DSRV: Reconfigurable Stream Runtime Verification With Context Transfers *

Morten Haahr Kristensen[1 [0009−0008−8467−7567]],
Thomas Wright[1 [0000−0001−8035−0884]], Mirgita Frasheri[1 [0000−0001−7852−4582]],
Cláudio Gomes[1 [0000−0003−2692−9742]], Peter Gorm Larsen[1 [0000−0002−4589−1500]],
and Lukas Esterle[1 [0000−0002−0248−1552]]

Department of Electrical and Computer Engineering, Aarhus University, Denmark
{mhk, thomas.wright, mirgita.frasheri, claudio.gomes, pgl,
lukas.esterle}@ece.au.dk

Abstract. As software systems adapt, receive updates, or acquire new requirements, their runtime monitors must often evolve as well. However, many stream-based runtime verification approaches require monitors to be restarted when their specifications change, causing internal monitor state and historical input data to be lost. To address this limitation, we present DSRV, a stream-based runtime verification language that supports dynamic monitor reconfiguration at runtime while preserving relevant monitor state across specification changes. We evaluate DSRV using a distributed leader-election case study featuring changing monitor roles and simulated failure scenarios. The results show that DSRV can update monitors without restarting them or losing historical context, while introducing minimal runtime overhead.

Keywords: Runtime Monitoring · Dynamic Software Updating · Stream-based Monitoring

1 Introduction

Runtime verification is well suited for providing lightweight assurance for systems at execution time, but it typically assumes that both the monitored system and the specification remain fixed while monitoring is in progress. In distributed and adaptive systems, this assumption can be too restrictive. Correct behaviour may depend on the current configuration of the system, including the roles of components, the communication structure, and the rules that govern coordination. As the system evolves during execution, for example through reconfiguration, dynamic software updates, policy changes, or autonomous adaptation, the assumptions under which a monitor was instantiated may no longer hold. A static monitor can then become misaligned with the behaviour it is intended

* The work presented here is supported by the RoboSAPIENS project funded by the European Commission's Horizon Europe programme under grant agreement number 101133807.

to check, even if the system itself continues to operate correctly under its new configuration. Simply restarting the monitor from an updated specification is often unsatisfactory, since this may discard internal monitoring state and thereby compromise the correctness of subsequent verdicts. Such scenarios are common and include autonomous vehicles transitioning between autonomous, semi-autonomous, and manual modes, each with fundamentally different safety obligations (lane-keeping, speed constraints, collision avoidance responsibilities) [17]; industrial robotic arms that switch between assembly, transport, and inspection tasks, requiring monitoring of different properties (force limits, positioning precision, vision thresholds) [20]; autonomous surveillance networks in which nodes take on various roles (observer or follower) [11]; or body sensor networks in which wearable nodes switch between normal monitoring, high-alert, and power-saving modes, checking different timing and sampling properties per mode [7].

In our previous work, published under the name DynSRV [18], we introduced Dynamically Updated Properties (DUPs), which allow selected expressions within a specification to adapt at runtime. While DUPs are useful when adaptation is local and anticipated, some dynamic scenarios require a more general form of adaptation, where specifications themselves may need to be replaced at runtime, with corresponding changes to inputs, outputs, and properties. This shift from targeted property updates to specification-level reconfiguration introduces the challenge of replacing the specification of a running monitor without restarting from scratch and without compromising the correctness of its verdicts.

To address this challenge, we introduce DSRV (pronounced "Deserve"), a stream runtime verification language with two key features. First, DSRV supports specification-level updates at runtime, allowing inputs, outputs, and properties to evolve as the monitored system reconfigures. This addresses scenarios where different system configurations require fundamentally different monitoring specifications, such as role changes in distributed protocols. Second, we present a context transfer mechanism based on dependency graphs that retains the minimal portion of the monitoring trace required to preserve solvability under the new specification. This enables seamless reconfiguration while ensuring that verdicts remain correct with respect to the new specification. We demonstrate these contributions through a running example of a RAFT-based leader election. Consensus algorithms such as RAFT are foundational to reliable distributed systems and are widely deployed in production environments; ensuring their correct operation is therefore a significant practical problem. Leader election is particularly interesting as an example since it is the most dynamic phase of the protocol: nodes transition through multiple roles, follower, candidate, and leader, each requiring distinct monitoring properties. This role-driven dynamism is representative of the broader class of systems in which monitoring requirements evolve alongside the system's configuration and align with the examples given earlier.

The remainder of the paper is structured as follows. Section 2 motivates the need for specification-level monitor reconfiguration through the RAFT example. Section 3 discusses related work. Section 4 presents the DSRV language and its reconfiguration model, together with the DSRV specifications we use for

monitoring RAFT. Section 5 introduces our context transfer mechanism and proves that it never makes expressions less solvable than reconfiguration from an empty context. Section 6 evaluates DSRV through the RAFT example and runtime benchmarks. Finally, Section 7 concludes the paper.

2 Running example and motivation

Runtime verification typically assumes that the behaviour being monitored is adequately captured by a single, static specification. However, in distributed systems with role-based architectures, components often transition between fundamentally different operational modes, each with distinct behavioural requirements, communication patterns, and correctness criteria. We argue that forcing such systems into a unified monitoring specification imposes unnecessary computational overhead, complicates maintenance, and violates information hiding principles that may be essential for security or modularity.

2.1 Running example: RAFT leader election.

We use the leader election algorithm from RAFT [21] as a running example. In RAFT, each node assumes one of three roles: a *Leader* coordinates the cluster via periodic heartbeats; a *Candidate* requests votes from peers to win election; and a *Follower* passively receives heartbeats and votes in elections. Nodes transition between roles dynamically during normal operation: a follower becomes a candidate when the leader fails; a candidate that wins the majority of votes becomes the new leader; and a leader steps down upon discovering a node with a higher term (a larger epoch number indicating a newer election). Each transition changes what constitutes correct behaviour for that node. Our goal is to monitor individual nodes to verify that they correctly follow the RAFT protocol according to their current role. Fig. 1 illustrates this scenario: after the leader fails, a new election takes place and the winning node must switch from follower-specific to leader-specific monitoring properties.

