DynSRV: Dynamically Updated Properties for Stream Runtime Verification*

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

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

Abstract. Systems that adapt to their environment or change based on new requirements pose challenges for runtime verification. Complexity is increased when the system needs to retain its internal state and continue monitoring while also updating properties or adding new ones during runtime. In this work, we propose DynSRV, a Stream Runtime Verification language that allows for dynamic updates of properties. A core benefit of this language is its capability to update properties at runtime without requiring a restart of the monitor, maintaining the internal state of the remaining properties. We formalise the semantics of our core primitives and demonstrate design patterns for allowing adaptations under certain constraints. Finally, we present an implementation of DynSRV and describe three memory strategies that balance memory usage and the ability to resolve dynamically added properties depending on historical data.

Keywords: Runtime Verification, Dynamic Properties, Stream Runtime Verification, Autonomous Systems, Dynamic Software Updating, Self-Adaptive Systems

1 Introduction

Motivation. How do we ensure continuous and accurate runtime monitoring when the system evolves during execution? If the system evolves in simple ways that can be captured in static Runtime Verification (RV) specifications then system evolution is not an issue. However, if significant behavioural changes are introduced by a human through Dynamic Software Updating (DSU) (see,

* The work presented here is supported by the RoboSAPIENS project funded by the European Commission's Horizon Europe programme under grant agreement number 101133807. In addition, the authors would like to thank Amalie Kaastrup-Hansen, Tobias Frejo Rasmussen, and Mikkel Kirkegaard for the fruitful discussions leading up to the paper.

e.g. [14]) or autonomously, then the RV specification must also be updated to ensure that the system is still being monitored correctly. Moreover, changes often entail that the system requirements have evolved [14], and if so, then the RV specification must also evolve to reflect these new requirements. An example of this is shown in [19], where a self-adaptive cloud-edge-end power distribution system requires the deployment of a state-machine based monitor that changes to reflect requirement changes in real-time, as the system reacts to evolving load demands, sensors failure, or maintenance events. Naturally, one possibility is restarting the monitor with an updated specification, but this is not always possible, as it involves loss of internal state potentially leading to incorrect verdicts [22].

Contribution 1. We propose and formally define DynSRV, a Stream Runtime Verification (SRV) language that allows monitors to be updated at runtime without requiring restarts or manual rewriting. Specifically, we introduce two primitives to DynSRV which enable expressing Dynamically Updated Properties (DUPs).

- `defer`(p) allows a RV property p to be specified at a later point in time, enabling exactly one dynamic update.
- `dynamic`(p) extends the concept of `defer`(p) by permitting continual updates, allowing the dynamic property to be modified multiple times throughout execution.

DUPs extend the concept of DSU to SRV by allowing specifications to evolve alongside the system without restarting. Unlike traditional DSU, which modifies the functional aspects of a running system, DUPs focus on changing the RV properties that the system is expected to satisfy.

Contribution 2. While DynSRV enables flexible adaptation, allowing arbitrary dynamic expressions within a monitor introduces challenges in reasoning about specification correctness. Thus, we propose *design patterns* for the specification of DUPs, enabling controlled adaptations, refinements, and demonstrating common adaptation patterns within DSU.

Contribution 3. We highlight the unique challenge presented by allowing adaptive SRV with DynSRV, such as ensuring consistency in monitoring results despite evolving specifications, managing historical data for updated properties, and developing performant interpreters that allow evaluating unforeseen properties.

2 Background & Related Work

We begin by linking the fields of DSU and Self-Adaptive Systems (SASs), which provided the motivation for this work, and we discuss how they relate to DUPs. We then recap the basic concepts of SRV before introducing existing works that express special cases of DUPs in the context of RV.

2.1 Dynamic Software Updating and Self-Adaptive Systems

DSU enables modifying running systems without stopping them, which is critical for applications like financial systems or web servers where downtime is costly. Key challenges include maintaining safety, supporting flexible updates, minimising overhead, and easing the developer's burden. Hicks, Moore, and Nettles [14] address these with a DSU system for C-like languages using type-safe dynamic patches and tools to aid patch creation and application.

SASs autonomously manage and adjust themselves to meet high-level goals [16]. Inspired by biological autonomic systems, they reduce manual intervention through capabilities like self-configuration, optimisation, healing, and modular architectural updates [27]. SASs use feedback loops and distributed components to monitor, analyze, plan, and execute changes in dynamic environments. Key challenges include goal specification, ensuring safety, and handling emergent behaviour.

DSU and SASs are interrelated: DSU enables runtime adaptation for SASs, while SASs frameworks can manage DSU to maintain stability during updates. Virtual machine-based DSU approaches, such as those presented in [25, 26, 15], inspired our monitor architecture, offering runtime control and transformation of system structures.

At design time, formal verification ensures DSU maintains safety and liveness properties. A foundational model by Bierman *et al.* [2] uses a λ-calculus with an update primitive to enable formal reasoning about dynamic updates. Despite extensive research (see surveys [23, 20, 28]), runtime updating of verification properties remains an underexplored area. The following sections address existing work on this topic.

Other works have developed temporal logic-based controller synthesis techniques to generate dynamic updates to a controller which satisfy temporal logic properties specifying both the expected new behaviour and the manner of the update. Nahabedian *et al.* [21] introduced an approach to synthesising controllers which satisfy new specifications as well as update strategies which ensures correct behaviour during the update; this corresponds to a *guided adaptation strategy* as proposed by Zhang and Cheng [29]. Finkbeiner, Klein, and Metzger [11] introduced LiveLTL, an extension of Linear Temporal Logic (LTL) for specifying desired behaviour before and after a dynamic update, and introduced a synthesis algorithm for synthesising updated controllers which meet the new requirements as well as any outstanding unsatified old requirements.

2.2 Dynamically Updated Properties

SRV is a lightweight RV approach that monitors systems producing continuous data streams. It processes input streams, i.e., sequences of event values, into verdict streams following a given specification.

LOLA [8] is a SRV language, inspired by LUSTRE and ESTEREL, supporting basic operations, conditionals, and time-offsets to enable temporal monitoring. LOLA uses a dependency graph to determine if a specification is "Efficiently

Monitorable", ensuring bounded memory usage. LOLA pioneered SRV and has influenced many subsequent languages, including TeSSLa [18].

