CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Journal Paper

Runtime verification of autopilot systems using a fragment of MTL-∫

André Pedro*
Jorge Sousa Pinto
David Pereira*
Luis Miguel Pinho*

*CISTER Research Centre

CISTER-TR-170802

2017/08/21

Runtime verification of autopilot systems using a fragment of MTL-∫

André Pedro*, Jorge Sousa Pinto, David Pereira*, Luis Miguel Pinho*

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: anmap@isep.ipp.pt, dmrpe@isep.ipp.pt, lmp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Current real-time embedded systems development frameworks lack support for the verification of properties using explicit time where counting time (i.e., durations) may play an important role in the development process. Focusing on the real-time constraints inherent to these systems, we present a framework that addresses the specification of duration properties for runtime verification by employing a fragment of metric temporal logic with durations. We also provide an overview of the framework, the synthesis tools, and the library to support monitoring properties for real-time systems developed in C++11. The results obtained provide clear evidence of the feasibility and advantages of employing a duration-sensitive formalism to increase the dependability of avionic controller systems such as the PX4 and the Ardupilot flight stacks.

© CISTER Research Center
www.cister.isep.ipp.pt

Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-017-0470-5

FMICS-AVOCS

Runtime verification of autopilot systems using a fragment of MTL-$\int$

André de Matos Pedro[1] · **Jorge Sousa Pinto**[2] · **David Pereira**[1] · **Luís Miguel Pinho**[1]

© Springer-Verlag GmbH Germany 2017

Abstract Current real-time embedded systems development frameworks lack support for the verification of properties using explicit time where counting time (i.e., durations) may play an important role in the development process. Focusing on the real-time constraints inherent to these systems, we present a framework that addresses the specification of *duration properties* for runtime verification by employing a fragment of *metric temporal logic with durations*. We also provide an overview of the framework, the synthesis tools, and the library to support monitoring properties for real-time systems developed in C++11. The results obtained provide clear evidence of the feasibility and advantages of employing a *duration-sensitive* formalism to increase the dependability of avionic controller systems such as the PX4 and the Ardupilot flight stacks.

Keywords Runtime verification · Metric temporal logic · Durations · Resource model · Hard real time system · Polynomial inequality

✉ André de Matos Pedro
anmap@isep.ipp.pt; apedro.1@gmail.com

Jorge Sousa Pinto
jsp@di.uminho.pt

David Pereira
dmrpe@isep.ipp.pt

Luís Miguel Pinho
lmp@isep.ipp.pt

[1] CISTER/INESC TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal

[2] HASLab/INESC TEC & Universidade do Minho, Braga, Portugal

1 Introduction

As the technology evolves, real-time embedded systems become more and more pervasive in our daily routines. Currently, the major effort in the research community working on controller design for real-time embedded systems is the design of *physical models* rather than *model synthesis* techniques and associated formal verification approaches [1]. Even when formal synthesis and verification methods are used, the techniques for enforcing time isolation are generally discarded and delegated to the capabilities of non-formally verified *real-time operating system* (RTOSs) [2]. Ideally, *Runtime Verification* (RV) [3] can increase the dependability of these systems by drawing verdicts at runtime that may be used to trigger recovery actions. By adopting RV techniques, developers may be able to reduce the usually expensive and time-consuming testing efforts; if used in collaboration with static verification methods, these techniques can increase the overall coverage of the system, by ensuring execution time correctness in those parts of the development where heavyweight static approaches (like model checking [4] and deductive verification [5,6]) fail. Such failures are common, due to well-known problems, *e.g.*, the state-space explosion problem inherent to model checking and the limitations of proof automation in deductive verification.

Yet another problem faced by developers of real-time embedded systems is *task overloading* in RTOSs. This problem is commonly addressed via fault-tolerant mechanisms such as resource-sharing algorithms [7] or imprecise computation algorithms [8,9], which may be used for recovering degraded systems in a way such that liveness is ensured. Most of these mechanisms are not formally verified, due to their inherent complexity and concurrency constraints. As an alternative, coupling automatically synthesized, correct-by-

Published online: 21 August 2017

construction *monitors* with the target application allows for transient overloads to be checked for safety.

In previous work, we have introduced the 3-valued restriction of *metric temporal logic with durations* (MTL-$\int$) [10], one of the most expressive and decidable temporal logics [11] that allows to reason about durations. Our *three-valued restricted metric temporal logic with durations* (RMTL-$\int_3$) is able to specify properties that are present in the majority of real-time systems, *e.g.*, the behavior of task schedulers [12] and resource models [13]. However, it is not capable of dealing with discrete analog controllers at the level of abstraction of differential equations and hybrid discrete/continuous states, which are the targets of *differential logic* [14] and *hybrid linear temporal logic* [15], respectively.

In this paper, we present a framework for performing RV of real-time embedded systems (targeting embedded X86 and ARM processors) that is based on the RMTL-$\int_3$ formalism [16]. *Cylindrical algebraic decomposition* (CAD) [17] was combined with RV for the first time in [16], where we propose an algorithm for the simplification of quantified formulas in RMTL-$\int_3$. The underlying idea is to simplify the formulas before the synthesis step and, at the same time, to increase the expressiveness of RV. However, at this time there exists no available algorithm that can be used directly in the RV framework introduced in this paper. Here, we describe the refinement steps of the previously proposed algorithm to cope with bare-metal *real-time systems* (RTS) as well as the rules to synthesize the RV architecture for such small devices. Our embedded framework consists in the *rmtld3synthcpp* tool for automatic monitor synthesization targeting C++11, the *rmtld3synthocaml* tool for Ocaml, and the RTMLib runtime library to support coupling monitors in bare-metal boards such as Pixhawk [2].

To the best of our knowledge, this is the first RV framework for real-time embedded systems that is able to cope with explicit time and durations, two of the essential concepts for the detection of anomalies [18] of hard real-time systems, as well as for monitoring resource sharing between working tasks. Instead of coupling monitors in a simple interrupt-driven way as described in [12], we provide a *hierarchy* for safe monitoring. We validate the feasibility of the practical use of our framework with two use cases that target the verification of the PX4 autopilot [2]. We instrument the controller source code in order to ensure that at runtime the possible overload of real-time tasks and the failure of the transient overload mechanisms are avoided, by enforcing time isolation.

The paper is organized as follows: Section 2 introduces the related work. Section 3 describes the logic formalism underlying our monitoring/synthesis framework, and the distribution functions employed for imprecise computation in the use cases. Section 4 introduces the framework, and Sect. 5 describes the synthesization tools and the RTMLib library. Section 6 then describes the formalization of two use cases, and our experiments for the PX4 platform are described in Sect. 7. Finally, Sect. 8 discusses the results achieved and future work.

2 Related work

In RV, handling sequences of events to reason about the duration of the computation tasks of an RTS (and the detection of timing anomalies) has always been a major concern for designers and developers. General mechanisms such as Eagle and RuleR [19] allow for the formulation of safety properties, but have limitations to express timing anomalies and reason about durations. Their major drawback is their *intrusiveness*. Most of these RV frameworks add calls to verification procedures into the application code. This modification of the target application may be problematic in safety critical systems if it is not planned from the very beginning of the system design, as they inject interferences that can break the original time specifications and lead to tasks missing deadlines, and possibly to unsafe behaviors. Furthermore, the frameworks do not ensure isolation between the monitors and the monitored application. Hence, a failure of the monitored application may very well impact the capability of the monitor to detect that failure.

Closely related to our work scope, we have RT-MaC [20] and Copilot [21]. The former supports the specification of real-time and probabilistic properties such as "the probability of a task missing its deadline is 0.1." However, probabilities are only related to the *temporal order* of the historical event occurrence and has nothing to do with the probability of a task overloading. For instance, expressing that "the probability of a task executing no more than 2 time units is 0.2" is not possible. Copilot on the other hand is composed of: (i) a strongly typed, synchronous stream-based DSL embedded in the Haskell functional programming language; (ii) an interpreter; and also (iii) a compiler that compiles specifications into small, constant-time and constant-space C monitors. However, this approach does not consider *monitor overloading*, since monitors are executed by an RTOS or called using a hardware timer interrupt.

