WALTZ: Formalizing the Dance Between Actors and Messages

Vladyslav Mikytiv[1], Bernardo Toninho[2], and Carla Ferreira[1]

[1] NOVA LINCS, NOVA University Lisbon, Portugal
[2] IST, INESC-ID, Portugal

Abstract. Software verification is a key principle for building and deploying reliable and correct software. Among verification techniques, runtime verification has emerged as a lightweight and complementary approach to traditional formal methods, providing assurance over a system under scrutiny. By performing checks while the system is running, runtime verification enables the dynamic monitoring of a system's behaviour. Monitoring actor-based systems is particularly interesting in the context of runtime verification, as the traces produced by these systems consist of the messages exchanged between different actors. In this work, we propose a specification language that enables the specification of interaction schemas and the properties of exchanged data in actor-based systems. This specification language allows us to identify actors with whom these actors communicate and the properties that the exchanged messages are intended to satisfy. Coupled with this specification language, we propose a monitor synthesis procedure that automatically generates executable code from the high-level specification language.

1 Introduction

Ensuring the production of accurate, reliable, and safe software is essential for developing robust software systems. This requirement has become increasingly critical as software continues to influence vital areas such as healthcare, transportation, and communication. To address the need for thorough verification of software behaviour and correctness, formal methods [12] have emerged. Formally ensuring the correctness of a given piece of software can be performed by several techniques, such as, *theorem proving* [10], *model checking* [13] and *testing* [17], each with its strengths and trade-offs.

Runtime verification (RV) is a dynamic approach that analyses a computing system's behaviour across various application scenarios [6,9,18,16]. It complements other verification techniques by providing an additional layer of assurance for software, allowing us to verify, enforce, and prevent failures through its runtime capabilities [15]. Additionally, RV can be used to gather information about a running system for later analysis and review [14]. The primary goal of RV is to check, during execution, whether a system run satisfies given correctness properties [16]. Achieving this involves several steps: observing the system, interpreting the data, verifying property satisfaction, issuing alerts on violations, and responding appropriately.

This challenge becomes even more pronounced in modern software, which increasingly adopts distributed architectures based on the actor model. In this programming model, autonomous entities interact via message passing to perform complex computations. While the actor model enhances scalability, fault tolerance, and modularity, it also complicates correctness assurance [3]. Traditional verification approaches, especially static analysis, often struggle with actor systems due to their reliance on dynamic, runtime interactions.

Runtime verification offers a practical approach to addressing the verification gap by enabling continuous monitoring of system executions against formally specified properties using dynamic monitors. These monitors act as runtime sentinels, consistently ensuring that the system complies with its specified properties throughout the program's execution.

Example 1. The system architecture consists of a central server responsible for routing incoming requests to another process. In this simplified implementation, the system accepts an input list and propagates the processing task to a worker responsible for removing duplicates from that list. The worker then propagates the new, duplicate-free list to another process that is responsible for sorting the given list. The system as a whole ensures the elimination of duplicate values and the sorting of that given list. The operational workflow is illustrated in Figure 1.

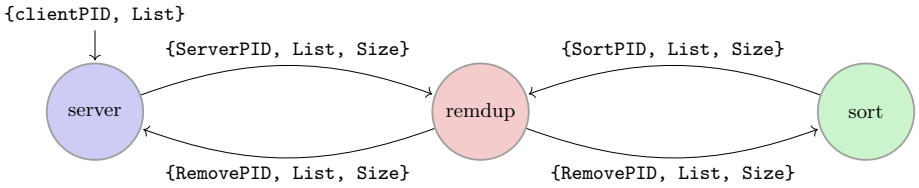

Fig. 1: Actor-based system that operates over a list

An interesting property to verify in this system is whether the size of the list after the `remove` process is indeed reduced or remains unchanged, in other words, whether the size of the list does not increase.

The size of the list after the remove operation does not increase $(\mathbf{P_1})$

We could also verify whether the process responsible for sorting the operation of a given list is indeed returning a sorted list to process `remove`.

The sort response to the remove process is a sorted list $(\mathbf{P_2})$

Additional properties can be defined according to the desired level of precision. If the properties are too generic, certain specific behaviours will not be captured. The flexibility of these properties depends on the needs and the level of verification we want to achieve. For example, the two properties above would

not capture the bug where the list is reduced to an empty one. It is up to the developer to exhaustively specify critical properties to check at runtime.

