Trace Partitioning and Local Monitoring for Asynchronous Components[*]

Duncan Paul Attard and Adrian Francalanza

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

Abstract. We propose an instrumentation technique for monitoring asynchronous component systems that departs from the traditional runtime verification set-up assuming a single execution trace. The technique generates partitioned traces that better reflect the interleaved execution of the asynchronous components under scrutiny, and lends itself well to local monitoring. We provide argumentation for the qualitative benefits of our approach, demonstrate its implementability for actor-based systems, and justify claims related to the applicability and efficiency gains via an empirical evaluation over a third party component-based system.

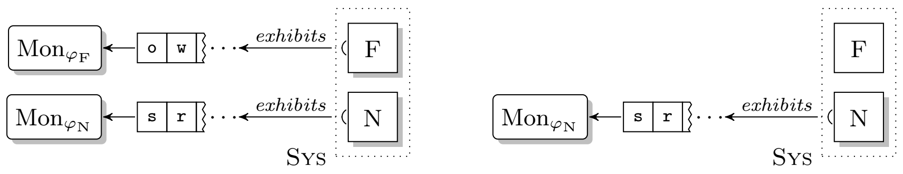
1 Introduction

Few systems are constructed in monolithic fashion these days. Rather, a considerable number are architected as *asynchronous components* [22,10,2] that execute independently to one another without recourse to a global clock or shared state; in place of the latter, components interact with one another via well-defined interfaces and non-blocking messaging [20]. Such software organisations encourage code reuse, ease incremental updates, naturally quarantine faults and engender graceful degradation, thus improving time-to-market.

At the same time, component-based systems pose new challenges for ensuring correctness. Their sheer size, dynamic structure, use of third party components, and inherent concurrent execution, complicate the use of traditional pre-deployment verification techniques, at times rendering them ineffective. Runtime Verification (RV) [24,14] is a lightweight post-deployment verification technique that circumvents a number of these obstacles, making it an appealing compromise when ascertaining software correctness. It uses monitors that *incrementally* analyse the behaviour of the running system (exhibited as a sequence of *trace events*) up to the current execution point, in order to determine whether a correctness specification is satisfied or violated.

Recent work [15,4,8,28,16] studies the application of online RV to *general* specification properties describing the branching structure of the system under scrutiny. This is of particular relevance to concurrent (component) systems with multiple executions. Since RV is not as expressive as exhaustive pre-deployment

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



Mon_{φ_F}
o
w
exhibits
F
Mon_{φ_N}
s
r
exhibits
N
Sys
F
Mon_{φ_N}
s
r
exhibits
N
Sys

(a) Local monitors analysing the traces for components F and N separately.

(b) Local monitor analysing the trace for component N; no monitor needed for F.

Fig. 1. Local monitors attached to independently executing components.

techniques such as Model Checking, this body of work is concerned with identifying monitorable (*i.e.,* can be verified at runtime) subsets of properties. But even for monitorable properties, the analytical power of the ensuing monitor analysis is still at the mercy of the trace the system decides to exhibit at runtime.

Example 1. Consider the logging system, Sys, consisting of two *independently-executing* components, F and N. Component F handles *file logging* operations, and is permitted actions open (`o`), close (`c`) and write (`w`), whereas component N manages *network logging* activities through send (`s`) and receive (`r`) messages. Additionally, both components may signal file or network-related problems by issuing error (`e`) actions. A possible correctness property is one that *"forbids* Sys *from sending messages at start-up".* When Sys exhibits the trace `s.o.w.r`, the monitor can detect a violation of this property. However, for a different execution interleaving (*e.g.* one producing the witness trace `o.s.w.r` where `s` is not the first event) the typical RV analysis would *not* be able to detect the fact that the system is *capable of* performing the initial action `s`. ■

In component architectures, there are instances where *additional* traces can be inferred from an observed trace; whenever these inferred traces are relevant to the correctness specification of interest, they help mitigate the aforementioned *lack of precision* of the technique. In the case of E.g. 1, we know that *(i)* the system consists of two components F and N whose execution can be arbitrarily interleaved, *(ii)* the trace events `o`,`c` and `w` can be uniquely attributed to F, whereas `s` and `r` can be accredited to N. A monitor can use this extra system information to permute the order of events in a witness trace. For example, from trace `o.s.w.r`, event `o` (generated by F) can be permuted with `s` (generated by N) to obtain the trace `s.o.w.r` which is also a *valid* trace that can be generated by the system and, crucially, provides evidence that Sys violates the property stated in E.g. 1. Inferring traces may also increase RV's *expressive power.* For instance, the property

At start-up, Sys *can neither send messages* nor *can it open files* (1)

is shown *not* to be monitorable in the setting of [15,16]: to detect a violation, it requires at least *two* witness execution traces, one showing that action `s` can be performed at start-up, another showing that `o` can be performed at start-up. Monitorability in traditional RV settings typically assumes *one* execution trace. For the case of F and N, from trace `o.s.w.r`, a monitor can infer the second trace `s.o.w.r` and *together* these can be used to determine the violation verdict.

Despite the benefits discussed above, trace inference for component-based systems induces additional runtime overheads and may require *unbounded* buffer space to record past trace events. Both aspects afflict the feasibility of the RV analysis, which often requires overheads to be *kept to a minimum*.

Contributions and synopsis. This paper argues that the aforementioned problems stem from the fact that traditional RV set-ups treat component-based systems as one monolithic block, artificially recording executions as one *universal trace*. Instead, we study instrumentation techniques that generate *multiple* traces, whereby events are partitioned to better reflect the asynchronous component structure of the system under scrutiny. As depicted in Fig. 1a, our proposed instrumentation technique would report the runtime execution of F and N as the *partitioned traces* `o.w` and `s.r`. These may be seen as a more compact representation of a number of universal traces in a traditional RV set-up, denoting both traces `o.s.w.r` and `s.o.w.r` mentioned earlier, but also other potentially relevant traces such as `o.s.r.w` and `s.r.o.w`.

