Form Methods Syst Des (2014) 44:203–239
DOI 10.1007/s10703-013-0199-z

Runtime verification of embedded real-time systems

Thomas Reinbacher · Matthias Függer · Jörg Brauer

Published online: 7 November 2013
© The Author(s) 2013. This article is published with open access at Springerlink.com

Abstract We present a runtime verification framework that allows on-line monitoring of past-time Metric Temporal Logic (ptMTL) specifications in a discrete time setting. We design observer algorithms for the time-bounded modalities of ptMTL, which take advantage of the highly parallel nature of hardware designs. The algorithms can be translated into efficient hardware blocks, which are designed for reconfigurability, thus, facilitate applications of the framework in both a prototyping and a post-deployment phase of embedded real-time systems. We provide formal correctness proofs for all presented observer algorithms and analyze their time and space complexity. For example, for the most general operator considered, the time-bounded Since operator, we obtain a time complexity that is doubly logarithmic both in the point in time the operator is executed and the operator's time bounds. This result is promising with respect to a self-contained, non-interfering monitoring approach that evaluates real-time specifications in parallel to the system-under-test. We implement our framework on a Field Programmable Gate Array platform and use extensive simulation and logic synthesis runs to assess the benefits of the approach in terms of resource usage and operating frequency.

Keywords Runtime verification · Embedded real-time systems · Past-time logics · Online monitoring

T. Reinbacher · M. Függer (✉)
Embedded Computing Systems Group, Vienna University of Technology, Treitlstrasse 3, 1040 Vienna, Austria
e-mail: fuegger@ecs.tuwien.ac.at

T. Reinbacher
e-mail: reinbacher@ecs.tuwien.ac.at

J. Brauer
Embedded Software Laboratory, RWTH Aachen University and Verified Systems International GmbH, Am Fallturm 1, 28359 Bremen, Germany
e-mail: brauer@verified.de

1 Introduction

Rigorous verification strategies are especially vital for the domain of safety-critical embedded real-time systems [48] where systems often do not only need to comply with a set of functional requirements but also—equally important—with tight timing constraints. Correct behavior of these systems is defined by the sequence of data they produce—either internally or at their physical outputs—complemented with their temporal behavior. The key idea behind formal verification techniques such as model checking [6, 22] is to exhaustively *check all executions* of a structure that is related to an implementation and its environment against given requirements, the latter of which are often formalized in terms of a temporal logic. Exhaustive analysis of programs, however, often suffers from practical infeasibility (due to state space explosion [21]) and/or theoretical impossibility (due to undecidability results).

In runtime verification [9], observers are synthesized to automatically evaluate the *current execution* of a system-under-test (SUT), typically from a formal specification in a logic that is suitable to cover certain forms of real-world specifications. The on-the-fly nature of runtime verification can be coupled with costly overhead [10, 56, 71]. Some mitigated overhead by reducing instrumentation points [34]; others ported the system and/or the observers to a more powerful architecture, such as database systems [8]. These artifacts of runtime verification are not compatible with embedded real-time systems running on ultra-portable hardware with power and performance limitations [65].

To evaluate specifications, runtime verification depends on observations of the state of the SUT. These observations are referred to as events and are input to the observer. However, the SUT's state typically is not directly observable.

An approach classically taken in runtime verification to obtain observations is to instrument the code base, a technique that has proven feasible for a number of high-level implementation languages such as C, C++, and Java [9, 39, 40, 64] as well as for hardware description languages such as VHDL and Verilog [4, 77]. Instrumentation can be done manually, or automatically by scanning programs for assignments and function calls at the level of the implementation language and then inserting hook-up functions that emit relevant events to an observer. However, for the domain of (safety-critical) embedded real-time systems, existing approaches, despite the considerable progress in the past, are not directly applicable; mainly due to the following limitations:

- Source code instrumentation of high-level languages can only capture events that are accessible from within the instrumented software system. Embedded systems [59] often include both hardware and mechanical parts; events from those might go unnoticed for an instrumenting runtime verification approach.
- The timing behavior of the SUT is altered by instrumentation [23, 34]. The additional runtime overhead may drastically impact the correctness of a heavy-loaded real-time application with tight deadlines. The same applies to memory consumption of resource constrained systems. The relevance of this argument is supported by the fact that restricted architectures are often used in critical environments[12, 33, 66], such as in nuclear power plants [28] and spacecrafts [30, Chap. 3].
- Instrumentation may make re-certification of the system onerous (e.g., systems certified for civil aviation after DO-178B [73]).
- In its present shape, runtime verification often analyzes the correctness of high-level code. However, to show that a high-level specification is correctly reproduced by the target system, it is further necessary to show the correctness of the translation of the high-level code into executable code, i.e., the compiler. Despite recent breakthroughs [52, 53], only

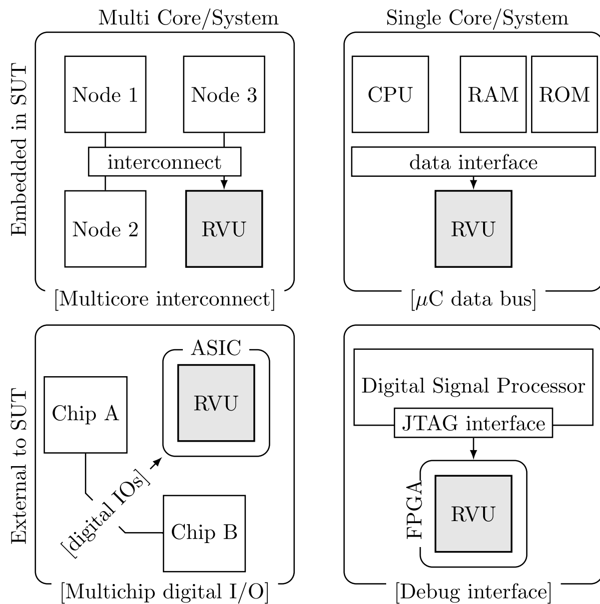


Fig. 1 Sample applications of an instance of the proposed framework, i.e., the Runtime Verification Unit (RVU). *Top left*: RVU embedded into a network-on-chip, monitoring data exchanged among network nodes; *Top right*: RVU connected to the data interface of a microcontroller IP-core, monitoring microcontroller behavior (software); *Bottom left*: RVU connected to digital interconnects among chips on a printed circuit board (PCB), monitoring data exchanged through digital I/Os; *Bottom right*: RVU running on an FPGA attached to a debug interface of a digital signal processor, monitoring changes of accessible registers and diagnosis indicators

few verified compilers are used in practice and flaws introduced by compilers [31, 55, 81] may remain undetected by existing approaches.

– Instrumentation at binary code level may circumvent the process of establishing correctness of the compiler. However, binary instrumentation is incomplete as long as a sound reconstruction of the control flow graph is not obtained from the binary. Despite being an active area of research [7, 35, 46, 67], generating sound yet precise results remains a challenge.

There exist, however, systems and applications [80], where the relevant events can be observed without the need to infuse additional functions into the high-level code. Consider, for example, an implementation of a network protocol, where the task is to check the correctness of data flow between two network nodes. It appears natural to place an additional (passive) node in the network that collects events sent over the network, rather than instrumenting the high-level code of the network nodes. The strength of an approach like this is that collecting of events is non-intrusive, at least, as long as the additional node is passive and does not actively participate in the communication. It is important to observe that information exchange among systems is often performed by standardized interfaces. This is especially the case for embedded real-time systems, at various levels of detail [59, Chap. 3]. For certain systems, wiretapping is the only option left to gain information of the state of the system, for example, if the design includes proprietary hardware or software components.

In the light of the discussion above, we proceed by defining requirements of a runtime verification framework targeting embedded real-time systems. We aim at a framework that is transparent to a hardware implementation, so as to be attached to or embedded into various SUTs. Examples of applications are outlined in Fig. 1. We summarize these special requirements as:

Stand-alone The runtime verification framework should not only be deployed during the testing phase of the product but also after the product is shipped. Therefore, it should

operate in a self-contained way and not depend on a powerful host computer that executes the observer.

Non-intrusive The resulting observers should be efficient enough to not alter the timing requirements of the SUT. From an algorithmic viewpoint, observers with an a-priory known execution time are of utmost importance so as to statically determine upper bounds of the execution time of the observer. From an implementation point of view, we need to provide measures to passively observe events from the SUT.

Timed To support correctness claims that involve timed properties, the framework should support expressive logics to formalize not only functional but also real-time requirements.

Reconfigurable For the testing phase, the framework should be reconfigurable without requiring to re-synthesize the whole hardware design, which may take dozens of minutes to complete, for example when targeting an Field Programmable Gate Array (FPGA) platform.

2 Contributions and roadmap

Our work can be seen as a response to overcome the above limitations that hinder the broad application of runtime verification to embedded real-time systems. This article provides the following contributions toward a *stand alone*, *non-intrusive*, *timed*, and *reconfigurable* hardware runtime verification approach:

(a) We present on-line observer algorithms that allow one to verify whether a past-time metric temporal logic (ptMTL) formula holds at (discrete) times $n \in \mathbb{N}_0$. The algorithms make use of basic operations only and are stated in a way that allows for a direct implementation in hardware, that can run without a host computer. By that our observers fulfill the timed and stand alone requirements.

(b) We formally prove the observers' correctness and derive bounds on their time complexity in terms of gate delays and their space complexity in terms of required memory bits. With n being the time an observer algorithm is executed and J a non-empty interval we obtain, for the most general of the presented observer algorithms, the ptMTL Since operator $\varphi_1 \, S_J \, \varphi_2$, a time complexity of $\mathcal{O}(\log_2 \log_2 \max(J \cup \{n\}))$, only. The observer's space complexity is dominated by the size of a list it needs to maintain. We show that the list's space complexity is at most $2\lceil \log_2(n) \rceil \cdot (2\max(J) - \min(J) + 2)/(2 + \mathrm{len}(J))$, where $\mathrm{len}(J) = \max(J) - \min(J)$. Both complexity results, as well as the fact that our algorithms refrain from loops and recursions and build on simple operations only, enable applications of our runtime verification framework on resource limited platforms that require predictable timing and memory consumption.

(c) We explain how to derive non-instrumenting efficient realizations of the proposed observer algorithms in hardware. The resulting hardware profits from the simplicity and low complexity of our highly-parallel observer algorithms. In contrast to instrumentation-based runtime verification techniques for software systems our observers are well suited to supervise hardware components. By that, in combination with (b), our observers fulfill the non-intrusive requirement. Although our algorithms are tailored for a hardware implementation, the observers can simply be adopted to run in software too. Reconfigurability of our observers is achieved by, instead of hardwiring the observers inputs and outputs according to their parse tree, letting a programmable, specifically tailored microprocessor control a pool of observers.

(d) To evaluate the effectiveness of our approach, we report on a throughout study of simulation traces and synthesis results of a full-fledged hardware implementation of the presented observer algorithms and discuss the scalability of our approach.

With regard to the contributions above, (a) and (b) are an extension of our work we presented at the International Conference on Runtime Verification [71], including detailed correctness proofs for our algorithms and (c) and (d) are unique contributions of this article. Contribution (c) builds on our previous work [69], where we presented a microprocessor designed to evaluate ptLTL specifications in a software-oriented fashion. Using this approach to check ptMTL specifications, however, requires a costly (cf. Sect. 3.3) rewriting to an equivalent ptLTL specifications. Instead, we show how to map the building blocks of our ptMTL observer algorithms into efficient hardware units. This enables our microprocessor to natively evaluate ptMTL specifications in real-time. Both (c) and (d) help us to put the presented real-time observer algorithms into industrial practice.

The contributions of this article are presented as follows. First, Sect. 3 is a primer on temporal logics, which sets the scene for the monitoring algorithms stated in Sect. 4. Section 5 details the key structures of the hardware design and Sect. 6 reports on experimental evidence. We continue with a survey of related work in Sect. 7 and conclude in Sect. 8.

3 Logics for runtime verification

We briefly summarize the temporal logics past-time linear temporal logic (ptLTL) and past-time metric temporal logic (ptMTL) which are used to specify properties in our framework. Both allow one to specify safety, past-time properties over executions. For further details, we refer the reader to more elaborate sources such as [2, 13, 32, 42, 51, 57].

3.1 Past-time linear temporal logic

A popular logic in runtime verification is the past-time fragment of LTL (ptLTL), mainly due to: (i) observer generation for ptLTL is straightforward [39, Sect. 5], and (ii) ptLTL can easily express typical specifications [54]. Even though past-time operators do not yield the expressive power of full LTL [32, Sect. 2.6], past-time operators often express desired properties from specifications [50, 54]. With $\bullet$ in $\{\wedge, \vee, \rightarrow\}$ and σ in the set Σ of atomic propositions, a formula φ is defined as:

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid \sigma \mid \neg\varphi \mid \varphi \bullet \varphi \mid \odot\varphi \mid \diamondsuit\varphi \mid \boxdot\varphi \mid \varphi\, \mathcal{S}_s\, \varphi \mid \varphi\, \mathcal{S}_w\, \varphi$$

Hereby, $\odot\varphi$ is the past-time analogue of next and referred to as *previously* φ. Likewise, $\diamondsuit\varphi$ is referred to as *eventually in the past* φ and $\boxdot\varphi$ as *always in the past* φ. The duals of the until and the weak-until operators are $\mathcal{S}_s$ and $\mathcal{S}_w$, i.e., *strong since* and *weak since*, respectively. Similar as in LTL [41, Theorem 1], ptLTL can be reduced to the propositional operators plus two past-time operators [58], e.g., to $\odot$ and $\mathcal{S}_s$. The satisfaction relation of a ptLTL specification can be defined as follows: Let $e = (s_t)_{t\geq 0}$ be an execution where s_t is a state of the system. Denote by e^n, for $n \in \mathbb{N}_0$, the *execution prefix* $(s_t)_{0\leq t\leq n}$. For a ptLTL formula φ, time $n \in \mathbb{N}_0$ and execution e, we define *φ holds at time n of execution e*, denoted

$e^n \models \varphi$, inductively as follows:

$$
\begin{array}{ll}
e^n \models true, & \\
e^n \not\models false, & \\
e^n \models \sigma, \text{ where } \sigma \in \Sigma & \text{iff } \sigma \text{ holds on } s_n, \\
e^n \models \neg\varphi & \text{iff } e^n \not\models \varphi, \\
e^n \models \varphi_1 \wedge \varphi_2 & \text{iff } e^n \models \varphi_1 \text{ and } e^n \models \varphi_2, \\
e^n \models \varphi_1 \vee \varphi_2 & \text{iff } e^n \models \varphi_1 \text{ or } e^n \models \varphi_2, \\
e^n \models \varphi_1 \rightarrow \varphi_2 & \text{iff } e^n \models \varphi_1 \text{ implies } e^n \models \varphi_2, \\
e^n \models \odot\varphi & \text{iff } e^{n-1} \models \varphi \text{ if } n > 0 \text{, and } e^0 \models \varphi \text{ otherwise}, \\
e^n \models \varphi_1 \, S_s \, \varphi_2 & \text{iff } \exists j (0 \le j \le n) : (e^j \models \varphi_2 \ \wedge \ \forall k (j < k \le n) : e^k \models \varphi_1).
\end{array}
$$

The above syntax can be augmented with a set of additional operators [42, 51] to provide a succinct representation of common properties that appear in practice:

$$\varphi ::= \uparrow \varphi \mid \downarrow \varphi \mid [\varphi, \varphi)_s \mid [\varphi, \varphi)_w$$

$\uparrow \varphi$ and $\downarrow \varphi$ are trigger conditions where $\uparrow \varphi$ stands for *start* φ (i.e., φ was false in the previous state and is true in the current state, equivalent to $\varphi \wedge \neg \odot \varphi$), $\downarrow \varphi$ for *end* φ (φ was true in the previous state and is false in the current state, equivalent to $\neg\varphi \wedge \odot\varphi$). The interval operators are strong *interval* $[\varphi_1, \varphi_2)_s$ (φ_2 was never true since the last time φ_1 was true, including the state when φ_1 was true, equivalent to $\neg\varphi_2 \wedge ((\odot\neg\varphi_2) \ S_s \ \varphi_1)$) and weak *interval* (equivalent to $\boxdot\neg\varphi_2 \vee [\varphi_1, \varphi_2)_s$) In the following we will only refer to the strong since and shortly write S instead of S_s. Checking whether a ptLTL formula holds at time $n \in \mathbb{N}_0$ in some execution $e = (s_t)_{t \ge 0}$ can be determined by evaluating only the current state s_n and the results from the predecessor state s_{n-1} [42]. For example, evaluating the invariant $\varphi = \boxdot \, \sigma$ on execution $e = (s_t)_{t \ge 0}$ can be done by:

$$
\begin{aligned}
e^n \models \boxdot\sigma &\Leftrightarrow \textstyle\bigwedge_{t=0}^{n} (\sigma \text{ holds on } s_t) \\
&\Leftrightarrow (e^{n-1} \models \boxdot\,\sigma) \wedge (\sigma \text{ holds on } s_n)
\end{aligned}
$$