This paper introduces a specification language for property verification at runtime in actor-based systems, addressing key challenges in expressing cross-actor properties. The language provides intuitive constructs that mirror the actor communication patterns and automatically manage state correlation across message exchanges. Properties are compiled into separate Erlang monitor processes that utilise the language's built-in tracing infrastructure [4], allowing for non-invasive monitoring by observing relevant messages without interfering with system execution or by slightly modifying the original code minimally. Our contributions include a language that naturally captures intra-actor properties, automatic state correlation for verifying complex interactions, implicit message filtering via signature matching to reduce specification overhead, and a compilation process that translates high-level specifications into efficient, standalone Erlang monitors.

2 Overview

A system's behaviour refers to how it operates and responds to specific external requests or stimuli. These triggers cause the system to process the input, resulting in changes to its internal state over time, and produce outcomes or effects which may also influence how the system behaves in future interactions. The behaviour of a system is something that one must take into account when working with it or, in this case, verifying it. Besides that, it is important to know how that behaviour is represented to interpret what the given system is doing [6].

2.1 System events and trace

The notion of what constitutes a trace does not have a formal unified definition in the runtime community, as each author defines it in a way that benefits their tool [14]. Most of the tools in RV hard-code the traces for simplicity.

The actor-based model is beneficial due to its intuitive structure, which allows for a clear understanding of traces that represent the history of messages exchanged between actors. This paradigm enables the observation of a running system and real-time analysis of its execution within a consistent framework of events and traces. While instrumentation is necessary, it is far more manageable compared to dealing with non-standard trace representations. With this in mind, we can define the possible actions in an actor-based system. Actors can exchange messages via `send` and `receive`, create new actors using `spawn`, forming parent-child links, and initialise processes. The interaction shown in Figure 1 can be directly mapped to actor-based code, as illustrated in the following pseudo-code:

```
receive {server, send, Msg1, remdup} ->
  case Msg1 of {ServerPID, List1, Size1} ->
    receive {remdup, send, Msg2, sort} ->
      case Msg2 of {RemovePID, List2, Size2} -> ...
```

In Erlang, each process has its mailbox where incoming messages are stored. In the actor-based model, sending a message from one process to another preserves the order between those two processes. However, when multiple processes send messages to the same recipient, the order in which messages arrive and are processed may vary, as it depends on the order of their arrival.

To address this challenge and avoid ambiguity when correlating requests and responses, Erlang and other actor-based languages rely on a standardised way for attaching unique references to messages. These references allow processes to associate messages with the correct requests, even if messages are processed out of order. All well-formed Erlang programs are expected to follow this convention. The language provides libraries, such as OTP, to abstract and manage references, promoting consistency across systems. In our work, we assume that the Erlang code adheres to these principles, as our focus is on verifying system properties rather than checking the structural correctness of the code itself.

3 Language definition and semantics

Our specification language is designed around three fundamental abstractions that directly correspond to the concepts of actor-based systems: actors as autonomous computational entities, message patterns as communication signatures, and state bindings that correlate information across multiple interactions.

These abstractions work together to provide a natural and intuitive way to express properties that span multiple actors. Broadly, the language consists of two key components. One dictates the control flow of messages, defining how messages are exchanged, chained, and related across different actors. The other focuses on verifying properties within the content of these messages, ensuring correctness within the established flows.

Definition 1 (WALTZ Formula). *The set of WALTZ formulas is defined by the following grammar:*

$$
\begin{aligned}
\varphi &::= \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \Omega(\varphi) \mid \varphi \,;\, \varphi \mid \alpha : \delta \\
\delta &::= \top \mid \bot \mid c \mid \neg\delta \\
\alpha &::= \mathtt{send}_{\mathtt{id}\to\mathtt{id}}\{\mathtt{M}\} \\
\mathtt{M} &::= \mathtt{T} \mid \mathtt{T}\,,\mathtt{M} \mid \{\mathtt{M}\} \\
\mathtt{T} &::= (\mathtt{cons} \mid \mathtt{var})
\end{aligned}
$$

Here, α represents an interaction signature in an actor-based system, c is a boolean constraint, and M is the signature of a message. The modal operator Ω checks different contexts (explained in detail later), functioning essentially as a recursive construct that enables the evaluation of properties over multiple independent message chains. The expression $\alpha : \delta$ indicates that a message with signature α must satisfy the constraint δ. If we want to simply specify the occurrence of a message without imposing any condition, we can use $\alpha : \top$, as every message trivially satisfies $\top$.

