor source file, Figure 3.7 shows also a second module, `launcher.erl`, that is generated automatically based on the specified system start up configuration. The launcher is tasked with the responsibility of starting the system and corresponding monitors in tandem. Said modules, together with other supporting tool-related source code files are afterwards compiled into executable modules (`.beam` files), which are then packaged and placed alongside other system binary files.

3.4 detectEr in Practice

We revisit the runtime monitoring tool depicted in Figure 3.7 from a user's perspective, and present a brief guide showcasing its main functionality. This guide, presented in the form of a tutorial, goes through the steps required

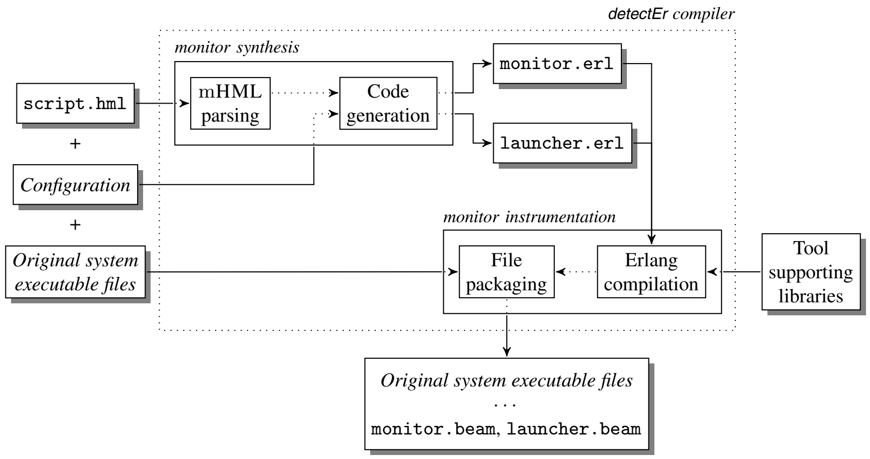


Figure 3.7 The monitor synthesis process and instrumentation pipeline.

to apply our tool to monitor an Erlang implementation of the client-server system seen earlier in Example 3.3.1. It shows how a simple (but useful) safety property can be scripted as a sHML formula, and compiled into a runtime monitor that is used to verify the incorrect and correct behaviour of the successor server illustrated in Figures 3.6a and 3.6b. cHML properties from Figure 3.4 can also be monitored for using the same sequence of steps.

The current prototype tool implementation is capable of instrumenting only one monitor inside the target system. Nevertheless, the tool's compilation and instrumentation processes were developed with extensibility in mind, and the steps that are outlined in the following tutorial will remain valid once the tool is extended to support multiple monitors. Although the example presented in this guide is fairly basic, it conveys the essence of how the tool should be applied in practice; more complex properties [8, 10] would be approached following the same instructions and procedures outlined in the coming sections.

3.4.1 Creating the Target System

The initial distribution of the tool is available from `https://bitbucket.org/duncanatt/detecter-lite`, and requires a working installation of Erlang. This guide 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 was used to create this tutorial, the steps below can be replicated on any other operating system.

3.4.1.1 Setting up the Erlang project

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

- `init`: Creates the standard Erlang project structure;
- `clean`: Removes Erlang `.beam` and other temporary files;
- `all`: Cleans and compiles the Erlang project;
- `instrument`: Synthesises and instruments monitors into the target system, given the HML script, target system binary directory, and application entry point configuration.

We begin by creating a target directory called `example`. This contains the client-server system Erlang project and all its associated source code files. At the root of the `example` directory, we also place the aforementioned makefile, since this is used to manage the build process of our simple Erlang application. The latest version of the makefile can be downloaded directly from the project site using wget:

```
duncan@term:/$ mkdir example
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, the standard Erlang directory structure is created 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
```

To avoid writing the Erlang server manually from scratch, the guide borrows a number of sample source code files that are included in the tool's distribution. For simplicity, we assume that the tool is set up in the same directory as our `example` project directory. The `plus_one` module that forms part of the tool distribution, implements a version of the successor server as described in Figure 3.6. This file, together with its dependencies should be copied into the `src` and `include` directories as shown below; these commands result in the creation of a directory structure that corresponds to the one shown in Figure 3.8a.