2.2 The case for reconfiguration.

A traditional approach would embed all three RAFT roles into a single specification, using conditionals to switch between role-specific properties. While this works, it forces every monitor to maintain streams for all roles simultaneously, despite only one role being active. In our RAFT example using a single specification, a follower monitor would have to subscribe to heartbeat responses (leader-only), wasting resources on messages that can never yield meaningful verdicts for its current role.

Production systems also evolve: protocol rules change, new roles are added, and monitoring logic must be updated accordingly. Without reconfiguration, specification changes require redeploying them to every monitor, which entails restarting all monitors and discarding accumulated state. For example, changing

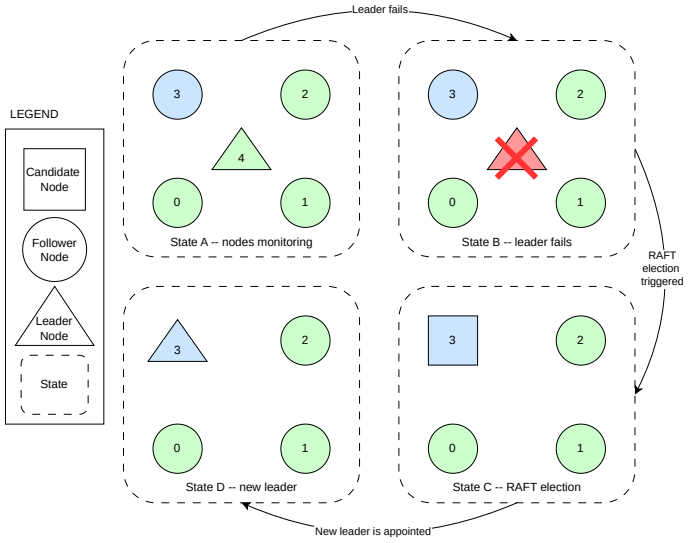


Fig. 1. Four states of a RAFT cluster (A–D). The blue node is under monitoring, green nodes are healthy, red nodes have failed. The monitored node transitions from follower (A) to candidate (C) to leader (D) as the election proceeds, adopting the corresponding specification at each step.

the number of votes a candidate must receive to become the new leader only affects candidate monitors, yet without reconfiguration every monitor in the system would need to restart. With DSRV, candidate nodes can be reconfigured independently, even while they remain candidates, allowing seamless updates.

The challenge is that reconfiguring a running monitor without losing state is non-trivial. Simply restarting with a new specification discards all accumulated context. A core contribution of DSRV is a mechanism for performing such context-aware reconfiguration, ensuring that monitors remain both efficient and correct across dynamic role changes.

3 Related work

To the best of our knowledge, no existing approach provides a framework for replacing full runtime verification specifications at runtime while preserving monitor state. The problem does however overlap with more general works in dynamic software updating, dynamic features in runtime verification languages, and a smaller number of works that have directly investigated dynamic updating of runtime monitors.

Dynamic Software Updating and Self-Adaptation Dynamic reconfiguration of monitors can be seen as a special case of dynamic software updating [1], which

attempts to update deployed software systems without restarting them. A number of works in this area have focused on the transfer of software state during software updates, including [23, 25]. These methods typically rely on controlled execution methods such as virtual machines to stop and replace running software atomically, whilst state mappings or transformations are used to map state from the old version to the new. ActivFORMS showed that dynamic software updating can also provide a basis for self-adaptive systems, by allowing a self-adaptive controller to update individual system components [16]. In this setting persistent state is of key importance as it enables seamless refinement of a system's behaviour whilst it is carrying out an ongoing task.

Dynamic Runtime Verification DSRV fits within a long tradition of existing formal-language approaches to runtime verification, and is most heavily influenced by stream-based languages such as Lola [10] and TeSSLa [8]. Whilst the vast majority of existing runtime verification languages focus on monitoring static properties, a number of languages and formalisms such as Lola 2.0 [12], Eagle [4], Quantified Event Automata [3], and LC_v [14] support dynamic spawning or modification of monitors based on the received stream data. HLola [22] extends this to *retrospective parameterisation*, enabling dynamic spawning of new monitors with access to historical stream data. Alternatively, Baumeister, Finkbeiner, and Scheerer [5] proposed a dynamic approach to a monitor's handling of input data by scheduling each input's sampling frequency based on an annotated specification.

Dynamic Updates of Runtime Monitors Apart from our previous work [18], only a handful of works have directly approached the problem of dynamically adapting runtime monitors without restarts. Carwehl et al. [7] propose dynamically adapting monitors to changing requirements without restarting, using predefined property adaptation patterns. Li et al. [19] propose a runtime verification framework for self-adaptive systems with runtime adaptations based on graph transformations over observer automata. This enables modifications such as changing timing constraints or event structures. However, none of these approaches tackles the problem of preserving monitor state across updates, which is the main contribution of our approach.

4 DSRV language

DSRV is an asynchronous discrete-time stream runtime verification language. A monitor evaluates a specification by transforming a set of input streams $I = \{p_1, \ldots, p_n\}$ into a set of output streams $O = \{o_1, \ldots, o_m\}$, where outputs represent verdicts or intermediate computations over the observed inputs. Orthogonally to the data streams, DSRV designates a reconfiguration stream $r \notin I \cup O$, enabling reconfigurations of the specification at runtime (detailed in Section 4.1).

