A Monitoring Tool for a Branching-Time Logic[⋆]

Duncan Paul Attard and Adrian Francalanza

CS, ICT, University of Malta, Malta
{duncan.attard.01,adrian.francalanza}@um.edu.mt

Abstract. We present the implementation of an experimental tool that automatically synthesises monitors from specifications written in mHML, a monitorable subset of the branching-time logic μHML. The synthesis algorithm is compositional *wrt.* the structure of the formula and follows closely a synthesis procedure that has been shown to be correct. We discuss how this compositionality facilitates a translation into concurrent Erlang monitors, where each individual (sub)monitor is an actor that autonomously analyses individual parts of the source specification formula while still guaranteeing the correctness of the overall monitoring process.

1 Introduction

Runtime Verification (RV) is a lightweight verification technique that compares the execution of a system against correctness specifications. Despite its advantages, this technique has limited expressivity and cannot be used to verify arbitrary specifications such as (general) liveness properties [6]. These limits are further explored in [3] *wrt.* the branching-time domain for a logic called μHML, describing properties about the *computational graph* of programs. The work identifies a syntactic logical subset called mHML, and shows it to be monitorable and maximally-expressive *wrt.* the constraints of runtime monitoring.

This paper discusses the implementation of a prototype tool that builds on the results of [3]. A pleasant byproduct of these results is the specification of a synthesis procedure that generates *correct* monitor descriptions from formulas written in mHML. Our tool investigates the implementability of this synthesis procedure, instantiating it to generate executable monitors for a specific general-purpose programming language. This instantiation follows closely the procedure described in [3], thereby giving us higher assurances that the generated executable monitors are indeed correct. Furthermore, we exploit the compositional structure of the procedure in [3] and refine the synthesis 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. Through our tool, we show how these concurrent components can be naturally mapped to Erlang [2] actors that monitor a running system with minimal instrumentation efforts.

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

Monitorable Logic Syntax

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

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

Monitor Syntax and Semantics

$$m \in \text{Mon} ::= v \;|\; \alpha.m \;|\; m_1 + m_2 \;|\; \mathsf{rec}\, x.m \;|\; x \qquad v \in \text{Verd} ::= \mathsf{end} \;|\; \mathsf{no} \;|\; \mathsf{yes}$$

$$\frac{}{v \xrightarrow{\alpha} v} \qquad \frac{}{\alpha.m \xrightarrow{\alpha} m} \qquad \frac{}{\mathsf{rec}\, x.m \xrightarrow{\tau} m[\mathsf{rec}\, x.m/x]} \qquad \frac{m_1 \xrightarrow{\mu} m_1'}{m_1 + m_2 \xrightarrow{\mu} m_1'}$$

Monitor synthesis

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

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

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

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

Fig. 1. The logic mHML, the monitor syntax, and compositional synthesis function.

This paper is structured as follows. Sec. 2 reviews the logic and synthesis procedure from [3]. Subsequently, Sec. 3 presents changes by which this synthesis procedure can achieve higher detection coverage. The challenges encountered while implementing a synthesis procedure that follows closely the formal description developed in Sec. 3, are discussed in Sec. 4. Finally, Sec. 5 concludes and briefly reviews related work.

2 Preliminaries

The syntax of $\psi \in$ mHML, a monitorable subset of μHML, is given in Fig. 1. It consists of two syntactic classes, sHML, describing *invariant* properties, and cHML, describing properties that hold *eventually* after a *finite* number of events. The logical formula $[\alpha]\theta$ states that *for all* system executions producing event α (possibly none), the subsequent system state must then satisfy θ, whereas the formula $\langle\alpha\rangle\pi$ states that there *exists* a system execution with event α whereby the subsequent state then satisfies π. *E.g.*, $[\alpha]\mathsf{ff}$ describes systems that *cannot* produce event α, whereas $\langle\alpha\rangle\mathsf{tt}$ describes systems that *can* produce event α. $\mathsf{max}\, X.\theta$ and $\mathsf{min}\, X.\pi$ *resp.* denote maximal and minimal fixpoints for recursive formulas; these act as binders for X in θ (*resp.* π), where we work up to α-conversion of bound variables while assuming recursive formulas to be guarded.