Lola 2.0 improves dynamic RV through dynamic parametrization, enabling quantification over objects and monitor spawning independent of observed instances, and retroactive parametrization, allowing monitors to revisit past events during execution [22]. While the new monitors support parameterizing existing expressions, DynSRV allows dynamically providing any syntactically valid expression that references valid input streams.

Barringer et al. [1] propose Eagle, a general logic framework supporting recursive monitoring rules with fixpoint semantics. Eagle supports dynamic monitor generation and logics like LTL, Metric Temporal Logic (MTL), and Statistical Contracts.

First order logic quantification in dynamically created objects in RV was explored by Havelund and Peled [13] and Sokolsky *et al.* [24] with LC_v. LC_v uses first-order and attribute quantifiers to track dynamic entities (e.g., tasks, sensors). This relates to Allocational Temporal Logic (ATL) using history-dependent automata [9].

Actor-based runtime verification [7, 6, 4, 5] has previously been applied to self-adaptive systems, using independent monitor actors that observe and react to behaviour asynchronously.

The most relevant related work in terms of goals (but not methods) is by Carwehl *et al.* [3], who propose dynamically adapting monitors to changing requirements without restarting the monitor. Their monitors are synthesized as automata with error states based on structured English specifications translated into MTL, whereas we use stream-based properties. During execution, a Runtime Verifier checks for violations, and when requirements change, a Requirements Manager applies predefined Property Adaptation Patterns (e.g., updating a time guard or updating events). In contrast, we support arbitrary property expressions as long as they are syntactically valid and use existing input streams. While they argue that adhering to fixed patterns leads to safer adaptations and view fully dynamic RV as undesirable, we take a different stance, and demonstrate through Contribution 2 that we can address these valid concerns while prioritising expressiveness.

3 Specification Language

3.1 Motivational Example

To provide a motivational example (Fig. 1), we consider future production lines where different products are manufactured by autonomously moving robots. The robots move around in the production hall and utilise the different tools available in order to produce the desired items. The robot has an understanding of the production process and which tools to utilise for each product. However, while the robot and the production line are developed in parallel, the robot will only get knowledge of the final layout and the respective locations of the different tools

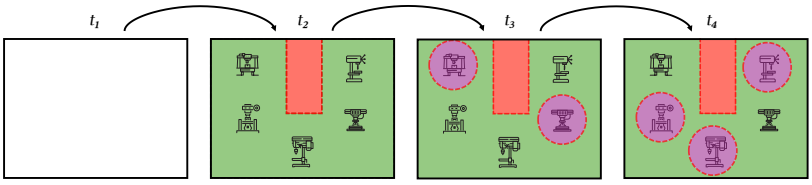


Fig. 1: Example of a production line at different time steps. First is the empty production hall, then the layout with the machines is added – a specific area is for robot maintenance (red square). Last two show which tools are allowed to be used during phases t_3 and t_4 (highlighted in purple).

upon completion of the production hall. Upon deployment, the robot will be given a product to manufacture, and it will start to move around the production hall. When the product is completed, the robot will receive a new product – potentially with a different requirement for the tools to be used. The robot will then have to adapt its plans, movement and overall behaviour to the new product. Finally, the robot is battery-operated and will need to recharge at certain intervals as well as undergo regular maintenance.

In this scenario, the robot uses the stream $l = \texttt{defer}(l_{in})$ for the layout of the production lines, respective locations of the different tools, and restricted areas, using `defer` since this configuration becomes immutable once it is made. At deployment time, $p = \texttt{dynamic}(p_{in})$ is used to change the rules determining which products the robot is allowed to manufacture as the production line evolves. Here, the new rules can use the information gathered from the layout stream. Finally, the verdict is available with $v = \texttt{update}(p_{init}, p)$ where an initial value of production rules is provided with `update`, which is detailed in Section 3.3.

This example highlights the need for DUPs in scenarios where the monitored system is subject to dynamic changes, and the monitoring properties must adapt accordingly. In addition, the specification can include static properties. For instance, the robot is also subject to regular maintenance and recharging within *at least* certain intervals (e.g., at least every 12 hours), requiring stateful properties to be monitored. As the stateful maintenance information would be lost if the monitor is restarted, a simple restart for each product update is not feasible. While this is only a simple example, the reader can imagine more complex scenarios with multiple robots and multiple products operating in parallel and potentially creating conflicts around resources and tools during execution.

3.2 Syntax

DynSRV defines monitors that transform a set of input streams $I = \{p_1, \ldots, p_n\}$ into a set of output streams $O = \{o_1, \ldots, o_m\}$. Each stream $s = (s_1, s_2, \ldots)$ is a sequence of typed values s_i in some domain $\mathbb{D}$. These domains $\mathbb{D}$ include booleans $\mathbb{B}$, integers $\mathbb{Z}$, floating point numbers $\mathbb{F}$, and, recursively, stream expressions in the DynSRV *expression domain* $\mathbb{E}[\mathbb{D}]$ which we will shortly define with values

in some domain $\mathbb{D}$. Streams may also take on a special value $\perp$ (pronounced *deferred*), denoting that no value was sent at the current time step.

A *specification* Φ over input streams and output streams $S = I \uplus O$ (where $\uplus$ denotes the *disjoint union*) consists of a set of equations

$$o_1 = \phi_{o_1} \quad \ldots \quad o_n = \phi_{o_n}$$

where the expressions ϕ_o are defined as stream expressions with output domain $\mathbb{D}$. Stream expressions $\phi \in \mathbb{E}[\mathbb{D}]$ are defined recursively to be a *basic expression*, a *DUP*, or a *DUP helper*. We elaborate on the precise semantics for DUPs in Section 3.3.