Partitioned traces are better suited to monitor *decentralisation*. For instance, Prop. 1 could be evaluated using two submonitors as in Fig. 1a, one analysing whether F starts by issuing event `o`, and another that checks if N produces `s`; these would then alert one another accordingly when independent detections are made. In the case of the property from E.g. 1, partitioning also allows us to *localise monitoring* to the subsystem of interest, as shown in Fig. 1b: this lowers runtime overheads since the local monitor needs to process less events to reach its verdict. Apart from making a case for partitioned traces and localised monitors, our contributions are:

- A unifying account of the types of runtime monitoring approaches that can be applied to generic instantiations of component-based systems, discussing the advantages enjoyed by local monitoring in this setting in Sec. 3;
- An investigation of the implementability of local monitoring in Sec. 4;
- A case study for a third-party component-based system validating our proposed technique from a performance standpoint in Sec. 5.

We conclude by discussing future and related work in Sec. 6.

2 Monitors and Specification

Runtime monitors are typically synthesised from property specifications expressed in a high-level formalism or logic. These specifications *finitely and unambiguously* describe the behaviour of interest for the system under scrutiny. Monitor *verdicts* are definite *non-retractable* judgements reached after analysing a finite prefix of the system execution trace, and correspond to property satisfactions or violations from which the monitor is synthesised. The monitor may

Syntax

$$\varphi, \phi \in \text{sHML} ::= \mathsf{tt} \quad | \quad \mathsf{ff} \quad | \quad \varphi \wedge \phi \quad | \quad [e]\varphi \quad | \quad \mathsf{max}\, X.\varphi \quad | \quad X$$

Semantics

$$[\![\mathsf{tt}]\!] \stackrel{\text{def}}{=} \textsc{Sys} \qquad [\![\mathsf{ff}]\!] \stackrel{\text{def}}{=} \emptyset \qquad [\![\varphi \wedge \phi]\!] \stackrel{\text{def}}{=} [\![\varphi]\!] \cap [\![\phi]\!]$$

$$[\![[e]\varphi]\!] \stackrel{\text{def}}{=} \Big\{ p \mid \forall p', \alpha.\ \big(p \stackrel{\alpha}{\Longrightarrow} p' \text{ and } \mathbf{match}(e, \alpha) = \sigma\big) \text{ implies } p' \in [\![\varphi\sigma]\!] \Big\}$$

$$[\![\mathsf{max}\, X.\varphi]\!] \stackrel{\text{def}}{=} \bigcup \{ S \mid S \subseteq [\![\varphi[X \mapsto S]]\!] \}$$

Fig. 2. The syntax and semantics of sHML.

also reach an *inconclusive* verdict whenever the trace exhibited by the system does not yield the necessary information for it to reach a definitive judgement.

Following [15,16], this paper uses the *safety* fragment of the branching-time logic μHML [23,1], called sHML (Safety HML), which has been shown to be monitorable and maximally expressive w.r.t. the constraints of runtime monitoring. The sHML syntax, given in Fig. 2, assumes a countable set of logical variables $X, Y \in \textsc{LVar}$, allowing formulae to recursively express largest fixpoints using the construct $\mathsf{max}\, X.\varphi$; this binds free instances of the variable X in φ. In addition to the standard constructs of truth, tt, falsehood, ff, and conjunction, $\varphi \wedge \phi$, the logic includes a necessity modality construct, $[e]\varphi$, where the term e can contain event *patterns* consisting of free variables that are matched and *bound dynamically* at runtime to specific system events α that carry data.

As in [15,16], the semantics of sHML, defined for *closed* formulae (*i.e.,* without free variables) is interpreted over Labelled Transition Systems (LTSs) — graphs modelling the branching behaviour of systems (see Fig. 3 for examples). Formally, a LTS is comprised of the triple $\langle \textsc{Sys}, \textsc{Act}, \longrightarrow \rangle$, consisting of a set of system states $p, q \in \textsc{Sys}$, a set of actions $\mu \in \textsc{Act}$ containing a distinguished *silent* action τ (used to represent unobservable actions) and *visible* actions α ranging over $\textsc{Act} \setminus \{\tau\}$, and finally, a ternary transition relation between states labelled by actions, $p \xrightarrow{\mu} q$. We use $p \Longrightarrow q$ to denote $p(\xrightarrow{\tau})^* q$, whereas $p \stackrel{\alpha}{\Longrightarrow} q$ is written in lieu of $p \Longrightarrow \cdot \xrightarrow{\alpha} \cdot \Longrightarrow q$. Formula tt is satisfied by all system states, whereas ff is satisfied by none. Conjunctions bear the standard set-theoretic meaning of intersection. Necessity formulae $[e]\varphi$ state that *for all* system executions producing event α (possibly none), pattern e must match α, yielding a set of bindings σ, and the subsequent system state must then satisfy $\varphi\sigma$ (*i.e.,* φ substituted with the bindings in σ). The recursive formula $\mathsf{max}\, X.\varphi$ is defined as the union of all the post-fixpoint solutions $S \subseteq \textsc{Sys}$ of φ; see [23,1] for details. A system state p satisfies formula φ whenever $p \in [\![\varphi]\!]$; conversely, it violates φ whenever $p \notin [\![\varphi]\!]$. In [15,16], the authors show that any sHML formula is monitorable for violations *exclusively* (*i.e.,* the monitor for φ can reach a rejection verdict whenever $p \notin [\![\varphi]\!]$). We note in passing that the full logic μHML contains other logical operators, such as disjunctions, $\varphi \vee \phi$, with the expected

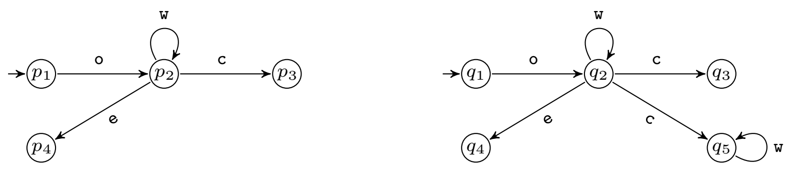


Fig. 3. Two LTSs depicting different behaviours of the file logging component in SYS.

interpretation; [15,16] show that disjunctions such as $\varphi \vee \phi$ are not monitorable for violations, even when the subformulae φ and ϕ are. Consult [15,16] for more details.