Monitors are expressed as a standard process calculus where $m \xrightarrow{\alpha} m'$ denotes a monitor in state m observing event α and transitioning to state m'. The action τ denotes internal transitions while μ ranges over α and τ. For instance, $m_1 + m_2$ denotes an external choice where $m_1 + m_2 \xrightarrow{\mu} m'$ if either $m_1 \xrightarrow{\mu} m'$ or $m_2 \xrightarrow{\mu} m'$ (Fig. 1 omits the symmetric rule). The only novelty is the use of verdicts v: persistent states that do not change when events are analysed, modelling the *irrevocability* of a verdict v (see [3] for details).

The synthesis function $(\!|-|\!)$ from MHML formulas to monitors is also given in Fig. 1. 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 *wrt.* the structure of the formula, *e.g.,* the monitor for $\psi_1 \wedge \psi_2$ is defined in terms of the submonitors for the subformulas ψ_1 and ψ_2. Finally, we highlight the fact that conditional cases used in the synthesis of conjunctions, disjunctions, necessity and possibility formulas, and maximal and minimal fixpoints are necessary to handle logically equivalent formulas and generate correct monitors.

Example 1. The SHML formula φ_1 describes the property stating that *"after any sequence of service requests* (req) *and responses* (ans)*, a request is never followed by two consecutive responses"*, *i.e.,* subformula $[\mathsf{ans}][\mathsf{ans}]\mathbf{ff}$. The synthesis function in Fig. 1 translates φ_1 into the monitor process m_1.

$$\varphi_1 = \mathbf{max}\, X.\big([\mathsf{req}]([\mathsf{ans}]X \wedge [\mathsf{ans}][\mathsf{ans}]\mathbf{ff})\big) \quad m_1 = \mathbf{rec}\, x.\big(\mathsf{req}.(\mathsf{ans}.x + \mathsf{ans}.\mathsf{ans}.\mathbf{no})\big)$$
$$\varphi_2 = \mathbf{min}\, X.\big(\langle\mathsf{ping}\rangle X \vee \langle\mathsf{cls}\rangle\mathbf{tt} \vee (\mathbf{min}\, Y.\mathbf{ff} \vee \langle\mathsf{cls}\rangle\mathbf{ff})\big) \quad m_2 = \mathbf{rec}\, x.\big(\mathsf{ping}.x + \mathsf{cls}.\mathbf{yes}\big)$$

The CHML formula φ_2 describes a property where after a (finite) sequence of ping events, the system closes a channel connection cls. The subformula $\mathbf{min}\, Y.\mathbf{ff} \vee \langle\mathsf{cls}\rangle\mathbf{ff}$ is semantically equivalent to $\mathbf{ff}$; accordingly the side conditions in Fig. 1 take this into consideration when synthesising monitor m_2. ■

Note that although the synthesis employs both acceptance and rejection verdicts, it only generates *uni-verdict* monitors that only produce acceptances or rejections, *never both*; [3] shows that this is essential for monitor correctness.

3 Refining the Monitor Synthesis

The first step towards implementing our tool involved refining the existing synthesis function to improve monitor detections. Specifically, there are cases where the synthesis function in Fig. 1 produces monitors with non-deterministic behaviour. For instance, monitor m_1 of Ex. 1 may exhibit the following behaviour:

$$\mathbf{rec}\, x.\mathsf{req}.\big(\mathsf{ans}.x + \mathsf{ans}.\mathsf{ans}.\mathbf{no}\big) \xrightarrow{\tau} \cdot \xrightarrow{\mathsf{req}} \mathsf{ans}.m_1 + \mathsf{ans}.\mathsf{ans}.\mathbf{no}$$

at which point, upon analysing action ans, it may non-deterministically transition to either m_1 or ans.**no**. The latter case can raise a rejection if it receives another ans event but the former case, *i.e.*, m_1, does *not* — this results in a missed detection. Although this behaviour suffices for the theoretical results required in [3], it is not ideal from a practical standpoint. The problem stems from a limitation in the choice construct semantics, $m_1 + m_2$, which forces a selection between submonitor m_1 or m_2 upon the receipt of an event.