3.2 Past-time metric temporal logic

MTL [2] extends LTL by replacing the qualitative temporal operators of LTL by quantitative operators that respect time bounds. Since we are interested in on-chip observer algorithms, progress of time is provided by the (possibly divided) chip's clock signal, resulting in a discrete time base $\mathbb{N}_0$.[1] Time bounds of quantitative operators are given in form of intervals: For t in $\mathbb{N}_0$ and t' in $\mathbb{N}_0 \cup \{\infty\}$, we write $[t, t')$ for the set $\{i \in \mathbb{N}_0 \mid t \le i < t'\}$ and, if t' in $\mathbb{N}_0$, $[t, t']$ for the set $\{i \in \mathbb{N}_0 \mid t \le i \le t'\}$. Similar to ptLTL, a restriction of MTL to its past time fragment (ptMTL) is of interest. Formally, a ptMTL formula φ is defined by:

$$\varphi ::= true \mid false \mid \sigma \mid \neg\varphi \mid \varphi \bullet \varphi \mid \varphi \ S_J \ \varphi$$

where $\sigma \in \Sigma$, $\bullet \in \{\wedge, \vee, \rightarrow\}$, and $J = [t, t']$ for some $t, t' \in \mathbb{N}_0$. The semantics of *true*, *false*, σ, $\neg\varphi$, and $\varphi \bullet \varphi$ are as before. Recall that in ptLTL $\varphi_1 \ S \ \varphi_2$ expresses φ_2 *was true in the past and since then* φ_1 *was true*. By way of contrast, satisfaction of $e^n \models \varphi_1 \ S_J \ \varphi_2$ in ptMTL, does not only depend on the observation that $\varphi_1 \ S \ \varphi_2$ holds in the current state, but also on (i) the

[1] In our framework, we thus assume time points to be from $\mathbb{N}_0$.

time n of the current state and (ii) the times $i \in \mathbb{N}_0$ since when $\varphi_1 S\, \varphi_2$ was observed to be true: for at least one such i, $e^i \models \varphi_2$, and $n - i \in J$ have to hold. Formally, we define:

$$e^n \models \varphi_1\ S_J\ \varphi_2 \ \text{ iff }\ \exists i(0 \le i \le n) : \big(n - i \in J \wedge e^i \models \varphi_2 \wedge \forall j(i < j \le n) :\ e^j \models \varphi_1\big)$$

Example Many real-time properties, such as

> "*If the system leaves the idle mode, it has received an according signal in the past 50 clock-cycles.*"

can be expressed in ptMTL. The above property, e.g., can be formalized by:

$$\big(\downarrow \text{(in idle mode)}\big) \rightarrow \big(\mathit{true}\ S_{[0,50]}\ \text{(received message)}\big)$$

Not surprisingly, determining satisfaction of an MTL (or ptMTL) formula is computationally more expensive than checking satisfaction of an LTL (or ptLTL) formula [78, Theorem 3.4].

3.3 Rewriting past-time metric temporal logic to past-time linear temporal logic

In a discrete time setting, there is an equivalent ptLTL formula for every ptMTL formula [57], directly leading to an observer algorithm for $\varphi_1 S_{[a,b]} \varphi_2$. With $\odot^i \varphi$ being $\odot$ applied i times to φ, a straightforward generic translation is given by the equivalence:

$$\begin{aligned} e^n \models \varphi_1\ S_{[a,b]}\ \varphi_2 \ &\Leftrightarrow\ \exists i(a \le i \le b) : ((\odot^i \varphi_2) \wedge (\odot^{i-1} \varphi_1) \wedge (\odot^{i-2} \varphi_1) \wedge \cdots \wedge \varphi_1) \\ &\Leftrightarrow\ \bigvee_{i=a}^{b} \qquad ((\odot^i \varphi_2) \wedge \textstyle\bigwedge_{j=0}^{i-1} (\odot^j \varphi_1)) \end{aligned}$$

In a hardware implementation, one can make use of shift-registers to store the relevant part of the execution path with regard to the truth values of φ_1 and φ_2. We will proceed by a sample implementation making use of the equivalence above.

Example Consider the ptMTL formula $\varphi_1\ S_{[3,9]}\ \varphi_2$. Rewriting the formula into a hardware implementation, requires two shift registers of length 9 and 8, respectively. With the equivalences from above, $e^n \models \varphi_1\ S_{[3,9]}\ \varphi_2$ can be rewritten into $\bigvee_{i=3}^{9}((\odot^i \varphi_2) \wedge \bigwedge_{j=0}^{i-1}(\odot^j \varphi_1))$, which can be realized by the optimized, hand-crafted circuit shown in Fig. 2. Observe that we do not need to store $\odot^0 \varphi_1$ and $\odot^0 \varphi_2$ explicitly, as they are immediately available. The circuit accounts for 15 two-input AND gates and six two-input OR gates. In a generalized setting, the proposed circuit requires the following resources:

- Shift registers (memory): $2 \times b - 1$
- Two-input AND gates: $2 \times b - a$
- Two-input OR gates: $b - a$

With parameters $a = 5$ and $b = 1500$, the circuit will occupy $3 \times b - 2 \times a = 3 \times 1500 - 2 \times 5 = 4490$ two-input gates, and $2 \times b - 1 = 2 \times 1500 - 1 = 2999$ flip-flops to implement the shift registers, resulting in a huge circuit.

It is important to observe that the chain of AND gates starting at $\odot^0 \varphi_1$ introduces a gate propagation delay [44, Chap. 9] Δ on the signal that is proportional to b and delays the output of the verdict $e^n \models \varphi_1\ S_{[a,b]}\ \varphi_2$. With a propagation delay δ_{AND} of a single AND gate of and an AND chain of length $b-1$, the total propagation delay equals to $\Delta = (b-1) \times \delta_{\text{AND}}$. The chain becomes the *critical path* of the circuit and lowers the achievable operational frequency of the observer design. This effect can be alleviated by introducing a pipeline, however, not without the cost of additional memory and control logic.

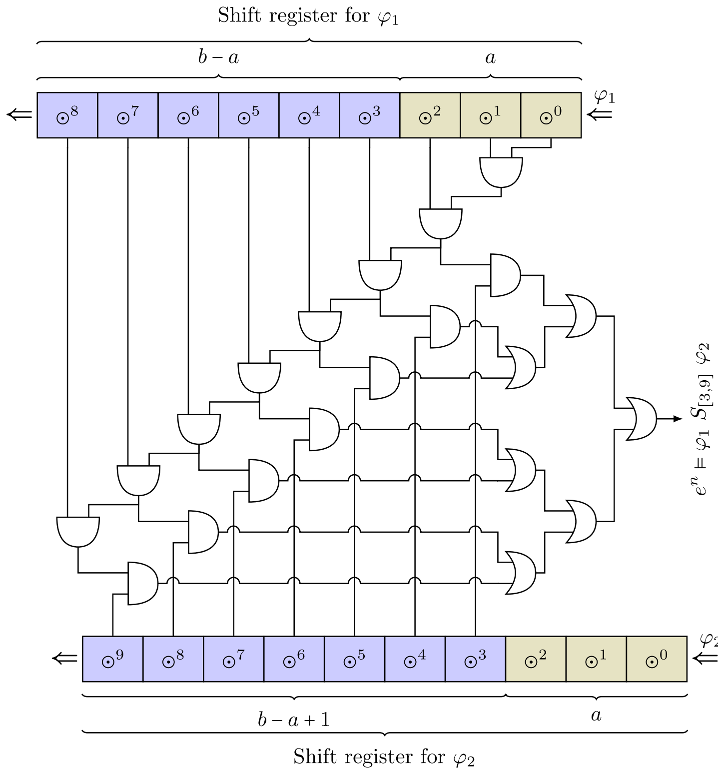


Fig. 2 Hardware realization of a rewriting from $\varphi_1 \ S_{[3,9]} \ \varphi_2$ to $\bigvee_{i=a}^{b}((\odot^i\psi) \wedge \bigwedge_{j=0}^{i-1}(\odot^j\varphi))$. The parameters a and b are set according to the interval in $\varphi_1 \ S_{[3,9]} \ \varphi_2$, i.e., $a=3$ and $b=9$, yielding $\bigvee_{i=3}^{9}((\odot^i\psi) \wedge \bigwedge_{j=0}^{i-1}(\odot^j\varphi))$

This supports that rewriting ptMTL to ptLTL, albeit theoretically possible, is costly and thus infeasible in practice with an application in mind where the satisfaction relation is checked on-the-fly, i.e., in parallel to the SUT. Rewriting, however, may prove feasible when the observer is executed on a powerful host computer with a capable term rewriting engine at hand, as studied in [72].

4 Observer design for real-time properties

In the following, we discuss the formal design of on-line observer algorithms for specifications in ptMTL in a discrete time model. The design is inspired by the observers described in [11] and extends work on observers for ptLTL [42] which have been built in hardware [63, 68]. We first give a high-level definition of the algorithms and turn to a hardware implementation in Sect. 5.

4.1 Decomposing a specification

In the following let $e=(s_t)_{t\geq 0}$ be an execution and φ a ptMTL formula. Further, let $J=[t,t']$, with $t,t' \in \mathbb{N}_0$, be a non-empty interval. An observer is an algorithm that, given input φ and

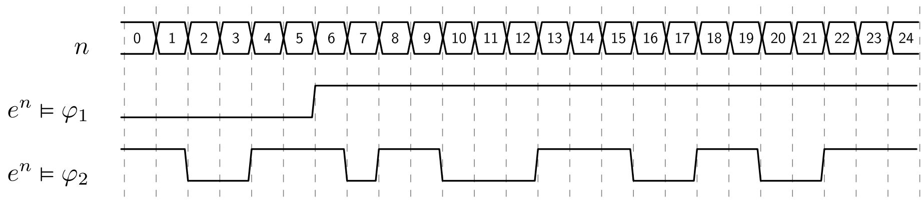


Fig. 3 Validity of $e^n \models \varphi_1$ and $e^n \models \varphi_2$ for prefix of execution e

execution e, at each time $n \in \mathbb{N}_0$, returns true if $e^n \models \varphi$, and false otherwise. We define the return value of our observer algorithm with input φ at time n by structural induction on ptMTL formula φ:

(i) $\varphi = \mathit{true}$ returns true.
(ii) $\varphi = \mathit{false}$ returns false.
(iii) $\varphi = \sigma$, where $\sigma \in \Sigma$ returns true if σ holds on s_n, and false otherwise.
(iv) $\varphi = \varphi_1 \bullet \varphi_2$ is true if $e^n \models \varphi_1 \bullet e^n \models \varphi_2$, where $\bullet \in \{\wedge, \vee, \rightarrow\}$, and false otherwise.
(v) If φ is a ptLTL formula, we apply the algorithms proposed in [41, 42].
(vi) For $\varphi = \varphi_1\ S_J\ \varphi_2$, we collect all times where φ_2 was true in the past and since then φ_1 remained true and store them in a list. At time n we check if there exists a time τ in the list such that $n - \tau \in J$. If such a τ exists we return true, and false otherwise.

Algorithms for cases (i)–(iv) are straightforward. For case (v), we use the algorithm of Havelund and Roşu [41, 42], for which a translation into hardware building blocks (specified in terms of VHDL) is known [68]. Finding an efficient algorithm to detect satisfaction of $e^n \models \varphi_1\ S_J\ \varphi_2$ requires more sophisticated reasoning, and is the topic of the next sections. We start with efficient observer algorithms for the time-bounded variants of the ptLTL modalities $\boxdot\varphi$ and $\diamonddot\varphi$ and later extend them to an efficient observer algorithm for $\varphi_1\ S_J\ \varphi_2$.

Running example In the following, we frequently refer to the execution given in Fig. 3, which describes satisfaction of the two formulas φ_1 and φ_2 over times $n \in [0, 24]$. We say *transition* _|‾ (*resp.* ‾|_) *of* φ *occurs at time* n iff $e^n \models \uparrow\varphi$ in case $n > 0$ and $e^0 \models \varphi$ otherwise (resp. $e^n \models \downarrow\varphi$ in case $n > 0$ and $e^0 \models \neg\varphi$ otherwise). In the running example, transition _|‾ of φ_1 occurs at time 6.

4.2 The invariant and exists previously operators

We first discuss specializations of the common operators $\diamonddot_J$ (exists within interval J) and $\boxdot_J$ (invariant within interval J). In accordance with [6] we define both operators in terms of the Since operator by:

$$\diamonddot_J\ \varphi \equiv \mathsf{true}\ S_J\ \varphi \qquad \boxdot_J\ \varphi \equiv \neg \diamonddot_J \neg\varphi \tag{1}$$

From a practical point of view, two instances of the *exists within interval* and the *invariant within interval* operators, namely *invariant previously* ($\boxbox_\tau$) and *exists previously* ($\diamonddiamond_\tau$), where $\tau \in \mathbb{N}_0$, are valuable. They have the intended meaning *at least once in the past* τ *time units* ($\diamonddiamond_\tau$) respectively *invariant for the past* τ *time units* ($\boxbox_\tau$), and are defined by $\diamonddiamond_\tau \equiv \diamonddot_{[0,\tau]}$ respectively $\boxbox_\tau \equiv \boxdot_{[0,\tau]}$.

For example, $(\uparrow\sigma_1) \rightarrow (\boxbox_{10}\,\sigma_2)$ expresses that whenever σ_1 becomes true, σ_2 holds at all 10 previous time units. For both $\diamonddiamond_\tau$ and $\boxbox_\tau$ we present simplifications that yield space- and time-efficient observers.

Algorithm 1 Observer for $\boxdot_\tau \varphi$. Initially, $m_{\boxdot_\tau \varphi} = \infty$.

1: At each time $n \in \mathbb{N}_0$:
2: **if** ⎍ transition of φ occurs at time n **then**
3: $\quad m_{\boxdot_\tau \varphi} \leftarrow n$
4: **end if**
5: **if** ⎍ transition of φ occurs at time n **then**
6: $\quad m_{\boxdot_\tau \varphi} \leftarrow \infty$
7: **end if**
8: return $\mathsf{valid}^{\boxdot}(m_{\boxdot_\tau \varphi}, \tau, n)$

Invariant previously ($\boxdot_\tau\ \varphi$) is transformed into $\neg(\mathsf{true}\ S_{[0,\tau]}\ \neg\varphi)$ by (1). An observer for $\boxdot_\tau\ \varphi$ requires a single register $m_{\boxdot_\tau \varphi}$ with domain $\mathbb{N}_0 \cup \{\infty\}$. Initially $m_{\boxdot_\tau \varphi} = \infty$. Note that an actual implementation of this observer algorithm clearly must restrict itself to a bounded domain $\{0, 1, \dots, N\} \cup \{\infty\}$, where N is chosen sufficiently large to cover the expected mission time of the system being analyzed. We will discuss implementation considerations of our observers in Sect. 5 and meanwhile assume unbounded domain registers.

For the observer in Algorithm 1, we define predicate $\mathsf{valid}^{\boxdot}(m, \tau, n)$ as:

$$\mathsf{valid}^{\boxdot}(m, \tau, n) \equiv \big(\max(n - \tau, 0) \geq m\big)$$

Intuitively, the predicate $\mathsf{valid}^{\boxdot}(m, \tau, n)$ holds, and thus the algorithm returns true at time n, iff the latest ⎍ transition of φ occurred before $n - \tau$ and no ⎍ transition of φ occurred since then until time n.