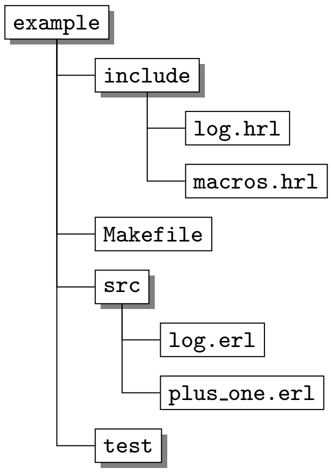
```
duncan@term:/example$ cd src
duncan@term:/example/src$ cp ../../detecter-lite/test/plus_one.erl .
duncan@term:/example/src$ cp ../../detecter-lite/src/mon/log.erl .
duncan@term:/example/src$ cd ../include/
duncan@term:/example/include$ cp ../../detecter-lite/include/* .
```

After the files have been copied successfully into their respective directories, the Erlang project can be built by invoking make:

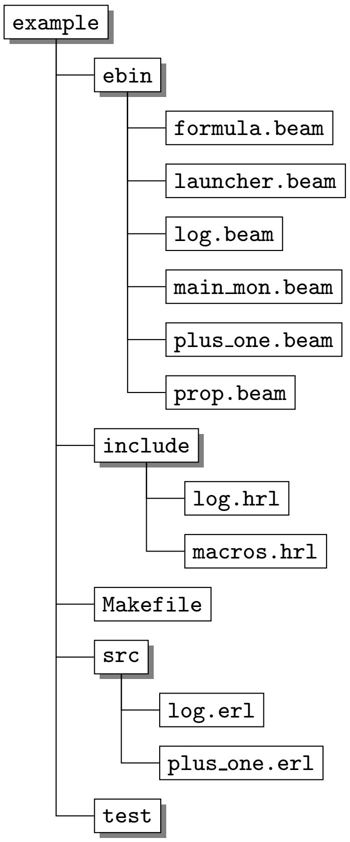
```
duncan@term:/example/include$ cd ..
duncan@term:/example$ make

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!
>------------------------------<
```



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



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

Figure 3.8 Creating the Erlang project directory structure.

3.4.1.2 Running and testing the server

With the build now completed, the `plus_one` successor server can be launched and tested. Since we have not developed a complete application, but only the server part, testing is conducted using the Erlang shell in place of a full client implementation. For illustrative purposes, the `plus_one` server may exhibit different behaviours at runtime depending on the flag it is started up with. Concretely, 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(bad)"
2
3 Erlang/OTP 18 [erts-7.2] [source] [smp:4:4] [async-threads:10] [kernel-poll:false]
4 Eshell V7.2  (abort with ^G)
5
6 [<0.33.0> - plus_one:22] - Started PLUS ONE server with initial value‘0’\,and\,mode ‘bad’.
7 1> _
```

The `plus_one` server is intentionally started using the startup flag `bad`, in order to simulate the *incorrect* server behaviour depicted in Figure 3.6a. This serves its purpose later when scripting the formula used to verify the server's behaviour. We can confirm that the server started up successfully by ensuring that the `plus_one` start up log (line 6) shows up in the terminal. Once loaded, the server can be tested by submitting requests to it using the Erlang `!` (send) operator (line 8 below). Following the protocol outlined in Example 3.3.1, the test request is sent to the process identified by the Erlang registered process name `plus_one`. This test request observes the tuple format $\{$`req`, *return_addr*, *value_to_increment*$\}$, where *return_addr* corresponds to the PID of the sender actor (in this case, the Erlang shell), and *value_to_increment* contains the actual numeric data payload, *i.e.,* the number the client wishes to increment. In Erlang, a process may obtain its own PID through the function call `self()`. Note that commands typed in the Erlang shell must terminate with a period symbol, otherwise these will not be processed.