We solve this problem by replacing external choice constructs with a parallel monitor composition construct, $m_1 \times m_2$ that allows *both* submonitors to process the event without excluding one another. The semantics of the new combinator is defined by the following rules (again we omit symmetric cases):

$$\frac{m_1 \xrightarrow{\alpha} m_1' \qquad m_2 \xrightarrow{\alpha} m_2'}{m_1 \times m_2 \xrightarrow{\alpha} m_1' \times m_2'} \qquad\qquad \frac{m_1 \xrightarrow{\alpha} m_1' \qquad m_2 \not\xrightarrow{\alpha} \qquad m_2 \not\xrightarrow{\tau}}{m_1 \times m_2 \xrightarrow{\alpha} m_1'}$$

$$\frac{m_2 \xrightarrow{\tau} m_2'}{m_1 \times m_2 \xrightarrow{\tau} m_1 \times m_2'} \qquad\qquad \frac{}{v \times m \xrightarrow{\tau} v}$$

The first rule states that both monitors proceed in lockstep if they can process the *same* action. The second rule states that if only one monitor can process the action and the other is stuck (*i.e.*, it can neither analyse action α, nor transition internally using τ), then the able monitor transitions while terminating the stuck monitor. Otherwise, the monitor is allowed to transition silently by the third rule. The last rule terminates parallel monitors once a verdict is reached.

We define a second synthesis function $[\![-]\!]$ by structural induction on the formula. Most cases are identical to those of $(\!|-|\!)$ in Fig. 1 with the exception of the two cases below, substituting the choice construct for the parallel construct:

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

The two monitor synthesis functions correspond in the sense of Thm. 1. In [3], verdicts are associated with logic satisfactions and violations, and thus Thm. 1 suffices to show that the new synthesis is still correct.

Theorem 1. *For all* $\psi \in$ 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 strucure 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. we obtain $[\![\psi_1]\!] \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} v$ (or $[\![\psi_2]\!] \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} v$) which suffices 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 2. For ψ_1 in Ex. 1, we now only have the following monitor behaviour:

$$\textbf{rec}\, x.\mathsf{req}.\big(\mathsf{ans}.x \times \mathsf{ans}.\mathsf{ans}.\textbf{no}\big) \xrightarrow{\tau} \cdot \xrightarrow{\mathsf{req}} \mathsf{ans}.m_1 \times \mathsf{ans}.\mathsf{ans}.\textbf{no} \xrightarrow{\mathsf{ans}} m_1 \times \mathsf{ans}.\textbf{no} \xrightarrow{\mathsf{ans}} \textbf{no}$$

■

4 Implementation

We implement a RV tool which analyses the correctness of concurrent programs developed in Erlang. Actions, in the form of Erlang trace events, consist of two types: outputs $i \,!\, d$ and inputs $i \,?\, d$, where i corresponds to process (*i.e.*, actor) identifiers (PID), and d denotes the data payload associated with the action in the form of Erlang data values (*e.g.*, PID, lists, tuples, atoms, *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 uppercase letter) in place of values; these are then bound to values when pattern-matched to actions at runtime. Action patterns require us to synthesise a slightly more general form of monitors with the following behaviour: if a pattern e matches a trace event action α, thereby binding a variable list to values from α (denoted as 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**.

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

mHML formulas are synthesised into *Erlang code*, following closely the synthesis function discussed in Sec. 3. In particular, we exploit the inherent concurrency features offered by Erlang together with the modular structure of the synthesis to translate submonitors into independent concurrent *actors* [2] that execute in *parallel*. An important deviation from the semantics of parallel composition specified in Sec. 3 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. 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 Sec. 3 for three main reasons: (i) monitors are univerdict, 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 [4], constitutes another minor departure from the semantics in Sec. 3, as the implementation uses a process environment that maps recursion variables to monitor terms. Erlang code for monitor **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 $x \mapsto m$.