Theorem 1 *For all $n \in \mathbb{N}_0$, the observer stated in Algorithm 1 implements $e^n \models \boxdot_\tau\ \varphi$.*

Proof We first observe the equivalences

$$\begin{aligned}
&e^n \models \boxdot_\tau\ \varphi \\
\Leftrightarrow\ &e^n \models \neg(\mathsf{true}\ S_{[0,\tau]}\ \neg\varphi) \\
\Leftrightarrow\ &\forall i (0 \leq i \leq n) : \big(n - i \in [0, \tau] \rightarrow e^i \models \varphi\big) \\
\Leftrightarrow\ &\forall i (0 \leq i \leq n) : \big(i \in n - [0, \tau] \rightarrow e^i \models \varphi\big) \\
\Leftrightarrow\ &\forall i : i \in [0, n] \cap [n - \tau, n] \rightarrow e^i \models \varphi \\
\Leftrightarrow\ &\forall i : i \in \big[\max(0, n - \tau), n\big] \rightarrow e^i \models \varphi.
\end{aligned} \tag{2}$$

Note that interval $[\max(0, n - \tau), n]$ is never empty. Thus equation (2) holds iff a ⎍ transition of φ occurred at a time at most $\max(0, n - \tau)$ and no ⎍ transition of φ occurred since then until time n. The theorem follows. □

Running example Consider $\psi \equiv (\uparrow \varphi_1) \rightarrow (\boxdot_2\, \varphi_2)$ on the execution in Fig. 3. Initially, $m_{\boxdot_2 \varphi_2} = \infty$. At time 0, φ_2 holds and thus $m_{\boxdot_2 \varphi_2} = 0$. The predicate $\mathsf{valid}^{\boxdot}(m_{\boxdot_2 \varphi_2}, 2, 0)$ holds, the algorithm returns true and we have that $e^0 \models \boxdot_2\, \varphi_2$. For similar arguments, at time 1, $e^1 \models \boxdot_2\, \varphi_2$. At time 2, a ⎍ transition of φ_2 occurs and we have $m_{\boxdot_2 \varphi_2} = \infty$. Since predicate $\mathsf{valid}^{\boxdot}(m_{\boxdot_2 \varphi_2}, 2, 2)$ does not hold, we have that $e^2 \not\models \boxdot_2\, \varphi_2$. For similar arguments, at time 3, $e^3 \not\models \boxdot_2\, \varphi_2$. Since a ⎍ transition of φ_2 occurs at time 4, $m_{\boxdot_2 \varphi_2} = 4$.

Again, $\mathsf{valid}^{\boxbox}(m_{\boxbox_2\varphi_2}, 2, 4)$ does not hold, thus, $e^4 \nvDash \boxbox_2\,\varphi_2$. The same is true for time 5, thus, $e^5 \nvDash \boxbox_2\,\varphi_2$. At time 6, $\uparrow\varphi_1$ becomes true and since $\mathsf{valid}^{\boxbox}(m_{\boxbox_2\varphi_2}, 2, 6)$ is true, we deduce $e^6 \vDash \psi$. For times n' prior to 6, (i.e., $0 \le n' < 6$), the left-hand side of the implication of ψ does not hold. We immediately have that $e^{n'} \vDash \psi$.

Exists previously ($\diamonddiamond_\tau\,\varphi$) From the equivalence $\diamonddiamond_\tau\,\varphi \equiv \neg \boxbox_\tau \neg\varphi$, we can immediately derive an observer for $\diamonddiamond_\tau\,\varphi$ from the observer for $\boxbox_\tau\,\varphi$. The resulting algorithm can straightforwardly be implemented by checking for a $\urcorner\!\llcorner$ (resp. $\lrcorner\!\ulcorner$) transition of φ instead of a $\lrcorner\!\ulcorner$ (resp. $\urcorner\!\llcorner$) transition of $\neg\varphi$ in line 2 (resp. line 5) and negating the output in line 8.

4.3 The invariant and exists within interval operators

We now present observers for the more general operators *invariant within interval J* ($\boxdot_J$) and *exists within interval J* ($\diamonddot_J$). Instead of a register (such as $m_{\boxbox_\tau\varphi}$ in case of the observer for $\boxbox_\tau\,\varphi$), both observers require a list of time point pairs. Clearly, an efficient implementation of this list is vital for an efficient observer. In the following, we present several techniques so as to keep the list succinct, whilst preserving validity of the observer. For a list l, we denote by $|l|$ its length, and by $l[k]$, where $k \in \mathbb{N}$, its kth element. We assume that elements are always appended to the tail of a list.

Invariant within interval ($\boxdot_J\,\varphi$) is transformed into $\neg(\mathsf{true}\ S_J\ \neg\varphi)$ by (1). An observer for $\boxdot_J\,\varphi$ requires a list $l_{\boxdot_J\varphi}$ of elements from $(\mathbb{N}_0 \cup \{\infty\})^2$. For a pair of time points $T \in (\mathbb{N}_0 \cup \{\infty\})^2$, we shortly write $T.\tau_s$ for its first component and $T.\tau_e$ for its second component. Initially, $l_{\boxdot_J\varphi}$ is empty. For the observer in Algorithm 2, we define predicate $\mathsf{valid}^{\boxdot}(T, n, J)$, with $T \in (\mathbb{N}_0 \cup \{\infty\})^2$, by:

$$\mathsf{valid}^{\boxdot}(T, n, J) \equiv \big(T.\tau_s \le \max\big(0, n - \max(J)\big)\big) \wedge \big(T.\tau_e \ge n - \min(J)\big),$$

and predicate $\mathsf{feasible}(T, n, J)$ as:

$$\mathsf{feasible}(T, n, J) \equiv \big(T.\tau_e - T.\tau_s \ge \mathrm{len}(J)\big) \vee \big(T.\tau_s = 0 \wedge T.\tau_e \ge n - \min(J)\big).$$

Intuitively, Algorithm 2 keeps track of all maximal intervals where φ holds whose length is large enough to potentially lead to the satisfaction of $\boxdot_J\,\varphi$. Whether this is the case is determined by the fact whether a tuple representation of an interval satisfies the $\mathsf{feasible}$ predicate. For large n, this means that an interval has to have length at least $\mathrm{len}(J)$.

We will deduce the correctness of the observer stated in Algorithm 2 from the correctness of a generalized algorithm, presented in Sect. 4.4, obtaining:

Theorem 2 *For all $n \in \mathbb{N}_0$, the observer stated in Algorithm 2 implements $e^n \vDash \boxdot_J\,\varphi$.*

Running example Consider $\psi \equiv (\uparrow\varphi_1) \to (\boxdot_{[3,4]}\varphi_2)$ and execution e of Fig. 3. At time 0, the element $(0, \infty)$ is inserted into $l_{\boxdot_{[3,4]}\varphi_2}$. The $\urcorner\!\llcorner$ transition of φ_2 at time 2 then leads to $l_{\boxdot_{[3,4]}\varphi_2} = ((0, 1))$, since $\mathsf{feasible}((0, 1), 2, [3, 4])$ holds. At time 4, another pair is added, resulting in $l_{\boxdot_{[3,4]}\varphi_2} = ((0, 1), (4, \infty))$. Since at time 6:

$$\begin{aligned}
&\mathsf{valid}^{\boxdot}\big(l_{\boxdot_{[3,4]}\varphi_2}[1], 6, [3, 4]\big) \Leftrightarrow (0 \le 6 - 4) \wedge (1 \ge 6 - 3) \quad \Leftrightarrow \mathsf{false}\\
&\mathsf{valid}^{\boxdot}\big(l_{\boxdot_{[3,4]}\varphi_2}[2], 6, [3, 4]\big) \Leftrightarrow (4 \le 6 - 4) \wedge (\infty \ge 6 - 3) \Leftrightarrow \mathsf{false}
\end{aligned}$$

we obtain $e^6 \nvDash \psi$.

Algorithm 2 Observer for $\boxdot_J \varphi$. Initially, $l_{\boxdot_J \varphi} = ()$.

```
1:  At each time n ∈ ℕ0:
2:  if _⌈ transition of φ occurs at time n then
3:     add (n, ∞) to l_{⊡Jφ}
4:  end if
5:  if ⌉_ transition of φ occurs at time n and l_{⊡Jφ} is non-empty then
6:     remove tail element (τs, ∞) from l_{⊡Jφ}
7:     if feasible((τs, n − 1), n, J) then
8:        add (τs, n − 1) to l_{⊡Jφ}
9:     end if
10: end if
11: return ⋁_{k=1}^{|l_{⊡Jφ}|} valid^⊡(l_{⊡Jφ}[k], n, J) in case n ≥ min(J) and true otherwise
```

Exists within interval ($\diamondminus_J \varphi$) From the equivalence $\diamondminus_J \varphi \equiv \neg \boxdot_J \neg\varphi$, we can easily derive an observer for $\diamondminus_J \varphi$ from the observer for $\boxdot_J \varphi$. As before, we obtain the observer by swapping ⌉_ and _⌈ transitions and negating the output.

4.4 The since within interval operator

An observer for $\varphi_1 \; S_J \; \varphi_2$ is obtained from a $\diamondminus_J$ observer and additional logic to reset the observer's list. Let l_S be an initially empty list. The $\varphi_1 \; S_J \; \varphi_2$ observer is stated in Algorithm 3. In case φ_1 holds at time n, the observer executes the same code as a $\diamondminus_J \varphi_2$ observer. In case φ_1 does not hold at time n, the list $l_{\varphi_1 S_J \varphi_2}$ is reset to contain only a single entry whose content depends on the validity of φ_2. Intuitively, for the maximum suffix where φ_1 holds Algorithm 3 keeps track of all maximal intervals where φ_2 holds whose length is large enough to potentially lead to the satisfaction of $\varphi_1 \; S_J \; \varphi_2$.

Theorem 3 *For all $n \in \mathbb{N}_0$, the observer in Algorithm 3 implements $e^n \models \varphi_1 \; S_J \; \varphi_2$.*

For the proof we introduce additional notation. For list l denote with $l \cdot T$, the list resulting from adding element T to the tail of list l. Further denote with l^n, where $n \in \mathbb{N}_0$, the state of Algorithm 3's list l_S in line 19 executed at time n. By $\bar{l}^n$ we denote the set $[0, n] \setminus \bigcup_{1 \le k \le |l|} [l[k].\tau_s, l[k].\tau_e + 1)$. For example, if $l^{10} = ((0,3),(5,8))$, then $\bar{l}^{10} = \{4, 9, 10\}$. We first show that the following proposition holds:

Proposition 1 *Consider Algorithm 3 without the feasibility check in line 8, i.e., replace this line with "if* true *then". For the modified algorithm the following is correct: For all $n \in \mathbb{N}_0$ and $i \le n$, $i \in \bar{l}^n$ holds iff both $e^i \models \varphi_2$ and for all k, $i < k \le n$, $e^k \models \varphi_1$.*

Proof The proof is by induction on $n \in \mathbb{N}_0$.

Begin ($n = 0$): Consider the four cases for φ_1 and φ_2:

Case (i): Assume $e^n \models \varphi_1$ and $e^n \not\models \varphi_2$. Then $l^n = ((0, \infty))$ and thus $\bar{l}^n = \varnothing$. Since $e^n \not\models \varphi_2$, the induction basis follows in this case.

Case (ii): Assume $e^n \models \varphi_1$ and $e^n \models \varphi_2$. Then $l^n = ()$ and thus $\bar{l}^n = \{0\}$. Since $e^n \models \varphi_2$, the induction basis follows in this case.

Algorithm 3 Observer for $\varphi_1\ S_J\ \varphi_2$. Initially, $l_{\varphi_1 S_J \varphi_2} = ()$.

1: At each time $n \in \mathbb{N}_0$:
2: **if** φ_1 holds at time n **then**
3:     **if** ⌝⌞ transition of φ_2 occurs at time n **then**
4:         add (n, ∞) to $l_{\varphi_1 S_J \varphi_2}$
5:     **end if**
6:     **if** ⌟⌜ transition of φ_2 occurs at time n **and** $l_{\varphi_1 S_J \varphi_2}$ is non-empty **then**
7:         remove tail element (τ_s, ∞) from $l_{\varphi_1 S_J \varphi_2}$
8:         **if** feasible$((\tau_s, n-1), n, J)$ **then**
9:             add $(\tau_s, n-1)$ to $l_{\varphi_1 S_J \varphi_2}$
10:         **end if**
11:     **end if**
12: **else**
13:     **if** φ_2 holds at time n **then**
14:         set $l_S = ((0, n-1))$ in case $n \neq 0$ and $l_S = ()$ otherwise
15:     **else**
16:         set $l_S = ((0, \infty))$
17:     **end if**
18: **end if**
19: return $\neg(\bigvee_{k=1}^{|l_{\varphi_1 S_J \varphi_2}|} \mathsf{valid}^{\boxdot}(l_{\varphi_1 S_J \varphi_2}[k], n, J))$ in case $n \geq \min(J)$ and false otherwise

Case (iii): Assume $e^n \nvDash \varphi_1$ and $e^n \nvDash \varphi_2$. The arguments are analogous to the arguments of case (i).

Case (iv): Assume $e^n \nvDash \varphi_1$ and $e^n \vDash \varphi_2$. The arguments are analogous to the arguments of case (ii).

Step $(n-1 \to n)$: Assume that the statement holds for $n-1 \geq 0$. We will show that it holds for n, too. Thereby we consider the same cases (i) to (iv) as in the induction basis.

Case (i): We distinguish two cases for φ_2: a ⌝⌞ transition of φ_2 (i.a) did, or (i.b) did not occur at time n.

In case of (i.a), $l^n = l^{n-1} \cdot (n, \infty)$. Thus $\overline{l}^n = \overline{l}^{n-1}$. Since $e^n \vDash \varphi_1$ but $e^n \nvDash \varphi_2$, the induction step follows in this case.

In case of (i.b), $l^n = l^{n-1}$. By the algorithm, the last element in l^n must be of the form (n', ∞) with $n' < n$. Thus $\overline{l}^n = \overline{l}^{n-1}$. Again, the induction step follows in this case.

Case (ii): We distinguish two cases for φ_2: a ⌝⌞ transition of φ_2 (ii.a) did, or (ii.b) did not occur at time n.

Now consider case (ii.a): If $l^{n-1} = ()$, $l^n = l^{n-1}$ holds, and thus $\overline{l}^n = \overline{l}^{n-1} \cup \{n\}$. Otherwise, the last element in l^{n-1}, say (n', ∞), with $n' \leq n$, is replaced with (n', n) in l^n. Again, $\overline{l}^n = \overline{l}^{n-1} \cup \{n\}$. In both cases, the induction step follows, as $e^n \vDash \varphi_1$ and $e^n \vDash \varphi_2$.

In case of (ii.b), $l^n = l^{n-1}$. By the algorithm, the last element in l^n, if it exists, must be of the form (n', n'') with $n' \leq n'' < n$. Thus $\overline{l}^n = \overline{l}^{n-1} \cup \{n\}$. Again, the induction step follows in this case.

Case (iii): By the algorithm, $l^n = ((0, \infty))$. Thus $\overline{l}^n = \emptyset$. Since $e^n \nvDash \varphi_2$, the induction step follows in this case.

Case (iv): By the algorithm, and since $n > 0$, $l^n = ((0, n-1))$. Thus $\overline{l}^n = \{n\}$. Since $e^n \models \varphi_1$, the induction step follows in this case. □

We are now in the position to prove Theorem 3.

Proof of Theorem 3 Consider the modified Algorithm 3 without feasibility check. By analogous arguments as in the proof of Theorem 1, we obtain

$$\begin{aligned}
&e^n \models \varphi_1 \, S_J \, \varphi_2 \\
&\Leftrightarrow \forall i : i \in [0, n] \cap [n - \max(J), n - \min(J)] \wedge (e^i \models \varphi_2) \wedge \forall k (i < k \leq n) : e^k \models \varphi_1 \\
&\Leftrightarrow \forall i : i \in [\max(0, n - \max(J)), n - \min(J)] \wedge (e^i \models \varphi_2) \wedge \forall k (i < k \leq n) : e^k \models \varphi_1 .
\end{aligned}$$

We distinguish two cases for n, namely (i) $n < \min(J)$, and (ii) $n \geq \min(J)$.

(i) In case $n < \min(J)$, interval $[\max(0, n - \max(J)), n - \min(J)]$ is empty, and $e^n \models \varphi_1 \, S_J \, \varphi_2$ is trivially false. Since the algorithm returns false in this case, the theorem follows for Algorithm 3 without the feasibility check for case (i).