A *basic expression* is one of the following:

- a constant $\phi \triangleq c$ for $c \in \mathbb{D}$
- a stream variable $\phi \triangleq v$ for any $v \in S$
- a function application $\phi \triangleq f(\psi_1, \ldots, \psi_n)$
  which lifts an arbitrary data-domain function $f : \mathbb{D}_1 \times \ldots \times \mathbb{D}_n \to \mathbb{D}$
- a temporal index $\phi \triangleq \psi[-j]$ for $\psi \in \mathbb{E}[\mathbb{D}], j \in \mathbb{N}$
  referring to the value of ψ at j time units in the past
- a conditional $\phi \triangleq \texttt{if}\ \sigma\ \texttt{then}\ \psi_1\ \texttt{else}\ \psi_2$ for $\sigma \in \mathbb{E}[\mathbb{B}]$, $\psi_1, \psi_2 \in \mathbb{E}[\mathbb{D}]$

A *DUP* is one of the following:

- a defer $\phi \triangleq \texttt{defer}(\psi)$ for $\psi \in \mathbb{E}[\mathbb{D}]$
  referring to a dynamic property which is $\perp$ until the first point at which ψ becomes available and behaves like ψ subsequently
- a dynamic $\phi \triangleq \texttt{dynamic}(\psi)$ for $\psi \in \mathbb{E}[\mathbb{D}]$
  referring to a dynamic property which behaves like the most recent value of ψ or is $\perp$ if none has been sent

A *DUP helper* is one of the following:

- a default $\phi \triangleq \texttt{default}(\psi, c)$ for $\psi \in \mathbb{E}[\mathbb{D}], c \in \mathbb{D}$
  which uses the default value c if ψ is $\perp$
- a when $\phi \triangleq \texttt{when}(\psi)$ for $\psi \in \mathbb{E}[\mathbb{D}]$
  which is false until the first time ψ is not $\perp$ and true thereafter
- an update $\phi \triangleq \texttt{update}(\psi_1, \psi_2)$ for $\psi_1, \psi_2 \in \mathbb{E}[\mathbb{D}]$
  which is ψ_1 until the first time ψ_2 is not $\perp$ and ψ_2 thereafter

Standard data-domain operators such as addition, multiplication, logical conjunction, disjunction, and comparison operators are supported as functions f lifted to stream expressions. These operators propagate $\perp$ values such that e.g. $42 + \perp = \perp$.

Furthermore, we define a specification to be *well-defined*, if it has no zero-time cycle of dependencies (similarly to [8]), that is, if any dependency cycle is guarded by a time index. This restriction is necessary for specifications to be monitorable.

3.3 Semantics of DUPs

In this section we define a mathematical semantics for DynSRV specifications Φ. We note that this follows a similar approach to the semantics of TeSSLa [18] and LOLA [8], whilst introducing novel definitions to handle DUPs.

First, we need to formalise the notion of streams, used for specification input and output. Streams s range over the time domain $\mathbb{T} \triangleq \mathbb{N}$ of natural numbers, and assign to each time point $t \in \mathbb{T}$ a stream value $s(t)$ in an appropriate data domain $\mathbb{D}$. We also need these to handle both deferred data $\bot$ (for dynamic properties) as well as *partiality*, which uses the special value ? to represent stream values which have not yet been computed.

Definition 1 (Stream). *A* partial stream *(or simply,* stream*) is a function* $s : \mathbb{T} \to \mathbb{D} \cup \{?, \bot\}$ *such that* $s(i) = ?$ *implies that for all* $j > i$ *we must have* $s(j) = ?$*. We denote the set of streams by* $\textsc{Stream} = [\mathbb{V} \to \mathbb{D} \cup \{?, \bot\}]$.

Additionally, we call a partial stream *total* if $\forall i \in \mathbb{T} : s(i) \neq ?$.

We define the input namespace $\text{in}(\Phi)$ consisting of the set of input variables, the output namespace $\text{out}(\Phi)$ consists of the set of output variables, and we define $\text{vars}(\Phi) = \text{in}(\Phi) \cup \text{out}(\Phi)$. We also define $\text{vars}(\phi)$, for any stream expression ϕ to be the set of all stream variables appearing in ϕ, and define $\mathbb{V}$ to be the set of all variable names[1]. This allows us to introduce *contexts*, representing an assignment of partial streams to some stream variables v in the set of all stream variables $\mathbb{V}$.

Definition 2 (Context). *A context is a partial function* $C : \mathbb{V} \rightharpoonup \textsc{Stream}$*. We denote the set of all such partial functions as*

$$\textsc{Context} \triangleq [\mathbb{V} \rightharpoonup \mathbb{T} \to \mathbb{D} \cup \{?, \bot\}] = [\mathbb{V} \rightharpoonup \textsc{Stream}].$$

That is, within a given context, for a stream variable $v \in \mathbb{V}$ in the domain of stream variables for which it is defined, we have a stream for this stream variable $C(v) : \mathbb{T} \to \mathbb{D} \cup \{?, \bot\}$. In particular, the inputs to a specification Φ can be provided via an *input context* C_{in} such that $\text{dom}(C_{\text{in}}) = \text{in}(\Phi)$.

We also define the *refinement* partial order on data values by setting $u \sqsubseteq v$ iff $v = ?$ implies $u = ?$. This extends elementwise to a partial order $\sqsubseteq$ on streams, and on contexts sharing the same domain.

Using this, we define the semantics of a specification Φ as the least fixed-point of a *single-step semantics*, which expands one recursive step of the stream equations, using refinement to gradually build streams covering the whole time domain.

Definition 3. *We define the single-step semantics for a specification* Φ *to be the function* $[\![\Phi]\!]_{\text{I}} : \textsc{Context} \to \textsc{Context} \to \textsc{Context}$ *defined such that*

$$[\![\Phi]\!]_1(C)(D)(v) = [\![\phi_v]\!]_1(C)(D \uplus C)$$

[1] To be concrete, we can set $\mathbb{V} = \mathbb{N}$ for countably many numerically-indexed variables.

for each $v \in \text{vars}(\phi)$, whilst the denotation function for a well-defined specification Φ given initial context $C = C_{\text{in}}$ to be the function $[\![\Phi]\!] : \textsc{Context} \to \textsc{Context}$ defined as the least-fixed point:

$$[\![\Phi]\!](C) = \mu D.\, [\![\Phi]\!]_1(C)(D).$$

under the refinement order $\sqsubseteq$.

We also define the shorthand $[\![\psi]\!](C) \triangleq [\![\Psi]\!](C)$ for the semantics of ϕ within the specification $\Psi \triangleq v = \psi$ where v is any fresh variable name.

The fixed-point in the above definition exists and is unique for well-defined specifications Φ by Kleene's fixed-point theorem since the definitions of the single-step semantics for individual operators – which we will give shortly – are monotone in the refinement order $\sqsubseteq$, and hence so is $[\![\Phi]\!]_1(C)$.