A key construct in WALTZ is the chain operator (`;`), which sequences two formulas. The expression $\alpha_1 : \delta_1 \ ; \ \alpha_2 : \delta_2$ specifies that a message matching α_1 must first be observed and satisfy δ_1, and only afterward should a message matching α_2 be observed and satisfy δ_2. This sequencing is essential for expressing the order in which messages should occur.

A signature (α) in an actor-based system represents a pattern for a message involving two different actor entities and a send operation (though other operations exist, we're focusing on send here). For example, the signature $\texttt{send}_{A \to B}$ `{A, B, {C, ok}}` specifies the operation type, the source and destination actors, and the structure of the message along with its payload. At runtime, messages sent in the system are checked against these signatures to ensure they match the defined pattern and whenever they match, the values will be bound to the variables at runtime.

3.1 Semantics

A system $\mathcal{S}$ has an alphabet Σ containing all its observable events. Given an alphabet Σ, a trace $\sigma = \sigma_1\sigma_2\ldots$, is a sequence of events containing elements of Σ. We denote by σ_i the i-th element of σ, σ^i is the suffix of σ starting from i, Σ^* is the set of all possible finite traces over Σ, and Σ^ω is the set of all possible infinite traces over Σ.

An interaction with the system produces a trace composed of various events, some of which are causally or logically related. We refer to such related sequences as belonging to the same **context**. A trace may contain multiple contexts, each representing an independent chain of events. Our goal is to verify properties over these individual contexts. Since event ordering is not always guaranteed and interactions may overlap, we assign a context identifier to each event to determine which events belong together. Formally, a message will be mapped to a context that represents a causal chain in the system. Let $\mathcal{C}$ be the set of all context identifiers (e.g., Δ_0, Δ_1). We introduce a function $\gamma : \Sigma \to \mathcal{C}$ that maps each event (a message) in our trace to a unique context identifier. Intuitively, $\gamma(\sigma_i)$ tells us which context the event σ_i belongs to. In practice, a new context is introduced into the system each time we re-enter a `receive` clause within a pattern-matching chain, as such:

```
a():
  receive {Msg} -> ..., a() end.
```

The main context is denoted by Δ_0, and additional sub-contexts may be created depending on the property being specified. Contexts in the system can be structured hierarchically, allowing for the modelling of nested behaviours. Specifically, we define a hierarchical relation between two contexts using the notation $\Delta_b \triangleright \Delta_a$, which denotes that context Δ_b is derived from context Δ_a. This hierarchical relationship captures the causal or structural dependency between contexts and it translates into `spawn` events within `receive` clauses:

```
a():
  receive {Msg} -> ..., spawn(b(), Context), a() end.
```

Definition 2 (WALTZ Semantics). *Let us state that $\sigma = \sigma_1\sigma_2... \in \Sigma^\omega$ be an infinite sequence of events and that $\mathcal{C}$ is the set that represents the contexts and the relations between them in the given trace, γ the function that maps messages to a context, and Δ is the current context. Then we can define the semantics of WALTZ as follows:*

$$\sigma, \Delta \models \alpha : \delta \text{ iff } \exists_{i \geqslant 0} : \sigma_i \models \delta \text{ and } \sigma_i \bowtie \alpha \text{ and } \gamma(\sigma_i) = \Delta$$
$$\sigma, \Delta \models \varphi \ ; \ \varphi' \text{ iff } \exists_{i \geqslant 0} : \sigma^i, \Delta \models \varphi \text{ and } \exists_{j \geqslant 0} : \sigma^j, \Delta \models \varphi' \text{ and } i < j$$
$$\sigma, \Delta \models \varphi \vee \varphi' \text{ iff } \sigma, \Delta \models \varphi \text{ or } \sigma, \Delta \models \varphi'$$
$$\sigma, \Delta \models \varphi \wedge \varphi' \text{ iff } \sigma, \Delta \models \varphi \text{ and } \sigma, \Delta \models \varphi'$$
$$\sigma, \Delta \models \Omega(\varphi) \text{ iff } \forall_{\Delta_k \blacktriangleright \Delta} : \sigma, \Delta_k \models \varphi$$

where $\bowtie$ is the operator that checks if a message has a given signature, and $\blacktriangleright$ is the operator that relates a sub-context with a higher one in the hierarchy.