(ii) In case $n \geq \min(J)$, interval $I = [\max(0, n - \max(J)), n - \min(J)]$ is non-empty. Thus $e^n \models \varphi_1 \, S_J \, \varphi_2$ holds iff there exists an $i \in I$ for which $e^i \models \varphi_2$ and for all $k, i < k \leq n$, $e^k \models \varphi_1$. From Proposition 1 we know that this is the case iff there exists an $i \in I$ with $i \in \overline{l}^n$. The latter is the case iff there exists no tuple (τ_s, τ_e) in l^n with $\mathsf{valid}^{\boxdot}((\tau_s, \tau_e), n, J)$. Since, for $n \geq \min(J)$, the algorithm returns true iff this is the case, the theorem follows for Algorithm 3 without the feasibility check for case (ii).

It remains to show that the theorem holds for Algorithm 3 with original line 8. If we can show that from $\neg\mathsf{feasible}((\tau_s, \tau_e), n, J)$ follows $\neg\mathsf{valid}^{\boxdot}((\tau_s, \tau_e), n', J)$, for all times $n' \geq n$, we may safely remove tuple (τ_s, τ_e) from the algorithm's list without changing the algorithm's return value.

Assume that $\mathsf{valid}^{\boxdot}((\tau_s, \tau_e), n', J)$ holds, with $n' \geq n$. We distinguish two cases for n': (a) $n' < \max(J)$ and (b) $n' \geq \max(J)$:

(a) In case $n' < \max(J)$, it follows from $\mathsf{valid}^{\boxdot}((\tau_s, \tau_e), n', J)$ that $T.\tau_s = 0$ and $T.\tau_e \geq n' - \min(J) \geq n - \min(J)$. Thus $\mathsf{feasible}((\tau_s, \tau_e), n, J)$ holds.

(b) Otherwise $n' \geq \max(J)$, and it follows from $\mathsf{valid}^{\boxdot}((\tau_s, \tau_e), n', J)$ that $T.\tau_s \leq n' - \max(J)$ and $T.\tau_e \geq n' - \min(J)$. Thus $T.\tau_e - T.\tau_s \leq \mathrm{len}(J)$ and thereby $\mathsf{feasible}((\tau_s, \tau_e), n, J)$.

The theorem follows. □

With the two definitions in (1), an observer algorithm implementing $e^n \models \boxdot_J \varphi$ can be deduced from Algorithm 3 by negating its input, its output, and replacing the if condition in line 2 by true. Since the obtained algorithm is equivalent to Algorithm 2, Theorem 2 immediately follows.

4.5 Garbage collection

Thus far, we did not consider housekeeping of either list so as to control the growth of the lists. It is important to appreciate that each timed operator has a bounded time-horizon on which it depends. This horizon can be exploited to eliminate pairs T from Algorithm 2 or Algorithm 3's lists that can neither validate nor invalidate the specification. Our garbage

collector works as follows: at any time $n \in \mathbb{N}_0$, we remove a tuple T from the list if the proposition

$$\mathsf{garbage}(T, n, J) \equiv T.\tau_e < n - \min(J)$$

holds. The main purpose of the garbage collector is to reduce the algorithms' space and time complexity: We will show that, by removing tuples, garbage collection considerably reduces the algorithms' space complexity. Further, observe that direct implementations of line 11 of Algorithm 2 and line 19 of Algorithm 3 require searches through a list. We will show that, with our garbage collector running in parallel to the observer algorithms, these lines reduce to checking the list's first element only. Thus we may replace the list in both algorithms by a simple queue, where elements are added only to its tail and read and removed only at its head.

In the following, we show the correctness of our garbage collection strategy for any of the proposed algorithms: We first show that if a tuple T is allowed to be removed by the garbage collector at time n, it cannot satisfy $\mathsf{valid}^{\boxdot}$ at that time or at any later time. It is thus safe to remove it from the list.

Lemma 1 *If* $\mathsf{garbage}(T, n, J)$, *then* $\neg\mathsf{valid}^{\boxdot}(T, n', J)$ *for all* $n \geq n'$.

Proof Assume that $\mathsf{garbage}(T, n, J)$ holds. Then $T.\tau_e < n - \min(J) \leq n' - \min(J)$. Since $T.\tau_e \geq n' - \min(J)$ is necessary for $\mathsf{valid}^{\boxdot}(T, n', J)$ to hold, the lemma follows. □

We next show that always a prefix of a list is removed. This allows the garbage collector to evaluate $\mathsf{garbage}$ iteratively, starting from the head of the list.

For that purpose we introduce additional notation. We write "..." for a potentially empty sequence of tuples. For example, $(\ldots, T, T', \ldots)$ denotes a list of length at least two, where T and T' are any two successive elements in this list.

Lemma 2 *Let* $l = (\ldots, T, T', \ldots)$ *be the list of any of the proposed observer algorithms at time* $n \in \mathbb{N}_0$. *If* $\mathsf{garbage}(T', n, J)$, *then* $\mathsf{garbage}(T, n, J)$.

Proof Assume that $\mathsf{garbage}(T', n, J)$ holds. Then $T'.\tau_e < n - \min(J)$. By observing that all of the proposed algorithms ensure that $T.\tau_e \leq T'.\tau_e$ for successive list elements T and T', we obtain $T.\tau_e < n - \min(J)$, i.e., $\mathsf{garbage}(T, n, J)$ holds. The lemma follows. □

We next prove an upper bound on the length of Algorithm 2 or Algorithm 3's lists. We start by showing that there is a minimum distance between successive elements in the algorithms' lists.

Lemma 3 *Let* $l = (\ldots, T, T', \ldots)$ *be the list of any of the proposed observer algorithms at time* $n \in \mathbb{N}_0$. *Then* $T.\tau_e + 2 \leq T'.\tau_s$.

Proof Consider Algorithm 2. By the algorithm, tuple T must have been added by line 8. For line 8 to add $T = (T.\tau_s, n - 1)$, transition $\urcorner\!\llcorner$ of φ must have occurred at time n. Thus the next tuple added to the list at a time $n' > n$ must have been of the form (n', ∞). Since, by the algorithm, then $T'.\tau_s \geq n'$ must hold, we further obtain $T'.\tau_s \geq (n - 1) + 2 = T.\tau_e + 2$. The lemma follows for Algorithm 2.

For Algorithm 3 the lemma follows by analogous arguments. □

Further the first element in the list that was not removed by the garbage collector cannot be of arbitrary age:

Lemma 4 *Consider a time-bounded formula* $\boxdot_J \varphi$, $\diamondsuit_J \varphi$, *or* $\varphi_1 \ S_J \ \varphi_2$. *Let* $l = (T, \dots)$ *be the list of the proposed respective observer algorithm at time* $n \in \mathbb{N}_0$, *after garbage collection has run at time* n. *Then* $T.\tau_e \geq n - \min(J)$.

Proof It must hold that $\mathsf{garbage}(T, n, J)$ is false, since otherwise T would have been removed by the garbage collector. Thus $T.\tau_e \geq n - \min(J)$. □

Lemma 5 *Let* l *be the list of any of the proposed observer algorithms at time* $n \in \mathbb{N}_0$, *after garbage collection has run at time* n, *and assume that* l *is non-empty. Let* $T^k = \ell[k]$, *for* $1 \leq k \leq |\ell|$. *Then* $T^k.\tau_e \geq n - \min(J) + (k-1)(2 + \mathrm{len}(J))$.

Proof The proof is by induction on the number $k \geq 1$ of the element in the list.

Begin ($k = 1$): Immediately follows from Lemma 4.

Step ($k - 1 \to k$): Assume that the statement holds for $k - 1 \geq 1$. We will show that it holds for k, too. By Lemma 3,

$$T^k.\tau_s \geq T^{k-1}.\tau_e + 2.$$

Because $k > 1$, it must hold that $T^k.\tau_s \neq 0$. Thus, by the algorithms, either $\mathsf{feasible}(T^k, n', J)$ must have held at time $n' \leq n$, when T^k was added to the list, or $T^k = (n', \infty)$. In both cases,

$$T^k.\tau_e \geq T^k.\tau_s + \mathrm{len}(J).$$

It follows that,

$$T^k.\tau_e \geq T^{k-1}.\tau_e + 2 + \mathrm{len}(J). \tag{3}$$

Combining (3) and the induction hypothesis

$$T^{k-1}.\tau_e \geq n - \min(J) + (k-2)\big(2 + \mathrm{len}(J)\big)$$

thus yields,

$$T^k.\tau_e \geq n - \min(J) + (k-1)\big(2 + \mathrm{len}(J)\big).$$

The lemma follows. □

We may now derive an upper bound on the number of list elements for all our observer algorithms:

Theorem 4 *Consider a time-bounded formula* $\boxdot_J \varphi$, $\diamondsuit_J \varphi$, *or* $\varphi_1 \ S_J \ \varphi_2$. *Let* l *be the list of the proposed respective observer algorithm at time* $n \in \mathbb{N}_0$, *after garbage collection has run at time* n. *Then* l *is of length at most*

$$\frac{2\max(J) - \min(J) + 2}{2 + \mathrm{len}(J)}.$$

Proof In case l is empty the lemma follows trivially. Assume $l = (T^1, \dots, T^k)$ is non-empty. We distinguish two cases for T^k:

(i) In case $T^k.\tau_e \neq \infty$, we obtain from Lemma 5,

$$T^k.\tau_e \geq n - \min(J) + (k-1)\big(2 + \mathrm{len}(J)\big). \tag{4}$$

Further, by the algorithms, a finite $T^k.\tau_e$ implies that

$$T^k.\tau_e \leq n - 1. \tag{5}$$

Combination of (4) and (5) yields

$$n - 1 \geq n - \min(J) + (k-1)\big(2 + \mathrm{len}(J)\big) \Leftrightarrow$$
$$k \leq \frac{\max(J) + 1}{2 + \mathrm{len}(J)} \leq \frac{2\max(J) - \min(J) + 2}{2 + \mathrm{len}(J)}.$$

The theorem follows for this case.

(ii) Otherwise, i.e., in case $T^k.\tau_e = \infty$, by the algorithms,

$$T^k.\tau_s \leq n \tag{6}$$

must hold. We obtain from Lemma 5,

$$T^{k-1}.\tau_e \geq n - \min(J) + (k-2)\big(2 + \mathrm{len}(J)\big),$$

and by Lemma 3,

$$T^k.\tau_s \geq n - \min(J) + 2 + (k-2)\big(2 + \mathrm{len}(J)\big). \tag{7}$$

Combination of (6) and (7) yields

$$n \geq n - \min(J) + 2 + (k-2)\big(2 + \mathrm{len}(J)\big) \Leftrightarrow$$
$$k \leq \frac{2\max(J) - \min(J) + 2}{2 + \mathrm{len}(J)}.$$

The theorem also follows for this case. □

4.6 Discussion of space and time complexity

We first give a bound on space complexity in terms of single-bit registers that are required by a hardware implementation of our observer algorithms. Clearly, the space complexity for an observer of ptMTL formula φ is the sum of the space complexity of its observers for all subformulas of φ, and its time complexity scales with the depth of the parse tree of φ. It is thus sufficient to state bounds for $\boxdot_J \varphi$, $\diamondminus_J \varphi$, and $\varphi_1 \, S_J \, \varphi_2$. In all these cases the respective observer algorithm's space complexity is dominated by the space complexity of the algorithm's list. Clearly the bit complexity of the τ_s or τ_e component of a tuple added by one of the proposed algorithms to its list before time $n \in \mathbb{N}_0$ is bounded by $\lceil \log_2(n) \rceil$. We thus obtain from Theorem 4 that for any of the time-bounded formulas $\boxdot_J \varphi$, $\diamondminus_J \varphi$, or

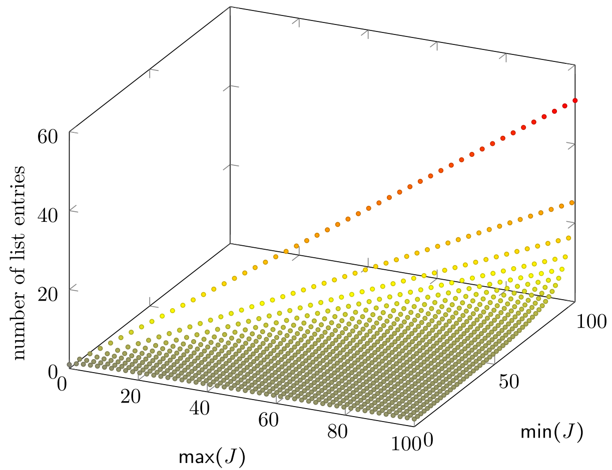


Fig. 4 Visualization of the space complexity bound $(2 \cdot \max(J) - \min(J) + 2)/(2 + \text{len}(J))$ for $0 \leq \min(J) \leq \max(J) \leq 100$ with 50 samples per axis

$\varphi_1 \, S_J \, \varphi_2$, our proposed observer algorithms, if executed at time $n \in \mathbb{N}_0$, have to maintain a list of space complexity at most:

$$2\lceil \log_2(n) \rceil \cdot \frac{2\max(J) - \min(J) + 2}{2 + \text{len}(J)}. \tag{8}$$

Figure 4 visualizes this bound, revealing that memory consumption is moderate for almost all cases, except for configurations where $\min(J) = \max(J)$, where space complexity grows linear in $\max(J)$. Note that $\log_2(n)$ is small for realistic experimental setups. For example, allowing to store 52 bit per tuple component is sufficient to check executions that are sampled with a 1 MHz clock during a period of over 140 years.

An alternative to storing absolute times in the observer's list, is to adapt the observer algorithms in a way such that only relative times are stored. While this potentially reduces the bound of Eq. (8) by substituting $\log_2(n)$ with $\log_2(\max(J))$, it requires updating of the list elements (as these then contain relative times) at every time $n \in \mathbb{N}_0$. Since this would require more complex hardware mechanism and result in a slower on-line algorithm, we decided not to follow this path in our hardware implementation.

We next show that garbage collection allows one to reduce time complexity of the proposed observers. The time-determining part of Algorithms 2 and 3 is the evaluation of the predicate $\mathsf{valid}^{\boxdot}$ for all list elements in line 11 and line 19 respectively. However, garbage collection makes it possible to only evaluate the predicate for the first element in the list, thus greatly improving time complexity of the proposed algorithms:

Lemma 6 *Let $l = (T, \ldots, T', \ldots)$ be the list of any of the observer algorithms at time $n \in \mathbb{N}_0$, after garbage collection has run at time n. Then $\neg\mathsf{valid}^{\boxdot}(T', n, J)$.*

Proof Assume by means of contradiction that $\mathsf{valid}^{\boxdot}(T', n, J)$ holds. Then $T'.\tau_s \leq \max(0, n - \max(J)) \leq \max(0, n - \min(J))$. For both Algorithms 2 and 3 we observe that $T.\tau_e < T'.\tau_s$ has to hold. Thus $T.\tau_e < \max(0, n - \min(J))$. Since neither Algorithms 2 nor 3 add tuples with a negative τ_s or τ_e component, we obtain that $T.\tau_e < n - \min(J)$ has to hold and by that $\mathsf{garbage}(T, n, J)$ holds. A contradiction to the fact that garbage collection has been run at time n: it would have removed tuple T in that case. The lemma follows. □

Since further there exist circuits that perform an addition of two integers of bit complexity $w \in \mathbb{N}$ within time $\mathcal{O}(\log_2(w))$ [47], and since evaluating the $\mathsf{valid}^{\boxdot}(T, n, J)$ and $\mathsf{garbage}(T, n, J)$ predicates at time $n \in \mathbb{N}_0$ requires addition of integers of bit complexity at most $\max(\log_2(n), \log_2(J))$, we arrive at an asymptotic time complexity of

$$\mathcal{O}\big(\log_2 \log_2 \max\big(J \cup \{n\}\big)\big),$$

for any of the observers $\boxdot_J \varphi$, $\diamondminus_J \varphi$, and $\varphi_1 \; S_J \; \varphi_2$ executed at time n.

5 Mapping the framework into hardware structures

In what follows, we elaborate design considerations to map the proposed runtime verification framework into hardware. Figure 5 shows the main modules of a hardware instance of the framework, i.e., the runtime verification unit (RVU). The design of the RVU is generic and can be attached to various SUTs, as shown in Fig. 1. We start with a discussion of how our RVU connects to existing systems and how we map registers and lists into primitive hardware structures. We then show how we derive the current time from a Real-Time Clock (RTC) and how we evaluate atomic propositions, before we show how to adapt an existing low-footprint, programmable ptLTL verification microprocessor to also evaluate ptMTL specifications using the observer algorithms described in Sect. 4.