Fig. 2 outlines the compilation steps required to transform a formula script file (`script.hml`) into a corresponding Erlang source code monitor implementation (`monitor.erl`). To be able to adhere the compositional synthesis of Sec. 3

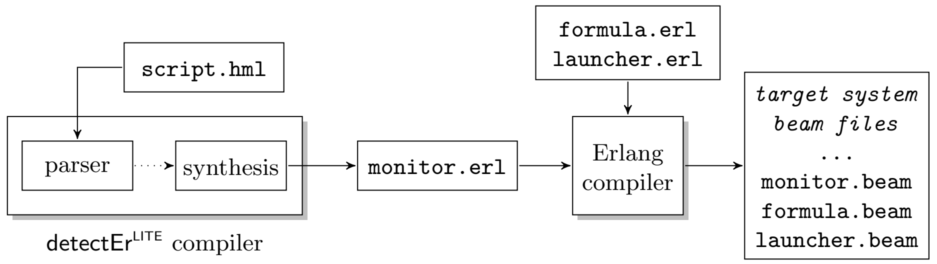


Fig. 2. The monitor synthesis process pipeline.

the tool had to overcome an obstacle attributed to pattern bindings. Specifically, in formulas such as $[e]\psi$ or $\langle e\rangle\psi$, subformula ψ may contain free (value) variables bound by the pattern e. For instance, in `[Srv ? {req, Clt}] [Clt ! ans]ff`, the Erlang monitor code for the subformula `[Clt ! ans]ff` would contain the free variable `Clt` bound by the preceding pattern `Srv ? {req, Clt}`. Since Erlang does *not* support dynamic scoping [2], 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 normally (using the static scoping of the *completed* monitor source code) via the standard Erlang compiler.

The tool itself, written in Erlang, is organised into two main modules. The synthesis in Fig. 2 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` (Fig. 2). Table 1 outlines the mapping for two of these constructs. Parallel composition is encoded by spawning two parallel actors (lines 2 - 3) followed by forking trace events to these actors for independent processing (line 4). 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 8) and executes the continuation

Monitor construct	`formula` module code

$[\![\psi_1]\!] \times [\![\psi_2]\!]$	`1 mon_and(Psi1, Psi2) ->`
	`2   fun(Env) ->  Pid1 = spawn_link(fun() -> Psi1(Env) end),`
	`3               Pid2 = spawn_link(fun() -> Psi2(Env) end),`
	`4               fork(Pid1, Pid2)`
	`5   end.`

$e.[\![\psi]\!]$	`6 mon_nec(ActMatcher) ->`
	`7   fun(Env) ->`
	`8     receive Act -> Psi = ActMatcher(Act),`
	`9                    Psi(Env) end`
	`10 end.`

Table 1. The Monitor constructs and the corresponding Erlang code (excerpt).

Synthesis subcase	`compiler` module function

$[\![\psi_1 \wedge \psi_2]\!] \overset{\textbf{def}}{=} \begin{cases} [\![\psi_1]\!] \text{ if } [\![\psi_2]\!] = \textsf{yes} \\ [\![\psi_2]\!] \text{ if } [\![\psi_1]\!] = \textsf{yes} \\ [\![\psi_1]\!] \times [\![\psi_2]\!] \text{ otherwise} \end{cases}$	`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       {join_tag(Tag1, Tag2),`
	`7         util:format("mon_and(~s,~s)", [Mon1, Mon2])}`
	`8   end;`

$[\![[e]\psi]\!] \overset{\textbf{def}}{=} \begin{cases} e.[\![\psi]\!] \text{ if } [\![\psi]\!] \neq \textsf{yes} \\ \textsf{yes} \quad \text{otherwise} \end{cases}$	` 9 synth({nec, Pat, Phi}) ->`
	`10   case synth(Phi) of`
	`11     {yes, _} -> {yes, "mon_tt()"};`
	`12     {Tag, Mon} ->`
	`13       Fun = util:format(`
	`14       "fun(Act) -> case Act of ~s -> ~s;`
	`15       _ -> mon_id() end end", [pat_to_str(Pat), Mon]),`
	`16       {join_tag(nec, Tag),`
	`17         util:format("mon_nec(~s)", [Fun])}`
	`18   end;`

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

monitor returned by `ActMatcher` in case of a successful match (line 9). 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 2 outlines the tight correspondence between this compilation and the synthesis function of Sec. 3. To encode 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 line 7 performs the necessary string processing and calls the function `mon_and` presented in Table 1. For formula $[\![[e]\psi]\!]$, the translation inserts directly the function corresponding to `ActMatcher` (lines 13 - 15) alluded to in Table 1 — this is passed as an argument to `mon_nec` 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 `Phi` (line 10). See Apps. A.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 Virtual Machine (EVM). Erlang directives instruct the EVM to report events of 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.