Intuitively, given a trace σ and the existing contexts $\mathcal{C}$, the following can be stated: σ, Δ satisfies $\alpha : \delta$, means that a message with signature α must be observed in the trace, it must also satisfy δ, and that message belongs to the context in which it is being evaluated (Δ).

We say that σ, Δ satisfies $\varphi; \varphi'$ if there exists a suffix of the trace starting at position i that satisfies φ, and another suffix starting at position j that satisfies φ', where the event at position j occurs after the event at position i. The operator ; allows the definition of message chains by enforcing an order between events and specifying the properties that each message in the chain must satisfy. This mechanism enables the specification of properties involving multiple actors by relating the contents of message payloads across events. Importantly, all events in the same chain share the same context.

σ, Δ satisfies the conjunction if it satisfies both properties in the conjunction, and it satisfies the disjunction if it satisfies at least one, sharing the same context.

The Ω operator is the most nuanced construct in the language, as it introduces the concept of evaluating different contexts and enables the verification of properties across independent contexts. Formally, a trace σ under the set of contexts $\mathcal{C}$, and current context Δ, satisfies $\Omega(\varphi)$ if, for all contexts Δ_k such that $\Delta_k \blacktriangleright \Delta$ (i.e., Δ_k was derived from Δ), the formula φ holds in each derived context. In short, $\Omega(\varphi)$ universally quantifies over all sub-contexts forked from the current one, requiring that φ be satisfied in each.

The verification process begins with a shared initial context that serves as the root for all subsequent context derivations. When Ω operators are nested, new sub-contexts are created within each other, establishing hierarchical relations. The number of contexts does not translate into the number of running monitors, as we have seen above, in the practical concept of context introduction.

3.2 Why context matters?

The context of a chain is a crucial concept for verifying the satisfiability of a formula. As stated previously, there is always a main context, denoted by Δ_0,

which always exist. If no sub-context is created, that is, no Ω operator exists to capture new contexts, then evaluation takes place entirely within Δ_0. Consider $\varphi_1 = \alpha : \delta$ and the following trace (where _ is an irrelevant message):

$$\sigma_1 = \_\ \mathtt{m}_1\ \_\ \_\ \mathtt{m}_2$$

In σ_1, all the messages with signature α belong to the same context (Δ_0). To satisfy φ_1, it is sufficient to check whether in the trace, there is at least one of them that satisfies the property δ. Some messages may not satisfy δ, but as long as at least one does, the formula is considered satisfied.

Now, consider the same trace, but suppose the goal is to check all messages with signature α that adhere to δ. In this case, the property would be defined as $\varphi_2 = \Omega(\alpha : \delta)$, and the contexts would change, as such:

$$\sigma_2 = \_\ \mathtt{m}_1\ \_\ \_\ \mathtt{m}_2$$

The objective is to verify property δ over all messages with signature α. In this example, there are two existing contexts, the context of the message $\mathtt{m}_1$ matching α, denoted by Δ_1, and the context of the message $\mathtt{m}_2$ matching α, denoted by Δ_2. These contexts, each represented by a different colour, are independent, which is precisely what is needed. Each occurrence of the message chain defined within the Ω operator is evaluated as a separate context, ensuring that the property is checked individually for every instance.

In φ_1, even if the first message matching the signature does not satisfy the property, the formula is still satisfied as long as at least one subsequent message does. This is not the case for φ_2, where it is required that the property holds for every independent chain. Now, let us consider trace σ_3 as:

$$\sigma_3 = \mathtt{m}_1\ \_\ \mathtt{m}_2\ \_\ \mathtt{m}_3\ \_\ \_\ \mathtt{m}_4\ \_\ \mathtt{m}_5$$

We can interpret σ_3 with respect to property P: *"For every client connected to the system, all requests sent by that client must have a positive payload"*. In WALTZ, this is expressed as: $\varphi_3 = \Omega(\alpha_1 : \top; \Omega(\alpha_2 : \delta))$.