Example 2. Fig. 3 depicts the LTSs of two possible implementations for component F from E.g. 1. The first one, rooted at state p_1, satisfies property φ_1 below: informally φ_1 describes implementations where, after opening (o) a logfile and performing an arbitrary number of writes (w), do not write to it once closed (c).

$$\varphi_1 = [\mathtt{o}]\big(\mathbf{max}\, X.([\mathtt{w}]X \wedge [\mathtt{c}][\mathtt{w}]\mathbf{ff})\big)$$

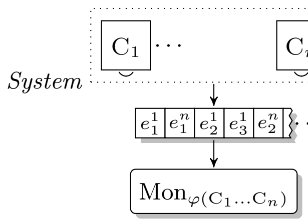
Interested readers can indeed check that $p_1 \in [\![\varphi_1]\!]$. The second implementation, rooted at q_1, describes non-deterministic behaviour once the logfile is closed, whereby it can either reach the inert state q_3 or state q_5, which allows further write operations. Although $q_1 \notin [\![\varphi_1]\!]$, the synthesised monitor for φ_1 of [15,16] depends on the runtime trace exhibited to determine the violation, where: *(i)* it reaches the violation verdict whenever q_1 produces a trace of the form $\mathtt{o.w^*.c.w^+}$, *(ii)* reaches an inconclusive verdict (and stops) if it sees the trace e, and *(iii)* continues monitoring for future events for traces of the form $\mathtt{o.w^*.c}$. ■

Example 3. The property stated in E.g. 1 is expressed as $[\mathtt{s}]\mathbf{ff}$ in sHML, whereas Prop. 1 from Sec. 1 is expressed as $[\mathtt{o}]\mathbf{ff} \vee [\mathtt{s}]\mathbf{ff}$ in the full logic μHML; in [15,16] this is shown to be non-monitorable. ■

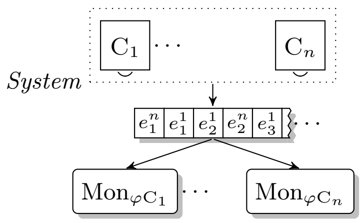
3 The Approach

Standard RV set-ups consist of the system under scrutiny, the instrumentation extracting and reporting the execution trace, and the monitor analysing this trace. As shown in Fig. 4a, execution events are typically collected as a *single universal* trace that describes the running system in its entirety. We propose an alternative instrumentation approach for asynchronous components whereby the individual execution of the constituent components is reported separately as *partitioned* traces, as depicted in Figs. 4c and 4d.

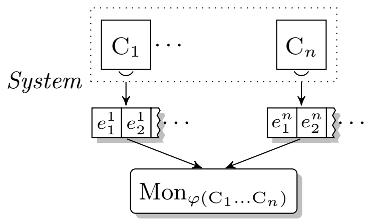
A partitioned trace gives an exclusive *localised view* for a subset of the system under scrutiny, delineated by the underlying system structure. Partitioned traces may be analysed individually, whenever this local view suffices, or in conjunction with other partitioned traces to form a *combined* trace. Note that in


System
C₁ ···
Cₙ
e^1_1 e^n_1 e^1_2 e^1_3 e^n_2 ···
$\mathrm{Mon}_{\varphi(\mathrm{C}_1\ldots\mathrm{C}_n)}$

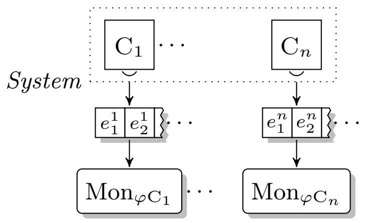
(a) Global monitor analysing the universal trace for components $\mathrm{C}_1, \ldots, \mathrm{C}_n$.


System
C₁ ···
Cₙ
e^n_1 e^1_1 e^1_2 e^n_2 e^1_3 ···
$\mathrm{Mon}_{\varphi\mathrm{C}_1}$ ···
$\mathrm{Mon}_{\varphi\mathrm{C}_n}$

(b) Local monitors analysing the universal trace for components $\mathrm{C}_1, \ldots, \mathrm{C}_n$.



(c) Global monitor analysing the partitioned trace for components $\mathrm{C}_1, \ldots, \mathrm{C}_n$.


System
C₁ ···
Cₙ
e^1_1 e^1_2 ···
e^n_1 e^n_2 ···
$\mathrm{Mon}_{\varphi\mathrm{C}_1}$ ···
$\mathrm{Mon}_{\varphi\mathrm{C}_n}$

(d) Local monitors analysing the partitioned trace for components $\mathrm{C}_1, \ldots, \mathrm{C}_n$.

Fig. 4. Four architectural set-ups characterising component-based runtime monitoring.

asynchronous settings, the merging of all the partitioned traces does *not* yield a unique combined trace, but rather a *set* of possible combined traces. Trace partitioning is advantageous whenever the correctness of a system comprised of asynchronous components is considered from a global view. First off, it does not taint the monitors's view of the system behaviour by reporting artificial orderings, which in turn impinge on the monitor's analytical precision (*e.g.* monitoring for $[\mathsf{s}]\mathbf{ff}$ from E.g. 1). Instead, since the aggregation of partitioned traces (efficiently) encode a set of combined (universal) traces, our proposed instrumentation provides *additional* information about the system's behaviour. This leads to more expressive RV set-ups in terms of the properties that can be monitored for at runtime (*e.g.* $[\mathsf{o}]\mathbf{ff} \vee [\mathsf{s}]\mathbf{ff}$ from Prop. 1 can be monitored in set-ups like Figs. 4c and 4d but *not* in classic set-ups like Fig. 4a); see Sec. 1 for discussion.

Second, trace partitioning yields other benefits in the form of local monitoring. Particularly, it permits the specification of *local properties* that describe the behaviour of a subset of components. *Local monitors* synthesised from these properties need *only* analyse events from single trace partitions in order to reach a verdict relating to the local property being considered. Note that local monitors may also execute w.r.t. a universal trace.