Recently, Bauer and colleagues [22], as well as Decker and colleagues [23], have proposed approaches that mix temporal logic with variants of first-order logic, with different restrictions. However, in both of these approaches the monitor decisions are based on the use of a SAT solver, which has a huge overhead, rendering it impractical for embedded systems. In [16], we have proposed a similar method, but focusing explicitly on a mix between *first-order logic of real numbers* ($\text{FOL}_{\mathbb{R}}$) and *metric temporal logic* (MTL). Instead of using a SAT solver, our approach relies on a *simplification algorithm* to be performed before execution, which removes

the quantifiers from the specification formulae, and brings them into a syntactic representation that eases the subsequent steps required for the complete monitoring generation process. However, the approach cannot be directly used in embedded hard RTSs without first refining its functional features, such as pattern matching and higher-order functions, to the languages supported by the bare-metal boards.

3 Preliminaries

Temporal specification. Timed temporal logics [24,25] extend classic temporal logics [26] with quantitative constructs, which makes them appropriate for reasoning about execution time requirements of RTSs. It is, however, far from being straightforward to select an appropriate logic for reasoning about timing requirements of realistic systems, as exemplified by current autopilot applications. First of all, because too much expressive power may easily result in undecidability [11], this is famously illustrated by MTL [25], a real-time extension of LTL [26]. A second difficulty is that, without guaranteeing the decidability of the logic, it might not be possible to devise an effective method for quantification removal. Moreover, the decidability result ensures that monitors synthesized from formulas will draw their verdicts and ideally terminate in a bounded amount of time, which is of outmost importance in RTSs when we are targeting execution time overloading of monitors.

Properties such as "the periodic task that performs the control loop has a duration per job no greater than 10 time units" or "the execution time that the monitor spends is less than 2 time units" cannot be specified, even in MTL, without knowing *a priory* the event triggering order of the system. In addition, formulas that can relate elapsed time for tasks such as "two tasks do not overload if the available load time checks $a = b-10$ or $a < b+10$" cannot be expressed in MTL. Note that a and b are variables that correspond to the execution time of each task, which is different from monotonic clocks (*e.g.*, those present in TPTL [24] formulas), but closer to the notion of stopwatches.

Our recent results [16] show that the inclusion of less-than relation $<$ over duration terms, to measure the duration of a certain formula with MTL, is an interesting abstraction for RTS and is also adequate for describing timing requirements. Quantifiers are simplified before the monitor synthesis process takes place, and because the relation $<$ is decidable and there exists a method to decompose quantified formulas into equivalent formulas without quantification (the CAD method, which decomposes polynomial inequalities by means of a quantifier removal algorithm), we are able to remove all the quantifiers statically, therefore avoiding that this process takes place at runtime, which could impose excessive overhead on the monitor.

However, the overall mechanism is not ready to be used in a bare-metal board such as Pixhawk: It requires some refinement steps, as well as guarantees about the constant-time execution of the generated monitors. Note that the monitor shall incrementally evaluate a system and draw a 3-valued verdict (YES, *no*, or *unknown*) in a deterministic bounded amount of time.

Let us now review the syntax and semantics of the adopted specification language RMTL-$\int_3$.

Definition 1 Let $\mathcal{P}$ be a set of propositions and $\mathcal{V}$ a set of logic variables. The syntax of RMTL-$\int_3$ terms η and formulas φ is defined inductively as follows:

$$\eta ::= \alpha \mid x \mid \eta_1 \circ \eta_2 \mid \int^{\eta} \varphi$$

$$\varphi ::= true \mid p \mid \eta_1 < \eta_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \varphi_1 \, \mathrm{U}_{\sim\gamma} \, \varphi_2 \mid \exists x \, \varphi$$

where $\alpha \in \mathbb{R}$, $x \in \mathcal{V}$ is a logic variable, $\circ$ stands for the arithmetic operators $+$ and $\times$, $\int^{\eta} \varphi$ is the duration of the formula φ in an interval, $p \in \mathcal{P}$ is an atomic proposition, U is a temporal operator with $\sim \in \{<, =\}$, $\gamma \in \mathbb{R}_{\geq 0}$, and the meaning of $\eta_1 < \eta_2$, $\varphi_1 \vee \varphi_2$, $\neg\varphi$, and $\exists x \, \varphi$ is defined as usual.

A *timed state sequence* κ is an infinite sequence of the form

$$(p_0, [i_0, i'_0[), (p_1, [i_1, i'_1[) \ldots,$$

where $p_j \in \mathcal{P}$, $i'_j = i_{j+1}$ and $i_j, i'_j \in \mathbb{R}_{\geq 0}$ such that $i_j < i'_j$ and $j \geq 0$. Let $\kappa(t)$ be defined as $\{p_j\}$ if there exists a tuple $(p_j, [i_j, i'_j[)$ such that $t \in [i_j, i'_j[$, and as $\emptyset$ otherwise. Note that there exists at most one such tuple.

A *logical environment* is any function $\upsilon : \mathcal{V} \to \mathbb{R}_{\geq 0}$. For any $x \in \mathcal{V}$, $r \in \mathbb{R}$, and logical environment υ, we will denote by $\upsilon[x \mapsto r]$ the logical environment that maps x to r and every other variable y to $\upsilon(y)$.

We will denote by Φ^3 the set of RMTL-$\int_3$ formulas, and by Γ the set of RMTL-$\int_3$ terms.

Definition 2 (RMTL-$\int_3$ Semantics) The truth value of a formula φ will again be defined relative to a model (κ, υ, t) consisting of a timed state sequence k, a logical environment υ, and a time instant $t \in \mathbb{R}_{\geq 0}$, and will now be one of the 3-values $\{\mathsf{tt}, \mathsf{ff}, \bot\}$. We will write $[\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \mathsf{tt}$ when φ is interpreted as true in the model (κ, υ, t), $[\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \mathsf{ff}$ when φ is interpreted as false in the model (κ, υ, t), and $[\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \bot$ otherwise. The auxiliary indicator function $1_{\varphi(\kappa,\upsilon)} : \mathbb{R}_{\geq 0} \to \{-1, 0, 1\}$ is defined as follows:

$$1_{\varphi(\kappa,\upsilon)}(t) = \begin{cases} 1 & \text{if } [\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \mathsf{tt}, \\ 0 & \text{if } [\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \mathsf{ff}, \\ -1 & \text{if } [\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \bot \end{cases}$$

The interpretation of the term η will be given by $\mathcal{I}[\![\eta]\!]_{3(\kappa,\upsilon)\,t} \in \mathbb{R} \cup \{\perp_{\mathbb{R}}\}$, as defined by the following rules. Whenever $\mathcal{I}[\![\eta]\!]_{3(\kappa,\upsilon)\,t} = \perp_{\mathbb{R}}$, this means that the term η is *infeasible*.

Rigid terms:

$\mathcal{I}[\![\eta_1]\!]_{3(\kappa,\upsilon)\,t}$ is defined as α if $\eta_1 = \alpha$, and as $\upsilon(x)$ if $\eta_1 = x$

Duration term:

If $\eta_1 = \int^{\eta_2} \phi$, then $\mathcal{I}[\![\eta_1]\!]_{3(\kappa,\upsilon)\,t}$ is defined as:

$$\begin{cases} \int_t^{t+\mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t}} 1_{\phi(\kappa,\upsilon)}(t')\,dt' & \text{if } \begin{array}{l}\mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t} \geq 0 \text{ and for all} \\ t'' \in [t, t+\mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t}],\ 1_{\phi(\kappa,\upsilon)}(t'') \in \{0,1\}\end{array} \\ \perp_{\mathbb{R}} & \text{otherwise} \end{cases}$$

Binary terms:

If $\eta_1 = \eta_2 + \eta_3$, then $\mathcal{I}[\![\eta_1]\!]_{3(\kappa,\upsilon)\,t}$ is defined as:

$$\begin{cases} \mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t} + \mathcal{I}[\![\eta_3]\!]_{3(\kappa,\upsilon)\,t} & \text{if } \mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t},\ \mathcal{I}[\![\eta_3]\!]_{3(\kappa,\upsilon)\,t} \in \mathbb{R} \\ \perp_{\mathbb{R}} & \text{otherwise} \end{cases}$$

If $\eta_1 = \eta_2 \times \eta_3$, then $\mathcal{I}[\![\eta_1]\!]_{3(\kappa,\upsilon)\,t}$ is defined as:

$$\begin{cases} \mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t} \times \mathcal{I}[\![\eta_3]\!]_{3(\kappa,\upsilon)\,t} & \text{if } \mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t},\ \mathcal{I}[\![\eta_3]\!]_{3(\kappa,\upsilon)\,t} \in \mathbb{R} \\ \perp_{\mathbb{R}} & \text{otherwise} \end{cases}$$

Turning to the interpretation of formulas, we define $[\![\varphi]\!]_{3(\kappa,\upsilon,t)}$ to be one of the three values in $\{\mathsf{tt}, \mathsf{ff}, \perp\}$, according to the following rules.