Streams. Let $\mathbb{T} \triangleq \mathbb{N}$ be the discrete time domain. A *stream* $s = (s_0, s_1, \ldots)$ is a sequence of typed values s_i drawn from a data domain $\mathbb{D}$, where each index $i \in \mathbb{T}$ corresponds to a discrete time step. The data domains $\mathbb{D}$ include booleans $\mathbb{B}$, integers $\mathbb{Z}$, floating-point numbers $\mathbb{F}$, and strings $\mathbb{S}$. For completeness, the data domain is also extended with a recursive expression domain for representing stream expressions as values; this is only needed for Dynamically Updated Properties, which are outside the scope of this paper, and we refer to [18] for the details. In addition to values from $\mathbb{D}$, a stream position may hold one of two special values: $\nexists$ (*no val*), indicating that no event was received on the stream at that time index; or $\bot$ (*deferred*), indicating that a value cannot yet be computed due to insufficient historical context (e.g., an evaluation of an expression referring further back than the start of the trace).

Definition 1 (Specification). *A specification Φ over a set of streams $V = I \uplus O$ (where $\uplus$ denotes disjoint union) consists of a set of equations*

$$o_1 = \phi_{o_1} \quad \ldots \quad o_m = \phi_{o_m},$$

where each ϕ_o is a stream expression with domain $\mathbb{D}$.

Stream expressions $\phi \in \mathbb{E}[\mathbb{D}]$ are defined recursively and include: constants $c \in \mathbb{D}$; stream variables $v \in V$; function applications $f(\phi_1, \ldots, \phi_k)$, lifting an arbitrary data-domain function $f : \mathbb{D}_1 \times \cdots \times \mathbb{D}_k \to \mathbb{D}$ pointwise to streams; temporal offset expressions[1] $v[i]$ for $v \in V$ and $i \in \mathbb{N}$, denoting the value of v at the i-th preceding evaluation step (defined below); and defaults $\mathbf{default}(\phi_1, \phi_2)$. Standard operators such as arithmetic, logical connectives, comparisons, and the conditional **if** ϕ_1 **then** ϕ_2 **else** ϕ_3 are supported as strict lifted functions: $f(\phi_1, \ldots, \phi_k)$ yields $\nexists$ if any argument is $\nexists$, otherwise $\bot$ if any argument is $\bot$, and otherwise the pointwise application of f. The $\mathbf{default}(\phi_1, \phi_2)$ operator is the sole exception, yielding ϕ_2 when ϕ_1 is $\bot$ and ϕ_1 otherwise, and is thus the only construct that eliminates $\bot$. Uniform strictness keeps logically equivalent encodings interchangeable, at the cost of requiring **default** wherever $\bot$ should not propagate: both the disjunction of *true* and $\bot$, and **if** *true* **then** 0 **else** $\bot$, evaluate to $\bot$. A specification is *well-defined* if it contains no zero-time dependency cycle (similarly to [10]), i.e., every dependency cycle is guarded by a temporal offset, which is necessary for the specification to be monitorable online. We refer to [18] for the formal denotational semantics of the underlying expression language, except for the $\nexists$ semantics, which the propagation rules above supply.

Asynchronous inputs. Synchronous SRV languages assume a global clock where every stream produces a value at every tick. DSRV relaxes this assumption: streams may be absent at a given time index, signalled by $\nexists$. This captures distributed settings where components observe at independent rates. The absence mechanism also makes it possible for the reconfiguration stream to be orthogonal, i.e., data streams simply carry $\nexists$ when a reconfiguration arrives. A time

[1] Unlike our previous work [18], temporal offsets apply only to stream variables, to simplify context transfers.

index at which one or more input streams carry a value is an *evaluation step*; at every other index the monitor performs no evaluation. We write $t_0 < t_1 < \cdots$ for the evaluation steps of a run. Absent values are resolved by *asynchronous lifting*, applied uniformly to every variable in V:

Definition 2 (Asynchronous Lifting). *The asynchronous lifting $\tilde{s}$ of a stream s applies zero-order hold semantics, retaining the most recently observed value whenever no event is present:*

$$\tilde{s}(i) \triangleq \begin{cases} \tilde{s}(i-1) & \text{if } s(i) = \nexists \text{ and } i > 0, \\ s(i) & \text{otherwise.} \end{cases} \tag{1}$$

At evaluation step t_n, the offset $v[i]$ thus denotes $\tilde{v}(t_{n-i})$, or $\bot$ if $n < i$; steps at which v is absent still advance the index and are read as the most recent value of v. Asynchronous lifting yields piecewise-constant signal semantics over a discrete clock, in the spirit of SRV languages such as TeSSLa [8] and RTLola [24], but without real-time timestamps. Consequently, DSRV encodes asynchrony entirely via $\nexists$ on a discrete clock, which simplifies language implementations, as they can be synchronous internally, with the lifting layer holding the last value whenever no event arrives. Gorostiaga, Danielsson, and Sánchez [13] use a comparable encoding, but additionally provide operators for jumping between encoded event positions.

4.1 Reconfiguration stream

The reconfiguration stream $r \notin I \cup O$ is orthogonal to the data streams: it carries specifications as values and triggers transitions between specifications at runtime, but cannot be referenced within stream expressions. Unlike data streams, whose values are drawn from a data domain $\mathbb{D}$, r carries values from $\mathcal{S} \subseteq \mathbb{S}$, where $\mathcal{S}$ denotes the set of well-formed specifications, encoded as strings.

Context. To preserve monitoring history across specification changes, DSRV maintains a *context* κ that records recent evaluation results for each stream variable.

Definition 3 (Context). *A context κ maps each variable $v \in V$ to a sequence of its past recorded values:*

$$\kappa(v) = (v_0, v_1, \ldots, v_k),$$

where v_0 is the most recently recorded value of v, v_1 the one before, and so on, up to a bound k determined by the maximum temporal offset applied to v in the specification. A value is recorded for every variable at every evaluation step, irrespective of whether that variable carried an event.

On initial startup the context is empty. At reconfiguration, when the monitor transitions from Φ to Φ', the context κ is transferred into κ' according to a

min-transfer principle: for each variable v present in both specifications, a prefix of $\kappa(v)$ is preserved, with the length of the prefix determined by the overlap between the dependency bounds of Φ and Φ'; for new variables $\kappa'(v) = ()$. This ensures that no more history is transferred than the new specification can use, while never discarding history that would be useful. The full definition and correctness argument are given in Section 5. The new monitor treats the transferred entries in κ' as if they had been observed under Φ', allowing temporal offsets in Φ' to resolve against pre-reconfiguration data. After reconfiguration, evaluation proceeds normally under Φ' using the history.