Property φ_3 uses two nested Ω operators, essential for correctly capturing the intended behaviour. The outer Ω tracks each client connection α_1, matching messages $\mathtt{m}_1$ and $\mathtt{m}_3$. The inner Ω tracks the sequence of requests α_2, matching the remaining messages. Without the second Ω, the formula would monitor all requests globally rather than per client, thus failing to isolate behaviours across individual connections. The resulting structure of contexts derived from this trace and property is illustrated in Figure 2, which shows the hierarchical relationships established by the nested Ω operators.

The context Δ_0 is the root context and is always present in any trace. For example, a simple formula like $\varphi = \alpha_1 : \delta$ is evaluated within this root context Δ_0. However, when a formula involves the Ω operator, it introduces a new context that inherits the context in which it was invoked. Each client initiates a chain of events matched by the formula $\alpha : \top; \Omega(\alpha_2 : \delta)$. Upon observing an event that satisfies $\alpha : \top$, a new context is created, denoted Δ_1, with Δ_0 as its parent, i.e., $\Delta_1 \triangleright \Delta_0$. This context captures the interaction scope for a single client.

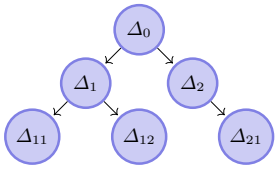

Fig. 2: The contexts of the trace σ_3

The formula then specifies another use of Ω, which applies to all subsequent messages from that client. For each message matching α_2, we create a new subcontext (e.g., Δ_{11}, Δ_{12}) under Δ_1, since we are now verifying whether each of those messages satisfies the property δ. These new contexts inherit from Δ_1, meaning $\Delta_{11} \blacktriangleright \Delta_1$ and $\Delta_{12} \blacktriangleright \Delta_1$. This hierarchical context structure enables us to handle traces where events from different chains or clients are interleaved, as satisfiability is always checked within the relevant context. Therefore, in σ_3, one of the interaction is $\mathtt{m}_1$; $\mathtt{m}_2$; $\mathtt{m}_4$ and the other one is $\mathtt{m}_3$; $\mathtt{m}_5$.

Managing these contexts can become cumbersome. However, following good coding practices helps simplify their management by ensuring that code references are handled adequately during message exchanges or by relying on standard communication modules, such as OTP. When such standards are not followed, we must introduce custom mechanisms to maintain and track contextual information by injecting additional code into the system.

4 Case studies

This section presents examples of actor-based systems and properties specified and verified using WALTZ, demonstrating the language's expressiveness in capturing a wide range of interactions.

4.1 List sorting and duplicate removal

Let us revisit the introductory example shown in Figure 1, which illustrates a system architecture where a central server routes incoming requests to subordinate processing actors. Knowing that α is $\mathtt{send}_{\mathtt{server}\to\mathtt{remdup}}$ and α' is $\mathtt{send}_{\mathtt{remdup}\to\mathtt{sort}}$ the property P_1 can be defined in as φ_1:

$$\varphi_1 = \Omega(\alpha\ \texttt{\{\_, \_, Size\}} : \top; \alpha'\ \texttt{\{\_, \_, Size'\}} : \texttt{Size' <= Size})(\mathbf{P_1})$$

This property captures all message chains that begin with a send operation from the `server` to the `remdup` process, followed by a message from `remdup` to `sort`. The specification captures the expectation that removing duplicates should not increase the size of the list. The specification enables this kind of cross-message and cross-actor data relationship by binding variables across the interaction chain. Because the formula is enclosed in a Ω operator, φ_1 will be evaluated across all such chains individually, each within its own context.