```
 8  1> plus_one ! {req, self(), 19}.
 9
10  [<0.33.0> - plus_one:41] - Received request with value ‘19’.
11  [<0.33.0> - plus_one:46] - Sending response with value ‘{resp,19}’, Current cnt ‘1’.
12  {req,<0.38.0>,19}
13  2> _
```

As can be gleaned from the logs above, the `plus_one` server receives the number ‘19’ as payload, and echoes back that same value to the shell (lines 10–11). A correct implementation of the server should have replied with a value of ‘20’, that corresponds to the client's request being incremented by ‘1’. The server's response can be extracted from the Erlang shell by invoking the `flush()` function to empty the shell's mailbox (line 14). After confirming that the server is working (incorrectly) as intended, the Erlang shell can be closed by typing "`q().`" at the terminal.

```
14  2> flush().
15  Shell got {resp,19}
16  ok
17  3> _
```

3.4.2 Instrumenting the Test System

We are now in a position to generate a monitor that verifies the safety property below, a generalisation of the property discussed earlier in Example 3.3.1:

> *"After any sequence of request-response interactions with arbitrary clients, the numeric payload contained in the server's response following a client request must never equal the one sent in the original client request."*

The monitor synthesised for this property should detect the violating behaviour exhibited by the `plus_one` server.

3.4.2.1 Property specification

Properties using our tool are specified in plain text files that are processed to produce monitors in the form of Erlang code. These, together with other supporting source files, are compiled to executable Erlang `.beam` files and copied into the target system's binary directory, `ebin`. As explained in Section 3.3.3, the tool also creates a `launcher` module that is used to bootstrap the system together with the synthesised monitor. Once loaded, the system executes as it normally would, while concurrently, the monitor passively observes the system's behaviour expressed in terms of the messages exchanged between it and its environment. A violation will be promptly flagged when discovered by the monitor analysing the trace generated by our successor server. The aforestated safety property can be scripted by pasting the sHML formula given below into a plain text editor, and saving it as `prop.hml` *in* the `example` directory.

```
max('X',
  [Srv ? {req, Clt, Num}][Clt ! {resp, Num}] ff
  &&
  [Srv ? {req, Clt, Num}][Clt ! {resp, Other}] 'X')
```

This *recursive* sHML formula makes use of a conjunction (`&&`) construct to express the two possible behaviours expected of the system. The violating behaviour, specified using `[`*Srv* ? {`req, `*Clt*`, `*Num*}`] [`*Clt* `!` {`resp, `*Num*}`] ff`, demands that a violation be flagged when the server *Srv* receives a request containing *Num* from client *Clt*, and returns to *Clt* the same value *Num*. The recursive (non-violating) behaviour, expressed by `[`*Srv* ? {`req, `*Clt*`, `*Num*}`] [`*Clt* `!` {`resp, `*Other*}`] 'X'`, requires the monitor to recurse whenever a request received from *Clt* is answered with some value *Other*, *i.e.,* not just the successor of *Num*. This is in line with the property above, as it requires the monitor to detect violations *only* when the same value of *Num* is returned by a server in reply to a client's request. Recursion, made possible by the maximal fixpoint construct `max('X', ...)` and the recursive variable `'X'`, allows the monitor to unfold repeatedly, thereby *continuously* analysing the system trace until the violating behaviour is detected. Note that the formula in `prop.hml` is an extension of the simpler property φ_3 from Example 3.3.1. In φ_3, the absence of recursion restricts the corresponding monitor to analyse, at most, two trace events before terminating. Note also that a more comprehensive interpretation of the aforementioned correctness property would of course require the formula to check that each number in the server's response is actually the successor of the one sent in the client's request, as discussed earlier in Example 3.3.2. This can be expressed by modifying line 4 in the above script to

```
max('X', ...  &&
  [Srv ? {req, Clt, Num}][Clt ! {resp, Other}] if Other =:= Num + 1 then 'X')
```

In what follows, we stick to the weaker variant of the property to simplify our presentation.

3.4.2.2 Monitor synthesis and instrumentation

The monitor corresponding to the sHML script created above is synthesised using the `instrument` target from the application makefile:

```
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,[bad]}"
```

The `instrument` target requires the following command line arguments:

- `hml`: The relative or absolute path that leads to the formula script file;
- `app-bin-dir`: The target application's binary base directory;
- `MFA`: The target application's entry point function, encoded as a {`Mod, Fun, [Args]`} tuple, where we specify the `plus_one` module's `start` function passing `bad` as argument, like previously.

Monitor synthesis and instrumentation (refer to Figure 3.7) results in the Erlang project directory structure shown in Figure 3.8b. All the original target system binaries remain untouched, and the `plus_one` server application can be still run without monitors, as before (see Section 3.4.1.2).

3.4.2.3 Running the monitored system

The instrumented system can be started up by using the automatically generated `launcher` module as shown:

```
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 Eshell V7.2  (abort with ^G)
5
6 [<0.34.0> - main_mon:38] - Started main monitor for processes/PIDs □.
7 [<0.33.0> - plus_one:22] - Started PLUS ONE server with initial cnt value '0' and mode 'bad'.
8
9 [<0.32.0> - main_mon:24] - System to be monitored started.
10 [<0.34.0> - main_mon:62] - Resolved procs □.
11 [<0.40.0> - formula:152] - mon_max adding var 'X' to formula env.
12 [<0.40.0> - formula:91] - mon_and spawned processes '<0.41.0>' and '<0.42.0>'.
13 [<0.34.0> - main_mon:84] - Starting main monitor loop.
14 1> _
```

As indicated by the above logs, the `plus_one` server and corresponding monitor are now executing in parallel with PIDs `<0.33.0>` and `<0.34.0>` that are dynamically assigned at runtime once the respective processes are spawned (lines 6–7). The synthesised monitor corresponding to the recursion in the formula of Section 3.4.2.1 eagerly unfolds one iteration of the formula (lines 10–11) exposing a conjunction construct at top level (see Francalanza *et al.* [21] for a detailed discussion of how recursion is handled in the synthesised monitors). The "conjunction monitor" `mon_and` spawns its two submonitor actors once it starts executing (line 12); these correspond to the violation submonitor created from subformula [*Srv* ? {`req`, *Clt*, *Num*}] [*Clt* ! {`resp`, *Num*}] `ff` and the recursive submonitor created from `[`*`Srv`* `?` {`req`, *`Clt`*, *`Num`*}`]` `[`*`Clt`* `!` {`resp`, *`Other`*}`]` `'X'`. As before, the server is tested using the same request, sent from the Erlang shell (line 15):