This depends on the single-step semantics for individual operators, which we define as follows for basic operators,

$$\begin{aligned}
[\![c]\!]_1(C)(D)(i) &\triangleq c \\
[\![v]\!]_1(C)(D)(i) &\triangleq D(v)(i) \\
[\![f(\psi_1,\ldots,\psi_k)]\!]_1(C)(D)(i) &\triangleq f([\![\psi_1]\!]_1(C)(D)(i),\ldots,[\![\psi_k]\!]_1(C)(D)(i)) \\
[\![\texttt{if } \sigma \texttt{ then } \psi_1 \texttt{ else } \psi_2]\!]_1(C)(D)(i) &\triangleq \begin{cases} [\![\psi_1]\!]_1(C)(D)(i) & \text{if } [\![\sigma]\!]_1(C)(D)(i) = \text{true} \\ [\![\psi_2]\!]_1(C)(D)(i) & \text{if } [\![\sigma]\!]_1(C)(D)(i) = \text{false} \\ \bot & \text{if } [\![\sigma]\!]_1(C)(D)(i) = \bot \\ ? & \text{if } [\![\sigma]\!]_1(C)(D)(i) = ? \end{cases} \\
[\![\psi[-j]]\!]_1(C)(D)(i) &\triangleq \begin{cases} [\![\psi]\!]_1(C)(D)(i-j) & \text{if } i \geq j \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

For the other functions, we first define duration restricted subsets of a context, which can be used to evaluate properties using only data available at a given point in time.

Definition 4. *Given a context C we define the duration-d prefix of C as the context $C|_d$ defined by*

$$C|_d(v)(i) \triangleq \begin{cases} C(v)(i) & \textit{if } i \leq d \\ ? & \textit{otherwise} \end{cases}$$

which we use to define the following two helper functions,

Definition 5. *For maximum duration i, expression ψ, and context C, we define the functions* $\text{first}, \text{last} : \mathbb{N} \times \mathbb{E}[\mathbb{D}] \times \textsc{Context} \to \mathbb{N} \cup \{\infty\}$ *defined by*

$$\begin{aligned}
\text{first}(i,\psi,C) &\triangleq \min\{\, j \in \mathbb{N} \mid [\![\psi]\!](C|_j)(j) \notin \{\bot,?\} \wedge j \leq i \,\} \\
\text{last}(i,\psi,C) &\triangleq \max\{\, j \in \mathbb{N} \mid [\![\psi]\!](C|_j)(j) \notin \{\bot,?\} \wedge j \leq i \,\}
\end{aligned}$$

where each of these functions is set to ∞ if $[\![\psi]\!](C|_j)(j) \in \{\bot,?\}$ for all j.

Then we define the single-step semantics of dynamic properties by,

$$[\![\texttt{defer}(\psi)]\!]_1(C)(D)(i) \triangleq \begin{cases} [\![\psi_j]\!]_1(C)(D)(i) & \text{if } i \geq j \\ \bot & \text{if } j = \infty \end{cases}$$

$$[\![\texttt{dynamic}(\psi)]\!]_1(C)(D)(i) \triangleq \begin{cases} [\![\psi_k]\!]_1(C)(D)(i) & \text{if } i \geq k \\ \bot & \text{if } k = \infty \end{cases}$$

where $j = \text{first}(i, \psi, C)$, $k = \text{last}(i, \psi, C)$, $\psi_j \triangleq [\![\psi]\!]_1(C|_j)(D)(j)$, and $\psi_k \triangleq [\![\psi]\!]_1(C|_k)(D)(k)$.

Finally, we define the semantics of each of the DUP helper functions by

$$[\![\texttt{update}(\psi_1, \psi_2)]\!]_1(C)(D)(i) \triangleq \begin{cases} [\![\psi_1]\!]_1(C)(D)(i) & \text{if first}(i, \psi_2, D) = \infty \\ [\![\psi_2]\!]_1(C)(D)(i) & \text{otherwise} \end{cases}$$

$$[\![\texttt{when}(\psi)]\!]_1(C)(D)(i) \triangleq \begin{cases} \text{false} & \text{if first}(i, \psi, C) = \infty \\ \text{true} & \text{otherwise} \end{cases}$$

$$[\![\texttt{default}(\psi, c)]\!]_1(C)(D)(i) \triangleq \begin{cases} [\![\psi]\!]_1(C)(D)(i) & \text{if } [\![\psi]\!]_1(C)(D)(i) \neq \bot \\ c & \text{otherwise} \end{cases}$$

4 Design Patterns with DUPs

In practical RV scenarios, system requirements change. Supporting such changes with a first-class language construct allows specifications to adapt systematically, and allows expressing which parts are allowed to adapt. With DUPs, not only can the specification itself evolve over time, but it also becomes possible to express meta-properties, i.e., properties about how the specification may change. This section presents design patterns for writing specifications with DUPs in DynSRV.

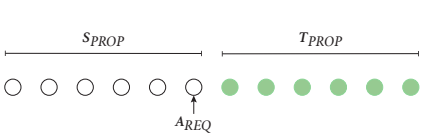
General Design Patterns

Open Property shows the most permissive use of `dynamic`, where the verdict v, directly reflects the incoming property p. In this case, it is up to the sender to ensure that the provided property is safe and valid.

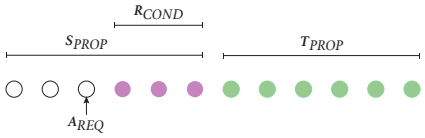
$$v = \texttt{dynamic}(p)$$

Weaken allows dynamic properties to weaken an existing requirement. In this example, accepting a new goal g normally requires the robot's battery level b to be above 30%. However, in emergencies such as a fire, strictly enforcing this threshold could block critical actions, such as evacuating an area or saving material, thus custom rules p are allowed.

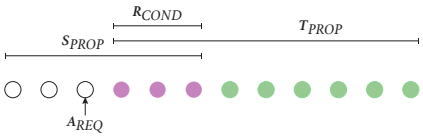
$$v = g \implies b > 30 \vee \texttt{default}(\texttt{dynamic}(p), \text{false})$$



(a) One-point adaptation, where T_{PROP} is used immediately upon arrival.



(b) Guided adaptation, where T_{PROP} is used when R_{COND} is satisfied.



(c) Overlap adaptation, where T_{PROP} is used alongside S_{PROP} until the R_{COND} is satisfied, whereafter only T_{PROP} is used.

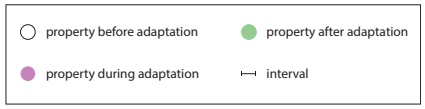


Fig. 2: Adaptation semantics proposed by Zhang and Cheng, figure adapted from [29] with minor modifications for SRV.