For example, given trace $\sigma = \mathtt{m}_1\ \mathtt{m}_2\ \mathtt{m}_3\ \mathtt{m}_4$, the set of contexts $\mathcal{C}$, and the mapping $\gamma(\mathtt{m}_1) = \Delta_1$, $\gamma(\mathtt{m}_2) = \Delta_1$, $\gamma(\mathtt{m}_3) = \Delta_2$, $\gamma(\mathtt{m}_4) = \Delta_2$. If $\mathtt{m}_1, \mathtt{m}_3 \bowtie \alpha_1$ and $\mathtt{m}_2, \mathtt{m}_4 \bowtie \alpha_2$ we will be able to verify at runtime whether the messages comply with our specification. Since we are working in a runtime context, the trace will grow infinitely, but we will evaluate our properties over the current observed trace, which is finite. By extracting the values of the message contents at runtime, we will check whether, for that given trace, the property is valid. The resulting pseudo-code of the generated monitor will be the following:

```
a():
  receive {server, send, Msg1, remdup} ->
    case Msg1 of {_, _, Size1} and Msg1 ∈ Δ ->
      receive {remdup, send, Msg2, sort} ->
        case Msg2 of {_, _, Size2} and Msg2 ∈ Δ  ->
          if (Size2 <= Size1) a()
          else false
```

Property φ_1 illustrates one of WALTZ's key strengths: referencing and comparing variables across different actor communications within the same chain, enabling, for example, the check that the `remdup` process is correctly removing duplicates by comparing the sizes of the lists. Since WALTZ relies on Erlang's built-in tracing mechanisms, it is not necessary to specify intermediate or unrelated messages; only the relevant interactions must be defined, and all other events will be ignored during monitoring.

We can also define properties that are more general, such as simply tracking whether the server at the end receives a sorted and duplicate-free list, or whether the `sort` process indeed sorts the list. For example, property P_2, introduced earlier, asserts that the `sort` process sends a sorted list to the `remdup` process:

$$\varphi_2 = \Omega(\mathtt{send}_{\mathtt{sort}\to\mathtt{remdup}}\ \mathtt{\{SortPID,\ List,\ Size\}} : \mathtt{isSorted(List)})(\mathbf{P_2})$$

Property φ_2 is relatively simple to verify and illustrates WALTZ's flexibility: developers can choose to model interactions at varying levels of specificity, from highly focused checks to broad system-level assertions, depending on their needs.

4.2 Clients and server connection

Example 2. Consider a system where a central server handles multiple client connections. Clients connect by sending a message `{myPID, hello}`, after which the server tracks them as connected. Once connected, clients send requests of the form `{myPID, Payload}`. To ensure server integrity, the payload must be a positive integer. We want to verify that for every client that connects, all subsequent requests contain only positive payloads. Using WALTZ we can specify φ_3, knowing that α and α' are $\mathtt{send}_{\mathtt{client}\to\mathtt{server}}$:

$$\varphi_3 = \Omega(\alpha\ \mathtt{\{MyPID,\ hello\}} : \top; \Omega(\alpha'\ \mathtt{\{MyPID,\ Payload\}} : \mathtt{Payload} \geqslant 0))$$

This formula can be interpreted as: for every message with signature `{myPID, hello}` that satisfies $\top$ (i.e, all such messages), we want to verify that all subsequent messages from the same client satisfy the property `Payload` $\geqslant 0$. The usage of contexts in WALTZ enables this, because $\Omega(\alpha : \top; \Omega(\alpha' : \delta))$ creates a new context for each client connection $\alpha : \top$, and within each of those, further sub-contexts for each client request $\Omega(\alpha' : \delta)$. This allows us to track and verify message chains for each client independently, even if their traces interleave.

Whenever we have contexts being created inside others, we `spawn` new sub-monitors that are linked together. They belong to the same context in the big picture, but we will allow them to have sub-contexts that are useful to specify richer properties over a given system.

4.3 Multiple workers

Example 3. Consider a system with a central server and multiple identical worker processes. The goal is simple: the server expects at least one worker to respond with a positive payload, we do not care which worker replies. This behaviour can be expressed in WALTZ property as φ_4, where `_` is the wildcard:

$$\varphi_4 = \texttt{send}_{\_\rightarrow\texttt{server}}\{\texttt{ok, Res}\} : \texttt{Res} \geqslant 0$$

All messages belong to the same root context Δ_0 since there is no Ω operator involved. We are checking for the existence of at least one message that satisfies the property. For a trace σ = `m m m`, where all messages match signature α, the property holds if any of these messages meet the condition. If, instead, we wanted all workers to return a positive result, the property would have to be:

$$\varphi_5 = \Omega(\texttt{send}_{\_\rightarrow\texttt{server}}\{\texttt{ok, Res}\} : \texttt{Res} \geqslant 0)$$

In this case, if even one message fails to satisfy the property, the entire property is violated. This is because each message is evaluated in its own context, so every message with the given signature is treated as a separate context for verification.