The initial distribution of the tool is available from `https://bitbucket.org/duncanatt/detecter-lite`, and requires a working installation of Erlang.

5 Conclusion

We discuss the implementation of a tool that synthesises and instruments asynchronous monitors from specifications written in mHML, a monitorable subset of the logic μHML. The implementation follows very closely a correct monitor synthesis specification described in [3]. This tight correspondence gives us high assurances that the executable monitors generated by our tool are also correct.

Discussion and Related work: Monitors form part of the trusted computing base of a system and generally, their correctness is *sine qua non* [5]. Despite its importance, tools prioritising this aspect often prove correctness for a high level abstraction of the monitor but do not put much effort towards showing that the *resp.* monitor implementation corresponds to this abstraction. To our knowledge, the closest work that attempts to bridge this correctness gap is [4], wherein the authors formalise an operational semantics of a subset of the target language and then show monitor correctness within this formalised language subset. Their tool shares a number of common aspects with our work (*e.g.,* they also synthesise subsets of μHML, use Erlang as a target language and also asynchronous instrumentation), but differs in a few main aspects: (*i*) we consider a substantially larger syntactic monitorable subset of μHML (*e.g.,* we can specify *positive* properties such as "the system can perform action α"); (*ii*) our notion of monitor correctness is formalised in terms of a language agnostic abstraction — a process calculus; (*iii*) we consider action patterns, which complicate the modularity of the synthesis process. In other related work, [1] explores *synchronous* monitor instrumentations within a similar setting to ours; this requires changes to the monitor and system code, which can potentially affect correctness.

References

1. I. Cassar and A. Francalanza. On Synchronous and Asynchronous Monitor Instrumentation for Actor-based Systems. In *FOCLASA*, pages 54–68, 2014.
2. F. Cesarini and S. Thompson. *Erlang Programming.* O'Reilly, 2009.
3. A. Francalanza, L. Aceto, and A. Ingólfsdóttir. On Verifying Hennessy-Milner Logic with Recursion at Runtime. In *Runtime Verification*, pages 71–86, 2015.
4. A. Francalanza and A. Seychell. Synthesising Correct Concurrent Runtime Monitors. *Formal Methods in System Design*, 46(3):226–261, 2015.
5. J. Laurent, A. Goodloe, and L. Pike. Assuring the Guardians. In *Runtime Verification*, pages 87–101, 2015.
6. Z. Manna and A. Pnueli. Completing the Temporal Picture. *Theoretical Computer Science*, 83(1):97–130, 1991.

A Appendix

In this appendix, we go through the steps required to monitor an existing system using the tool extended in this paper. 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.

A.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.

Remark 1. The current prototype implementation of the tool supports the instrumentation of a single monitor inside the target system. As the tool's compilation and synthesis processes were developed with extensibility in mind, the steps presented below remain valid once it is enhanced to support multiple monitors.

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
```

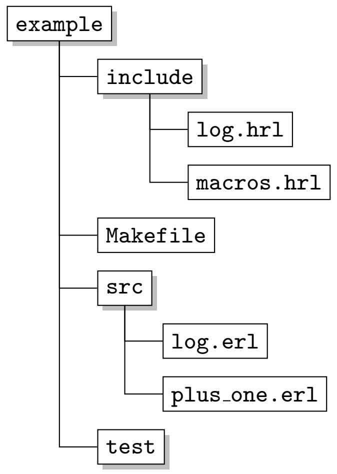
Instead of writing an Erlang server ourselves, we reuse the `plus_one.erl` module included in the tool's distribution. If you have not yet downloaded it, refer to the instructions provided at `https://bitbucket.org/duncanatt/detecter-lite`. For simplicity, we assume that the tool is set up in the same 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 Fig. 3a.