Strengthen allows dynamic properties to strengthen existing requirements. In the example, T is the current time, T_m is the scheduled maintenance time, and p represents dynamic rules that further constrain the maintenance window. For instance, these rules might shorten the service interval if the battery degrades or if the robot moves farther from its charging station.

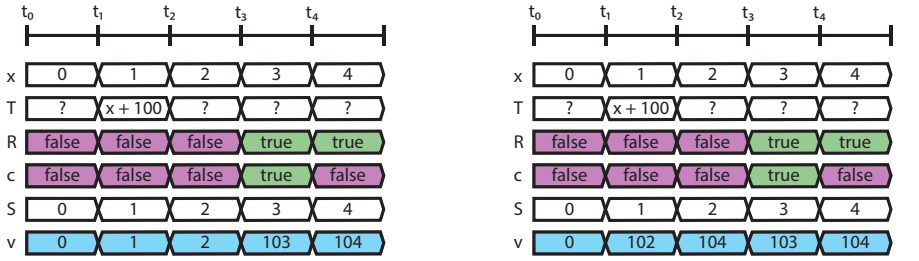
$$v = T < T_m \wedge \texttt{default}(T < \texttt{dynamic}(p), \text{true})$$

Refinement allows refining an existing property with a new one, where the verdict reflects whether the refinement is valid. In this example, b represents an update of the original condition b_c to a new property p once sent. The refinement expression r evaluates whether the new property remains valid within the context of the original condition. Notably, r is a tautology ($b_c \implies b_c$) when no new property has been provided. The verdict v combines the updated condition b and the refinement r, where true means the requirements are met. If the new property evaluates to $\bot$, the verdict becomes false.

$$b_c = b_{in} > 30 \qquad b = \texttt{update}(b_c, \texttt{defer}(p))$$
$$r = b \implies b_c \qquad v = \texttt{default}(b \wedge r, \text{false})$$

Adaptation Patterns

In their works on A-LTL, Zhang and Cheng [29] formalised the semantics of three commonly occurring adaptation semantics: *one-point*, *guided*, and *overlap*



(a) Trace demonstrating guided adaptation. The verdict stream v switches to T after R is satisfied.

(b) Trace demonstrating overlap adaptation. The verdict stream v combines S and T until R is satisfied, after which it uses only T.

Fig. 3: Example traces for guided and overlap adaptation.

adaptation[2]. This is depicted in Fig. 2. To highlight the expressiveness of DynSRV, we demonstrate how the latter two can be expressed with DUPs. One-point adaptation is trivial with DynSRV, as it immediately applies the new property, which is the default behaviour of `defer` and `dynamic`.

Guided Adaptation allows a new property to not be used immediately but await for a restriction condition to be satisfied, as depicted in Fig. 2b. This allows the system to delay the application of a new property until appropriate.

The example below demonstrates a specification implementing guided adaptation in DynSRV, with the corresponding trace in Fig. 3a. An integer stream is used for the verdict to more clearly represent its distinct states.

$$S = x \qquad R = \texttt{when}(T) \wedge c \vee \texttt{default}(R[-1], \text{false})$$
$$c = x == 3 \qquad v = \texttt{if}\ R\ \texttt{then}\ \texttt{defer}(T)\ \texttt{else}\ S$$

Here, x and T are input streams, with T representing a property received at time step 1. This property is not applied to the verdict v immediately but is gated by R, which becomes true when c holds after the property is received – at time step 3. The disjunction with $\texttt{default}(R[-1], \text{false})$ ensures that once R holds, it remains true thereafter. The verdict v switches to the deferred property $\texttt{defer}(T)$ only after R holds; otherwise, it yields from S.

Overlap Adaptation allows a new property to be used alongside the original property until a condition is satisfied, as depicted in Fig. 2c. Once this condition holds, the new property is used exclusively.

[2] In Zhang and Cheng's [29] semantics, adaptation may involve a delay between receiving and applying an adaptation request to ensure the program is in a safe state. This is not required in DynSRV, as it is stateless.

The example below shows a DynSRV specification implementing this behaviour, with its trace depicted in Fig. 3b:

$$S = x \qquad R = \texttt{when}(T) \wedge c \vee \texttt{default}(R[-1], \text{false})$$
$$c = x == 3 \qquad v = \texttt{if } \neg\texttt{when}(T) \texttt{ then } S \texttt{ else if } \neg R \texttt{ then } S + T \texttt{ else } T$$

Unlike guided adaptation, overlap adaptation introduces a transition phase where the original stream and the new one are combined $(S + T)$[3] until the condition R becomes true. Initially, the verdict v yields from S. When a new property is received at time step 1 via T, the verdict combines S and T until R is satisfied at time step 3. After that, the verdict uses only T.

5 Memory Management and History with DUPs

Efficient online monitoring with SRV has long been a focus of the community. Newer languages such as TeSSLa [18] guarantee Bounded Memory (BM) by disallowing future stream indexing – a restriction also adopted by DynSRV. Here, we define BM to mean that at each monitoring step, the monitor's memory usage does not grow with the length of the trace. While memory usage may change dynamically (e.g., when new properties are added), it must remain independent of the trace length. Allowing DUPs in DynSRV introduces a trade-off for ensuring BM, as dynamic expressions may require access to historical data that would otherwise be discarded as a part of the memory management strategy. In static SRV languages, the Dependency Graph (DG) can be used to safely discard unused history once it is no longer needed to resolve equations. In contrast, DUPs can introduce new data dependencies at runtime, potentially referencing historical values that have already been discarded to free memory. This dynamic behaviour makes it impossible to guarantee both BM usage and optimal resolution of expressions introduced by DUPs. Even if a dynamic expression at time t could be resolved to a non-$\perp$ value given the full trace, the monitor may still return $\perp$ if the necessary data has been discarded. This section defines *solvable* stream expressions in relation to memory management and presents three strategies that demonstrate the trade-off.

Solvable Stream Expressions
For this section, we use the notation that $\phi[-j]$ refer to a stream expression ϕ annotated with an optional temporal index $-j$, where $j = 0$ when no explicit index is given.

[3] Here, addition is used as an example; other operators may apply depending on context.