Example 4. Property $[\mathsf{s}]\mathbf{ff}$ from E.g. 1 can be seen as a local property describing the behaviour of component N. In fact, a local monitor can be synthesised from $[\mathsf{s}]\mathbf{ff}$ accordingly; this, in turn, is able to reach a rejection verdict just by analysing the partitioned trace for N. Property φ_1 from E.g. 2 can also be seen as a local property that describes the behaviour of component F from E.g. 1. ■

Executing a local monitor on a universal trace, as shown in Fig. 4b may still lead to detections in cases where the component interleaving prioritises the events of interest (*e.g.* the universal trace $\mathtt{s.r.o.w}$ permits a violation detection for the local property $[\mathtt{s}]\mathbf{ff}$), but extraneous events may affect precision, as discussed in Sec. 1. In practice, one may be able to regain degrees of precision via trace filtering at the monitor level, where conceptually, this equates to converting a local property into a *global* one that accounts for the events of other components.

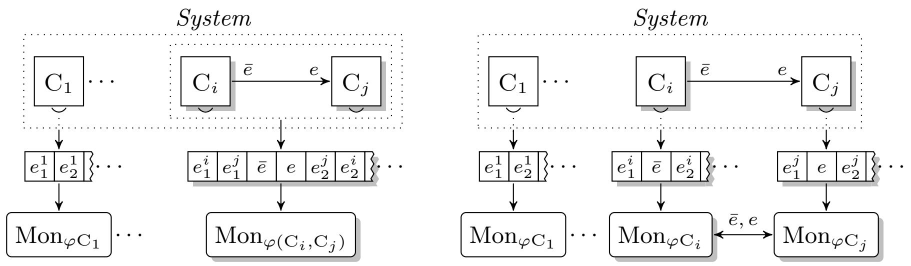
Example 5. The precision of monitoring for the local property $[\mathtt{s}]\mathbf{ff}$ over a universal trace in E.g. 1 can be enhanced by converting it into the global property $\mathbf{max}\, X.([\mathtt{o}]X \wedge [\mathtt{w}]X \wedge [\mathtt{c}]X \wedge [\mathtt{s}]\mathbf{ff})$ that handles extraneous events from F, and reverting to the monitoring set-up of Fig. 4a from that of Fig. 4b. By contrast, monitoring for $[\mathtt{s}]\mathbf{ff}$ on the partitioned trace of N as in Fig. 4d does not require any trace filtering, lowering runtime overheads. Note that the constructed global property $\mathbf{max}\, X.([\mathtt{o}]X \wedge [\mathtt{w}]X \wedge [\mathtt{c}]X \wedge [\mathtt{s}]\mathbf{ff})$ evaluated over a universal trace still does not attain the precision of $[\mathtt{s}]\mathbf{ff}$ monitored locally, due to the *common* event $\mathtt{e}$ that may be generated by *either* of components F or N. Since one cannot infer its provenance at the level of a universal trace, $\mathtt{e}$ cannot be filtered out and, correspondingly, an "ignore $\mathtt{e}$" subformula $[\mathtt{e}]X$ cannot be added as another conjunction to the global property $\mathbf{max}\, X.([\mathtt{o}]X \wedge [\mathtt{w}]X \wedge [\mathtt{c}]X \wedge [\mathtt{s}]\mathbf{ff})$ without compromising correctness. Contrastingly, in Fig. 4d, event $\mathtt{e}$ is automatically suppressed when exhibited by F, but considered when exhibited by N. ■

The benefits of local monitoring over partitioned traces are enjoyed when global properties can be *reformulated* in terms of local properties. Whenever global properties *cannot* be fully localised due to dependencies across the various components, these can still be synthesised in a *decentralised* fashion to exploit the underlying partitioned trace instrumentation set-up. We illustrate this next.

Example 6. Consider the global sHML formula $[\mathtt{o}]\mathbf{ff} \wedge [\mathtt{s}]\mathbf{ff}$ stating that, *"on start-up, the system can never produce an open event, nor can it produce a send event"*. For the set-up in Fig. 1, the property is violated whenever F produces event $\mathtt{o}$ *or* N produces event $\mathtt{s}$. Accordingly, $[\mathtt{o}]\mathbf{ff} \wedge [\mathtt{s}]\mathbf{ff}$ can be reformulated as two local properties, $[\mathtt{o}]\mathbf{ff}$ and $[\mathtt{s}]\mathbf{ff}$, that are runtime verified by two independent local monitors analysing the respective partitioned trace of interest, as in Fig. 4d, and flagging as soon *either* one detects a violation from its local trace.

Moreover, recall the formula $[\mathtt{o}]\mathbf{ff} \vee [\mathtt{s}]\mathbf{ff}$ from E.g. 3. Although it cannot be synthesised in terms of local monitors (that reach verdicts by *exclusively* analysing their own trace partition), one can still runtime verify the formula in a decentralised fashion using monitors that individually analyse subformulae $[\mathtt{o}]\mathbf{ff}$ and $[\mathtt{s}]\mathbf{ff}$, and communicate with each other once a detection is made by *both*. Decentralised monitors can *collaboratively* reach a violation verdict only when separate local detections have been made by participating monitors and are *shared* with others. This arrangement constitutes an instance of Fig. 4c. ■

A general approach to localising monitoring over partitioned traces begets further advantages. Decomposing global properties into *smaller* local subproperties improves the *maintainability* of specification scripts, since the latter tend



(a) Treating the interacting components C_i and C_j as a single entity.

(b) Attaching communicating local monitors to each of C_i and C_j.

Fig. 5. Local monitoring for interacting components using partitioned traces.

to be less complex and more lightweight to instrument. In an effort to increase precision, global properties may be reformulated to account for the potential interleaving due to the underlying asynchronous structure (*e.g.* changing $[\mathsf{o}]\mathbf{ff} \vee [\mathsf{s}]\mathbf{ff}$ into $[\mathsf{o}][\mathsf{s}]\mathbf{ff} \wedge [\mathsf{s}][\mathsf{o}]\mathbf{ff}$); this however tends to complicate specifications.

Property decomposition also makes scripts *extensible*, since changes in existing correctness requirements do not necessitate substantial refactoring of global formulae, but merely amendments that are administered to specific local formulae; adding new components into the system carries similar benefits. Segregated monitors over partitioned traces as in Fig. 4d are better equipped to handle *failures* which, in component systems, typically affect only a component subset. For instance in E.g. 6, the failure of component F does not prevent the monitor at N from making its detections; this renders the whole set-up *fault-tolerant*. By constrast, global monitoring relies on a central trace processing model that can be crippled by the smallest of partial system failures.