Basic formulae:

If ϕ is p, then $[\![\phi]\!]_{3(\kappa,\upsilon,t)}$ is tt if $p \in \kappa(t)$, ff if $p \notin \kappa(t)$ and $\kappa(t) \neq \emptyset$, and $\perp$ if $\kappa(t) = \emptyset$.

Relation operator:

If ϕ is $\eta_1 < \eta_2$, then $[\![\phi]\!]_{3(\kappa,\upsilon,t)}$ is defined as:

$$\begin{cases} \mathsf{tt} & \text{if } \mathcal{I}[\![\eta_1]\!]_{3(\kappa,\upsilon)\,t},\ \mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t} \in \mathbb{R}, \text{ and } \mathcal{I}[\![\eta_1]\!]_{3(\kappa,\upsilon)\,t} < \mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t} \\ \mathsf{ff} & \text{if } \mathcal{I}[\![\eta_1]\!]_{3(\kappa,\upsilon)\,t},\ \mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t} \in \mathbb{R}, \text{ and } \mathcal{I}[\![\eta_1]\!]_{3(\kappa,\upsilon)\,t} \geq \mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t} \\ \perp & \text{if } \mathcal{I}[\![\eta_1]\!]_{3(\kappa,\upsilon)\,t} = \perp_{\mathbb{R}} \text{ or } \mathcal{I}[\![\eta_2]\!]_{3(\kappa,\upsilon)\,t} = \perp_{\mathbb{R}} \end{cases}$$

Boolean operators:

If ϕ is $\neg\varphi$, then $[\![\phi]\!]_{3(\kappa,\upsilon,t)}$ is tt if $[\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \mathsf{ff}$, ff if $[\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \mathsf{tt}$, and $\perp$ otherwise.
If ϕ is $\varphi_1 \vee \varphi_2$, then $[\![\phi]\!]_{3(\kappa,\upsilon,t)}$ is tt if $[\![\varphi_1]\!]_{3(\kappa,\upsilon,t)} = \mathsf{tt}$ or $[\![\varphi_2]\!]_{3(\kappa,\upsilon,t)} = \mathsf{tt}$, ff if $[\![\varphi_1]\!]_{3(\kappa,\upsilon,t)} = \mathsf{ff}$ and $[\![\varphi_2]\!]_{3(\kappa,\upsilon,t)} = \mathsf{ff}$, and $\perp$ otherwise.

Temporal Operators:

If ϕ is $\varphi_1\ \mathrm{U}_{\sim\gamma}\ \varphi_2$, then $[\![\phi]\!]_{3(\kappa,\upsilon,t)}$ is defined as:

$$\begin{cases} \mathsf{tt} & \text{if there exists } t' \text{ such that } t < t' \sim t+\gamma,\ [\![\varphi_2]\!]_{3(\kappa,\upsilon,t')} = \mathsf{tt} \text{ and} \\ & \quad \text{for all } t'',\ t < t'' < t',\ [\![\varphi_1]\!]_{3(\kappa,\upsilon,t'')} = \mathsf{tt} \\ \mathsf{ff} & \text{if for all } t',\ t < t' \sim t+\gamma, \\ & \quad [\![\varphi_2]\!]_{3(\kappa,\upsilon,t')} \neq \mathsf{ff} \text{ implies that} \\ & \quad\quad \text{there exists } t'' \text{ such that } t < t'' < t',\ [\![\varphi_1]\!]_{3(\kappa,\upsilon,t'')} = \mathsf{ff} \text{ and} \\ & \quad [\![\varphi_2]\!]_{3(\kappa,\upsilon,t')} = \mathsf{ff} \text{ implies that there exists no } t'' \\ & \quad \text{such that } t < t'' < t' \text{ or} \\ & \quad\quad \text{there exists } t'' \text{ such that } t < t'' < t',\ [\![\varphi_1]\!]_{3(\kappa,\upsilon,t'')} = \mathsf{ff} \\ \perp & \text{otherwise} \end{cases}$$

Existential operator:

If ϕ is $\exists x\ \varphi$, then $[\![\phi]\!]_{3(\kappa,\upsilon,t)}$ is defined as:

$$\begin{cases} \mathsf{tt} & \text{if there exists a value } r \in \mathbb{R} \text{ such that } [\![\varphi]\!]_{3(\kappa,\upsilon[x\mapsto r],t)} = \mathsf{tt} \\ \mathsf{ff} & \text{if for all } r \in \mathbb{R},\ [\![\varphi]\!]_{3(\kappa,\upsilon[x\mapsto r],t)} = \mathsf{ff} \\ \perp & \text{there exits } r \in \mathbb{R} \text{ such that } [\![\varphi]\!]_{3(\kappa,\upsilon[x\mapsto r],t)} = \perp \text{ and} \\ & \quad \text{there exists no } r \in \mathbb{R} \text{ such that } [\![\varphi]\!]_{3(\kappa,\upsilon[x\mapsto r],t)} = \mathsf{tt} \end{cases}$$

Note that the semantics of the until operator is strict and non-matching [24]. We will write $(\kappa, \upsilon, t) \models_3 \varphi$ when $[\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \mathsf{tt}$, and $(\kappa, \upsilon, t) \not\models_3 \varphi$ when $[\![\varphi]\!]_{3(\kappa,\upsilon,t)} = \mathsf{ff}$. In what follows we will often write $\eta_1 = \eta_2$ for $\neg(\eta_1 < \eta_2) \wedge \neg(\eta_2 < \eta_1)$. We will use the following abbreviations:

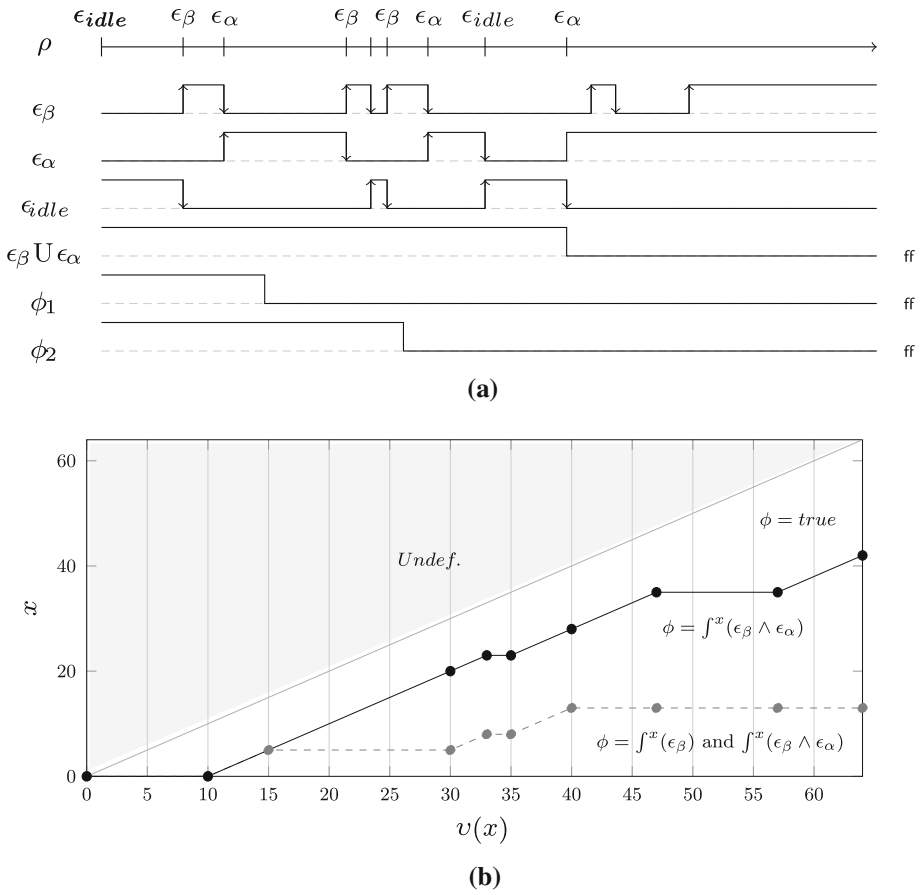

Fig. 1 Evaluation of formulas based on a sequence (**a**) and respective evaluation of duration terms (**b**)

$\varphi \wedge \psi$ for $\neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi$ for $\neg\varphi \vee \psi$, $\mathtt{tt}$ for $\varphi \vee \neg\varphi$, $\mathtt{ff}$ for $\varphi \wedge \neg\varphi$, $\Diamond_{\sim\gamma}\, \varphi$ for $\mathtt{tt}\ \mathrm{U}_{\sim\gamma}\ \varphi$, and $\Box_{\sim\gamma}\, \varphi$ for $\neg(\mathtt{tt}\ \mathrm{U}_{\sim\gamma}\ \neg\varphi)$.