Definition 6 (Effective Index). *Given a stream expression $\phi[-j]$ and stream variable $s \in \texttt{vars}(\phi)$, the* effective index $\texttt{index}(\phi, s, j)$ *is defined as:*

$$\texttt{index}(\phi, s, j) = \begin{cases} j + j' & \text{if } \phi = s[-j'], s \in \texttt{vars}(\phi) \\ j & \text{if } \phi = \texttt{when}(\psi) \vee \\ & \quad \phi = \texttt{default}(\psi, c) \\ \texttt{index}(\psi, s, j) & \text{if } \phi = \texttt{defer}(\psi) \vee \\ & \quad \phi = \texttt{dynamic}(\psi) \\ \texttt{index}(\psi, s, j + j') & \text{if } \phi = \psi[-j'] \\ \max(\{\texttt{index}(\psi, s, j) \mid & \text{if } \phi = f(\psi_1, \ldots, \psi_n) \\ \quad \psi \in \{\psi_1, \ldots, \psi_n\}\}) & \\ \max(\{\texttt{index}(\sigma, s, j), & \text{if } \phi = \texttt{if } \sigma \texttt{ then } \psi_1 \\ \quad \texttt{index}(\psi_1, s, j), \texttt{index}(\psi_2, s, j)\} & \qquad \texttt{else } \psi_2 \\ \max(\{\texttt{index}(\psi_1, s, j), & \text{if } \phi = \texttt{update}(\psi_1, \psi_2) \\ \quad \texttt{index}(\psi_2, s, j)\} & \\ j & \text{otherwise} \end{cases}$$

Effective Index is useful for reasoning about expressions with nested temporal indices. We highlight the case with $\texttt{when}(\psi)$ and $\texttt{default}(\psi, c)$ expressions, which do not introduce new temporal indices and can therefore be used in practice to relax the requirements for *solvable* introduced below.

Definition 7 (Dependency Graph). *Let $s_x \in O, s_y \in S$ be streams and $\phi[-j]$ the expression assigned to s_x. A* Dependency Graph (DG) *is a weighted and directed multigraph $G = (S, E)$, with edges $(s_x, s_y, k, T) \in E$ iff the equations for s_x contains s_y as a subexpression with effective index $k = \texttt{index}(\phi, s_y, j)$, and the edge was introduced at monitor step T.*

The need for T in the DG definition becomes apparent when considering Dynamic Dependency Graphs (DDGs) in strategy 3 below. Until then, it can be assumed that $T = 0$.

Note that if the specification contains DUPs, e.g., if $x = \texttt{dynamic}(p)$ then $(x, p, 0, 0) \in E$, but the properties sent at runtime to p are not. As a result, the DG must be extended to track new dependencies introduced by DUPs, as demonstrated with the strategies below, such that the dynamically received expressions can become *solvable*.

Definition 8 (Solvable). *Let $k = \texttt{index}(\phi, s, j)$ be the effective index of ϕ for stream variable $s \in \texttt{vars}(\phi)$. A stream expression $\phi[-j]$ is said to be* solvable *at monitor step t if there exists an edge $(s', s, j', T) \in E$ such that:*

$$t \geq T + k \wedge j' \geq k,$$

and s evaluated at monitor step $(t - k)$ is not equal to $\bot$.

Intuitively, the first term $t \geq T + k$ ensures that the monitor has progressed sufficiently in steps since the dependency was introduced to solve the expression. The second term $j' \geq k$ ensures that there exists an edge in the DG with a sufficiently large effective index such that the kth last value of s is not discarded.

Theorem 1 claims that a solvable stream expression evaluates to a non-$\bot$ value at monitor step t. The proof follows by structural induction on ϕ and is written out below.

Theorem 1. *Let ϕ be a stream expression that is solvable at monitor step t and may contain DUPs instantiated with solvable stream expressions $(\psi_1, \ldots, \psi_N)$. Then, ϕ evaluated at monitor step t is not equal to $\bot$.*

Proof. Assume ϕ is a stream expression that is solvable at monitor step t, potentially containing DUPs receiving solvable subexpressions $(\psi_1, \ldots, \psi_N)$. We proceed by structural induction on the stream expression ϕ:

- Basic expressions: By definition of solvable, all stream variables referenced by ϕ at their respective time indices are not equal to $\bot$ at monitor step t. Therefore, if ϕ is a basic expression it cannot evaluate to $\bot$ at monitor step t, since basic expressions only evaluate to $\bot$ when a referenced stream variable is $\bot$ at that time step.
- $\texttt{default}(\psi, c)$ never evaluates to $\bot$, because it yields ψ if $\psi \neq \bot$, and defaults to the constant $c \in \mathbb{D}$ otherwise.
- $\texttt{when}(\psi)$ is guaranteed to evaluate to a value in $\mathbb{B}$.
- $\texttt{update}(\psi_1, \psi_2)$ evaluates to ψ_2 if $\psi_2 \neq \bot$, which is guaranteed by the definition of solvable.
- DUPs: If ψ_i is a DUP, it is either $\psi_i = \texttt{defer}(\psi_i')$ or $\psi_i = \texttt{dynamic}(\psi_i')$. For ψ_i to be solvable, ψ_i' must also be solvable. By induction, this means that ψ_i' is either a non-DUP expression that evaluates to a non-$\bot$ value at step t or a DUP that is solvable, meaning it will evaluate to a non-$\bot$ value at step t.

Therefore, ϕ evaluated at monitor step t is not equal to $\bot$.

When writing DynSRV specifications, considering when a stream expression is not solvable is crucial, as an $\bot$ verdict at runtime may not be desirable. We now present three memory management strategies that have different trade-offs between memory efficiency and trace availability (keeping expressions solvable as often as possible).

Strategy 1 – Discard BM: Retain the Entire History
Favoring trace availability, this strategy introduces unbounded time dependencies to every other stream in the specification when a DUP is present:

$$E_{\text{DUP}} = \bigcup_{s \in O} \{(s, s', j, 0) \mid \text{hasDUP}(s), s' \in S \setminus \{s\}, j \in \mathbb{N}\}$$
$$G = (S, E \cup E_{\text{DUP}})$$

where hasDUP(s) indicates that stream s's expression contains a DUP.

Using this DG to retain data ensures that any stream expression received dynamically through a DUP is solvable at any monitor step t, if the monitor has progressed sufficiently and none of the referenced stream variables are $\bot$ at the specific monitor steps. That is, the condition $j' \geq k$ from Definition 8 is always met. However, this strategy sacrifices memory efficiency, as any specification

involving DUPs will no longer have BM.