5 Monitor synthesis and instrumentation

The core of RV is the generation of monitors from high-level specification languages [6]. In this section, we present the instrumentation and orchestration within the system to couple it with our monitors. Monitors are active entities that analyse a system and produce a verdict based on the system's execution trace. The framework is illustrated in Figure 3.

Our approach uses outline monitoring for actor-based systems, which is less intrusive compared to some other methods [11]. Monitors are implemented as separate entities, enabling us to observe system behaviour without changing the existing code, unless that code diverges from standard actor-based practices. In such cases, we may need to perform semi-manual or automatic code injection. While this outline method introduces more overhead than inline monitoring, it

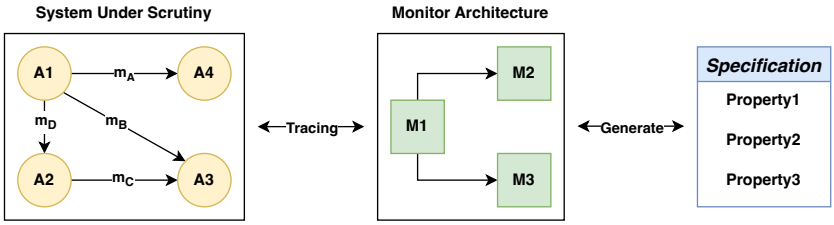

Fig. 3: Instrumentation framework

becomes essential in scenarios where modifying or accessing the system code is not possible. In actor-based models, outline monitoring is a natural fit, as monitors can function as independent actors that communicate externally rather than being embedded within the already existing code. By leveraging the built-in tracing features of Erlang [4], it is possible to capture all interactions in the system, including message sending, message reception, and abrupt failures. To achieve this, the monitor compiled from WALTZ is responsible for tracking all specified interactions and verifying whether the system satisfies the defined properties.

Thanks to Erlang's inherent scalability and its "let it crash" philosophy [4], we can effectively manage sub-monitor failures and ensure reliable property verification. By maintaining links between the main monitor and all sub-monitors, violations of nested properties, such as those involving $\Omega(...\Omega)$, can be accurately detected. The language's architecture naturally supports this structure, enabling robust monitoring guided by the language's own fault-tolerant design.

The correctness of the generated monitors can be evaluated through inspection of the generated code, even in the absence of formal correctness proofs. The direct correspondence between the property specifications and the monitor implementation makes it straightforward to verify that the generated code accurately implements the specified property logic.

The monitor compilation process is still in development, but the case studies presented demonstrate the practical value and expressive power of WALTZ. These examples illustrate how the language integrates seamlessly with actor-based systems, allowing for precise, scalable, and non-intrusive monitoring of complex interaction patterns. This showcases its potential for real-world applications. So far, progress includes a working compiler that translates the specification language into an Erlang monitor. Further work is currently underway in instrumentation and context generation. For more information on the project, the reader can visit `Actorchestra`.

6 Related work

The field of runtime verification has produced numerous approaches for monitoring system behaviour against formal specifications, each addressing different aspects of the verification challenge with varying degrees of success.

`MonPoly` [8] is a runtime verification tool that analyses traces and properties defined in a fragment of metric first-order temporal logic (MFOTL) [7]. A key strength of `MonPoly` is that monitors return satisfying variable assignments, not just boolean verdicts, providing deeper diagnostic insights. A later version supports full MFOTL, but with significant performance costs [19]. However, `MonPoly` reads traces from pre-processed log files in a specific format, limiting its integration with live systems.

`detectEr` [5] is a runtime verification tool designed specifically for Erlang, using its native tracing mechanisms to monitor properties defined in a monitorable fragment of μHML [1,2]. Unlike tools that process pre-generated logs, like `MonPoly`, `detectEr` operates on live system traces, enabling real-time verification and verdict generation. While it shares a similar goal with our approach, it faces key limitations. First, its specification language treats each actor as an isolated black box, restricting the ability to express properties that span multiple actors, preventing the relating of variables across actor interactions. Second, in `detectEr`, users must explicitly declare which messages to ignore. While feasible for simple systems, this becomes error-prone and cumbersome in large systems with many concurrent interactions.