Component interaction synchronises the involved parties, and this interaction is often recorded in the respective traces as an event and its dual (*e.g.* a *write* event in one partitioned trace and a corresponding *written* event in the other). This establishes a partial ordering on events *across* partitioned traces. For instance, Fig. 5a depicts components C_i and C_j generating event sequences $e^i_1.\bar{e}.e^i_2$ and $e^j_1.e.e^j_2$ resp., where event e is the dual of $\bar{e}$. In a combined trace of these partitioned traces, events e^i_1 and e^j_1 may occur in any order relative to one another; the same applies to events e^i_2 and e^j_2. However, event e^i_1 must always precede e^j_2, and similarly e^j_1 must always precede e^i_2, where the synchronising events $\bar{e}$ and e act as a delimiter for the possible permutations.

In a partitioned trace set-up, the problem of monitoring for properties that span over multiple communicating components can be circumvented by choosing to treat the interacting components as *one* component subset generating a *single* partitioned trace for the respective components. This then allows them to be *locally* monitored, as shown in Fig. 5a with $\mathrm{Mon}_{\varphi(\mathrm{C}_i,\mathrm{C}_j)}$ monitoring for the local property concerning these components. Alternatively, individual monitors can also be attached to C_i and C_j as shown in Fig. 5b, though these must then *communicate* between themselves in order to determine the relative ordering of the events exhibited by each component in relation to $\bar{e}$ and e. In the sequel, we favour arrangement Fig. 5a since this leads to local monitoring of Fig. 4d.

4 The Implementability of Local Monitoring

We demonstrate the implementability of our local monitoring approach by considering actors in Erlang [3,9] which constitute an instance of asynchronous component systems. Erlang is a general-purpose, concurrent programming language where *actors* are concurrent units of decomposition that do not share any mutable memory. They interact with one another via *asynchronous* messages and change their internal state based on the messages received. Actors are implemented as lightweight processes that are identified via unique process IDs (PIDs). Every actor owns a message queue, called a *mailbox*, to which messages are sent in a non-blocking fashion; subsequently, these messages can be selectively consumed by the the recipient actor at any stage.

Our implementation conveniently utilises the tracing mechanism proffered by the Erlang Virtual Machine (EVM) to obtain local trace events. Tracing via the EVM [9] makes it possible to observe actor behaviour *without* modifying the system through commonly used instrumentation techniques such as Aspect-Oriented Programming (AOP). It can be *selectively* applied to specific groups of actors simultaneously, enabling one to independently target different subparts of the system and attain partitioned traces as described in Sec. 3. A *traced* actor generates event messages describing the nature of the trace events (*e.g.* function calls, message sends and receives, *etc.*). These trace messages are directed by the EVM to the mailbox of a specifically designated *tracer* actor. Tracing serves as the basis for a number of utilities, including Erlang's text-based tracing facility dbg [3], and the third-party monitoring tool Recon [18].

The set-up in Fig. 4d can be naturally phrased in terms of Erlang actors. In our tool implementation, actors are used to represent both the system components $\mathrm{C}_1, \ldots, \mathrm{C}_n$ and their associated monitors $\mathrm{Mon}_{\varphi \mathrm{C}_1}, \ldots, \mathrm{Mon}_{\varphi \mathrm{C}_n}$: trace event collection is handled by the EVM as explained above, where the role of tracers is assumed by monitor actors. Our tool also handles dynamic reconfiguration of component systems by adjusting the local monitoring set-up of Fig. 4d accordingly. In fact, actor systems typically are not static, since actors may terminate and new actors may be spawned. Our implementation can either terminate or dynamically assign new local monitors to the spawned actors, thereby scaling the monitoring organisation accordingly. A rudimentary implementation of this monitoring mechanism can give rise to race-conditions. Specifically, system (actor) components that require monitoring may spawn and forge ahead executing *before* their associated monitors have been properly attached, potentially leading to a loss of trace events. To avoid this, the tool opts for a *synchronous* monitor instantiation procedure that *pauses* the components that require monitoring until their associated monitors have been created and started correctly. Synchronisation takes place via instrumented source code instructions inside the monitored components which communicate with a special coordinating actor that manages the initialisation sequence of components and their corresponding monitor actors. We have integrated our implementation within the detectEr tool [4] which synthesises monitors from property descriptions expressed as sHML formulae.

We conjecture that such an implementation arrangement does not favour any particular language or property specification formalism, nor is it tied to the unit of decomposition employed by the host language's programming paradigm. In the absence of a tracing mechanism such as that offered by Erlang, one can resort to instrumentation techniques including intermediate code-level (*e.g.* AspectJ) or proxy-based (*e.g.* Spring AOP) weaving [25].

5 Experimental Evaluation

Local monitoring over partitioned traces induces *lower* performance overheads. We substantiate this claim through a series of empirical experiments performed over an Erlang third-party component-based software called Ranch [19]. Ranch is a socket acceptor pool for TCP protocols that can be used to build custom network applications (*e.g.* the Cowboy HTTP web server [19] uses Ranch to manage its client connections). Our evaluation permits us to: *(i)* explore the applicability of local monitoring by identifying cases where it can be used (non-artificially) to monitor third-party software, and also *(ii)* investigate whether it can be *feasibly* applied to real-world scenarios.

5.1 Monitoring for the Ranch Connection Protocol

Performance tests on Ranch were conducted using a number of sHML properties designed to push the application to its limits, the better to assess how local monitoring behaves under usages typical of production environments. The properties target various aspects of Ranch, and focus mainly on communication exchanges, *i.e.,* sends and receives (denoted by **!** and **?**), that take place between different components inside Ranch.

For instance, the following *recursive* local property describes behaviour for a fragment of the Ranch connection protocol used by acceptor components and the connections supervisor, from the acceptor's point of view:

$$\begin{aligned}&\mathbf{max}\,X.\big(\\ &\qquad [\mathit{ConnsSup}\,\mathbf{!}\,\{\texttt{ranch\_conns\_sup}, \texttt{start\_protocol}, \mathit{Acpt}, \mathit{Sock}\}]\ ❸\\ &\qquad\quad ([\mathit{Acpt}\,\mathbf{stp}\,\texttt{killed}]\,\mathbf{ff} \wedge [\mathit{Acpt}\,\mathbf{?}\,\mathit{ConnsSup}]\,X\ ❺)\quad \big)\end{aligned} \tag{2}$$

In this protocol (see Fig. 6), *acceptors* wait on a port for incoming connections. When a connection is established ❷, the acceptor exchanges its newly acquired client socket information with the *connections supervisor* `pid_r` ❸. Consequently, a *protocol handler* is spawned by the connections supervisor so that ownership of the client socket is transferred to the handler ❹, permitting it to engage in direct communication with the client from that point onwards ❻.