Strategy 2 – Preserve BM: Statically Specifying Dependencies of DUPs To retain memory efficiency, an alternative approach is to restrict DUPs by requiring users to explicitly annotate their potential temporal dependencies in advance. For example, the expression $\texttt{dynamic}(p, \{(x, -4), (y, -2)\})$ declares that the dynamically introduced property p may depend on stream x up to 4 steps back in time, and on stream y up to 2 steps. A similar change could be made for `defer`. The DG is then defined as:

$$E_{\text{DUP}} = \bigcup_{s \in O} \text{declaredDUP}(s)$$
$$G = (S, E \cup E_{\text{DUP}})$$

where declaredDUP(s) returns the set of edges that are explicitly declared as dependencies of s. This strategy allows each property in the specification to maintain BM, even when using DUPs. However, this comes at the cost of expressiveness as it becomes possible to introduce expressions that never become solvable, specifically, expressions where the condition $j' \geq \texttt{index}(\phi, s, j)$ does not hold, causing them to always evaluate to $\bot$.

Strategy 3 – Preserve BM: Dynamically Update Dependencies To balance memory efficiency and trace availability, we propose a DDGs, which extends static DGs by adding dependencies introduced by DUPs at runtime. The DG becomes a time-dependent structure, where the edges are updated based on the declared dependencies of received expressions.

We define the DDG as a stream of DGs that is updated according to the received expressions:

$$E_{\text{DUP}}(t) = \bigcup_{\substack{s \in O \\ \text{where } s \text{ is assigned } e}} \text{dep}(s, t, [\![s]\!]\,(\text{last}(t, e, [\![\Phi]\!]\,C_{in})))$$
$$G(t) = (S, E \cup E_{\text{DUP}}(t))$$

where $[\![s]\!]\,(\text{last}(t, e, [\![\Phi]\!]\,C_{in}))$ denotes the last received expression for stream s at time t in the context $[\![\Phi]\!]\,C_{in}$, and $\text{dep}(s, T, e)$ returns the set of dependencies introduced by the expression e assigned to stream s at monitor step T.

The DDG is used in our DynSRV implementation, detailed in Section 6, to determine how much history to retain at each step. By updating the DDG accordingly, the condition $j' \geq \texttt{index}(\phi, s, j)$ from Definition 8 is guaranteed for new properties as there exists a j' equal to j. However, the monitor step T from Definition 7 is crucial here: If a new expression ψ referencing stream variable s at effective index k arrives at step T, and at step $T-1$, s had no incoming edges in the DDG, then ψ is not solvable before time $T+k$, and may evaluate to $\bot$ until then. While this approach offers weaker trace availability than the previous two strategies, it preserves bounded memory and supports the full expressiveness of DUPs.

6 Implementation and Performance

DynSRV is implemented in Rust as part of the RoboSAPIENS trustworthiness language framework [17], which is a general framework for implementing stream-based languages. The framework is implemented as a modular runtime, with a parser layer that parses specifications into an Abstract Syntax Tree, an extensible execution layer that allows for multiple runtime engine and language semantics to be implemented, and a flexible IO layer allowing input and output streams to be transmitted several sources including files, MQTT, and ROS topics. The full source code is available at [4].

As discussed in Section 5, DUPs impose some additional requirements on the implementation of the language compared to existing SRV languages. We meet these requirements via two runtime engine implementations: a constraint-based similar to LOLA [8], and a novel stream-based which translates the specification into a collection of asynchronous Rust actors that communicate over channels. The latter engine features some key design decisions:

- Dependencies between stream variables are handled dynamically, with a publisher/subscriber model used to propagate input values to dependent streams.
- The lifetimes of stream values are handled automatically via Rust's ownership model and the use of channels to communicate between actors. This means that there is no central constraint store or garbage collection step.

This dynamic model does impose some performance challenges since specifications cannot easily be compiled to specifically optimized Rust code for a single specification (as in [12]) and has to keep track of dependencies at runtime.

In Fig. 4, we compare the monitoring of a dynamically introduced property to the same property introduced statically. We also compare the impact of the time during the run at which the dynamic property was introduced. This shows a significant, but constant overhead factor for the use of dynamic properties in this scenario, with this overhead factor increasing the later the dynamic property is introduced in the trace. However, in this case, the overall monitoring performance is still sufficient for real-time monitoring, with $100,000$ events being monitored in under 350 milliseconds in the worst case. All benchmarks use integer inputs read from a file and were carried out using an Intel i7-1370P CPU with 32 GiB RAM, and averaged over 10 runs.

7 Conclusion

In this paper, we have demonstrated how DUPs can be supported with our SRV language DynSRV. Section 2 highlights how DynSRV differs from most related work by enabling truly dynamic expressions to the system's existing properties, whereas prior approaches primarily focus on adapting specific system behaviours.

The semantics presented in Section 3.3 define the denotational meaning of our DUPs primitives, while Section 5 compliments this by describing different

[4] `https://github.com/INTO-CPS-Association/robosapiens-trustworthiness-checker`

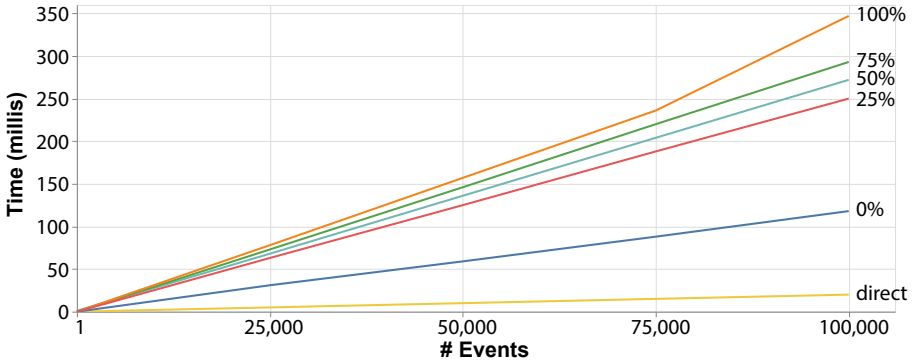


Fig. 4: Time taken to monitor a deferred property $z = \texttt{default}(\texttt{defer}(e), \text{true})$ with the property $e = x \wedge y$ introduced at a certain percentage of the run. This is compared to compared to directly monitoring the static property $z = x \wedge y$.

memory management strategies based on the trade-off between memory usage and trace availability. Design patterns in Section 4 demonstrate how to weaken, strengthen, and refine properties, as well as describe how to express well-known DSU adaptation patterns.