5.1 Interfacing the system under test

Our runtime verification unit (see Fig. 5) connects to various systems through wiretapping of the SUT's communication interfaces, as outlined in Fig. 1. The attachment to these communication interfaces is application specific. In its current shape, we implemented bus interfaces for systems operating with: RS-232 (serial port), CAN (vehicle bus), Wishbone (System-on-Chip interconnect), I^2C (multimaster serial bus), and JTAG (boundary scan) variants.

5.2 Registers and lists of pairs of time points

Registers are implemented by, for example, linking multiple flip-flops. The width of such a register equals to the width of the (upper bounded) time points issued by the RTC plus two additional bits. These additional bits enable indication of overflows when performing arithmetics on time points and indication of the special value ∞. For lists of pairs of time points, we turn to block RAMs, which we organize as ring buffers. Each ring buffer is managed by a unit that controls its read pointer (RP) and its write pointer (WP).

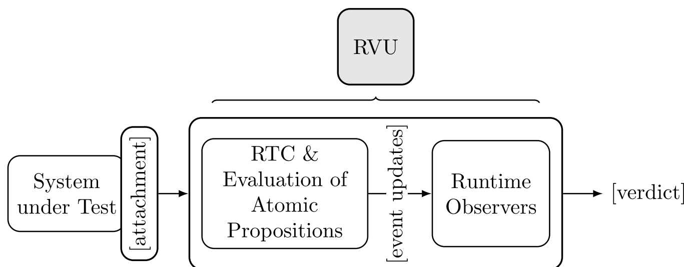


Fig. 5 The runtime verification unit (RVU) and its architecture

5.3 Real-time clock

The progression of time is measured by a digital clock, i.e., the real-time-clock (RTC), which contains a counter and an oscillation mechanism that periodically increments the counter [48, Chap. 3]. For an on-chip RVU solution, the oscillation mechanism can also be bounded to the global system clock of the SUT. Note that the design also allows for an instantiation of a fully external clock which is decoupled from the SUT, such as a GPS receiver. Time points are internally stored in registers of width $w = \lceil \log_2(N) \rceil + 2$, where N is the maximum time (in terms of ticks of the RTC) expected to occur during a run of the SUT. The two additional bits enable indication of overflows when performing arithmetical operations on time points and indication of ∞.

Note that our proposed algorithms (cf. Sect. 4) make use of absolute time points, i.e., we store time points for both _|‾ and ‾|_ transitions of an event e. In contrary, we could also use a mixed representation of absolute and relative time points, i.e., store the absolute time points of the _|‾ transition of event e and then count the duration of e (the number of clock ticks until the ‾|_ transition occurs). While the latter would help to improve the average-case memory requirements in a software-oriented implementation, the former is superior in terms of a hardware implementation: In a hardware design, memory needs to be statically assigned at design time; thus registers have to be of width w rendering the benefits of relative time points. Further storing relative time points would require an additional counter of width w for all atomic propositions and subformulas that use time points.

5.4 Evaluation of atomic propositions

Ideally, with respect to expressiveness of the supported specifications, atomic propositions include arbitrary equalities, inequalities, and disequalities over variables in the state of the SUT. To arrive at a responsive framework, however, an observer needs to guarantee that it finishes evaluation of atomic propositions within a tight time bound. It is therefore necessary to establish a balance between (hardware) complexity of the resulting observer and expressiveness. To achieve this balance, we restrict the class of atomic propositions supported by our framework in a way inspired by the so-called logahedron abstract domain [45], frequently used in the field of abstract interpretation [24].

Specifically, the class of supported atomic propositions consist of conjunctions of linear constraints, where each constraint ranges over two variables. In addition, each variable can be negated and multiplied by a power of two. In our implementation, we support atomic propositions that are restricted linear constraints ranging over values transferred through an interface of the SUT. Specifically, atomic propositions are of the form $(\pm 2^n \cdot v_1 \pm 2^m \cdot v_2) \bowtie c$, where v_1 and v_2 are application specific symbols, $c, n, m \in \mathbb{Z}$ and $\bowtie \in \{=, \neq, \leq, \geq, >, <\}$. For example, when the RVU is connected to a microcontroller data bus (cf. Fig. 1), v_1 (and v_2) can be interpreted as the value stored in a memory location, which in turn, maps to a program variable.

In [68, Sect. 3] we showed how to build circuits (see Fig. 6) that evaluate such linear constraints, with a minimum time penalty. We will use the term AtChecker to refer to such a circuit. It comprises an operands register to fetch new data from the SUT interface, two shifter units to implement multiplication and division by a power of two, an arithmetic unit (i.e., an adder) and a comparator stage. For every atomic proposition of the ptMTL formula, one such unit is instantiated. To evaluate the hardware requirements of AtChecker units, we synthesized the respective circuits with the industrial logic synthesis tool ALTERA QUARTUS II for an Altera Cyclone IV EP4CE115 FPGA device. A single AtChecker unit consumes 290 logic elements (0.25 % of the available logic elements) and can run with a clock frequency of up to $f_{max} = 128$ MHz.

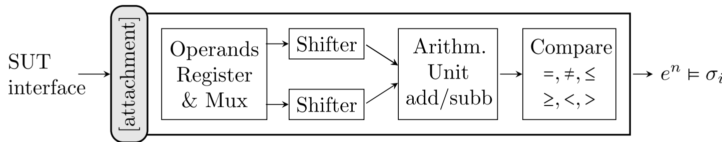


Fig. 6 An AtChecker unit to evaluate an atomic proposition σ_i in hardware

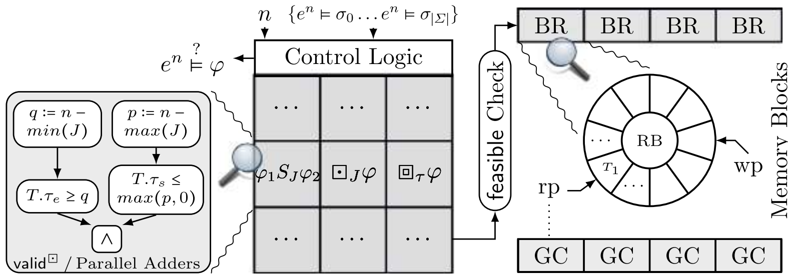


Fig. 7 Hardware runtime observers for ptMTL specifications; abbreviations: garbage collector (GC), block ram (BR), ring buffer (RB), read pointer (rp), and write pointer (wp)

Example Consider the ptMTL formula $\varphi = (\uparrow (2 \cdot v_1 + v_2 \leq 68)) \rightarrow (\boxdot_{[5,10]}(4 \cdot v_3 = 20 \vee v_4 = 40))$. Assume that the runtime verification framework is instantiated as shown in the top-right part of Fig. 1, i.e., it monitors a microcontroller core. The atomic propositions $\{\sigma_1, \sigma_2, \sigma_3\}$ of φ are: $\sigma_1 \equiv (2 \cdot v_1 + v_2 \leq 68)$, $\sigma_2 \equiv (4 \cdot v_3 = 20)$, and $\sigma_3 \equiv (v_4 = 40)$. The symbols $v_1, \dots, v_4$ relate to memory locations stored in the microcontroller RAM. Together with debug information from the compiler they can be linked to high-level language symbols, e.g., C code variables. Evaluating $\{\sigma_1, \sigma_2, \sigma_2\}$ requires three AtChecker blocks. For example, to evaluate σ_1, an AtChecker is configured to load new data from the SUT interface as soon as new values for either v_1 or v_2 are transferred. Its shifter is programmed to shift v_1 one position to the left and the arithmetic unit so as to calculate the sum of $2 \cdot v_1$ and v_2. The comparator then compares this result with the constant 68 and finally outputs the truth value of σ_1 at the current time point n.

5.5 Runtime observers

Figure 7 shows the hardware architecture to evaluate ptMTL operators. A pool of statically synthesized hardware observers is interconnected by a control logic to resemble the parse tree of the specification φ. For each operator we use Theorem 4 to statically assign sufficient memory to it.

Evaluating the observer algorithms' predicates Subtraction and relational operators as required by the predicates feasible, garbage, and valid can be built around adders. Observe that, when $\mathsf{Add}(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle, \mathsf{c})$ is a ripple carry adder for arbitrary length unsigned vectors $\langle \mathsf{a} \rangle$ and $\langle \mathsf{b} \rangle$ and c the carry in, then a subtraction of $\langle \mathsf{a} \rangle - \langle \mathsf{b} \rangle$ is equivalent to $\mathsf{Add}(\langle \mathsf{a} \rangle, \langle \overline{\mathsf{b}} \rangle, 1)$. Relational operators can be built around adders in a similar way [49, Chap. 6]. For example (left part of Fig. 7), $\mathsf{valid}^{\boxdot}((\tau_e, \tau_s), n, J)$ is implemented using five w-bit adders: one for $q := n - \min(J)$, one for $r := T.\tau_e \geq q$, one to calculate $p := n - \max(J)$ and two to calculate $t := T.\tau_s \leq \max(p, 0)$. Finally, the unit outputs the verdict $t \wedge r$, where t and r

are calculated in parallel. To evaluate $\mathsf{valid}^{\boxdot}(m, \tau, n)$ the unit uses three w-bit adders, one to determine $q := n - \tau$, one for $p := q > 0$, and a third to either calculate $r := q \geq m_{\boxdot_\tau \varphi}$ or $r := 0 \geq m_{\boxdot_\tau \varphi}$, depending on the truth value of p. Finally, the validity checker outputs the verdict r to the ptLTL evaluation unit. Note that, for the actual implementation, we do not explicitly calculate $q := n - \min(J)$ through an adder. Instead, the design is configured with an absolute time point that signalizes the end of the *startup phase*, which equals to $\mathsf{max}(J) + 1$. A dedicated signal is cleared at reset and asserted once $n = \mathsf{max}(J) + 1$, therefore, replacing an adder by a more resource friendly comparator circuit in the implementation for the $\mathsf{valid}^{\boxdot}((\tau_e, \tau_s), n, J)$ predicate.

Lists and garbage collection For a list $l_{\boxdot_J \varphi}$ we turn to block RAMs (abundant on contemporary FPGAs) which are organized as ring buffers (right in Fig. 7). Each ring buffer has a read (rp) and a write pointer (wp). To insert a time point pair that satisfies $\mathsf{feasible}((\tau_s, n - 1), n, J))$, wp is incremented to point to the next free element in the ring buffer. The GC then adjusts rp to indicate the latest element with regard to n and J that is *recent enough*. In a fresh cycle (indicated by a changed time point n), the GC loads (τ_s, τ_e) using rp, which is incremented iff $\mathsf{garbage}((\tau_s, \tau_e), n, J)$ holds.

Control logic and modularity The control logic as shown in Fig. 7 allows one to easily reconnect hardware observers according to the specification's parse tree, which entails that the specification can be modified (within resource limitations) without re-synthesizing the whole design, which could take tens of minutes for FPGA designs.

5.6 A microcomputer to evaluate ptMTL and ptLTL specifications

In the following, we discuss a low footprint, reconfigurable microcomputer design that uses AtChecker blocks and the hardware observer blocks to evaluate arbitrary ptLTL and ptMTL formulas. The microcomputer, called μSpy, is configured with a binary program that controls and configures the building blocks depending on the formula to be evaluated. This configuration-based design of the μSpy proves elegant in a dynamic setting, such as product testing in early development phases, where the specification is subject to frequent changes [70]. Modifying the specification then only requires to download a new program to the μSpy. The hardware design of the μSpy is shown in Fig. 8 and builds on our previous work [68, 70] where we showed how to evaluate ptLTL formulas on such an architecture. An additional component (ptMTL observers) implements the control logic needed to instantiate ptMTL hardware observers to cover the time-bounded operators of the specification.

Workflow A (GUI-based) observer-generation application on a host computer compiles a ptMTL specification φ into a triple $\langle \Pi, C_a, C_m \rangle$, where C_a is a configuration for the AtChecker, C_m is a configuration for the pool of time bounded MTL operators and Π is a native program for the μSpy.

The synthesis of a configuration for the μSpy, denoted by $\langle \Pi, C_a, C_m \rangle$, from φ requires the following steps:

(1) We use the ANTLR parser generator [61] to parse φ. This step yields an abstract syntax tree (AST) that represents the specification.
(2) After some pre-processing of the AST, we determine the m subformulas $\varphi_1, \ldots, \varphi_m$ of φ by using a post-order traversal.
(3) For each subformula φ_i, $1 \leq i \leq m$:

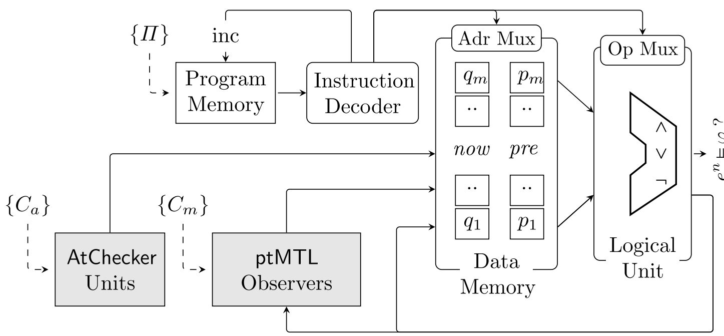


Fig. 8 The μSpy architecture. AtChecker units as in Fig. 6 and ptMTL observers as in Fig. 7

- If φ_i is an atomic proposition, instantiate an AtChecker block and add its configuration to C_a.
- If φ_i is a ptLTL formula, we use the approach shown in [68, 70] to generate a native instruction for the μSpy and add the instruction to Π.
- If φ_i is a ptMTL formula, we instantiate the corresponding observer hardware block, generate the hardware block's configuration and a native instruction for the μSpy. We add the configuration to C_m and the instruction to Π.

After running steps (1–3) of the synthesis procedure, the resulting configuration $\langle \Pi, C_a, C_m \rangle$ is then transferred from the host computer to the hardware platform where the μSpy is instantiated on, e.g., from the host computer through an Universal Serial Bus (USB) to an FPGA. We note that the host computer is only required to generate such a configuration for the current specification, but is not required during monitoring.

Instruction set architecture The μSpy is a pipelined microcomputer organized as a classical Harvard architecture. Its Instruction Set Architecture (ISA) supports 22 opcodes to handle ptLTL and ptMTL operators, where each instruction word is 40 bits long. It contains the opcode, addresses of two operands, an interval address, and a further address to select a private memory space for ptMTL operators. The first two bits from the operands address denote the source of the operands data which can be a memory location, i.e., the location in the data memory where the result of the respective subformula is held, an atomic proposition or an immediate value, which can be *true* or *false*. The additional fields *Interval Address* and *List Address* are necessary for the ptMTL operators only. A single instruction word for the μSpy is 40 bit long and is structured as follows:

OpCode	Addr. Operand 1	Addr. Operand 2	Interval Addr.	List Addr.
5 bit	2+8 bit	2+8 bit	8 bit	7 bit

Architectural features The μSpy manages two memories $p[0, \dots, m-1]$ and $q[0, \dots, m-1]$, one containing the evaluations of all m subformulas of φ (generated in a post-order traversal of the parse tree of φ; in step (1) of the synthesis procedure) in the current and in the previous execution cycle (i.e., time points n and $n-1$). This allows for space and time efficient evaluation of formulas whose parse tree is a directed acyclic graph, and not

Table 1 μSpy clock-cycles for Boolean, ptLTL, and ptMTL operators

Logic	Operator	μSpy clock cycles
Boolean	$\neg\varphi$	1
	$\varphi_0 \bullet \varphi_1, \bullet \in \{\wedge, \vee\}$	1
ptLTL	$\odot\varphi$	1
	$\varphi_1 \, S \, \varphi_2$	1
ptMTL	$\boxdot_\tau \varphi \mid \Diamond_\tau \varphi$	2
	$\boxdot_J \varphi \mid \Diamond_J \varphi$	4
	$\varphi_1 \, S_J \, \varphi_2$	4

necessarily a tree. For example, to evaluate the formula $\varphi \equiv (\uparrow \sigma_1) \equiv \sigma_1 \wedge \neg \odot \sigma_1$, one is not required to evaluate both σ_1 and $\odot\sigma_1$ independently, and thus σ_1 twice. Rather, we will have two registers of length 1, i.e., $p[0]$ holds the result of σ_1 from the previous round and $q[0]$ from the current round. The μSpy then fetches both $p[0]$ and $q[0]$ and executes the instruction that represents the operator $\uparrow$, which maps to the Boolean operation $(q[0] \oplus p[0]) \wedge q[0]$, namely, σ_1 did toggle its truth value ($q[0] \oplus p[0]$ holds) and σ_1 is true in the current state ($q[0]$ holds). Each instruction is processed through a four-stage pipeline (*fetch*, *load*, *calc*, and *write back*). All stages except the calc stage require one clock cycle per instruction, the execution time of the calc stage depends on the operator and requires from one to four clock cycles.