Figure 1a intuitively illustrates the meaning of the RMTL-$\int_3$ formulas with temporal operators. The diagram contains a sequence ρ; three event releases ϵ_β, ϵ_α, and ϵ_{idle}; and the respective truth value of the logic formulas $\epsilon_\beta \,\mathrm{U}\, \epsilon_\alpha$, $\phi_1 = \int^{|\rho|}(\epsilon_\beta \wedge \epsilon_\alpha) \leq 10$, and $\phi_2 = \int^{|\rho|}(\epsilon_\beta) \leq 10$, where $|\rho|$ is the size of the sequence. In Fig. 1b, we can conclude that the formula $\forall x\ \int^x(\epsilon_\beta) \leq \int^x(\epsilon_\beta \wedge \epsilon_\alpha)$ in the finite interval $[0, 64)$ is interpreted as true. This figure illustrates the intuition behind the duration terms. The graph depicts the formula $\int^x(\epsilon_\beta)$ and $\int^x(\epsilon_\beta \wedge \epsilon_\alpha)$, which allows us to visually check the formula $\forall x\ \int^x(\epsilon_\beta) \leq \int^x(\epsilon_\beta \wedge \epsilon_\alpha)$ in the finite interval $[0, 64)$.

Figure 1b illustrates the intuition behind duration terms. The graph depicts the formulas $\int^x(\epsilon_\beta)$ and $\int^x(\epsilon_\beta \wedge \epsilon_\alpha)$, which allows us to visually check the formula $\forall x\ \int^x(\epsilon_\beta) \leq \int^x(\epsilon_\beta \wedge \epsilon_\alpha)$ in the finite interval $[0, 64)$. We conclude that the formula is interpreted as true in that interval.

Encoding Uncertainty. Fragments of temporal logic are commonly extended to incorporate probabilistic information. It is common to write a property such as "the probability of the event A stopping before the event B starts is 0.1." However, this requires defining a completely new semantics and operators for a temporal logic that allows for such properties to be encoded. Encoding uncertainty in MTL is not possible by default. We will now motivate the presence of the duration operator in the RMTL-$\int_3$ logic by showing that one of the advantages of having duration terms, with respect to what happens in traditional temporal logics, is precisely the ability to encode uncertainty.

Duration terms allow us to have random variables that, instead of getting a verdict based on a strict decision (say, "duration of a task is greater than 0 and less than 10"), allow us to express something about the temporal deviation of the system. This is significantly important in order to shape the temporal behavior of a task and provides the required system time's adaptability based on that temporal deviation.

As a motivation case, we adopt the schedulability analysis of RTSs. This analysis is based on the assumption that there exists an upper bound for the execution time of each task; however, this upper bound (and the same applies to the lower bound) can be too pessimistic, far from the average execution of the system. It is here that uncertainty can play a role, providing us with more information about the duration of the tasks.

Regarding the upper bound, it usually corresponds to the *worst-case execution time* of the various tasks, and

calculating this implies considering more pessimistic schedulability tests [7] (*i.e.*, producing negative verdicts under schedulable tasksets). Markov processes, such as *generalized semi-Markov processes* (GSMPs), are suitable models for specifying uncertainty of RTSs when schedulers are time-triggered, and the exact response times are not possible to obtain. Such a process is able to deal with any continuous probabilistic distribution under the restriction of the structure imposed on clocks. Furthermore, it allows us to construct a more realistic model of the target system, containing not only knowledge about the lower and upper bounds, but also a complete distribution of the response time of the system. Random variables (with any distribution) can be used for predicting the next state of the system. In this way, the next expected state can be estimated based on the *probability density function* (pdf) of the duration of each task and on the current state, up to the point at which the monitor executes. Based on these facts, we adopt the above notions in order to verify at runtime that the system satisfies the bounded uncertainty of the time execution using formulas in RMTL-$\int_3$. Note that only certain distributions can be encoded in this logic. For that, we rely on mixtures *Beta* distributions to encode uncertainty of real-time tasks using polynomial inequalities. The pdf of the *Beta* distribution is defined by

$$P(x) = \frac{(1-x)^{\beta-1}\, x^{\alpha-1}}{B(\alpha,\beta)},$$

where $B(p,q) = \frac{(p-1)!\,(q-1)!}{(p+q-1)!}$ is the *Beta* function, and $\alpha, \beta > 0$.

Verifying these properties at runtime is quite interesting, since it allows us to get runtime data and test whether there is a feasible deviation from the expected behavior. Examples of application of the *Beta* distribution are given in Sect. 6.

4 RV with RMTL-$\int_3$

In this section, we present a RV framework for embedded RTSs based on the novel RV monitoring model that will be described in Sect. 5. The latter contains the constraints/rules from the application side that allow us to synthesize a proper architecture for monitors. These rules are used to configure the target application to be executed in a multiprocessor embedded system or over a classic single processor from the AVR or ARM-M families of embedded processors. The support is given by the RTMLib [27] library that allows us to execute monitors in a lock-free and wait-free manner, which is very useful to guarantee deadlock-free RV operation.

Our toolchain is depicted in Fig. 2. As input, we have a set of formulas that will be converted to monitors using a one-to-one correspondence. From these formulas, we generate Ocaml and `C++11` source code as well as tests for `C++11` implementation that are automatically generated from the Ocaml synthesis, which corresponds to the dependence between both synthesis tools and identified by the dashed arrow. Tests and synthesized monitors are merged and compiled using the gcc toolchain including the support library RMTLib. This binary will run under NuttX OS. Otherwise, the compiled code from the synthesis Ocaml tool is executed in a common x86 operating system.

Operationally, each monitor can share resources (*e.g.*, memory and processors) with other monitors or may execute in isolation (using its own processor and memory partition), which is part of the specification of the RV monitoring model. The monitors have different execution rules that may change at execution time and rules for their operation.

- Execution rules are step-based (for iterative/tail recursive monitors; for an arbitrary number $n \in \mathbb{N}$ of execution steps), symbol-based (for explicit symbol consumption in automata formalisms), time-based (a timed bound in discrete execution time for execution of general-purpose monitors). Based on this, we can change the execution of the monitor at runtime in a dynamic way (a feature provided by RTMLib).
- Operation rules are time-triggered or event-triggered; the idea is to generate runtime verifiers depending of the target RTS. The modes of operation/execution are assigned according to the RV model.

For hard RTS, we use the step-based rule combined with a time-triggered rule. Note that there is no explicit architecture for monitoring, and different RV rules produce different monitor architectures, depending on the target systems and the provided RV monitoring model.

Synthesis Algorithm Refinement. The evaluation algorithm proposed for RMTL-$\int_3$ in [16] uses functional programming language features such as *pattern matching* and *higher-order functions*, in particular *fold* operations.

Let $\mathbf{K}$ be a set of sequences κ, Υ a set of logic environments υ, and $\mathbb{R}_{\geq 0}$ the domain of a time instant t (analogous to the model (κ, υ, t)). Let us first consider the lambda functions, as already defined in [16], such as $\mathsf{Compute}_{(\vee)}::(\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \to \Phi^3 \to \Phi^3 \to \mathbb{B}_3$, $\mathsf{Compute}_{(\neg)}::(\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \to \Phi^3 \to \mathbb{B}_3$, $\mathsf{Compute}_{(\mathrm{U}_<)}::(\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \to \mathbb{R}_{\geq 0} \to \Phi^3 \to \Phi^3 \to \mathbb{B}_3$, and $\mathsf{Compute}_{(\int)}::(\mathbf{K} \times \Upsilon) \to \mathbb{R}_{\geq 0} \to \mathbb{R} \to \Phi^3 \to \mathbf{D}$, that evaluate formula schemes of the form $\psi_1 \vee \psi_2$, $\neg\psi$, $\psi_1 \,\mathrm{U}_{<\gamma}\, \psi_2$, and $\int^{\eta} \psi$, respectively. Note that $(\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0})$ is a model (consisting of a sequence in $\mathbf{K}$, a logic environment in Υ, and a time instant in $\mathbb{R}_{\geq 0}$), $\mathbf{D}$ the set $\mathbb{R}_{\geq 0} \cup \{\bot_{\mathbb{R}}\}$, Φ^3 is a set of three-valued formulas, $\mathbb{B}_3$ is the set of three values $\{\mathsf{tt}, \mathsf{ff}, \bot\}$, and $\mathbf{B}_4$ is a four-valued set defined by $\mathbb{B}_3 \cup \{\mathfrak{r}\}$, where $\mathfrak{r}$ is the fourth symbol of the four-valued set. Pattern matching features are currently