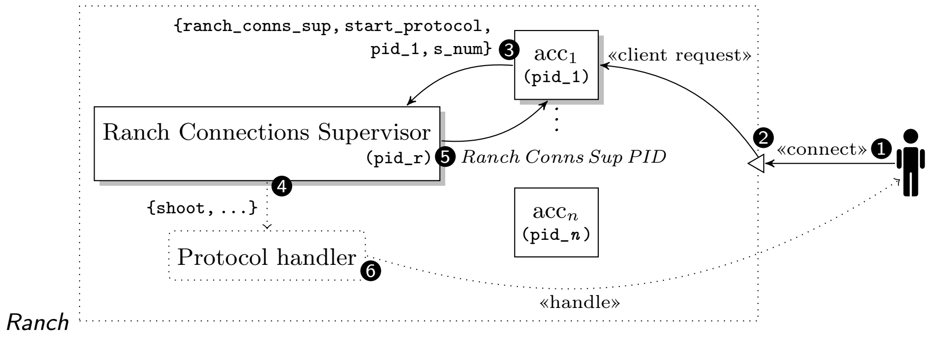

{ranch_conns_sup, start_protocol, pid_1, s_num} ❸
acc$_1$ (pid_1)
«client request»
Ranch Connections Supervisor
(pid_r) ❺ Ranch Conns Sup PID
❷ «connect» ❶
❹ {shoot, ...}
acc$_n$ (pid_n)
Protocol handler ❻
«handle»
Ranch

Fig. 6. The Ranch connection protocol used to handle incoming client connections.

Upon successful creation of the protocol handler, the connections supervisor acknowledges back to the acceptor ❺. Fig. 6 shows a Ranch configuration consisting of n acceptors with its first acceptor with PID `pid_1` servicing a new client connection.

Prop. 2 employs pattern matching as explained in Sec. 2 to dynamically bind the formula variables to the process and socket identifiers. Concretely, the pattern [*ConnsSup* **!** {`ranch_conns_sup`,`start_protocol`, *Acpt*, *Sock*}] matches the client socket information sent by acceptor `pid_1` (encoded as the Erlang tuple {`ranch_conns_sup`, `start_protocol`, `pid_1`, `s_num`}) to the supervisor `pid_r`, binding the pattern variables *ConnsSup*, *Acpt* and *Sock* to the respective values `pid_r`, `pid_1` and `s_num` ❸. Following this, the acceptor may either crash, thus matching the second necessity subformula [`pid_1` **stp** `killed`] **ff** *violating* Prop. 2, or receive an acknowledgement message from the connections supervisor, matching the third necessity subformula [`pid_1` **?** `pid_r`] ❺.

5.2 Experiment Set-up and Design

Our evaluation focusses on global properties that can be cleanly decomposed into a set of local properties which can be verified against a partitioned trace. Each experiment was conducted as a series of performance benchmarks where local properties were monitored over individually executing components, and the results were in turn compared against those yielded by monitoring the original global property over the entire system. Global properties were monitored using the detectEr tool developed in [4], whereas their decomposed local constituents were handled by the tool extension reported in Sec. 4. These two set-ups correspond to Fig. 4a and Fig. 4d resp. Performance was judged on: *(i)* the system's memory consumption in MB, *(ii)* its CPU usage, given as a percentage, and *(iii)* the system response time in milliseconds. Each experiment is presented in Fig. 7, plotting the results of the performance parameter (*e.g.* CPU utilisation) under consideration (y-axis) against the local and global monitoring benchmarks (x-axis). We also include the *unmonitored* system measurements as a baseline.

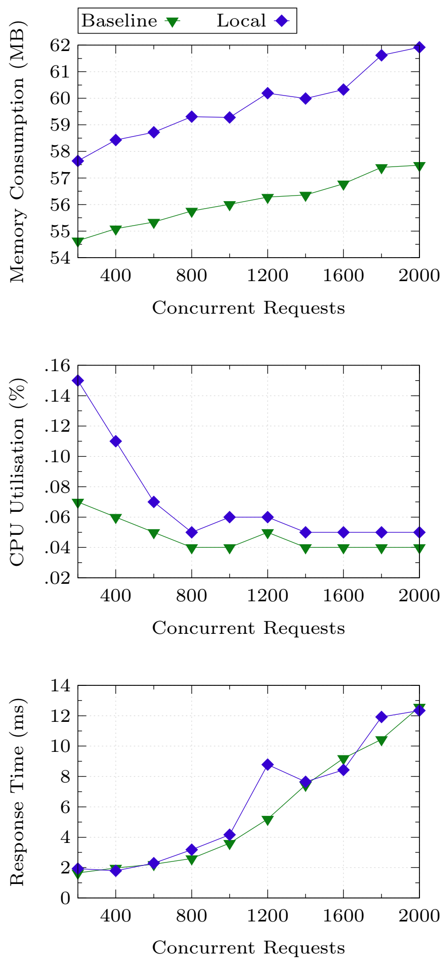
Data Collection and Precautions. An *experiment* refers to a set of ten performance benchmarks, each performed by load testing individual system configurations (*e.g.* the system with local monitors, *etc.*) using a series of concurrent requests, commencing at 200 and progressing up to 2000 in steps of 200 (*i.e.,* $200, 400, \ldots, 2000$). Results for repetitions of sets of ten experiments were averaged to obtain the plots shown in Fig. 7. A number of precautions were taken to ensure the accuracy and repeatability of our results: *(i)* ten *repeated readings* for each experiment were taken, after calculating the *coefficient of variation* (*i.e.,* $\frac{\sigma}{\bar{x}} \times 100$) for different sets of experiment repetitions (*e.g.* five, ten, fifteen, *etc.*) showed negligible variability between the data sets obtained with ten repetitions and above, *(ii)* optimisations such as *garbage collection* were switched off so that the readings obtained clearly underscore the differences between local and global monitoring, *(iii)* performance spikes in the initial set of data points due to the lazy start up of the internal VM infrastructure were eliminated by issuing a series of *warm-up requests* before the actual benchmarks tests were performed.