7 Conclusion and future work

WALTZ offers a specification language and monitoring framework for actor-based systems, introducing the concept of context-aware properties to track and relate message interactions across actors. This allows for expressive intra-actor verification, supports implicit filtering, and enables rich reasoning. By unifying high expressiveness with scalable monitoring, WALTZ provides a strong foundation for runtime verification in concurrent and distributed environments.

To strengthen the practical and theoretical impact of our language, several key extensions are planned: (1) establishing formal correctness proofs for generated monitors would ensure reliability over the monitor implementations; (2) exploring the notion of blame assignment. Stating what happened wrong is more powerful than simply stating that something went wrong [20]. WALTZ's structure already supports detailed property definitions, allowing users to isolate where violations occur. This opens the door to integrating blame assignment directly into the language, enabling monitors to return informative verdicts rather than binary outcomes; (3) investigating whether it is possible to monitor systems that do not follow OTP conventions without requiring code injection. OTP provides a standard structure for building actor-based systems in Erlang, including mechanisms like unique references in message exchanges. These built-in features are crucial for creating and associating contexts, making monitoring significantly more feasible. In contrast, non-OTP systems lack such consistency, posing greater challenges for external monitoring; (4) assessing the non-monitorability [9] aspects of runtime verification. Since not all properties can be monitored at runtime, exploring these limits could provide deeper insights into the expressiveness and capabilities of WALTZ.

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in monitorability: from branching to linear time and back again **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290365
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: An operational guide to monitorability. In: Ölveczky, P.C., Salaün, G. (eds.) Software Engineering and Formal Methods. pp. 433–453. Springer International Publishing, Cham (2019)
3. Agha, G.A., Thati, P., Ziaei, R., Bowman, H., Derrick, J.: Actors: a model for reasoning about open distributed systems. Formal methods for distributed processing: a survey of object-oriented approaches pp. 155–176 (2001)
4. Armstrong, J.: Programming Erlang : Software for a Concurrent World. The Pragmatic Bookshelf, Raleigh, NC : (2013), http://digital.casalini.it/9781680504330
5. Attard, D.P., Cassar, I., Francalanza, A., Aceto, L., Ingolfsdottir, A.: A runtime monitoring tool for actor-based systems. Behavioural Types: from Theory to Tools pp. 49–76 (2017)
6. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to Runtime Verification, pp. 1–33. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
7. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2) (May 2015). https://doi.org/10.1145/2699444
8. Basin, D.A., Klaedtke, F., Zalinescu, E.: The monpoly monitoring tool. RV-CuBES **3**, 19–28 (2017)
9. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for ltl and tltl. ACM Trans. Softw. Eng. Methodol. **20**(4) (Sep 2011). https://doi.org/10.1145/2000799.2000800
10. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
11. Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A.: A survey of runtime monitoring instrumentation techniques. Electronic Proceedings in Theoretical Computer Science **254**, 15–28 (Aug 2017). https://doi.org/10.4204/eptcs.254.2
12. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. **28**(4), 626–643 (Dec 1996). https://doi.org/10.1145/242223.242257
13. Clarke, E., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order reduction. International Journal on Software Tools for Technology Transfer **2**, 279–287 (01 1999). https://doi.org/10.1007/s100090050035
14. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. **23**(2), 255–284 (Apr 2021). https://doi.org/10.1007/s10009-021-00609-z
15. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? International Journal on Software Tools for Technology Transfer **14** (06 2011). https://doi.org/10.1007/s10009-011-0196-8
16. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009). https://doi.org/https://doi.org/10.1016/j.jlap.2008.08.004, the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07)

17. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley Publishing, 3rd edn. (2011)
18. Pnueli, A., Zaks, A.: Psl model checking and run-time verification via testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 573–586. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
19. Schneider, J., Basin, D., Krstić, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification. pp. 310–328. Springer International Publishing, Cham (2019)
20. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering **42**(8), 707–740 (2016). https://doi.org/10.1109/TSE.2016.2521368