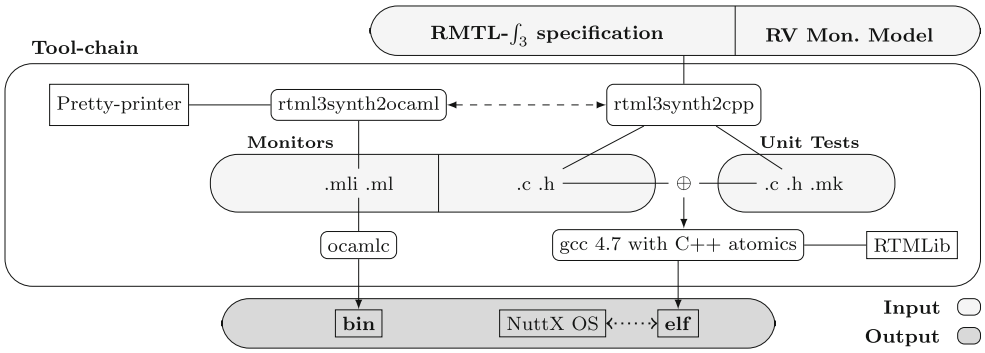

Fig. 2 Toolchain overview

not included in imperative programming languages such as `C++11`. Henceforth, and for the sake of compatibility with `C++11`, we adapt that algorithm as follows:

- the pattern matching constructions are statically erased and fully encoded into the generated monitors;
- the fold functions are encoded as *iterators* over the structure of traces;
- the remaining functions are encoded as `C++11` *lambda functions*.

Pattern matching is simplified over the inductive structure of the formulas. For instance, the formula $a \rightarrow \int^{10} b$ is implemented without pattern matching by composition over the structure of the formula. For that, we need to define some new `C++11` lambda functions such as $compute_p$::$\mathbf{P} \rightarrow (\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \rightarrow \mathbb{B}_3$, $compute_\neg$::$((\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \rightarrow \mathbb{B}_3) \rightarrow (\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \rightarrow \mathbb{B}_3$, $compute_\int$::$\mathbb{R} \rightarrow ((\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \rightarrow \mathbb{B}_3) \rightarrow (\mathbf{K} \times \Upsilon) \rightarrow \mathbb{R}_{\geq 0} \rightarrow \mathbf{D}$, and

$$
\begin{aligned}
compute_\vee :: ((\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) & \\
& \rightarrow ((\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \rightarrow \mathbb{B}_3) \rightarrow \mathbb{B}_3) \\
& \rightarrow (((\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \rightarrow \mathbb{B}_3) \\
& \rightarrow (\mathbf{K} \times \Upsilon) \rightarrow \mathbb{R}_{\geq 0} \rightarrow \mathbf{D}) \\
& \rightarrow (\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \rightarrow \mathbb{B}_3.
\end{aligned}
$$

Note that they encode the pattern matching (all required combinations for a given formula) instead of accepting RMTL-$\int_3$ formulas as input arguments. The generated function that corresponds to $a \rightarrow \int^{10} b$ is then the lambda function

$$
\begin{aligned}
&\lambda m.\ compute_\vee\ \left(compute_\neg\ \left(compute_p\ a\right)\right) \\
&\quad \left(compute_\int\ 10\ \left(compute_p\ a\right)\right)\ m
\end{aligned}
$$

where m is the model defined in `C++11` as `TraceIterator <int> iter, struct Environment env,` and `timespan t`. Note that $\lambda x.\ fun$ is defined in `C++11` as the expression `[](x){fun}`.

Let us now focus on the U operator. Porting to `C++11`, the function $\mathsf{Compute}_{(\mathrm{U}_<)}$, responsible for the synthesis of the until operator, requires defining a number of auxiliary `C++11` functions. As an example, the function eval_fold:: $(\mathbf{K} \times \Upsilon \times \mathbb{R}_{\geq 0}) \rightarrow \Phi^3 \rightarrow \Phi^3 \rightarrow \mathbf{K} \rightarrow \mathbf{B}_4$, as provided in the original RMTL-$\int_3$ evaluation algorithm [16], is defined in `C++11` as shown in Listing 1. We remark that the synthesized function (eval_fold $(\kappa, \upsilon, t)\ \phi_1\ \phi_2\ \varkappa$) is originally defined by

$$
fold\ \left(\lambda v\ (p, (i, t')) \rightarrow \mathsf{eval\_b}\ (\kappa, \upsilon, t' - \epsilon)\ \phi_1\ \phi_2\ v\right) \mathfrak{r}\ \varkappa,
$$

where ϕ_1 and ϕ_2 are formulas that were statically coded as the `C++11` lambda functions eval_b (of which there exist as many as there are occurrences of until operators, since each one contains different formulas), $\varkappa$ is the original trace sequence that is mapped into the iterator $iter$ of Listing 1, i is the lower bound of the interval (i, t'), ϵ is the minimum precision of a float, and $\mathfrak{r}$ is a proper mark for release if the until evaluation gives us an unknown value, identified in `C++11` by `FV_SYMBOL`, respectively. The operators $U_<$, $<$, and duration terms $\int^\eta \varphi$ may now be fully implemented using the `C++11` lambda functions. The existential operator does not need to be treated since we assume the existence of a simplification algorithm that decomposes a quantified formula into a non-quantified formula as in [16]. The output of this tool is a monitor written in the `C++11` programming language and composed by several source files, and the input is a configuration file containing an RMTL-$\int_3$ formula to be synthesized. The `rmtld3synth` synthesis tool for these operators, written in the Ocaml programming language [28], is fully described in [29]. The reader is referred to the example in Appendix 1 for further details and a worked out example.

The synthesized monitors have polynomial time complexity [16]. In order to analyze the space complexity of the synthesized monitors, we first note that the synthesis algorithm produces monitors written using pure lambda functions. Following [16], each formula ψ in RMTL-$\int_3$ to be synthesized, of length m_ψ, will originate a set of

```
auto eval_fold = []( struct Environment env, timespan t,
      TraceIterator<int> iter) -> four_valued_type
{
 return std{:}{:}accumulate
(
  iter.begin(), iter.end(), pair<four_valued_type,
       timespan>( FV_SYMBOL, t ),
  [&env, eval_b]( const pair<four_valued_type, timespan>
        a, Event<int> e )
  {
    return make_pair( eval_b( env, a.second, a.first ),
         a.second + e.getTime() );
  }
 ).first;
};
```

Listing 1 eval_fold synthesis in C++11

λ-expressions whose global size is in $\mathcal{O}(m_\psi)$, and whose mutual recursion pattern (or call graph) is free of cycles, since the invocations follow the structure of the formula ψ. Execution of these λ-expressions relies on a functional, stack-based mechanism, and it follows that the number of push/pop operations performed will be in $\mathcal{O}(m_\psi)$. The required stack size will thus be linear in m_ψ and *constant in the input trace size*. Therefore, the generated monitoring algorithms have *constant-space complexity* regarding the trace size, as our experimental results will confirm.

5 RV monitoring model

In this section, we describe how monitors are linked to buffers and tasks via the specialized *RunTime Embedded Monitoring Library* (RTMLib) and then discuss how timing guarantees are enforced in practice by the adopted hierarchy of monitors.

Linking monitors with RTMLib. Monitors are executed in a simple embedded monitoring framework, which we named the RTMLib [27]. These monitors use circular buffers as the data structure to hold a trace, and they have a certain periodicity. The framework ensures that monitors retrieve events from circular buffers respecting their partial order, in a lock- and wait-free manner. Note that several buffers are used in a composition as described in [30] for the reference architecture; more details on the implementation of RTMLib can be found in the documentation in [27]. Monitors execute as higher-priority tasks and are constantly interfering with the application. However, such interference is predictable and constant, since each monitor can execute in constant time that depends on the structure of the formula.[1]

Knowledge of the length of the circular buffers is required at compile time, and for that we define a *bound* over temporal formulas, allowing us to determine a map from time to event size. The calculation of temporal bounds for formulas of RMTL-$\int_3$ is then achieved by a recursive algorithm that traverses the inductive structure of the formulas. We now give two examples of the calculation of an upper bound for a given formula and the construction of a flow graph for a given time window.