Execution time per operator Due to the pipelined design of the μSpy any ptLTL operator is executed within a single clock cycle in the pipeline stage. The additional overhead for list management and garbage collection required for the ptMTL operators require an additional one to three clock cycles. Due to a data forwarding strategy from the execution to the load stage in the pipeline, no further pipeline stalls are necessary and the pipeline is guaranteed to be optimally filled. Table 1 summarizes the execution times for various Boolean, ptLTL, and ptMTL operators.

Example Consider the ptMTL property $\varphi \equiv (\uparrow (2 \cdot v_1 + v_2 \leq 68)) \to (\boxdot_{[5,10]}(4 \cdot v_3 = 20 \vee v_4 = 40))$. As in the example of Sect. 5.4, the atomic propositions $\{\sigma_1, \sigma_2, \sigma_3\}$ of φ are evaluated by three AtChecker units. The subformulas $\uparrow (2 \cdot v_1 + v_2 \leq 68)$ and $(4 \cdot v_3 = 20 \vee v_4 = 40)$ are checked by the μSpy. For example, the value of σ_1 and the result of σ_1 from time $n-1$ is used by the calc stage, which decides if $\uparrow \sigma_1$ holds at the current time. The process is similar to determine the truth value of $\sigma_2 \vee \sigma_3$, the result of which is used as input to calculate $\boxdot_{[5,10]} (\sigma_2 \vee \sigma_3)$. The observer block is configured through the interval memory so as to represent $J = [5, 10]$. The output of the $\boxdot_{[5,10]} (\sigma_2 \vee \sigma_3)$ calculation is then the input to the final ptLTL computation, i.e., $\varphi \equiv (\uparrow \sigma_1) \to (\boxdot_{[5,10]}(\sigma_2 \vee \sigma_3))$.

6 Evaluation

To demonstrate the feasibility of our approach, we implemented the presented algorithms for ptMTL monitoring by means of the μSpy on an FPGA platform. In the current implementation, subformulas are evaluated sequentially as they appear in the specification's parse tree. Since the observer blocks are executed in sequence, their logic elements can be reused and it

suffices to equip the μSpy with only one $\boxdot_\tau \varphi$, one $\boxdot_J \varphi$, and one $\varphi_1 S_J \varphi_2$ hardware observer block and assign memory according to the number of subformulas.[2] The implementation is a synchronous register-transfer-level VHDL design, which we both simulated in MENTOR GRAPHICS MODELSIM and synthesized for various FPGAs using the industrial logic synthesis tool ALTERA QUARTUS II.[3]

6.1 Simulation results

We conducted several simulation runs of the VHDL implementation of the μSpy unit when monitoring different ptMTL formulas with randomly generated inputs, representing the execution traces of an SUT. The simulation runs cover several combinations of the ptLTL operators $\uparrow$, $\odot$, and $\varphi_1 S_s \varphi_2$ as well as the time-bounded ptMTL operators $\boxdot_\tau \varphi$, $\diamondsuit_J \varphi$, and $\varphi_1 S_J \varphi_2$. The truth values of the involved atomic propositions $\{\sigma_0, \sigma_1, \sigma_2\}$ were generated by placing 1000 truth value transitions with uniformly distributed interarrival times on the discrete timeline. In all simulated executions, our implementation behaved as specified. To increase confidence in the implementation, we used an automatic test suite, which checks the generated executions not only with the μSpy, but also with (i) a software implementation of our observer algorithms and (ii) a naive offline monitoring algorithms following the semantics definition of ptLTL and ptMTL. We run this setup with a set of sample specifications and compared the output of the three implementations and iteratively fixed remaining bugs. We used traditional line coverage metrics to assess the test progress. A rigorous, formal correctness analysis of the μSpy implementation, however, is still an open issue.

In what follows, we discuss two representative simulation runs involving the $\boxdot_\tau$ and the S_J operator. To make the simulation traces accessible, Table 2 summarizes all relevant hardware signals and their intended meaning. We further use the following annotation for the internal state of the μSpy: $m(x)$ denotes the location in the observer's data memory at address x, $a(x)$ denotes the xth atomic proposition and $i(x)$ specifies the interval stored at address x in the observer's interval memory.

(a) *Invariant previously* $\boxdot_\tau \varphi$ We setup the framework so as to evaluate the ptMTL formula:

$$\varphi_1 \equiv (\uparrow \sigma_0) \rightarrow (\boxdot_5 \sigma_1)$$

The property is then translated by the host application into the following binary program for the μSpy:

```
01011 0000000000 0000000000 00000000 0000000 // rising edge at a(0)
10001 0000000001 0000000000 00000001 0000000 // [[]] a(1), i(1), mem 0
00110 1000000000 1000000001 00000000 0000000 // m(0) -> m(1)
11111 1000000010 0000000000 00000000 0000000 // output result m(2)
```

and into the following data for the interval memory:

```
0000000000000000 0000000000000110 // startup phase duration: 6
0000000000000000 0000000000000101 // [0, 5]
```

[2]In our experiments, we opted for a resource efficient design of the μSpy. A configuration of the μSpy with multiple ptMTL hardware observers immediately makes an evaluation of several subformulas in parallel possible, however, increase resource requirements.

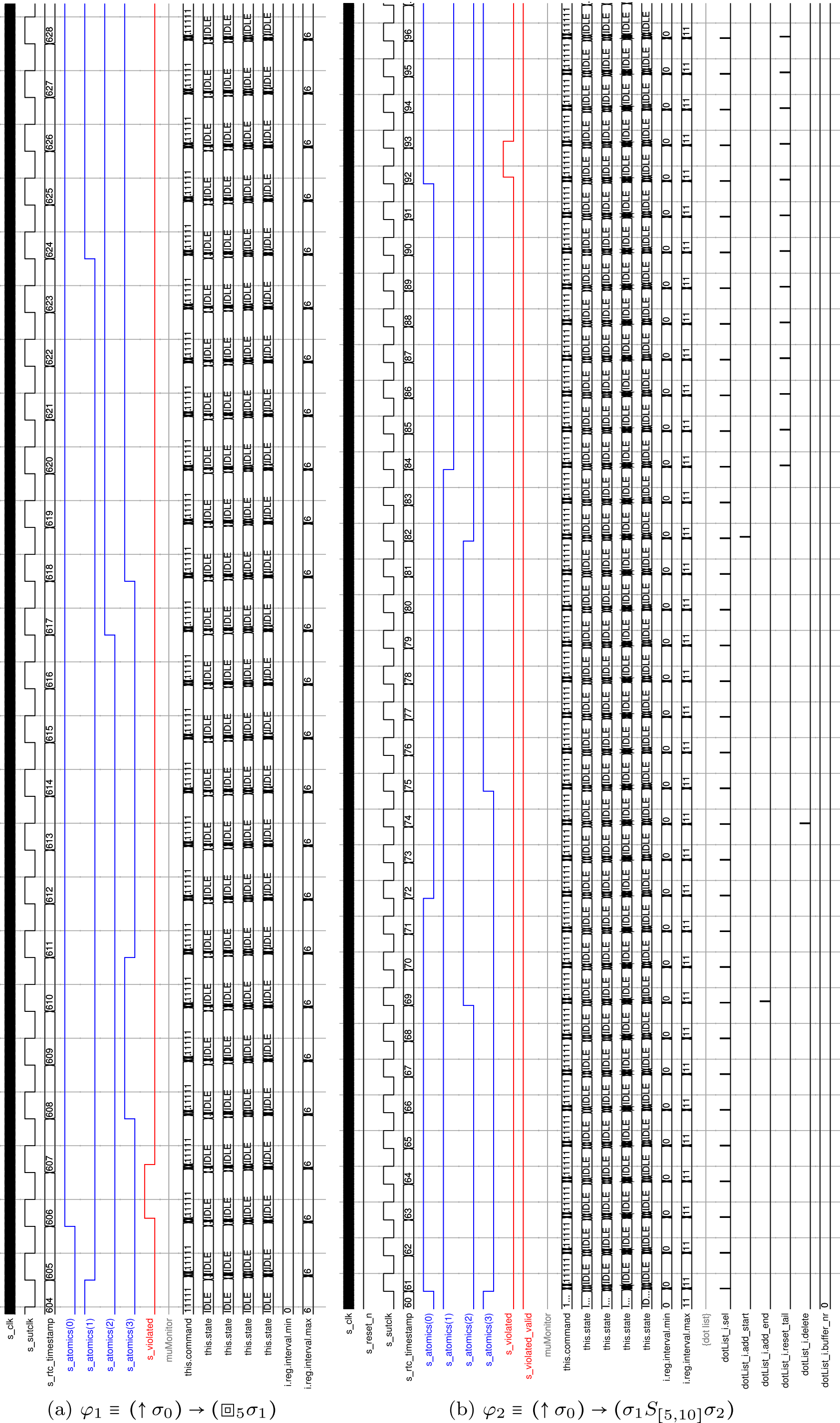
[3]Tools can be downloaded from http://www.mentor.com and http://www.altera.com.

Table 2 Simulation signals and their meaning; *AH* = *Active High* (issued when high); *AL* = *Active Low* (issued when low), and *RTC* = *Real Time Clock*

Signal Name	Unit	Meaning
s_clk	RVU	system clock of the RV framework
s_reset_n		asynchronous reset of the RV framework (AL)
s_sut_clk		system clock of the SUT
s_rtc_timestamp	RTC	ctr. value of the real-time clock (i.e., time point n)
s_atomic(0)	SUT	truth value of atomic proposition # 0, σ_0 (AH)
s_atomic(1)		truth value of atomic proposition # 1, σ_1 (AH)
s_atomic(2)		truth value of atomic proposition # 2, σ_2 (AH)
s_atomic(3)		truth value of atomic proposition # 3, σ_3 (AH)
s_violated	RVU	monitoring output $e^n \models \varphi$ (AH)
command	μSpy	instruction (op-code) for the μSpy
state		state of the fetch stage state machine
state		state of the load stage state machine
state		state of the calc stage state machine
state		state of the write back stage state machine
interval_min		$\mathsf{min}(J)$ (in RTC ticks)
interval_max		$\mathsf{max}(J)$ (in RTC ticks)
sel	List	select the list specified by buffer_nr (AH)
add_start		add start (_|‾) time point to the list (AH)
add_end		add end (‾|_) time point to the list (AH)
set_tail		clear list and add new entry (AH)
reset_tail		clear list and add entry with time point 0 (AH)
drop_tail		remove tail element from the list (AH)
delete		remove head element from the list (AH)
buffer_nr		id of the currently used list (AH)

The binary program consists of three subformulas and a dedicated *end* instruction. The interval memory holds two entries, the first denotes the duration of the start-up phase in RTC clock cycles and the second entry holds $\tau = 5$ for the $\boxdot_5$ operator. The startup phase signal is then used to implement the check whether $n - \tau \geq 0$ in the $\mathsf{valid}^{\boxdot}(m, \tau, n)$ predicate.

The simulation screenshot in Fig. 9a shows a section of the simulated VHDL entities. At time point $n = 606$, we see a _|‾ transition of s_atomic(0) which makes the premise of the implication true. As s_atomic(1) does not hold for all times within the interval $[601, 606]$, $e^{606} \not\models \boxdot_5 \sigma_1$ and the implementation correctly asserts the violated signal. According to Algorithm 1, the next _|‾ transition of s_atomic(1) at time $n = 617$ is stored in the $m_{\boxdot}$ memory of the $\boxdot$ operator. At the next _|‾ transition of s_atomic(0) at time $n = 624$ the premise of the implication holds and $\mathsf{valid}^{\boxdot}$ is evaluated as follows: $624 - 5 \geq 617$, yielding true, thus, $e^{624} \models \varphi_1$.



(a) $\varphi_1 \equiv (\uparrow \sigma_0) \rightarrow (\boxdot_5 \sigma_1)$

(b) $\varphi_2 \equiv (\uparrow \sigma_0) \rightarrow (\sigma_1 S_{[5,10]} \sigma_2)$

Fig. 9 Simulation traces for φ_1 and φ_2 extracted from ModelSim

(b) *Since within interval* $\varphi_1 \, S_J \, \varphi_2$ We setup the framework so as to evaluate the ptMTL formula:

$$\varphi_2 \equiv (\uparrow \sigma_0) \rightarrow (\sigma_1 \, S_{[5,10]} \, \sigma_2)$$

The property is then translated by the host application into the following binary program for the μSpy:

```
01011 0000000000 0000000000 00000000 0000000
        // rising edge at a(0)
10011 0000000001 0000000010 00000001 0000000
        // a(1) S a(2), i(1), mem 0
00110 1000000000 1000000001 00000000 0000000
        // m(0) -> m(1)
11111 1000000010 0000000000 00000000 0000000
        // output result m(2)
```

and into the following data for the interval memory:

```
0000000000000000 0000000000001011
        // startup phase duration: 11
0000000000000101 0000000000001010 // [5, 10]
```

The instruction memory contains three instructions corresponding to the three operators in the formula. Figure 9b shows a snippet of the corresponding simulation trace. At time $n = 69$ a _/‾ transition of s_atomic(2) is detected and according to Algorithm 3, $n - 1 = 68$ is added to the list l_S of the S observer which is triggered by the *add_end* signal. At time $n = 74$ the predicate garbage evaluates to true (since $(68 < 74 - \min(5, 10))$ holds) and triggers the deletion of the element in the list. The signal *delete* is asserted. The ‾_ transition of s_atomic(2) at time $n = 82$ triggers the adding of the interval-start time point to l_S (see Algorithm 3 line 4). Consequently $(82, \infty)$ is the new head element of l_S. Starting from time $n = 84$ on s_atomic(1) and s_atomic(2) are false, which, according to Algorithm 3, sets the list to $(0, \infty)$. This is done through the *reset_tail* signal. At time $n = 92$ we see a _/‾ transition of s_atomic(0) which yields $e^{92} \vDash (\uparrow \sigma_0)$. The valid$^{\boxdot}$ predicate evaluates as follows: $(0 \leq 92 - \max(5, 10)) \wedge (\infty \geq 92 - \min(5, 10))$, yielding true. Finally, we obtain $e^{92} \nvDash \varphi_2$ and the *violated* signal is asserted.

6.2 Performance study

Recall, that our hardware implementation uses one hardware module for $\boxdot_\tau \, \varphi$ and $\Diamond_\tau \, \varphi$ observers, one for the $\boxdot_J \, \varphi$ and $\Diamond_J \, \varphi$ observers, and one for $\varphi_1 S_J \varphi_2$ observers. The latter two modules both require lists of the same size, therefore, scale identically with respect to operating frequency, logic elements, and required memory size. We thus treated them equally within the performance study.