Monitor configuration. With the context in place, a monitor configuration captures the full evaluation state:

Definition 4 (Monitor Configuration). *A monitor configuration at time index $i \in \mathbb{T}$ is a tuple $C = (\Phi, \kappa)$ where:*

- *Φ is the active specification over $I \uplus O$*
- *κ is the context recording the evaluation history of each variable in V*

Orthogonality invariant. DSRV enforces a strict separation between data events and reconfiguration events, i.e., a reconfiguration cannot coincide with data observations on any input stream. Formally, data events and reconfiguration events are mutually exclusive:

$$\forall i \in \mathbb{T}.\ r(i) \neq \nexists \implies \forall p \in I.\ p(i) = \nexists.$$

A time index i with $r(i) \neq \nexists$ is called a *reconfiguration step*. At a reconfiguration step, the monitor produces no output. Since no data events arrive, there is nothing to evaluate, and the sole effect of the step is to reconfigure the monitor. This invariant eliminates the ambiguity of deciding whether data arriving alongside a reconfiguration should be evaluated under the old specification Φ or the new Φ'.

Reconfiguration transitions. A reconfiguration event $r(i) = \varphi$ at time i triggers a transition between monitor configurations:

Definition 5 (Reconfiguration). *Let $C = (\Phi, \kappa)$ be the monitor configuration at time $i - 1$. A reconfiguration event $r(i) = \varphi$ encoding specification Φ' induces the transition*

$$C \xrightarrow{r(i)=\varphi} C' = (\Phi', \kappa'),$$

where κ' is the context transferred from κ according to Section 5. The new specification Φ' governs evaluation from time $i + 1$ onward.

A naive approach to reconfiguration would preserve the full evaluation history for every stream, ensuring that no information is lost. This is infeasible in practice because histories grow without bound. At the other extreme, discarding all history on reconfiguration is safe but forces the monitor to wait for new steps

before temporal offsets become resolvable, even if the reconfiguration only introduces a small change, such as adding a stream or modifying a single property. We address this in Section 5 by using the dependency structure of the old and new specifications to determine how much history to preserve. This is bounded by what the old specification already required, and we prove that it never makes expressions less solvable than reconfiguration from an empty context.

4.2 Example specifications

Below, we present DSRV specifications for monitoring a single node in the RAFT leader election example[2]. Each specification captures correctness properties for one of the three roles, and observes only the streams relevant to that role. Without reconfiguration, these properties can be combined into a single combined specification which selects the active properties via the node's current role

```
if role == "leader" then
  // Leader spec
else if role == "candidate" then
  // Candidate spec
else
  // Follower spec
```

or alternatively, with reconfiguration we can switch to the correct specification for the current role at runtime.

For the leader, the specification has a single input stream carrying heartbeat messages with fields `hb_term` and `hb_receiver`. Verdict `v` monitors that consecutive heartbeats target distinct nodes during the same election term, ensuring some fairness in leader–follower communication.

```
v = default(hb_term == hb_term[1], true) =>
    !default(hb_receiver == hb_receiver[1], false)
```

Temporal offsets `[1]` read prior values of `hb_term` and `hb_receiver`. `default(x, v)` yields the value `v` when `x` is $\bot$, which happens on the first step after a role transition.[3] The operators `!`, `=>`, and `||` denote lifted logical negation, implication, and disjunction.

For the candidate, the specification has two input streams: `node_voter_count` which carries the node's internal vote count, and a vote message stream carrying structured messages with fields `current_term`, `msg_id`, and `vote_granted`, from which all auxiliary streams are derived. We monitor that the internal vote count deviates by at most one from the number of successfully granted votes counted directly within the specification. This slack accounts for the two input streams arriving asynchronously. The verdict is defined as

[2] The full specifications can be found at https://github.com/mortenhaahr/DSRV_leader_election/tree/main/tests/fixtures/trustworthiness_checker/case_study.

[3] In the combined specification, the leader branch explicitly evaluates to `true` on the first step after promotion from candidate.

```
v = node_voter_count == counted_votes - 1 ||
    node_voter_count == counted_votes ||
    node_voter_count == counted_votes + 1
```

where `counted_votes` is an auxiliary stream that counts how many successfully granted votes the candidate has received. `counted_votes` is defined as,

```
counted_votes =
  if reset_vote then 1
  else if vote_granted && new_msg then
    default(counted_votes[1], 0) + 1
  else
    default(counted_votes[1], 0)
```

stating that if the vote count must be reset, we start counting from 1 (since the current message is a new vote grant); if we have a new message granting a vote, we increment the count; otherwise, we keep the count unchanged. The remaining auxiliary streams are defined as follows,

```
reset_vote = !default(current_term == current_term[1], false)
new_msg    = !default(msg_id == msg_id[1], false)
```

to determine when to reset the vote count due to a change in election term, and whether a genuinely new message has been received (based on the request response message ID).

For the follower, we monitor that each node votes for a candidate at most once per term. The specification uses the same vote message stream as the candidate example and reuses the auxiliary streams `new_msg` and `reset_vote` defined above. For simplicity, we exemplify this property for a single peer, whilst a more complete specification could replicate the same structure for every peer. The verdict is defined as

```
v = !is_new_vote || reset_vote ||
    !default(term_vote_granted[1], false)
```

where the auxiliary stream `is_new_vote` signals a new vote-grant message,

```
is_new_vote = new_msg && vote_granted
```

and `term_vote_granted` records whether the follower has already granted a vote to this peer during the current term

```
term_vote_granted =
  if reset_vote then is_new_vote
  else if is_new_vote then true
  else default(term_vote_granted[1], false)
```

5 Context transfer

Context transfer is the mechanism by which, during a reconfiguration transition from monitor configuration $C = (\Phi, \kappa)$ to $C' = (\Phi', \kappa')$, the new context κ'

inherits the evaluation histories of variables from κ. Stream runtime verification languages that target *efficiently monitorable* specifications [10] must already maintain, for each stream, a bounded history of past evaluations whose size is statically determined by the specification and independent of trace length. This is required because expressions containing temporal offsets such as $v[i]$ must look back i evaluation steps. DSRV reuses these existing bounded histories as the source of the new context κ' at reconfiguration, with the amount of history to preserve for each variable determined by the dependency structure of the old and new specifications.