Example 1 Let us consider a trace and the formula $a\ \mathrm{U}_{<10}\ (b\ \mathrm{U}_{<10}\ c)$, containing propositions a, b, c evaluated at time $t = 0$. Based on the semantics of temporal operators, we achieve the timing bounds $t \in]0, 10[$ and $t \in]0, 20[$, respectively. These time bounds are intervals where the truth values resulting from the evaluation of formulas may change. By the semantic nature of temporal operators, we know that for any $t \notin]0, 10[\cup]0, 20[$ the truth value is maintained constant, which gives us the desired bound for changes of the evaluation value.

Example 2 In order to estimate the amount of time required from the system under observation to couple monitors in a safe manner, we can use a pessimistic approach based on the assumption of a maximum inter-arrival time of events in the system, or we can pre-compute the flow graph of the application. Based on these, we are able to infer how many events will be triggered in a certain time interval. To exemplify the specific case of the latter, we define a time window given by a certain formula using the previous approach. Then, we create a flow graph of the entire system and fix the starting point of the system as depicted in the partial flow pattern of the events (ranging from symbol A to M) under monitoring in Fig. 3. From label α to β, where α corresponds to the beginning of the execution and β corresponds to the end of the execution, we have the flow of the main task composed by three paths (the task that manages the autopilot controller), and from label *1* to *4*, we have the optional task (a time-triggered task for device drivers execution that need to execute at least 1 time in a second). The optional task has two times the period of the main task. In summary, we have at most four events between α and β and the optional task executes twice between them. The figure also depicts the dependencies of events and allows us to estimate the required relative time for some events.

Altogether, these examples combine temporal settings of the monitors and the system itself: the first one give us the amount of time that we need to wait for a verdict (minimum time granularity); the second one helps us to find the period for a monitor based on the time behavior of the system under monitoring as well as to estimate the *worst-case execution time* (WCET) of the monitor (i.e., the time complexity times a constant).

Timing guarantees of hard real-time systems are commonly pessimistic [7]. Given that, it is not good to have monitors always executing in constant time since they may

[1] By constant time, we mean that a monitor executes the same number of CPU cycles at each invocation.

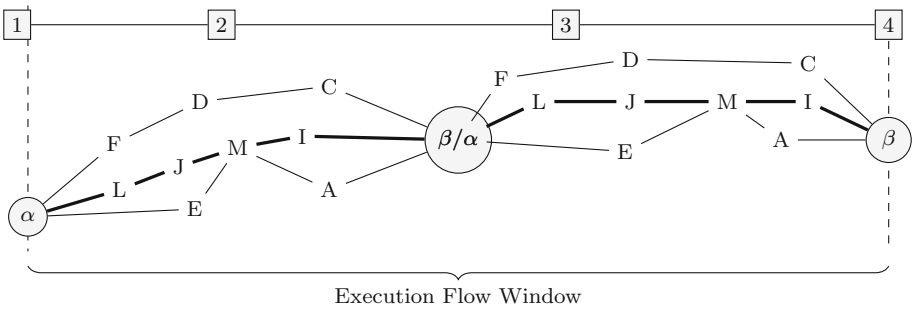

Fig. 3 Flow graph of the system-enabled events defined in a time window

consume more time than required in average. In order to produce coherent timing verdicts of monitors without assuming any specific scheduler, a hierarchy of monitors should be employed. The main monitor is required to execute in constant time to supervise the other monitors, that can be executing without any restriction of time. Given that, as time elapses the main monitor will ensure the timing guarantees of the other monitors, and these monitors will supervise the main application. Now, we are able to use our framework to settle on any real-time scheduler.

A hierarchy supervisor monitor is based on the idea of guaranteeing a monitor that is correct-by-construction and executes in constant time and constant space. This allows us for adaptability of new monitors, as well as new system functions. To give a constant-time implementation of a monitor, we need to fix the sample size for the trace that the supervisor monitor uses for incremental evaluation and use the symbol-based execution for arbitrary n steps. However, we do not have guarantees that the maximum delay detection will be ensured. For that we need to consider the rate of the events that scheduler and monitors trigger. This is relatively simple since monitors are time- or event-triggered or both. Since counting events is performed in constant time, we have a monitor that will count the events and verify whether they respect the amount of events allowed by the system. Note that this is safe by itself since the assumption is also monitored.

Note that none of the related works described in Sect. 2 have a focus on a hierarchy of trusted monitors. At most, they assume that the monitors execute as fast as possible, and in the absence of an RTOS, the scheduling is achieved by assigning the hardware interrupt routines hard-coded for each task or subroutine.

6 Specification for synthesis with RMTL-$\int_3$

The adopted formalism supports an explicit notion of time that is required for the timing analysis of RTSs. Support of inequalities, durations and quantification over these, increases the expressiveness of classic temporal logics to specify explicit timing settings, filling a gap in the common specification languages for RTSs. Increasing the expressiveness of temporal logics may introduce decidability issues; the interest of decidable fragments, like RMTL-$\int_3$, is that the existence of an effective procedure that always evaluates any formula in any model as a truth value is guaranteed. In practice, the existence of this procedure implies that a monitor always terminates drawing a verdict, which is indeed important in runtime monitoring applications, and even more important in the context of hard real-time systems.

Let a be a coefficient represented by a logic variable. Duration terms of the form $a \times \int^{\eta_1} \psi_1$ can be synthesized if the coefficient a is constrained by polynomial inequalities, or if the coefficient a with distribution *Beta* or *Dirichlet* is employed. Under these restrictions, the `rmtld3synth` tool is able to generate monitors that evaluate conditional probabilities of random actions of RTSs. For instance, these monitors can be used to monitor the inflation and the deflation of imprecise tasks, which is required when imprecise computation models are employed. Moreover, the degradation of the system can also be specified by defining liveness properties such as "a task cannot execute for less that 5 time units in one interval of 100 time units."

Two use cases for monitoring of the Ardupilot autopilot framework are described in this paper. The first is a simple case that exemplifies the quantification of linearly constrained duration formulas, to illustrate how to generate monitoring conditions in C++. Use Case (2) explores how to encode uncertainty by using polynomial inequalities to constrain quantified duration formulas.

Use Case (1):

Resource models (RM) are mechanisms provided to establish amounts of shared resources to be consumed by working tasks in RTSs. Normally, these mechanisms focus on time consumption and ensure *time isolation* between different tasks or sets of tasks. *Periodic RMs* are defined by their *replenishment period* and *budget supply*. Budgets are dynamically available as the time elapses and are replenished at certain defined periods. *Elastic RMs* are an extension of periodic RMs containing *elastic coefficients*, similar to *spring coefficients* in physics. They describe how the execution time of a task can be temporally deflated or inflated by applying n-D geometric region constraints (polynomial inequalities) over resource budgets. These restricted coefficients allow for

the system's underload and overload to be controlled. Spring coefficients, which are seen as logic variables, define the rate (or constraint) of inflation and deflation of a resource (in our case, processing time) and can be changed during execution. In this use case, these coefficients are governed by linear inequality constraints, which dictate the under- and overloading conditions of a certain set of tasks.

Example 3 Consider the formula

$$0 \leq a \times \int^{\pi_1} \psi_1 + b \times \int^{\pi_2} \psi_2 \leq \frac{1}{4}\theta$$

that specifies the resource constraints of two RMs where coefficients are managed according to the linear equation $a = 1 - b$ for $a, b \geq \frac{1}{4}$, that ψ_1,ψ_2 are two formulas describing the event releases of two distinct tasks, and that θ is the allowed execution time for the RMs. Informally, the formula specifies that both resource models have different budgets when both execute at the same time, which in practice is the case when both RMs interfere in the system. To find the conditions for monitoring, we need to quantify the formula, yielding a new formula

$$\exists_{\{a,b\}} \left(a = 1 - b \wedge a > \frac{1}{4} \wedge b > \frac{1}{4} \wedge 0 \leq a \times \int^{\pi_1} \psi_1 + b \times \int^{\pi_2} \psi_2 \leq \frac{\theta}{4} \right).$$

Later, after applying the simplification algorithm described in [16], we generate the monitoring conditions from Example 3, as follows:

$$\begin{array}{lll} \left(\int^{\pi_1} \psi_1 = 0 \wedge 0 \leq \int^{\pi_2} \psi_2 < \theta\right) & \vee & \left(0 < \int^{\pi_1} \psi_1 < \frac{\theta}{4} \wedge 0 \leq \int^{\pi_2} \psi_2 < \theta - 3\int^{\pi_1} \psi_1\right) \quad \vee \\ \left(\int^{\pi_1} \psi_1 = \frac{\theta}{4} \wedge 0 \leq \int^{\pi_2} \psi_2 \leq \frac{\theta}{4}\right) & \vee & \left(\frac{\theta}{4} < \int^{\pi_1} \psi_1 < \frac{\theta}{3} \wedge 0 \leq \int^{\pi_2} \psi_2 < \frac{\theta - \int^{\pi_1} \psi_1}{3}\right) \quad \vee \\ \left(\int^{\pi_1} \psi_1 = \frac{\theta}{3} \wedge \theta - 3\int^{\pi_1} \psi_1 \leq \int^{\pi_2} \psi_2 < \frac{\theta - \int^{\pi_1} \psi_1}{3}\right) & \vee & \left(\frac{\theta}{3} < \int^{\pi_1} \psi_1 < \theta \wedge 0 \leq \int^{\pi_2} \psi_2 < \frac{\theta - \int^{\pi_1} \psi_1}{3}\right), \end{array}$$

where ψ_1 and ψ_2 are both simplified formulas.