```
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/* .
```

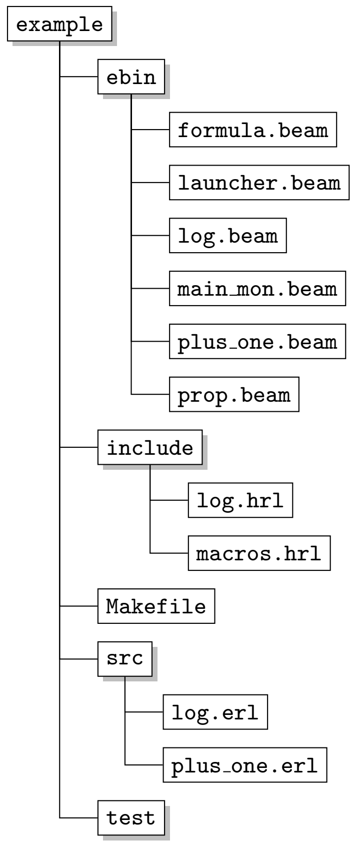
Once all files are copied in place, the whole project can be built by invoking `make`:

```
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:



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



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

```
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 Apps. A.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 from 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.

A.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. 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 $[\mathit{Server}\,?\,\{\texttt{request}, \mathit{Client}, \mathit{Request}\}]\,[\mathit{Client}\,!\,\{\texttt{result}, \mathit{Request}\}]\,\texttt{ff}$ 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 $[\mathit{Server}\,?\,\{\texttt{request}, \mathit{Client}, \mathit{Request}\}]\,[\mathit{Client}\,!\,\{\texttt{result}, \mathit{Result}\}]$ `'X'` states that the monitor ought to 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 back 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 Apps. A.1.

The resulting instrumented system results in the project depicted in Fig. 3b. 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 observation, 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 sub-monitor tries to unfold the next computation (line 27), the left sub-monitor 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 sub-monitor to terminate due to a pattern mismatch (line 25), and the right sub-monitor to unfold recursively in preparation for the next trace events (line 26).

A.3 Co-safety Properties

The monitor synthesised previously from the safety property in Apps. A.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 Apps. A.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 Apps. A.2: the "disjunction monitor" `mon_or` (PID $\langle 0.40.0\rangle$) spawns its left (PID $\langle 0.41.0\rangle$) and right (PID $\langle 0.42.0\rangle$) 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).

```
14  1> lists:foreach(fun(N) -> plus_one:request(N) end, lists:seq(1, 1000)).
15  ...
16  ...
17  [<0.240.0> - formula:106] - mon_or spawned processes '<0.241.0>' and '<0.242.0>'.
18
19  [<0.241.0> - formula:136] - mon_pos evaluating action:
20                              {recv,<0.35.0>,{request,<0.38.0>,101}}.
21  [<0.242.0> - formula:136] - mon_pos evaluating action:
22                              {recv,<0.35.0>,{request,<0.38.0>,101}}.
23  [<0.241.0> - formula:136] - mon_pos evaluating action:
24                              {send,<0.38.0>,{stop,limit_reached}}.
25  [<0.242.0> - formula:136] - mon_pos evaluating action:
26                              {send,<0.38.0>,{stop,limit_reached}}.
27  [<0.241.0> - formula:76] - mon_tt matched 'tt' action.
28  [<0.242.0> - formula:59] - mon_id no match.
29  [17/5/2016 20:03:25, INFO - <0.34.0> - main_mon:110] -
30
31  Main monitor/tracer received 'tt' - *** Satisfaction detected! ***
32
33  2> _
```

A.4 Correct Property Synthesis

We present a final example aimed at showcasing the generation of correct monitors from mHML formulas according to the synthesis function refined in Sec. 3. 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 Sec. 3, 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_nec(fun(Act) ->
  case Act of
    {recv, Server, {request, Client, Request}} ->
      formula:mon_nec(fun(Act1) ->
        case Act1 of
          {send, Client, {result, Request}} -> formula:mon_ff();
          _ -> formula:mon_id()
        end
      end);
    _ -> formula:mon_id()
  end
end)
```

An exhaustive test suite, `compiler_tests.erl` located in the EUnit `tests` directory within the distribution of our tool considers and tests all the possible side conditions handled by the refined synthesis function in Sec. 3. The interested reader is encouraged to explore these tests in order to appreciate the inner workings of the monitor generation process.

A.5 Trying Out Other Properties

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.

We invite the reader to experiment with the other safety and co-safety properties located in the `mhml_tests.erl` module included with the distribution of the tool. Comments and suggestions are welcome, and can be directed to us through the project's Jira page at `https://bitbucket.org/duncanatt/detecter-lite/issues`.