```
15  1> plus_one ! {req, self(), 19}.
16
17  [<0.33.0> - plus_one:41] - Received request with value '19'.
18  [<0.41.0> - formula:120] - mon_nec evaluating action: {recv,<0.33.0>,{req,<0.38.0>,19}}.
19  [<0.42.0> - formula:120] - mon_nec evaluating action: {recv,<0.33.0>,{req,<0.38.0>,19}}.
20  [<0.33.0> - plus_one:46] - Sending response with value '{resp,19}', Current cnt '1'.
21
22  {req,<0.38.0>,19}
23  [<0.41.0> - formula:120] - mon_nec evaluating action: {send,<0.38.0>,{resp,19}}.
24  [<0.42.0> - formula:120] - mon_nec evaluating action: {send,<0.38.0>,{resp,19}}.
25  [<0.41.0> - formula:67] - mon_ff matched 'ff' action.
26  [<0.42.0> - formula:180] - mon_var retrieving var 'X' from formula env and recursing.
27  [<0.34.0> - main_mon:113] -
28
29  Main monitor/tracer received 'ff' - *** Violation detected! ***
30
31  2> _
```

The violation (PID `<0.41.0>`) and recursive (PID `<0.42.0>`) submonitor processes acquire trace events from their parent "conjunction monitor" process `mon_and` as soon as new trace events are reported by the EVM. For instance, the trace event generated by the message {`req, self(), 19`} sent from the shell is forwarded by `mon_and` to its child submonitors (lines 18–19). Next, the `plus_one` server computes the result and sends it back to the Erlang shell (line 20). This causes the second trace event to be generated by the system and reported by the EVM's tracing mechanism; once again this trace event is forwarded to, and processed by both submonitors (lines 23–24). At this point, the recursive submonitor tries to unfold in preparation for the next computation (line 26), while the violation submonitor flags a violation verdict **ff** (line 25), which is in turn sent to the main monitor. As a *single* detection suffices to ensure a global verdict, the main monitor terminates accordingly with **ff** (line 29); consult the work by Attard *et al.* [4] for reasons on why this is the case.

3.4.2.4 Running the correct server

So far, the `plus_one` successor server has been intentionally launched in `bad` mode in order to demonstrate how violations are handled by our monitor. We now re-instrument the system in order to emulate the correct successor server behaviour depicted in Figure 3.6b; invoking the `instrument` target differs only in the MFA tuple used to start the server, where instead of bad, the flag good is used:

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

The server should now behave correctly, and return the successor value of any numeric payload that we choose to send to it from 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   Eshell V7.2  (abort with ^G)
5
6   [<0.34.0> - main_mon:38] - Started main monitor for processes/PIDs [].
7   [<0.33.0> - plus_one:22] - Started PLUS ONE server with initial cnt value '0' and mode 'good'.
8
9   [<0.32.0> - main_mon:24] - System to be monitored started.
10  [<0.34.0> - main_mon:62] - Resolved procs [].
11  [<0.40.0> - formula:152] - mon_max adding var 'X' to formula environment.
12  [<0.40.0> - formula:91] - mon_and spawned processes '<0.41.0>' and '<0.42.0>'.
13  [<0.34.0> - main_mon:84] - Starting main monitor loop.
14  1> _
15  1> plus_one ! {req, self(), 19}.
16
17  [<0.33.0> - plus_one:41] - Received request with value '19'.
18  [<0.41.0> - formula:120] - mon_nec evaluating action: {recv,<0.33.0>,{req,<0.38.0>,19}}.
19  [<0.42.0> - formula:120] - mon_nec evaluating action: {recv,<0.33.0>,{req,<0.38.0>,19}}.
20  [<0.33.0> - plus_one:46] - Sending response with value '{resp,20}', Current cnt '1'.
21
22  {req,<0.38.0>,19}
23  [<0.41.0> - formula:120] - mon_nec evaluating action: {send,<0.38.0>,{resp,20}}.
24  [<0.42.0> - formula:120] - mon_nec evaluating action: {send,<0.38.0>,{resp,20}}.
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> _
```

When the client request {req, self(), 19} is submitted to the server from the Erlang shell (line 15), this again generates a response from the server answering back with the tuple {resp,20}. Although the sequence of trace events is similar to the ones in Section 3.4.2.3, the data in these events is *different*: the server response now carries value '20' as opposed to '19'. This causes the violation submonitor to terminate with an inconclusive verdict (line 25) and the recursive submonitor to unfold (line 26) in preparation for the next trace events. Stated otherwise, no violation is detected by the monitor up to the current point of execution.

3.5 Conclusion

We have presented an overview of detectEr from the perspective of a user wishing to employ this tool to verify Erlang systems at runtime. The tool automatically synthesises monitoring code from specifications written in the monitorable subset of the Hennessy-Milner Logic with maximal and minimal fixpoints [25, 19]. The monitoring code which is then instrumented to run alongside the system under scrutiny infers specification satisfactions or violations by analysing the runtime execution trace exhibited by the system. One salient aspect of the tool is that the instrumentation employs the tracing facility of the host language virtual machine. It therefore requires no access to system source code and relies only on the application's binary files. The execution of the monitor and the system being analysed is decoupled — this may lead to late (satisfaction or violation) detections from the monitor. In spite of this, the lightweight instrumentation approach adopted by detectEr leaves the target system binaries untouched, thus making it possible to employ our tool in cases where (commercial) software with licenses and/or support agreements explicitly forbid the modification of binary code.