5.3 Results and Analysis

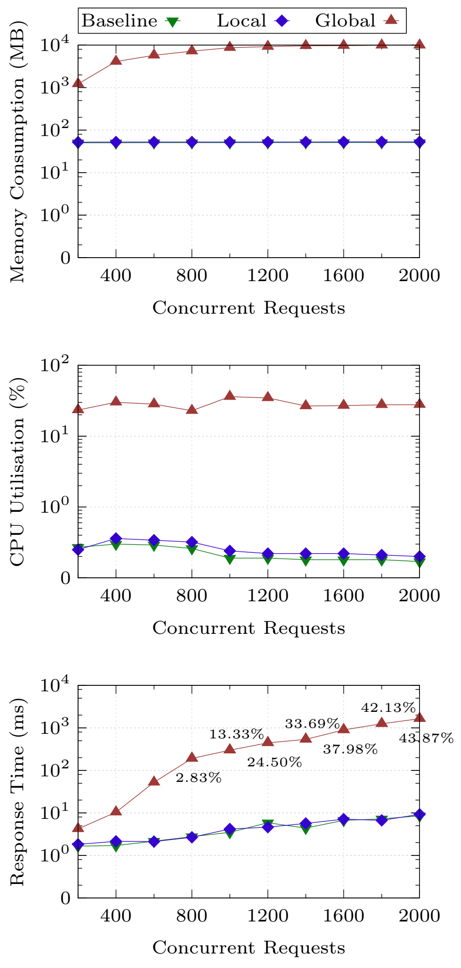
Fig. 7 shows the experiment results for monitors synthesised from formulae such as Prop. 2 using two Ranch configurations: one with a hundred acceptors, and one with four. All experiments were conducted on a 3.1 GHz Intel Core i7 processor with 16GB of memory.

Realistic Ranch configuration. We first applied local monitoring to the *recommended* Ranch set-up configured with one hundred acceptors [19]. The plots for the unmonitored and locally monitored Ranch set-up in Fig. 7a show that the memory and CPU-related overheads induced by local monitoring are reasonably low and exhibit an analogous rate of change to those of the unmonitored system. This suggests that the resource consumption overheads due to local monitoring follow those of the unmonitored system, and in such cases, one would be able to *forecast* the extra system resource requirements that would be introduced by local monitoring. Response times measure the *observable* impact of monitoring on the behaviour of the system; the plot in Fig. 7a shows that the performance impact of local monitoring is imperceptible for the benchmarks considered. Note that for all three parameters, evaluating global monitoring on this Ranch set-up was not possible because it consistently led to resource exhaustion.

Other configurations. Our attempts at evaluating global monitoring on Ranch configured with one hundred acceptors were stymied by the high amount of overheads. To investigate which settings would permit us to test global monitoring suitably, we used various Ranch configurations with different numbers of acceptors. These trials revealed that only Ranch configurations having less than five acceptors could be reliably benchmarked without crashing. Fig. 7b shows the memory, CPU and response time plots for Ranch with four acceptors (in log scale). Whereas the overheads induced by global monitoring are infeasibly higher than the baseline, those resulting from local monitoring are decidedly closer to the measurements obtained for the unmonitored system. Specifically, in Fig. 7b



(a) Performance measurements for Ranch with one hundred acceptors (linear scale).



(b) Performance measurements for Ranch with four acceptors (log scale).

Fig. 7. Memory, CPU and response time benchmarks for two Ranch configurations.

we achieved an average memory overhead of 4.54% and an average CPU overhead of 14.22%. The response time plot in Fig. 7b additionally displays the percentage of *request failures*, where each value represents the ratio of failed requests that resulted when the benchmarked system was unable to cope with the number of concurrent requests due to errors such as TCP connection refusals, timeouts and broken pipes. Global monitoring degrades response behaviour both quantitatively (response times) and qualitatively (failed responses), whereas local monitoring induces negligible overheads without provoking any failed requests.

6 Conclusion

We have presented a novel monitoring technique for asynchronous components that generates partitioned traces reflecting the interleaved execution of the constituent components under scrutiny. We argued how this yields benefits on many fronts. In particular, we demonstrated that the proposed instrumentation set-up lends itself better to local monitoring. Our approach is, in part, inspired by distributed monitoring settings where partitioned traces come about naturally due to physical constraints such as the absence of global clocks. In our case, however, we have the added benefit of controlling the trace partitioning level, coalescing tracing for tightly coupled components so as to attain local monitoring (see discussion for Fig. 5a) — this cannot be achieved for physically distributed components such as those in [26]. Local monitoring is used in a variety of application domains such as session types [7,21], where monitors are attached to individual channel endpoints. To our knowledge, the overhead gains of such a set-up, as opposed to a global approach, have never been validated for these domains and we expect our results to be applicable. As future work, we plan to extend our study to distributed settings and investigate aspects such as trace reconstructions that increase RV's precision and expressivity.

Related Work. There are various RV approaches for asynchronous components where trace events are collected globally and monitors analyse system behaviour as a single universal trace [4,17]. Even though these monitors decentralise their analysis via concurrent submonitors, the correctness of the system in question is still perceived globally, and thus they classify as the set-up depicted in Fig. 4a.