5.1 Dependency graphs and dependency bounds

A DSRV specification induces a weighted dependency graph that encodes how output variables depend on other variables through temporal offsets.

Definition 6 (Dependency Graph). *The dependency graph of a specification Φ over $V = I \uplus O$ is the directed graph $G_\Phi = (V, E)$ where E contains an edge $v \to u$ whenever $u \in O$ and the defining expression of u references v with a temporal offset (i.e., u's definition contains $v[i]$ for some $i \in \mathbb{N}$).*

Only direct edges are needed because the context κ stores evaluated histories for *all* streams, including intermediate outputs.

For each edge $v \to u$, let $w(v, u) \in \mathbb{N}$ be the largest temporal offset used to reference v in the defining expression of u. The *dependency bound* of a variable $v \in V$ is

$$\ell_\Phi(v) \triangleq \max\{\, w(v, u) \mid v \to u \in E \,\},$$

with $\ell_\Phi(v) = 0$ if v has no outgoing edges. This captures how many past values of v must be retained to resolve all temporal offsets in Φ. The context $\kappa(v)$ stores $\ell_\Phi(v)$ past values of v; older entries are never accessed and can be discarded. Consequently, an online monitor can implement this context with space bounded by the specification, independent of the trace length, similarly to *efficiently monitorable* specifications [10].

5.2 The min-transfer principle

When reconfiguring from Φ to Φ', the monitor must decide how much of each variable's history to preserve. The answer follows from the dependency bounds of the two specifications.

Definition 7 (Context Transfer). *Let Φ and Φ' be specifications with contexts κ and κ', respectively. For $v \in V \cap V'$, let $\kappa(v)|_j$ denote the j most recent entries of $\kappa(v)$, or the entire history if fewer than j entries have been accumulated. The transferred context is defined by:*

$$\kappa'(v) = \begin{cases} \kappa(v)|_{\min(\ell_\Phi(v), \ell_{\Phi'}(v))} & \text{if } v \in V \cap V', \\ () & \text{otherwise.} \end{cases}$$

The rationale for the use of min is straightforward. The old context $\kappa(v)$ stores at most $\ell_{\Phi}(v)$ entries, while the new specification needs at most $\ell_{\Phi'}(v)$. Hence $\min(\ell_{\Phi}(v), \ell_{\Phi'}(v))$ is the ideal overlap, since no larger transfer is possible by definition, and no smaller transfer is desirable, since discarding available history that Φ' could use would only delay the evaluation of temporal offsets after reconfiguration. We formalise this intuition in Theorem 1.

Definition 8 (Solvable). *A stream expression ϕ is* solvable *at time t with context κ if its evaluation does not yield $\perp$. Since temporal offsets apply only to variables, ϕ is solvable precisely when, for every temporal offset $v[i]$ in ϕ, $|\kappa(v)| \geq i$ at time t.*

Example 1. Let Φ define $z = x[i]$ and Φ' define $z = x[j]$, with $i, j \in \mathbb{N}$. Assume x does not occur elsewhere in Φ or Φ'. Then $\ell_{\Phi}(x) = i$, $\ell_{\Phi'}(x) = j$, and $\ell_{\Phi}(z) = \ell_{\Phi'}(z) = 0$. If $i \geq j$, the j most recent entries of $\kappa(x)$ are transferred, hence $x[j]$ is solvable from the first step after reconfiguration. If $i < j$, only the i most recent entries are transferred. The new monitor must accumulate $j - i$ evaluation steps before $x[j]$ becomes solvable.

We now show that context transfer never makes expressions less solvable than they would be with an empty context.

Theorem 1 (Preservation of Solvability). *Let ϕ be a stream expression in Φ' and let t_m be the first evaluation step after reconfiguration. For every temporal offset $v[k]$ in ϕ, the number of additional evaluation steps after t_m required for $v[k]$ to become solvable under transferred context κ' is at most that required under empty context.*

Proof. Consider a temporal offset $v[k]$ appearing in ϕ. With empty context, $v[k]$ requires exactly k evaluation steps before it becomes solvable, since the context contains no prior entries. With transferred context, $\kappa'(v)$ contains the $\min(\ell_{\Phi}(v), \ell_{\Phi'}(v))$ most recent entries of $\kappa(v)$, or the entire history if fewer entries have been accumulated (Definition 7). Hence $v[k]$ requires only $\max(0, k - |\kappa'(v)|)$ additional evaluation steps, which, since $|\kappa'(v)| \geq 0$, is at most k. □

By Definition 8, ϕ is solvable precisely when every offset in it is. Since the bound holds for each offset individually, it holds for their maximum, and therefore for the expression as a whole.

5.3 Specification compatibility across reconfiguration

It is worth emphasising that Theorem 1 concerns solvability only, as it guarantees that transferred context does not delay evaluation, but says nothing about whether the transferred values are compatible with the new specification. Without reconfiguration this question does not arise, since every run begins with $\kappa(v) = ()$ for all $v \in V$, which makes it simpler to assess whether a specification is correct than when $\kappa(v) \neq ()$. Under reconfiguration the specification author

must instead reason about what κ' contains, which raises challenges that do not occur in the static setting. Naming is one such challenge, as a variable v may keep the same name across reconfigurations while its intended meaning changes, and if both specifications refer to v with a temporal offset its history will still be carried over. Such mismatches can be avoided by using different names for semantically distinct variables, or by letting the specification author control which variables retain their history.