In Fig. 4a, we can see regions where the RMs are able to consume resources or not. For instance, the resource B cannot consume any time units if resource A consumes 10 units, and the resource A can only consume more than 4 units if the resource B consumes less than 2 time units, due to resource constraints. For the case of both resources consuming 2.5 units each, the difference between the sum and the execution time indicates that the interference of both resource models executing concurrently is at most 5 time units (it is identified by the dashed region). Intuitively, this constraint means that one resource needs to be deflated when the other resource is inflated and conversely. Note that different regions can be found by modifying the constraints of the scale factor $\frac{1}{4}$, or any of the θ, a or b parameters.

Since in Example 2 only a quantification over durations was observed, let us now give an example combining multiple polynomial inequalities and temporal operators.

Example 4 Consider the constraint $Cons_1(x, y)$ defined by

$$\left(\left(8x + 5(y-1)^3 < 0 \wedge y \leq 1\right) \vee \left(5x^3 + 15x + 8y < 15x^2 + 5 \wedge x \leq 1\right)\right) \wedge \left(5x \geq 1 \vee x^2 + y^2 > \frac{1}{25}\right),$$

as depicted in Fig. 4b for different x and y variables. To provide different time restrictions for operation modes of an autopilot, we assign different variants of constraint formulas $Cons_i$ with index $i \in \{1, 2, 3\}$. This means that each mode of operation has different resources available. The functional and explicit time behavior is defined as follows:

$$\square_{<\infty} \bigwedge_{i \in \{1,2,3\}} \left(mode_i \rightarrow Cons_i \left(\int^{\pi_1} \psi_1, \int^{\pi_2} \psi_2 \right) \right),$$

where ψ_1, ψ_2 are formulas to be measured according to the periods π_1 and π_2, respectively, and $mode_i$ is the proposition that evaluates to true when mode i begins. Figure 5 shows the constraints for the three modes of operation. Considering that context switches influence the mode of operation, we can have a scenario such as "if the RM A measured by the variable y has more than 10 context switches per each period, then the $mode_1$ is unsafe and $mode_2$ or $mode_3$ should be used."

The $mode_2$ should be avoided if context RM A has more than 100 context switches per second. Using this specification, we can include the overhead of the RMs in the specification without the need to measure it at runtime, as it requires several symbols and indirectly increases the monitor space and time consumption.

Use Case (2): A conditional probability for a given duration measure for tasks can be specified using this formalism. We will next evaluate the likelihood of the remaining tasks in a system to be unscheduled, based on the overload of a certain task. This example applies in the context of RMs monitoring and also of imprecise computation monitoring. Let a be defined as a coefficient with uncertainty. Any probability distribution that can be described using polynomial inequalities can be encoded

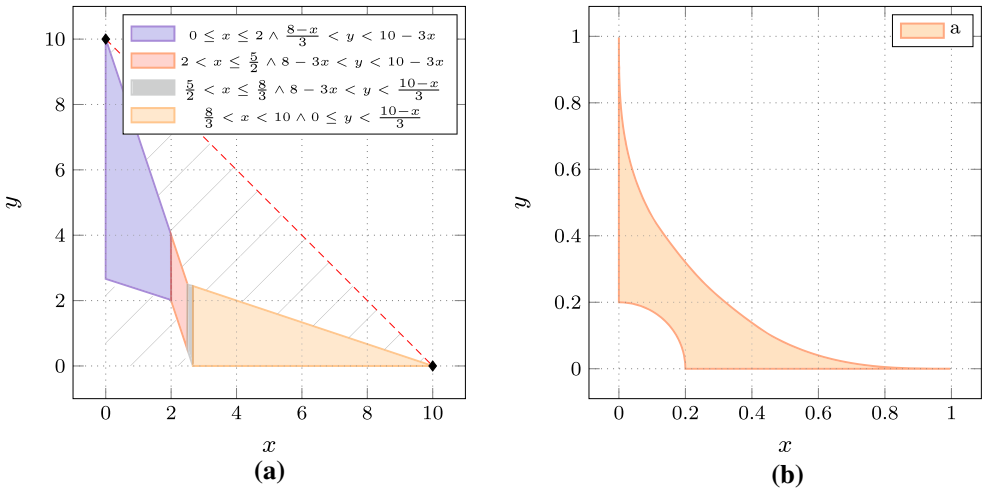

Fig. 4 **a** Regions of decomposed inequalities with duration x, y and $\theta = 10$. **b** Polynomial inequality constraint with normalized measures for two variables

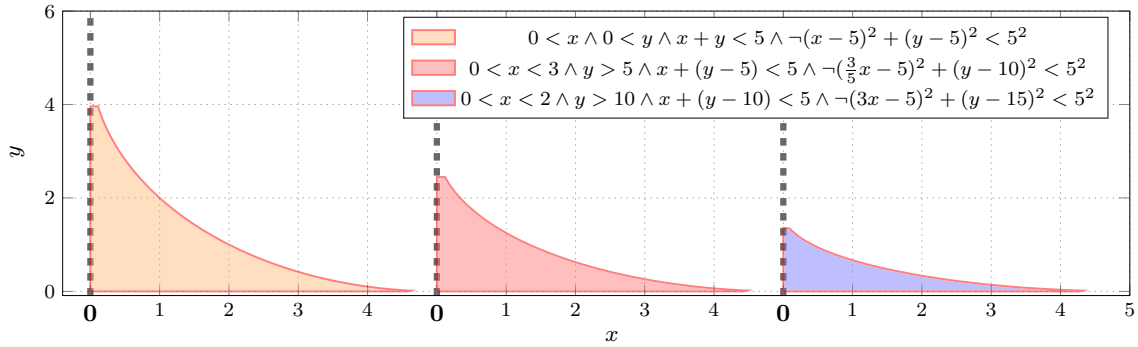

Fig. 5 Three regions defined by $Cons_i$ for $i \in \{1, 2, 3\}$

using this approach. Here we will focus on the *Beta* distribution only, but other interesting distributions, such as multinomial and Dirichlet distributions, could be equally used.

Let X and Y behave as two random variables with distribution $Beta(a_i, b_i)$ for $i \in 0, 1$. To encode these random variables in RMTL-$\int_3$, we define the *Beta pdf* as a constraint of the form $\frac{\hat{f}(1-x,\beta-1)\,\hat{f}(x,\alpha-1)}{C_\beta}$, where C_β is simplified and equal to $B(\alpha, \beta)$, and $\hat{f}$ is the power function. Power functions can be encoded in RMTL-$\int_3$ with the following axiom $y = \sqrt[a]{x^b} \Leftrightarrow x^b = y^a \vee y = x^{\frac{b}{a}}$, for any $x, y \in \mathbb{R}_{\geq 0}$, $a, b \in \mathbb{Q}_{>0}$. Any function $\hat{f}$ may now be encoded in RMTL-$\int_3$. The *Beta distribution* $p = f_{\beta,\alpha}(x)$ is now fully defined by $y^{a_1} = (1-x)^{b_1} \wedge z^{a_2} = x^{b_2} \wedge \frac{y \times z}{C_\beta} = p$, where $a_i, b_i \in \mathbb{N}$, $i \in \{1, 2\}$ are solutions of the formulas $\frac{a_1}{b_1} = \beta - 1$ and $\frac{a_2}{b_2} = \alpha - 1$, and p stands for the probability of the logical variable x in the interval $[0, 1]$.