A hardware instantiation of the μSpy with the standard configuration of

- Time point width in bits 32
- Number of supported $\boxdot_\tau \, \varphi \mid \Diamond_\tau \, \varphi$ subformulas 64
- Number of supported $\boxdot_J \, \varphi \mid \Diamond_J \, \varphi \mid \varphi_1 S_J \varphi_2$ subformulas 64
- Number of list entries for each $\boxdot_J \, \varphi \mid \Diamond_J \, \varphi \mid \varphi_1 S_J \varphi_2$ subformula 256
- Program memory size 256×40 bit

requires a total of 1297 logic elements and 1.075.392 memory bits (132 kByte) and allows for a maximum operating frequency f_{max} of 106 MHz (for the slow timing model at 85 °C) on an Altera Cyclone IV EP4CE115 FPGA. The operating frequency can easily be increased

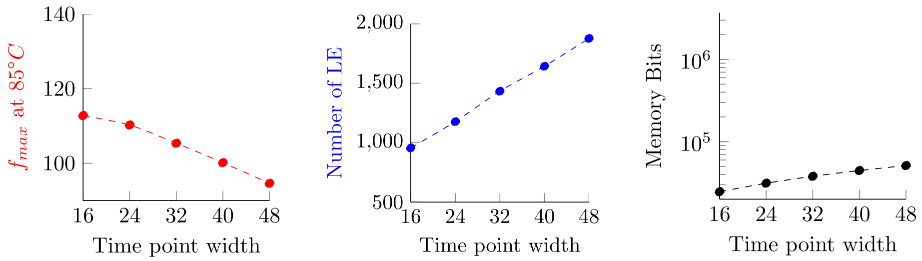


Fig. 10 Maximum operating frequency f_{max}, number of logic elements (LE), and required memory bits versus time point width, assuming a fixed number of 64 subformulas of type $\boxdot_{\mathcal{T}}\ \varphi \mid \diamondsuit_{\mathcal{T}}\ \varphi$ and 1 of type $\boxdot_J\ \varphi \mid \diamondsuit_J\ \varphi \mid \varphi_1 S_J\ \varphi_2$

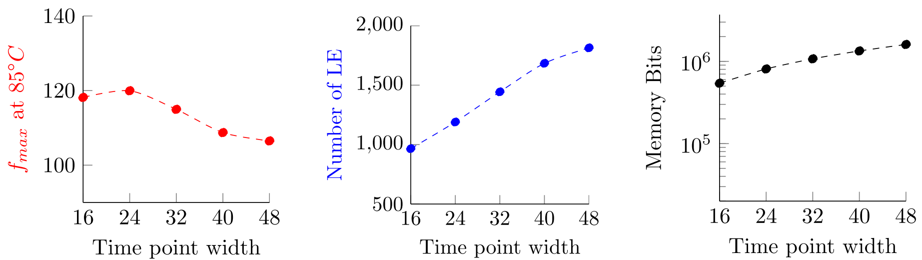


Fig. 11 Maximum operating frequency f_{max}, number of logic elements (LE), and required memory bits versus time point width, assuming a fixed number of 1 subformula of type $\boxdot_{\mathcal{T}}\ \varphi \mid \diamondsuit_{\mathcal{T}}\ \varphi$ and 64 of type $\boxdot_J\ \varphi \mid \diamondsuit_J\ \varphi \mid \varphi_1 S_J\ \varphi_2$

by moving to a more powerful FPGA architecture. For example, when synthesizing the design for an Altera Stratix V FPGA, we obtain a maximum operating frequency f_{max} of 230 MHz.

Scalability We synthesized the μSpy with different parameters to assess its scalability with regard to the width of the time points as well as the maximum number of ptMTL subformulas supported by the μSpy. We ran the synthesis with default settings so as to not obscure measurements by tool-specific optimizations. For example, when running synthesis optimized for speed, we naturally obtained results with higher operating frequencies but also with a higher number of logic elements. The influence of the *time point width* on the synthesized designs is shown in Figs. 10 and 11. Both figures show the scalability of the operating frequency, number of logic elements and required memory bits with respect to time point width. To asses scalability of each of the observer modules we built one variant supporting 64 (respectively 1) subformula(s) of type $\boxdot_{\mathcal{T}}\ \varphi \mid \diamondsuit_{\mathcal{T}}\ \varphi$ and 1 (respectively 64) subformula(s) of type $\boxdot_J\ \varphi \mid \diamondsuit_J\ \varphi \mid \varphi_1 S_J\ \varphi_2$. For both design variants, operating frequency and required number of logic elements scale linearly with comparable slope. For example, doubling the time point width from 24 to 48 bit increases the number of logic elements by about 60 %, whereas we observe only a 13 % decrease in the achievable maximum operating frequency. However, the design is still considerably small. Even with a time point width of 48 bit, it requires only 1.4 % of the available logic elements on our (low-end) target FPGA. For the number of required memory bits we observe a significant difference for both variants: Since the hardware module for evaluating $\boxdot_J\ \varphi \mid \diamondsuit_J\ \varphi \mid \varphi_1 S_J\ \varphi_2$ operators is equipped

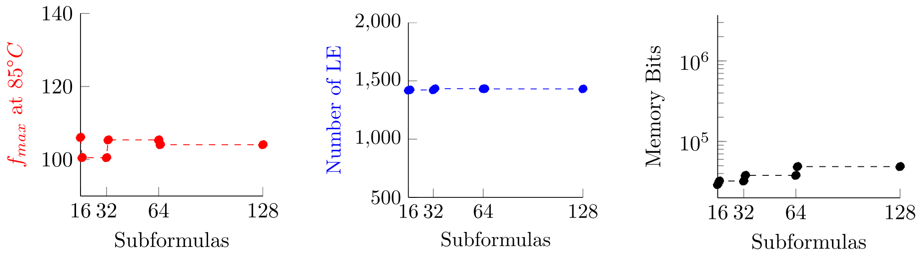


Fig. 12 Maximum operating frequency f_{max}, number of logic elements (LE), and required memory bits versus number of subformulas of type $\boxdot_\tau\ \varphi \mid \diamonddot_\tau\ \varphi$, assuming a fixed number of 1 subformula of type $\boxdot_J\ \varphi \mid \diamonddot_J\ \varphi \mid \varphi_1 S_J\, \varphi_2$ and time point width 32

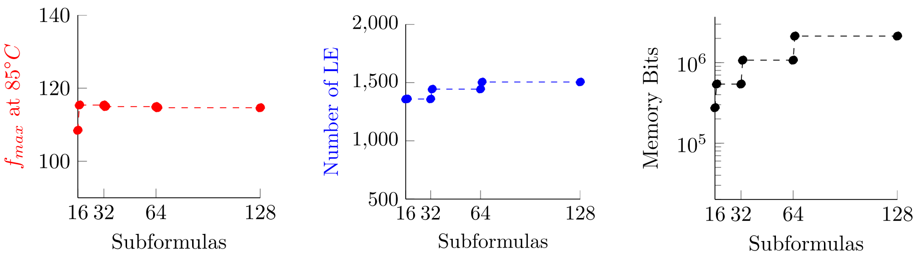


Fig. 13 Maximum operating frequency f_{max}, number of logic elements LE, and required memory bits versus number of subformulas of type $\boxdot_J\ \varphi \mid \diamonddot_J\ \varphi \mid \varphi_1 S_J\, \varphi_2$, assuming a fixed number of 1 subformula of type $\boxdot_\tau\ \varphi \mid \diamonddot_\tau\ \varphi$ and time point width 32

with a memory to store a list of time points for each of the supported $\boxdot_J\ \varphi \mid \diamonddot_J\ \varphi \mid \varphi_1 S_J\, \varphi_2$ subformulas, the required memory bits increase significantly faster in the variant supporting 64 such subformulas than in the version supporting only a single such subformula.

Figures 12 and 13 show the influence of the *number of supported subformulas* of type $\boxdot_\tau\ \varphi \mid \diamonddot_\tau\ \varphi$ and of type $\boxdot_J\ \varphi \mid \diamonddot_J\ \varphi \mid \varphi_1 S_J\, \varphi_2$. For that purpose, we built variants supporting a varying number of subformulas of type $\boxdot_\tau\ \varphi \mid \diamonddot_\tau\ \varphi$ and only one subformula of type $\boxdot_J\ \varphi \mid \diamonddot_J\ \varphi \mid \varphi_1 S_J\, \varphi_2$, and vice versa. One immediately sees that the number of supported subformulas of both types is not a limiting factor with respect to operating frequency and number of logic elements, as both stay almost constant. This is mainly due to the design choice we made for the μSpy, where we implemented the predicates, checks, and control logic required to evaluate either $\boxdot_\tau\ \varphi \mid \diamonddot_\tau\ \varphi$, $\boxdot_J\ \varphi \mid \diamonddot_J\ \varphi$, or $\varphi_1 S_J\, \varphi_2$ subformulas only once and reuse this hardware blocks every time the μSpy executes an opcode for a time-bounded subformula. To put this results in perspective, trimming the design of the μSpy to evaluating ptLTL specifications only accounts for 294 logic cells (23 % of the original design) and an f_{max} of 122 MHz (114 % of the original design). The situation is different for the required memory. It increases significantly with the number of supported subformulas: For each additional supported subformula, a sufficiently large memory block has to be added to the design. Clearly this leads to larger increases for subformulas that require to store lists of time points (cf. Fig. 13) than those that require to store only a single time point (cf. Fig. 12).

7 Related work

This section surveys related work by focusing on frameworks and tools, theoretical results on observer algorithms, and approaches that perform runtime verification either in or of hardware designs.

Frameworks and tools Watterson and Heffernan [80] review established and emerging approaches for monitoring (software) executions of embedded systems; calling for future work on runtime verification approaches that utilize existing chip interfaces to provide the observations as events to an external monitoring system. Pike et al. [64] worked on runtime verification for real-time systems by defining observers in a data-flow language, which are compiled into programs with constant runtime and memory. If the original system is periodically schedulable with some safety margin, the monitored system can be shown to be schedulable, too. This approach targets software only, whereas we monitor a combination of embedded software and hardware components. Hardware observers that simply probe one or more internal signals have been known in literature for a few decades. An early instance thereof is the non-interference monitoring and replay mechanism by Tsai et al. [79]. Their monitoring system is based on the MC6800 processor that records the execution history of the target system. A dedicated replay controller then replays stored executions, which supports test engineers in low-level debugging. Although we share a similar idea of probing internal signals, our framework detects specification violations on-the-fly, rather than replaying traces from some execution history.

The Dynamic Implementation Verification Architecture (DIVA) exploits runtime verification at intra-processor level [5]. Whenever a DIVA-based microprocessor executes an instruction, the operands and the results are sent to a checker which verifies correctness of the computation; the checker also supports fixing an erroneous operation. Chenard [19] presents a system-level approach to debugging based on in-silicon hardware checkers. The work of Brörkens and Möller [18] is akin to ours in the sense that they also do not rely on code instrumentation to generate event sequences. Their framework, however, targets Java and connects to the bytecode using the Java Debug Interface (JDI) so as to generate sequences of events.

BusMOP [62] generates observers for ptLTL on FPGAs, which are connected to the Peripheral Component Interconnect (PCI). The commercial Temporal Rover system [29] implements observers for MTL formulas, but the implementation and algorithms used are not published.

Observer algorithms We restrict our survey to ptMTL observer algorithms for past time logics in the discrete-time setting.

Thati and Roşu [78] presented an on-line observer for MTL formulas ψ. Their idea is to reduce the problem of deciding whether $e^n \models \psi$ to deciding several instances of $e^{n'} \models \psi'$, where ψ' is a subformula of ψ and $n' \leq n$. Thereby for each subformula $\varphi_1 S_{[a,b]} \varphi_2$ of ψ, the formulas $\varphi_1 S_{[a-1,b-1]} \varphi_2$, $\varphi_1 S_{[a-2,b-2]} \varphi_2$, $\ldots$, $\varphi_1 S_{[0,b-a]} \varphi_2$, $\ldots$, $\varphi_1 S_{[0,0]} \varphi_2$ are defined to be subformulas of ψ. For example, in case $\psi \equiv \varphi_1 S_{[1,3]} \varphi_2$, where φ_1 and φ_2 are atomic propositions, the reduced formulas of ψ are φ_1, φ_2 as well as $\varphi_1 S_{[0,2]} \varphi_2$, $\varphi_1 S_{[0,1]} \varphi_2$, and $\varphi_1 S_{[0,0]} \varphi_2$. Denoting by m the number of subformulas an MTL formula ψ is reduced to, the space complexity of their observer is within $\mathcal{O}(m 2^m)$ and its time complexity is within $\mathcal{O}(m^3 2^{3m})$ for each time n in $\mathbb{N}_0$, the observer is executed. For the special cases of $\psi \equiv \varphi_1 \ S_J \ \varphi_2$, the observer still requires a memory of at least $2m \geq 2 \max(J)$ bit. While this bound is incomparable in general to our bound, for large values of $\max(J)$ we immediately obtain that our

solution has less memory complexity. For example for $\varphi_1 S_{[5,1500]}\varphi_2$ the solution in [78] requires at least 3000 bit of memory, whereas our observer requires 208 bit, assuming (upper bounded) time points of 52 bit.

Maler et al. [57] presented an on-line observer algorithm for $\varphi_1\ S_J\ \varphi_2$ that is based on having active counters for each event of φ_2. Divakaran et al. [26] improved the number of counters of bit width $\log\max(J)$ to $2\lceil\min(J)/(\mathrm{len}(J))\rceil + 2$ and proved that any Since observer realized as a timed transition system must use at least $2(\lceil\min(J))/(\mathrm{len}(J))\rceil + 1$ clocks. While their space complexity is incomparable to ours in general, their solution is very resource intensive for a hardware realization: While we may store list values in cheap RAM blocks, their solution requires to store the current counter values in registers, since their values are incremented at every time step. Further, one can show by simple algebraic manipulations that:

Proposition 2 *For all intervals J, with $0 \le \min(J) < \max(J) < \infty$,*

$$\frac{2\max(J) - \min(J) + 2}{2 + \mathrm{len}(J)} - \frac{\min(J)}{\mathrm{len}(J)} \le 2. \tag{9}$$

Proposition 3 *For all intervals $J = [a, a+1]$, where $a \in \mathbb{N}_0$,*

$$\frac{\min(J)}{\mathrm{len}(J)} - \frac{2\max(J) - \min(J) + 2}{2 + \mathrm{len}(J)} = \frac{2}{3}(a-2).$$

From Proposition 2 immediately follows that our observer requires at most two tuples in addition to the (counter) tuples required by Divakaran et al.'s observer. On the other hand, it follows from Proposition 3 that there exists a choice of parameters where our observer requires significantly less memory.

In contrast to the solution presented by Divakaran et al. [26], our solution is tailored to a discrete time base, dictated by our application domain: not only that at the hardware level a (discrete) system clock is naturally available, but also adding and comparing fractions would incur a significant overhead with respect to latency and circuit size. Nonetheless, our algorithms also work in the dense time domain with only two small modifications: (i) instead of running the algorithms at every time $n \in \mathbb{N}_0$, they need to be executed at every transition of an input signal, and (ii) the term "$n-1$" must be replaced by "n" in Algorithms 2 and 3. By analogous proofs we obtain that, in this case, list ℓ is of size at most $(\max(J))/(\mathrm{len}(J)) + 1$ tuples, which is at most one more than the number of clocks required by the Since observer by Divakaran et al. [26].

Basin et al. [11] present a (discrete time) point-based observer for formula $\varphi_1\ S_J\ \varphi_2$ which runs in time $\mathcal{O}(\log\max(J \cup \{n\}))$ if executed at time $n \in \mathbb{N}_0$. Their algorithm, however, requires memory in the order of $\max(J)$. They further presented an interval-based observer algorithm for $\varphi_1 S_J \varphi_2$ with space complexity comparable to our solution. However, the algorithm is clearly motivated with a software implementation in mind, whereas we aim at efficient (highly parallel) circuit implementations. For example, for an arbitrary ptMTL formula φ, our time-complexity bounds scale with the depth of the parse tree of φ, in case the μSpy executes observer algorithms in parallel, and with the number of nodes in the parse tree of φ, in case the μSpy executes observer algorithms sequentially. By contrast, the bounds in [11] scale with the fourth power of the number of nodes in the parse tree of φ. Further, a direct implementation of their algorithm would require considerable hardware overhead, as it makes use of doubly-linked lists to store and manipulate time points. In comparison, our ring buffer design can easily be mapped to block RAM elements that are abundant on modern day FPGAs.

Hardware observers In previous work, we have shown that ptLTL can, within certain bounds, be checked in hardware running at the same frequency as the SUT [68]. Assertion-based verification (ABV) [36] gained momentum in industrial-strength hardware verification, especially driven by the emerge of the Property Specification Language (PSL). PSL is based on LTL, augmented with regular expressions, thus, we will not compare our work to PSL monitoring algorithms but rather to the hardware architecture of the resulting checkers. Existing work largely aims at synthesizing hardwired circuits out of various temporal specifications, whereas our approach (a) focuses on ptMTL specifications and (b) aims at providing a reconfigurable framework that has also applications in testing and not only as hard-coded observer. Translations from PSL into hardware either follow the modular or the automata based synthesis.

In the modular approach [14, 15, 25, 27, 60], sub-circuits for each operator are built and inter-connected according to the parse tree of the PSL expression being monitored. These circuits then output a pair of signals indicating the status of the assertion. Boulé and Zilic [15] present a hardware-checker generator capable of supporting ABV, by translating PSL to hardware language descriptions that can be included into the source design. The input to their circuit generator is the source file of the design under test (DUT). This limits their approach to designs where the source is available, whereas our framework can be attached to a variety of targets (cp. Fig. 1), even third party proprietary systems. Unfortunately, their algorithms lack a complexity analysis. Borrione et al. [14] describe a method of translating properties of the PSL foundation layer into predefined primitive components. A component is a hardware unit, consisting of a checking window and an evaluation block. They make use of shift register chains in the checking window block to trigger the execution of the evaluation block. Primitive components representing a timed operator (e.g., within in the next τ time units), need to individually count the number of elapsed time points. Das et al. [25] presented a modular approach by decomposing System Verilog Assertions (SVA) into simple communicating parallel hardware units that, when connected together, act as an observer for a SVA. Morin-Allory and Borrione [60] describe a generation of synthesizable hardware from regular expressions included in PSL. Drechsler [27] describes an approach to synthesize checkers for online verification of SoC designs through chains of shift registers, but does not allow for checking arithmetic relations among bit-vectors. For hardware designs, these specifications are often directly available from the specification [75].