Another challenge is that some specifications produce correct verdicts only when evaluated from the start of a trace. A stream introduced by Φ' begins from $\kappa'(v) = ()$ and therefore behaves as though the trace began at the reconfiguration, so a stream counting how many times an event has occurred is correct for a fresh run but counts only from the point at which it was introduced, and violations depending on the earlier occurrences go undetected. The monitor cannot repair this, as recovering the missing values would require retaining the entire trace. Specifications can instead be designed to converge on the intended values, for example by guarding a stream with an explicit reset, as `counted_votes` is by `reset_vote` in Section 4. DSRV provides no language constructs for stating and checking such requirements, and supplying them remains an open research direction. Until then, we regard careful specification design, both for the initial specification and for its replacements, as the practical way to avoid these mismatches, and ensuring semantic compatibility remains the responsibility of the specification author, not the monitor.

6 Evaluation

In this section we evaluate our approach, first by applying reconfigurable monitoring to the RAFT leader election case study, and then by measuring the overhead of reconfiguration and the scaling properties of context transfers.

6.1 RAFT leader election

In this paper the RAFT protocol has been implemented in Python as a standalone simulation[4], focusing on the leader election component of RAFT (hence, omitting the log replication component). The simulation uses an internal simulated clock, models communication messages between different RAFT nodes, and simulates network latency for messages between nodes. The probabilistic aspects of RAFT, such as the node timeouts and the network latencies, are simulated using pseudorandom numbers based on a set simulation seed, allowing for reproducible simulations of the whole RAFT system.

We consider a test scenario RAFT in which node 4 becomes disconnected and is unable to send and receive messages between times 250ms and 550ms, showing the role of each of the nodes. We ran this scenario for 700ms with

[4] The source-code for the simulation and replication of the results can be found at https://github.com/mortenhaahr/DSRV_leader_election/.

simulated clock ticks of 5ms, a RAFT heartbeat interval of 15ms, and uniformly distributed message latencies between 10 and 30ms. An example run for this scenario shown in Fig. 2. We see that the leader, node 4, becomes disconnected from the rest of the nodes at 250ms causing a new leader (node 3) to first become a candidate and then become elected as leader. After time 550 the former leader node 4 rejoins the system and is demoted to a follower.

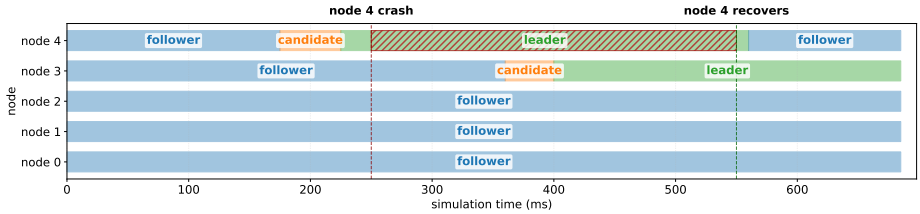


Fig. 2. An example simulation run from the RAFT scenarios, showing the role of each node over the simulation period.

We built a harness to enable online runtime monitoring of messages sent and received in the RAFT simulation via a DSRV runtime monitor. This harness forwards RAFT messages to the DSRV monitor and collects monitoring verdicts via MQTT publish/subscribe topics [2]. We used this to compare two different specifications for node 3:

1. We monitor a single combined DSRV specification which selects the follower, candidate, and leader specifications based on the node's current role;
2. We use a reconfigurable DSRV monitor which is reconfigured to monitor the current role of the node whenever it changes.

Monitoring results In both the combined specification and reconfigurable monitoring scenarios we observed that the DSRV properties held throughout the duration of the simulation for 100 different simulation seeds. This demonstrates that we are able to monitor the RAFT node in a reconfigurable manner throughout each of its roles, and the algorithm performed correctly under a variety of message timings. We also compared the number of messages received and sent from the DSRV monitor in Fig. 3. This shows a substantial variation of the number of messages sent/received for different simulation seeds, including a number of outlier runs which sent/received substantially more messages than average. We also saw a substantial reduction in the number of sent/received messages for the reconfigurable monitor, which was able to avoid subscribing to unnecessary input topics or computing and outputting unnecessary verdicts.

6.2 DSRV language benchmark

We evaluate the runtime cost of DSRV with the Criterion benchmarking framework [9] on six specifications of varying complexity: `simple_and` (Boolean

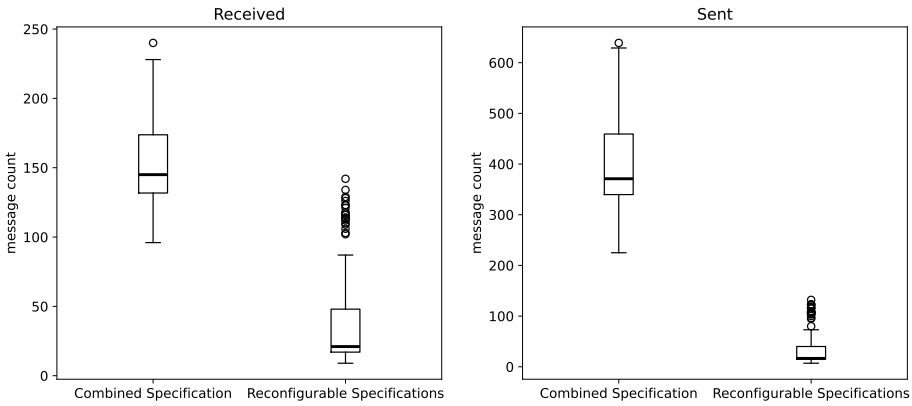


Fig. 3. Distribution of RAFT messages received and DSRV verdict messages sent over 100 simulated runs. Outliers (beyond 1.5 times inter-quartile range from the mean) are shown as circles.