We consider DynSRV to be especially relevant given the increasing number of systems that require DSU and SASs in our environment and the corresponding need for correctness and safety assurances. While the adaptation for some of these systems is simple enough to be specified statically, others may change in ways where we do not necessarily know what the behaviour in certain situations precisely looks like as shown by Esterle and Brown in [10]. Even if we could specify all possible adaptations in advance, state-space explosion makes it practically infeasible. Our language allows deferring certain parts of the verification to runtime, allowing users to specify new properties as the system evolves, similar to DSU, or allowing a SASs to autonomously specify updated properties as the system evolves.

In the future, we intend to further evaluate the language for industrial use cases, including the case studies provided by the RoboSAPIENS project [17]. We also hope to extend the core language to add full support for asynchronous, timed properties and distributed monitoring of properties. As an additional direction, more work could be carried out to understand and improve the performance characteristics of DUPs based on more realistic benchmarks and properties. For instance, techniques such as Just In Time compilation could potentially reduce the performance impact of dynamic properties compared to static properties.

References

1. 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
2. Bierman, G., Hicks, M., Sewell, P., Stoyle, G.: Formalizing Dynamic Software Updating
3. 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
4. Cassar, I., Francalanza, A.: On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In: Integrated Formal Methods, pp. 176–192 (2016). https://doi.org/10.1007/978-3-319-33693-0_12
5. Cassar, I., Francalanza, A.: Runtime Adaptation for Actor Systems. In: Runtime Verification, pp. 38–54 (2015). https://doi.org/10.1007/978-3-319-23820-3_3
6. Clark, T., Kulkarni, V., Barat, S., Barn, B.: A Homogeneous Actor-Based Monitor Language for Adaptive Behaviour. In: Programming with Actors: State-of-the-Art and Research Perspectives, pp. 216–244 (2018). https://doi.org/10.1007/978-3-030-00302-9_8
7. Clark, T., Kulkarni, V., Barat, S., Barn, B.: Actor Monitors for Adaptive Behaviour. In: Proceedings of the Innovations in Software Engineering Conference, pp. 85–95 (2017). https://doi.org/10.1145/3021460.3021469
8. 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
9. Distefano, D., Rensink, A., Katoen, J.-P.: Model Checking Birth and Death. In: Proceeings of IFIP International Conference on Theoretical Computer Science (TCS), pp. 435–447 (2002). https://doi.org/10.1007/978-0-387-35608-2_36
10. Esterle, L., Brown, J.N.: The Competence Awareness Window: Knowing what I can and cannot do. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), pp. 62–63 (2020). https://doi.org/10.1109/ACSOS-C51401.2020.00031
11. Finkbeiner, B., Klein, F., Metzger, N.: Live Synthesis. Innovations Syst Softw Eng **18**(3), 443–454 (2022). https://doi.org/10.1007/s11334-022-00447-5
12. Finkbeiner, B., Oswald, S., Passing, N., Schwenger, M.: Verified Rust Monitors for Lola Specifications. In: Runtime Verification, pp. 431–450 (2020). https://doi.org/10.1007/978-3-030-60508-7_24
13. 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
14. Hicks, M., Moore, J.T., Nettles, S.: Dynamic software updating. ACM SIGPLAN Notices **36**(5), 13–23 (2001). https://doi.org/10.1145/381694.378798
15. 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
16. Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/mc.2003.1160055

17. Larsen, P.G., Ali, S., Behrens, R., Cavalcanti, A., Gomes, C., Li, G., De Meulenaere, P., Olsen, M.L., Passalis, N., Peyrucain, T., Tapia, J., Tefas, A., Zhang, H.: Robotic safe adaptation in unprecedented situations: the RoboSAPIENS project. Research Directions: Cyber-Physical Systems **2** (2024). https://doi.org/10.1017/cbp.2024.4
18. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: Runtime Verification of Non-Synchronized Real-Time Streams. In: Proceedings of the Annual ACM Symposium on Applied Computing, pp. 1925–1933 (2018). https://doi.org/10.1145/3167132.3167338
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 (2024). https://doi.org/10.1016/j.hcc.2024.100255
20. Lounas, R., Mezghiche, M., Lanet, J.-L.: Formal Methods in Dynamic Software Updating: A Survey. International Journal of Critical Computer-Based Systems **9**(1–2), 76–114 (2019). https://doi.org/10.1504/IJCCBS.2019.098794
21. Nahabedian, L., Braberman, V., D'Ippolito, N., Honiden, S., Kramer, J., Tei, K., Uchitel, S.: Assured and Correct Dynamic Update of Controllers. In: 2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 96–107 (2016). https://doi.org/10.1145/2897053.2897056. https://ieeexplore.ieee.org/document/7830552 (visited on 07/25/2025)
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. Seifzadeh, H., Abolhassani, H., Moshkenani, M.S.: A survey of dynamic software updating. Journal of Software: Evolution and Process **25**(5), 535–568 (2012). https://doi.org/10.1002/smr.1556
24. Sokolsky, O., Sammapun, U., Lee, I., Kim, J.: Run-Time Checking of Dynamic Properties. Electronic Notes in Theoretical Computer Science **144**(4), 91–108 (2006). https://doi.org/10.1016/j.entcs.2006.02.006
25. 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
26. Walton, C., Krl, D., Gilmore, S.: An Abstract Machine for Module Replacement. In: Proceedings of the Workshop on Principles of Abstract Machines (1998)
27. Weyns, D.: Engineering Self-Adaptive Software Systems – An Organized Tour. In: Proceedings of the IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W), pp. 1–2 (2018). https://doi.org/10.1109/FAS-W.2018.00012
28. Wong, T., Wagner, M., Treude, C.: Self-adaptive systems: A systematic literature review across categories and domains. Information and Software Technology **148**, 106934 (2022). https://doi.org/10.1016/j.infsof.2022.106934
29. Zhang, J., Cheng, B.H.C.: Using Temporal Logic to Specify Adaptive Program Semantics. Journal of Systems and Software **79**(10), 1361–1369 (2006). https://doi.org/10.1016/j.jss.2006.02.062