In the automata based approach [4, 16, 17, 37, 38, 56], state machines are synthesized that check a property during simulation. The generated automata are generally of non-deterministic nature. To avoid a blowup of the automaton capable of monitoring formulas that are required to hold for a certain number of clock cycles, additional counters are inserted. However, this is only feasible if the output language natively supports non-deterministic finite automata (NFA), unfortunately, major hardware descriptions languages (e.g., Verilog and VHDL) do not. Consequently, observers need to be converted to a deterministic finite automaton (DFA) first, which, in the worst case, yields an exponential blowup of the resulting DFA in the size of the NFA [43]. This theoretical limitations were also reflected in the experiments of Straka et al. [76] where they report on an attempt to verify trivial properties of a simple counter, where the resulting observers synthesized by FoCs [1] from a PSL specification requires 120 logic slices whereas the resources for the counter itself accounts only for 3 slices. This performance issues motivate them to turn to a self-made tool to design on-line checkers instead of using existing toolchains. Lu and Forin [56] present a compiler from PSL to VERILOG, which translates a subset of PSL assertions (SPSL, a C-language binding for PSL [20]) about a software program (written in C in their approach) into hardware execution blocks for an extensible MIPS processor, thus allowing for transparent runtime verification without altering the program under investigation.

The synthesized verification unit is generated by a property rewriting algorithm developed by Roşu and Havelund [72]. Atomic propositions are restricted to a single comparison operator only. For comparison, our approach supports more complex relations among memory values in the atomic propositions, thus yielding greater flexibility and expressiveness in the specification language. Armoni et al. [4] describe an automata-theoretic construction based on determinization for unrestricted temporal logic, i.e., ForSpec [3]. They showed how to obtain deterministic compilation targeting dynamic verification that is as close as possible to the nondeterministic compilation of temporal assertions.

8 Conclusion

We presented an on-line runtime verification framework to check a ptMTL formula on executions with discrete time domain. At the framework's heart is an observer design for the time-bounded Since operator and the special cases of *exists*/*invariant previously* and *within interval*. Correctness proofs of all presented algorithms have been given and bounds on their time and space complexity have been proven. The promising complexity results are mainly due to the integration of a garbage collection and a filtering strategy that automatically drop events that can neither validate nor invalidate the specification.

We further discussed a reconfigurable hardware realization of our observer algorithm that provides sufficient flexibility to allow for changes of the monitored specification without necessarily re-synthesizing the hardware observer. Reconfigurability is indeed a valuable property of the presented approach since logic synthesis is itself a very time-consuming task. To demonstrate the feasibility of our approach for practical applications, we implemented the algorithms on a Field Programmable Gate Array. The predictable and low resource requirements of the presented hardware solution together with its reconfigurability support the application in the diagnosis of embedded real-time systems during execution time.

Based on the framework presented in this article, we plan to investigate the following directions: *who guards the guardians?* [74] is a legitimate question with regard to the implementation of our runtime verification unit. Whereas we gave a formal correctness analysis for the algorithms itself, however, doing so for the implementation is an open issue. Additionally, we plan to extend our work to (bounded) future time MTL specifications.

Acknowledgements The work of Thomas Reinbacher and Matthias Függer has been supported within the FIT-IT project CevTes managed by the Austrian Research Agency FFG under grant 825891 and (partially) supported by the Austrian Science Foundation (FWF) under project S11405 (RiSE). The work of Jörg Brauer has been, in part, supported by the DFG Cluster of Excellence on *Ultra-high Speed Information and Communication*, German Research Foundation grant DFG EXC 89 and by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems*. The authors want to thank Dejan Nickovic, Andreas Steininger, Kristin Y. Rozier, and Johann Schumann for helpful discussions. Additionally, the authors want to thank Andreas Hagmann, Johannes Geist, and Patrick Moosbrugger for their help with the hardware implementation and experiments.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. Abarbanel Y, Beer I, Gluhovsky L, Keidar S, Wolfsthal Y (2000) FoCs: automatic generation of simulation checkers from formal specifications. In: CAV. LNCS, vol 1855. Springer, Berlin, pp 538–542

2. Alur R, Henzinger TA (1990) Real-time logics: complexity and expressiveness. In: LICS. IEEE, New York, pp 390–401
3. Armoni R, Fix L, Flaisher A, Gerth R, Ginsburg B, Kanza T, Landver A, Mador-Haim S, Singerman E, Tiemeyer A, Vardi MY, Zbar Y (2002) The Forspec temporal logic: a new temporal property-specification language. In: TACAS. Springer, Berlin, pp 196–211
4. Armoni R, Korchemny D, Tiemeyer A, Vardi M, Zbar Y (2006) Deterministic dynamic monitors for linear-time assertions. In: Formal approaches to software testing and runtime verification. LNCS, vol 4262. Springer, Berlin, pp 163–177
5. Austin TM (1999) DIVA: a reliable substrate for deep submicron microarchitecture design. In: MICRO. IEEE, New York, pp 196–207
6. Baier C, Katoen JP (2008) Principles of model checking. MIT Press, Cambridge
7. Bardin S, Herrmann P, Védrine F (2011) Refinement-based CFG reconstruction from unstructured programs. In: VMCAI. Springer, Berlin, pp 54–69
8. Barre B, Klein M, Soucy-Boivin M, Ollivier PA, Hallé S (2012) MapReduce for parallel trace validation of LTL properties. In: RV. LNCS. Springer, Berlin
9. Barringer H, Falcone Y, Finkbeiner B, Havelund K, Lee I, Pace GJ, Rosu G, Sokolsky O, Tillmann N (eds) (2010) Runtime verification—first international conference, proceedings. LNCS, vol 6418. Springer, Berlin
10. Bartocci E, Grosu R, Karmarkar A, Smolka S, Stoller S, Zadok E, Seyster J (2012) Adaptive runtime verification. In: RV. LNCS. Springer, Berlin
11. Basin D, Klaedtke F, Zălinescu E (2011) Algorithms for monitoring real-time properties. In: RV. LNCS, vol 7186. Springer, Berlin, pp 260–275
12. Bate I, Conmy P, Kelly T, McDermid J (2001) Use of modern processors in safety-critical applications. Comput J 44(6):531–543
13. Bauer A, Leucker M, Schallhart C (2010) Comparing LTL semantics for runtime verification. J Log Comput 20(3):651–674
14. Borrione D, Liu M, Morin-Allory K, Ostier P, Fesquet L (2005) On-line assertion-based verification with proven correct monitors. In: ICICT, pp 125–143
15. Boulé M, Zilic Z (2005) Incorporating efficient assertion checkers into hardware emulation. In: ICCD. IEEE Computer Society Press, Los Alamitos, pp 221–228
16. Boulé M, Zilic Z (2006) Efficient automata-based assertion-checker synthesis of PSL properties. In: High-level design validation and test workshop. Eleventh annual IEEE international, pp 69–76
17. Boulé M, Zilic Z (2008) Automata-based assertion-checker synthesis of PSL properties. ACM Trans Des Autom Electron Syst 13(1)
18. Brörkens M, Möller M (2002) Dynamic event generation for runtime checking using the JDI. Electron Notes Theor Comput Sci 70(4):21–35
19. Chenard JS (2011) Hardware-based temporal logic checkers for the debugging of digital integrated circuits. PhD thesis, McGill University
20. Cheung PH, Forin A (2007) A C-language binding for PSL. In: Proceedings of the 3rd international conference on embedded software and systems. ICESS '07. Springer, Berlin, pp 584–591
21. Clarke EM (2009) My 27-year quest to overcome the state explosion problem. In: LICS. IEEE Computer Society Press, Los Alamitos, p 3
22. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge. ISBN 0262032708
23. Colombo C, Pace GJ, Schneider G (2009) Safe runtime verification of real-time properties. In: FORMATS. LNCS, vol 5813. Springer, Berlin, pp 103–117
24. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL
25. Das S, Mohanty R, Dasgupta P, Chakrabarti P (2006) Synthesis of system verilog assertions. In: DATE, vol 2, pp 1–6
26. Divakaran S, D'Souza D, Mohan MR (2010) Conflict-tolerant real-time specifications in metric temporal logic. In: TIME, pp 35–42
27. Drechsler R (2003) Synthesizing checkers for on-line verification of system-on-chip designs. In: ISCAS, vol 4, pp IV-748–IV-751
28. Druilhe A, Daumas F, Nguyen T (2010) Formal verification of an FPGA emulation of the motorola 6800 microprocessor. In: NPIC&HMIT. American Nuclear Society, New York, pp 1316–1325
29. Drusinsky D (2003) Monitoring temporal rules combined with time series. In: CAV. LNCS, vol 2725. Springer, Berlin, pp 114–118
30. Dvorak D (ed) (2009) NASA study on flight software complexity. NASA office of chief engineer
31. Eide E, Regehr J (2008) Volatiles are miscompiled, and what to do about it. In: EMSOFT. ACM, New York, pp 255–264

32. Emerson EA (1990) Temporal and modal logic. In: Handbook of theoretical computer science, vol B. MIT Press, Cambridge, pp 995–1072
33. Engblom J (2001) On hardware and hardware models for embedded real-time systems. In: RTSS
34. Fischmeister S, Lam P (2010) Time-aware instrumentation of embedded software. IEEE Trans Ind Inform 6(4):652–663
35. Flexeder A, Mihaila B, Petter M, Seidl H (2010) Interprocedural control flow reconstruction. In: APLAS. LNCS, vol 6461. Springer, Berlin, pp 188–203
36. Foster H, Lacey D, Krolnik A (2003) Assertion-based design, 2nd edn. Kluwer Academic, Norwell
37. Gheorghita S, Grigore R (2005) Constructing checkers from PSL properties. In: CSCS'05 international conference on control systems and computer science, pp 757–762
38. Gordon M, Hurd J, Slind K (2003) Executing the formal semantics of the accellera property specification language by mechanised theorem proving. In: CHARME 2003. LNCS, vol 2860. Springer, Berlin, pp 200–215
39. Havelund K, Roşu G (2004) An overview of the runtime verification tool Java PathExplorer. Form Methods Syst Des 24(2):189–215
40. Havelund K (2008) Runtime verification of C programs. In: TestCom/FATES. Springer, Berlin, pp 7–22
41. Havelund K, Roşu G (2004) Efficient monitoring of safety properties. Int J Softw Tools Technol Transf 6:158–173
42. Havelund K, Rosu G (2002) Synthesizing monitors for safety properties. In: TACAS. LNCS. Springer, Berlin, pp 342–356
43. Hopcroft JE, Motwani R, Ullman JD (2006) Introduction to automata theory, languages, and computation. Addison-Wesley Longman, Reading
44. Horowitz P, Hill W (1980) The art of electronics. Cambridge University Press, Cambridge. ISBN 0521370957
45. Howe J, King A (2009) Logahedra: a new weakly relational domain. In: ATVA. LNCS, vol 5799. Springer, Berlin, pp 306–320
46. Kinder J, Veith H, Zuleger F (2009) An abstract interpretation-based framework for control flow reconstruction from binaries. In: VMCAI. LNCS, vol 5403. Springer, Berlin, pp 214–228
47. Kogge PM, Stone HS (1973) A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans Comput 22(8):786–793
48. Kopetz H (2011) Real-time systems, 2nd edn. Springer, Berlin
49. Kroening D, Strichman O (2008) Decision procedures: an algorithmic point of view. Springer, Berlin
50. Laroussinie F, Markey N, Schnoebelen P (2002) Temporal logic with forgettable past. In: LICS. IEEE, New York, pp 383–392
51. Lee I, Kannan S, Kim M, Sokolsky O, Viswanathan M (1999) Runtime assurance based on formal specifications. In: PDPTA, pp 279–287
52. Leroy X (2006) Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL. ACM, New York, pp 42–54
53. Leroy X (2009) A formally verified compiler back-end. J Autom Reason 43:363–446
54. Lichtenstein O, Pnueli A, Zuck L (1985) The glory of the past. In: Logics of programs. LNCS, vol 193. Springer, Berlin, pp 196–218
55. Lindig C (2005) Random testing of C calling conventions. In: AADEBUG. ACM, New York, pp 3–12
56. Lu H, Forin A (2007) The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Tech rep MSR-TR-2007-99, Microsoft Research
57. Maler O, Nickovic D, Pnueli A (2005) Real time temporal logic: past, present, future. In: FORMATS, pp 2–16
58. Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems. Springer, Berlin
59. Marwedel P (2011) Embedded system design. Springer, Berlin. ISBN 9789400702578
60. Morin-Allory K, Borrione D (2006) Proven correct monitors from PSL specifications. In: DATE, pp 1–6
61. Parr TJ, Quong RW (1995) ANTLR: a predicated-ll(k) parser generator. Softw Pract Exp 25:789–810
62. Pellizzoni R, Meredith P, Caccamo M, Rosu G (2008) Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In: RTSS, pp 481–491
63. Pellizzoni R, Meredith P, Caccamo M, Rosu G (2008) Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In: RTSS, pp 481–491
64. Pike L, Goodloe A, Morisset R, Niller S (2010) Copilot: a hard real-time runtime monitor. In: RV. LNCS, vol 6418. Springer, Berlin, pp 345–359
65. Pike L, Niller S, Wegmann N (2011) Runtime verification for ultra-critical systems. In: RV. LNCS, vol 7186. Springer, Berlin, pp 310–324
66. Puschner P (2002) Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In: Proceedings of the 2nd Euromicro international workshop on WCET analysis, Department of Computer Science, University of York

67. Reinbacher T, Brauer J (2011) Precise control flow reconstruction using boolean logic. In: EMSOFT. ACM, New York, pp 117–126
68. Reinbacher T, Brauer J, Horauer M, Steininger A, Kowalewski S (2011) Past time LTL runtime verification for microcontroller binary code. In: FMICS. LNCS, vol 6959. Springer, Berlin, pp 37–51
69. Reinbacher T, Brauer J, Horauer M, Steininger A, Kowalewski S (2012) Runtime verification of microcontroller binary code. Sci Comput Program (in press)
70. Reinbacher T, Brauer J, Schachinger D, Steininger A, Kowalewski S (2011) Automated test-trace inspection for microcontroller binary code. In: RV. LNCS, vol 7186. Springer, Berlin, pp 239–244
71. Reinbacher T, Függer M, Brauer J (2013) Real-time runtime verification on chip. In: Qadeer S, Tasiran S (eds) RV. LNCS, vol 7687. Springer, Berlin, pp 110–125
72. Roşu G, Havelund K (2005) Rewriting-based techniques for runtime verification. Autom Softw Eng 12(2):151–197
73. RTCA/DO-178B (1992) Software considerations in airborne systems and equipment certification
74. Schumann J, Srivastava A, Mengshoel O (2010) Who guards the guardians? Toward V&V of health management software. In: RV. LNCS, vol 6418, pp 399–404
75. Shimizu K, Dill DL, Hu AJ (2000) Monitor-based formal specification of PCI. In: FMCAD. Springer, Berlin, pp 335–353
76. Straka M, Kotásek Z, Winter J (2008) The design of hardware checkers for verification and diagnostic purposes. In: CSE, pp 320–327
77. Tabakov D, Rozier KY, Vardi MY (2012) Optimized temporal monitors for SystemC. Form Methods Syst Des 41(3):236–268
78. Thati P, Roşu G (2005) Monitoring algorithms for metric temporal logic specifications. Electron Notes Theor Comput Sci 113:145–162
79. Tsai JJP, Fang KY, Chen HY, Bi Y (1990) A noninterference monitoring and replay mechanism for real-time software testing and debugging. IEEE Trans Softw Eng 16:897–916
80. Watterson C, Heffernan D (2007) Runtime verification and monitoring of embedded systems. IET Softw 1(5):172–179
81. Yang X, Chen Y, Eide E, Regehr J (2011) Finding and understanding bugs in C compilers. In: PLDI. ACM, New York, pp 283–294