conjunction), `rec_moving_average` (a recursively defined moving average), `maple_index` (a more involved property using nested temporal offsets), and three stream-indexing `sindex1`, `sindex10`, `sindex100` with different temporal offsets (1, 10, and 100 respectively). The latter family stresses the context κ along a single, controllable axis. Each specification is exercised on synthetic input traces of size $N \in \{1, 100, 1000, 2500, 5000, 7500, 10000\}$ messages, with the reconfiguration message frequency r varied across $\{0, 1, 5, 10, 20, 50, 100\}\%$. The monitor processes $N + r \cdot N$ stream events in total, where $r \cdot N$ is the number of events between two consecutive reconfigurations. For every (spec, N, r) configuration we run two variants: the full DSRV runtime (*CT on*), and a variant in which context transfer is disabled and each reconfiguration discards all monitor state (*CT off*). The CT-off variant isolates the cost of transferring κ from the cost of reconfiguration. Measurements use 10 Criterion samples per configuration, except for the benchmark used to calculate the CT-cost statistics in Fig. 5, in which 200 samples were used. All measurements were obtained on a Lenovo ThinkPad P14s Gen 6 (Intel Core Ultra 7 255H, 32 GB DDR5-5600 RAM) running Ubuntu 24.04 LTS.

From the experiment data with $r = 0$, we measured that DSRV processes input messages at 23–26 μs per message ($\sim$39 000–44 000 messages/second). Fig. 4 reports the reconfiguration overhead at $N = 10\,000$, normalised against the $r = 0$ baseline: at $r = 100\%$, a reconfiguration step takes roughly 24 times as long as an evaluation step. Fig. 5 compares CT-on and CT-off and finds no statistically significant difference in mean iteration time for any of the six specifications (Holm-adjusted Welch t-tests [15, 26], $\alpha = 0.05$); the largest mean difference is +1.1% (`rec_moving_average`). A 200-sample re-run for `sindex10` makes the CT effect detectable in some configurations, but the largest overhead is +2.4% (at

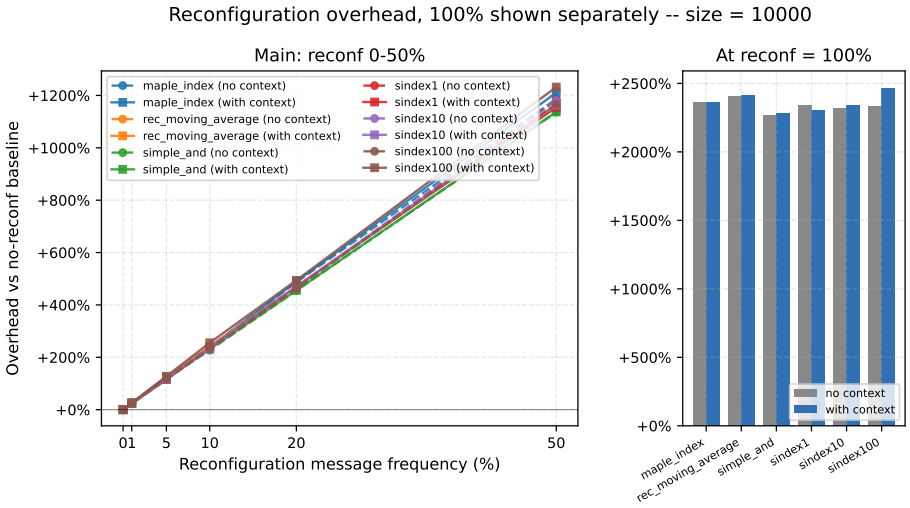


Fig. 4. Reconfiguration overhead at $N = 10\,000$, normalised against the $r = 0$ baseline. Left: $r \in [0, 50]\%$. Right: $r = 100\%$ shown separately to avoid compressing the left plot.

$N = 7\,500$). Across the tested specifications, maintaining the bounded context κ adds no significant runtime cost.

7 Conclusion

In this paper we introduced DSRV, a stream-based runtime verification language that supports dynamic reconfiguration of monitors at runtime. This enables changing the monitored specification via a dedicated reconfiguration stream. We showed how context can be transferred to the reconfigured monitor, preserving the solvability of stream expressions whilst transferring a minimal portion of the monitor history.

We demonstrated DSRV on a RAFT leader-election case study, monitoring nodes as they transition between follower, candidate, and leader roles. In the RAFT demonstration, the monitored properties held throughout each simulation run, confirming that DSRV monitors the system correctly across role changes. Results showed a significant reduction in the total number of messages sent and received by the runtime monitor compared to a monitor for a single combined specification without reconfiguration. In the language benchmark, steady-state evaluation sustains approximately 39 000–44 000 messages per second, whilst the additional cost of maintaining and transferring context was not statistically detectable above measurement noise. Specifically, processing each reconfiguration takes roughly 24 times as long as a standard monitor event, but this cost is amortized over long steady-state phases.

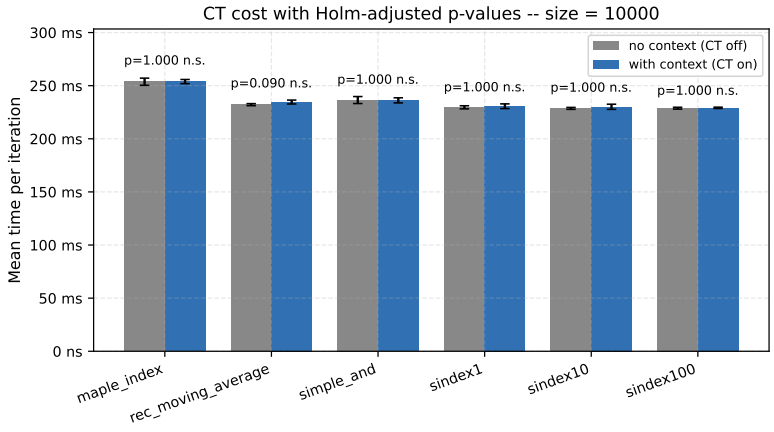


Fig. 5. Mean iteration time at $r = 0$ for input size $N = 10\,000$, with and without context transfer. Error bars show ± 1 standard deviation. The annotated p-values are Holm-adjusted Welch t-tests of the CT-on vs. CT-off means. None of the differences are significant at the $\alpha = 0.05$ level.