3.5.1 Related and Future Work

Apart from being a manifestation of the work due to Francalanza *et al.* [19], the tool detectEr was also used as a starting point for a number of other investigations. Cassar *et al.* [11] explored choreographed reconfigurations for submonitors as means to lower the monitoring computational overhead, whereas in subsequent work [8], the authors also explored modifications to the tool to be able to synchronise more closely the executions of the system and the monitor, thereby avoiding problems associated with late detections. In other work by Cassar *et al.* [10], the investigators consider extensions to the tool that enable the runtime analysis to administer adaptation actions to the system once a violation is detected. Following this work, the authors also developed a type-based approach [9] to ensure that runtime adaptations are administered correctly by the tool. We are presently considering tool extensions that enable monitoring analysis to be distributed across sites and also alternative monitor synthesis procedures that guarantee a degree of property enforcement.

There has also been an extensive body of work [16, 28] on the runtime checking of session types. Lange *et al.* [24] demonstrate the correspondence between session types and a fragment of the modal μ-calculus, which has been previously shown by Larsen [25] to be a reformulation of the logic

μHML. Crucially, the monitors we study consider the system from a global level. By contrast, the aforementioned works project global multiparty session types to local endpoint types, which are then synthesised into local monitors that analyse traffic at individual channel endpoints.

Acknowledgments This work was partly supported by the project "TheoFoMon: Theoretical Foundations for Monitorability" (nr.163406-051) of the Icelandic Research Fund.

References

[1] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge Univ. Press, New York, NY, USA, first edition, 2007.

[2] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987.

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

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

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

[6] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.

[7] Andreas Klaus Bauer and Yliès Falcone. Decentralised LTL Monitoring. In *FM*, volume 7436 of *LNCS*, pages 85–100. Springer, 2012.

[8] Ian Cassar and Adrian Francalanza. On Synchronous and Asynchronous Monitor Instrumentation for Actor-Based Systems. In *FOCLASA*, volume 175 of *EPTCS*, pages 54–68, 2014.

[9] Ian Cassar and Adrian Francalanza. Runtime Adaptation for Actor Systems. In *RV*, volume 9333 of *LNCS*, pages 38–54. Springer, 2015.

[10] Ian Cassar and Adrian Francalanza. On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In *IFM*, volume 9681 of *LNCS*, pages 176–192. Springer, 2016.

[11] Ian Cassar, Adrian Francalanza, and Simon Said. Improving Runtime Overheads for detectEr. In *FESCA*, volume 178 of *EPTCS*, pages 1–8, 2015.

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

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, first edition, 1999.

[14] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model Checking and the State Explosion Problem. In *LASER*, volume 7682 of *LNCS*, pages 1–30. Springer, 2011.

[15] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. Elarva: A Monitoring Tool for Erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374. Springer, 2011.

[16] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design*, 46(3):197–225, 2015.

[17] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[18] Adrian Francalanza. A Theory of Monitors. In *FoSSaCS*, volume 9634 of *LNCS*, pages 145–161. Springer, 2016.

[19] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. On Verifying Hennessy-Milner Logic with Recursion at Runtime. In *RV*, volume 9333 of *LNCS*, pages 71–86. Springer, 2015.

[20] Adrian Francalanza, Andrew Gauci, and Gordon J. Pace. Distributed System Contract Monitoring. *J. Log. Algebr. Program.*, 82(5–7):186–215, 2013.

[21] Adrian Francalanza and Aldrin Seychell. Synthesising Correct Concurrent Runtime Monitors. *Formal Methods in System Design*, 46(3):226–261, 2015.

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

[23] Orna Kupferman. Variations on Safety. In *TACAS*, volume 8413 of *LNCS*, pages 1–14. Springer, 2014.

[24] Julien Lange and Nobuko Yoshida. Characteristic Formulae for Session Types. In *TACAS*, volume 9636 of *LNCS*, pages 833–850. Springer, 2016.

[25] Kim Guldstrand Larsen. Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *Theor. Comput. Sci.*, 72(2&3):265–288, 1990.

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

[27] Zohar Manna and Amir Pnueli. Completing the Temporal Picture. *Theor. Comput. Sci.*, 83(1):91–130, 1991.

[28] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed Runtime Monitoring for Multiparty Conversations. In *BEAT*, volume 162 of *EPTCS*, pages 19–26, 2014.

[29] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. Marq: Monitoring at Runtime with QEA. In *TACAS*, volume 9035 of *LNCS*, pages 596–610. Springer, 2015.

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