Parameteric Trace Slicing (PTS) [11,5] is a monitoring technique whereby a universal trace is *projected* into subtraces called *trace slices*, based on parametric specifications, *i.e.,* properties that are specified in terms of *parametrised* symbolic events whose parameters are bound to values from concrete events in the universal trace. Slicing is mainly concerned with filtering events from a universal trace so as to obtain *local* monitors as in Fig. 4b. PTS differs from our work in these respects: *(i)* projection is *not* partitioning since an event may be assigned to multiple subtraces (*i.e.,* their local monitors may overlap w.r.t. events), *(ii)* subtraces are described at the specification level whereas partitioning works at the instrumentation level, *(iii)* parametric specifications typically describe *replicated* component behaviour sharing a common structure, whereas we are able to partition non-replicated components, *(iv)* since PTS works on a universal trace, events that cannot always be attributed to a unique component (*e.g.* event e in F and N of E.g. 1 and E.g. 5) cannot be filtered as selectively. ELarva [13] can be seen as an instance of PTS applied to asynchronous components. It also targets Erlang actors and is implemented using the EVM's native tracing mechanism as in Sec. 4. However, ELarva relies on a universal trace, and through an application-level routing mechanism, multiplexes events from the trace to monitors attached to different components. The parametric properties specified per spawned actor facilitate dynamic monitor creation, but the centralised trace processing mechanism induces unnecessary bottlenecks that may hamper gains obtained from monitor parallelism. We are unaware of any PTS used for branching-time specifications of asynchronous components.

The closest work to ours is [6,12] where global LTL formulae are monitored locally over partitioned traces. The authors propose and evaluate a decentralised approach that decomposes a given global LTL specification into smaller subproperties that analyse separate trace partitions and communicate amongst themselves to handle subformula dependencies accordingly, as depicted in Fig. 4c. They also show that local monitoring yields lower monitoring overheads. The work differs from ours w.r.t. the following aspects: *(i)* they consider synchronous systems, governed by a global clock that yields a *unique* combined trace from the respective partitioned traces; asynchronous settings are richer and typically yield multiple combined traces, *(ii)* the logic considered describes execution *traces* whereas we consider a logic describing properties over *programs*; we show how the multiple combined traces inferred in asynchronous settings can be exploited to increase the monitor's precision and monitorability of such properties, *(iii)* they require a complete partitioning of trace events in order to automate formula decentralisation, whereas we allow components to share trace events (*e.g.* components F and N from E.g. 1 can both exhibit event e), *(iv)* the evaluation in [6,12] focusses on decentralised communicating monitors that still regard correctness from a global perspective (analogous to $[\mathsf{o}]\mathbf{ff} \vee [\mathsf{s}]\mathbf{ff}$ from E.g. 6); our evaluation rather concentrates on properties that can be cleanly decomposed into local ones that fully capitalise on trace partitioning, *(v)* their tool assumes a fixed number of components that remains constant throughout execution as opposed to ours, which can handle dynamic partitioning as well.

Partitioned traces are also used for monitoring shared-state concurrency programs such as [27], where decentralised monitors attached to different executing threads collect and analyse events locally and actively collaborate in order to build a combined representation of the present system state. The data exchange between monitors takes place when shared variables are accessed (for reading or writing) by the executing threads; this can be seen as an instance of the set-up depicted in Fig. 4c. By contrast, our work concentrates on studying local monitors over such partitioned traces, as discussed in Fig. 4d. In particular, we assess the performance impact of local monitoring, whereas performance issues are not a focus of [27]. Instead they study the detection and prediction of particular types of safety properties. As in earlier work by the same authors [26], the investigation is conducted in terms of a linear-time epistemic logic that describes execution traces, whereas we consider a logic describing the branching program behaviour as a computation graph, which gives us scope for inferring other parts of the computation graph from the path observed at runtime.

References

1. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge Univ. Press, New York, NY, USA (2007)
2. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7, 1–72 (1997)

3. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
4. Attard, D.P., Francalanza, A.: A Monitoring Tool for a Branching-Time Logic. In: RV. LNCS, vol. 10012, pp. 473–481 (2016)
5. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: FM. LNCS, vol. 7436, pp. 68–84 (2012)
6. Bauer, A.K., Falcone, Y.: Decentralised LTL Monitoring. In: FM. LNCS, vol. 7436, pp. 85–100 (2012)
7. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring Networks through Multiparty Session Types. In: FORTE. LNCS, vol. 7892, pp. 50–65 (2013)
8. Cassar, I., Francalanza, A.: On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In: IFM. LNCS, vol. 9681, pp. 176–192 (2016)
9. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly Media (2009)
10. Chappell, D.: Enterprise Service Bus: Theory in Practice. O'Reilly Media (2004)
11. Chen, F., Rosu, G.: Parametric Trace Slicing and Monitoring. In: TACAS. LNCS, vol. 5505, pp. 246–261 (2009)
12. Colombo, C., Falcone, Y.: Organising LTL Monitors over Distributed Systems with a Global Clock. In: RV. LNCS, vol. 8734, pp. 140–155 (2014)
13. Colombo, C., Francalanza, A., Gatt, R.: Elarva: A Monitoring Tool for Erlang. In: RV. LNCS, vol. 7186, pp. 370–374 (2011)
14. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? STTT 14(3), 349–382 (2012)
15. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: On Verifying Hennessy-Milner Logic with Recursion at Runtime. In: RV. LNCS, vol. 9333, pp. 71–86 (2015)
16. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Monitorability for the Hennessy-Milner Logic with Recursion. Formal Methods in System Design pp. 1–30 (2017)
17. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors. Formal Methods in System Design 46(3), 226–261 (2015)
18. Hebert, F.: Recon. `http://ferd.github.io/recon`, accessed: 2017-03-13
19. Hoguin, L.: 99s. `http://ninenines.eu`, accessed: 2017-03-13
20. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional (2003)
21. Jia, L., Gommerstadt, H., Pfenning, F.: Monitors and Blame Assignment for Higher-order Session Types. In: POPL. pp. 582–594. ACM (2016)
22. Josuttis, N.M.: SOA in Practice: The Art of Distributed System Design: Theory in Practice. O'Reilly Media (2007)
23. Larsen, K.G.: Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. Theor. Comput. Sci. 72(2&3), 265–288 (1990)
24. Leucker, M., Schallhart, C.: A Brief Account of Runtime Verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
25. Safonov, V.O.: Using Aspect-Oriented Programming for Trustworthy Software Development. Wiley-Interscience (2008)
26. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: ICSE. pp. 418–427 (2004)
27. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Decentralized runtime analysis of multithreaded applications. In: IPPS (2006)
28. Vella, A., Francalanza, A.: Preliminary Results Towards Contract Monitorability. In: PrePost@IFM. EPTCS, vol. 208, pp. 54–63 (2016)