Looking forward, we envision future work applying DSRV in the context of self-adaptive systems. For example, we could extend the framework of Baxter et al. [6], which attempts to formally model architectural aspects of self-adaptive loops, with reconfigurable runtime monitors which check their behaviour at runtime. We hope to leverage runtime reconfiguration to update the specification of the self-adaptive system as it evolves to meet new requirements.

References

1. Ahmed, B.H., Lee, S.P., Su, M.T., Zakari, A.: Dynamic software updating: a systematic mapping study. IET Software **14**(5), 468–481 (2020). https://doi.org/10.1049/iet-sen.2019.0201
2. Banks, A., Gupta, R.: MQTT Version 3.1.1, (2014). https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. OASIS Standard. OASIS Open.
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods, pp. 68–84. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: Verification, Model Checking, and Abstract Interpretation, pp. 44–57 (2004). https://doi.org/10.1007/978-3-540-24622-0_5
5. Baumeister, J., Finkbeiner, B., Scheerer, F.: "Active Monitoring with RTLola: A Specification-Guided Scheduling Approach". In: Runtime Verification. Springer Nature Switzerland, Sept. 2025, pp. 181–201. https://doi.org/10.1007/978-3-032-05435-7_11.

6. Baxter, J., van Acker, B., Kristensen, M., Wright, T., Cavalcanti, A., Gomes, C.: "Formal Architectural Patterns for Adaptive Robotic Software". In: Fundamental Approaches to Software Engineering. Springer Nature Switzerland, 2025, pp. 145–165. ISBN: 9783031909009. https://doi.org/10.1007/978-3-031-90900-9_8.
7. Carwehl, M., Vogel, T., Rodrigues, G.N., Grunske, L.: Runtime Verification of Self-Adaptive Systems with Changing Requirements. In: IEEE/ACM Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 104–114 (2023). https://doi.org/10.1109/SEAMS59076.2023.00024
8. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: Temporal Stream-Based Specification Language. In: Massoni, T., Mousavi, M.R. (eds.) Formal Methods: Foundations and Applications, pp. 144–162. Springer International Publishing, Cham (2018)
9. Criterion, https://docs.rs/criterion/latest/criterion/. [Accessed 01-06-2026].
10. D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H., Mehrotra, S., Manna, Z.: LOLA: Runtime Monitoring of Synchronous Systems. In: Proceedings of the International Symposium on Temporal Representation and Reasoning, pp. 166–174 (2005). https://doi.org/10.1109/TIME.2005.26
11. Esterle, L., King, D.W.: Loosening Control—A Hybrid Approach to Controlling Heterogeneous Swarms. ACM Trans. Auton. Adapt. Syst. **16**(2) (2022). https://doi.org/10.1145/3502725
12. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A Stream-Based Specification Language for Network Monitoring. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification, pp. 152–168. Springer International Publishing, Cham (2016)
13. Gorostiaga, F., Danielsson, L.M., Sánchez, C.: Unifying the Time-Event Spectrum for Stream Runtime Verification. In: Runtime Verification, pp. 462–481 (2020). https://doi.org/10.1007/978-3-030-60508-7_26
14. Havelund, K., Peled, D.: Runtime Verification: From Propositional to First-Order Temporal Logic. In: Runtime Verification, pp. 90–112 (2018). https://doi.org/10.1007/978-3-030-03769-7_7
15. Holm, S.: A Simple Sequentially Rejective Multiple Test Procedure. Scandinavian Journal of Statistics **6**(2), 65–70 (1979). http://www.jstor.org/stable/4615733
16. Iftikhar, M.U., Weyns, D.: ActivFORMS: Active Formal Models for Self-Adaptation. In: Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 125–134 (2014). https://doi.org/10.1145/2593929.2593944
17. Koopman, P., Wagner, M.: Autonomous Vehicle Safety: An Interdisciplinary Challenge. IEEE Intelligent Transportation Systems Magazine **9**(1), 90–96 (2017). https://doi.org/10.1109/MITS.2016.2583491
18. Kristensen, M.H., Wright, T., Gomes, C., Esterle, L., Larsen, P.G.: DynSRV: Dynamically Updated Properties for Stream Runtime Verification. In: International Conference on Runtime Verification, pp. 101–119 (2025). https://doi.org/10.1007/978-3-032-05435-7_7
19. Li, Y., Duan, X., Xu, Y., Zhao, C.: Dynamic assessment approach for intelligent power distribution systems based on runtime verification with requirements updates. High-Confidence Computing **5**(2), 100255 (2025). https://doi.org/10.1016/j.hcc.2024.100255
20. Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: A survey. ACM Computing Surveys (CSUR) **52**(5), 1–41 (2019)

21. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX annual technical conference (USENIX ATC 14), pp. 305–319 (2014). https://doi.org/10.5555/2643634.2643666
22. Pedregal, P., Gorostiaga, F., Sánchez, C.: A Stream Runtime Verification Tool with Nested and Retroactive Parametrization. In: Runtime Verification, pp. 351–362 (2023). https://doi.org/10.1007/978-3-031-44267-4_19
23. Subramanian, S., Hicks, M., McKinley, K.S.: Dynamic Software Updates: A VM-centric Approach. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1–12 (2009). https://doi.org/10.1145/1542476.1542478
24. Torfah, H.: Stream-Based Monitors for Real-Time Properties. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification, pp. 91–110. Springer International Publishing, Cham (2019)
25. Walton, C., Krl, D., Gilmore, S.: An Abstract Machine for Module Replacement. In: Proceedings of the Workshop on Principles of Abstract Machines (1998)
26. Welch, B.L.: The Generalization of 'Student's' Problem When Several Different Population Variances Are Involved. Biometrika **34**(1-2), 28–35 (1947). https://doi.org/10.1093/biomet/34.1-2.28