Intuitively, the idea is to specify non-deterministic actions based on the information provided at execution time. For instance, a system can change its *modus operandis* if for some reason the probability of a given overload is greater than a certain fixed probability threshold. Note that these probabilistic inequality constraints will be used as monitoring conditions. The generation of monitoring conditions based on simplification approaches, as in Use Case (1), is only required if quantifiers are applied.

Let us consider without loss of generality the case of two tasks, where the first one may have a chance to overload, and the second one should avoid this by self-deflating. The specification of probabilistic coefficients that allows elasticity when overload situations occur is encoded by

$$a = \int^{\lambda} \psi_1 \wedge \Box_{<\infty} \left(\left(f_{\beta,\alpha}(a) < \frac{3}{4} \right) \rightarrow \left(\Diamond_{<\lambda+\lambda_2} \psi_d \right) \right),$$

where ψ_d is defined as $(\int^{\lambda_2} \psi_2) < b \times d$, a and b are restricted by one polynomial inequality constraint (e.g., $a = b + 1$), d is the maximum allowed execution time for a task, and ψ_1, ψ_2 are the formulas defined for each of the two tasks (e.g., conjunction of propositions for specifying a certain task or RM).

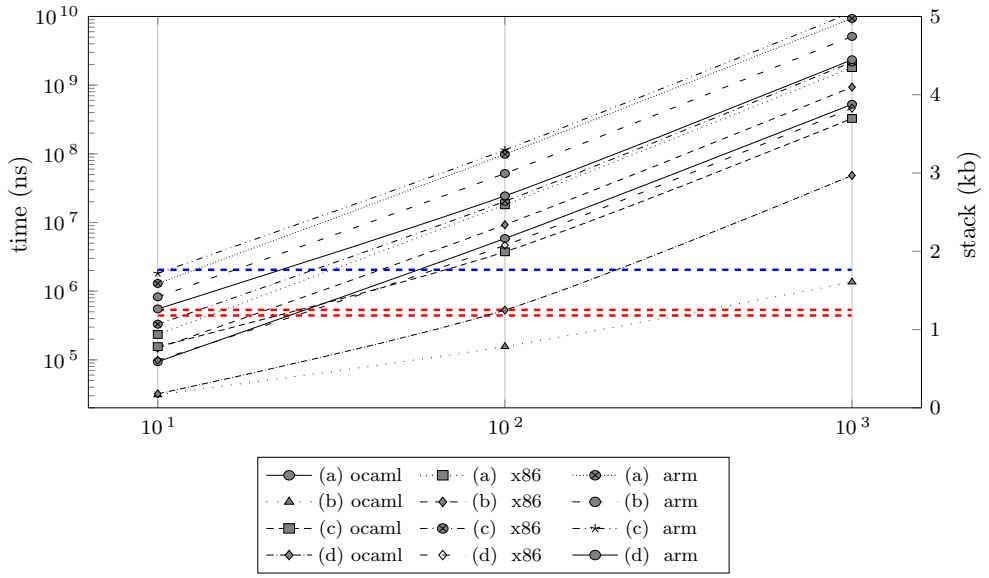

Fig. 6 Comparison of implementations/architectures

7 Experimental results

Before discussing the experimental results for the presented use cases, we compare our results with the ones presented in [16], where we show that one element takes in average 400 ns to be processed using an Intel x86 machine. For that, we re-use the Ocaml source code to compare with our present setting.

For comparing both implementations, we have used the following set of RMTL-$\int_3$ formulas: (a) $true\ U_{\leq t}\ \phi$ (eventually); (b) $\phi \rightarrow \Box_{\leq t}\ \psi$ (bounded invariance); (c) $\Box_{\leq t}\ \int^t \phi \leq \beta$ (limited duration); and finally (d) $\phi \rightarrow \int^t \psi \leq \beta$ (bounded duration). For each formula, we have used different trace sizes ranging from 10 to 10^3. The traces that we consider are selected as the traces that maximize the execution time of each formula evaluation. We have run the experiments in two distinct architectures, namely the ARM(armv7) and the x86(i686) architectures, whereas the OCaml experiments were only performed in the x86 architecture.

The PixHawk [2] board is the target platform to execute periodic monitors that were synthesized from RMTL-$\int_3$ formulas into `C++11`. We have also tested the same implementation using an Intel Core i3-3110M at 2.40 GHz CPU with 8 GB of RAM memory, and running Windows 10 Embedded x86 in a virtual machine running on a Fedora 23 X86'64 host.

In the case of the PixHawk board, we have only 256 kb of memory RAM for the overall system, and we assign at most 90% of the processor usage for these monitoring experiments. From the experimental results presented in Fig. 6, we can conclude that such monitors execute in polynomial time as the trace increases, which is in accordance with the theoretical results presented in [16]. [2] The stack consumption is also acceptable for the PixHawk board. The constant upper dashed line is the maximum stack consumption of 1.76 kb for the formula (c), and the other two lines represent the lower bounds of the reaming three formulas that have a very similar stack usage.

Different lines are depicted in Fig. 6. They correspond to different execution times and stack experiments: The lines tagged with "ocaml" refer to the execution of the original evaluation algorithm using Ocaml; the ones tagged with "x86" are the execution time of the `C++11` implementation in the same platform of the Ocaml test; and finally, the ones tagged with "arm" refer to the execution time of the `C++11` implementation in the PixHawk board. In these experiments, we do not consider more than two nested *until* operators, which is indeed a common pattern of formulas for the specification of embedded systems—we do not have any evidence of how deep nested U operators can be used in a real application scenario.

Case study experiments. Ideally, lightweight controller systems should use elastic execution time for tasks, in order to enable the required adaptability for reducing overload situations.

Ardupilot [31] supports several platforms such as AVR, ARM (based on NuttX[32]), and X86 (based on the Linux kernel). Recently, Ardupilot has adopted nonlinear Kalman filters for the *attitude and heading reference system* (AHRS).

[2] The instructions to generate the `C++11` code files that are the output of the use case experiments are fully detailed in [29] .

It is a demanding process that can only be executed in the PixHawk board. For this ARM architecture, two versions are available to perform the same tasks as in imprecise computation definitions. The faster one adopts *direction cosine matrix* (DCM), which is sufficient for the majority of the cases (but is less accurate). The slower version reveals that AHRS can be much better for heavy copters. Ardupilot for the AVR architecture contains several sub-tasks that are scheduled using cyclic scheduling rules. It uses the *Hardware Abstraction Library* (HAL) to communicate with the devices directly, using interrupt-driven routines. However, Ardupilot for PixHawk uses the HAL to communicate with device drivers that are implemented as separate tasks running on NuttX. The RTOS runs a single main task as defined by the AVR architecture and, instead of using interrupt-driven routines, uses four optional tasks that should be executed at least once each second. These optional tasks have different purposes, such as controlling the IO, the UART, and managing timing events and storage (system drivers). The main task contains sub-tasks that execute cyclically in different frequencies ranging from 20 to 400 hz, dictated by the defined cyclic scheduler. The execution rule for sub-tasks is: *based on the predicted WCET, an optional task will execute if there exists available time*.

For designing a safe autopilot, we are required to ensure time-space isolation. This is crucial for autopilot tasks that have not been formally verified, or are still undergoing testing. To the best of our knowledge, none of the currently available autopilot systems for radio control copters have been formally verified. They may well generate absurd values due to *hardware failures* and are susceptible to *introduced code attacks*, via radio-frequency telemetry links [33].

Let us now analyze the impact of the use cases in the Ardupilot firmware. Use Case (1) is composed of several disjunctions, meaning that each branch of the formula can take different execution times. However, the results demonstrate that these formulas are not out of the scope of the previous experiments. The stack usage is 3.4 kb for Use Case (1) and 4.3 kb for the formula proposed in Use Case (2). Based on that, the execution times are on average faster than the worst case considered. Usually, the monitor increases its execution time as more events are triggered, which means that if the set of events selected for a system is subdivided in different buffers (when possible), then the monitoring will generate lower overheads. However, the impact of the overheads in the Ardupilot is *not* negligible. The overhead generated in the system is 10us/1s for the instrumentation of two sub-tasks and is 50ms/1s for the monitor (the sub-tasks have a period of 10ms and 5ms, respectively). We also have an idle time of about 40% percent. Monitor buffer length is fixed to 100 elements, which is the value obtained according to the pre-calculated time interval required for the formulas under synthesis, and we consider a maximum inter-arrival time of 1ms. The monitors execute with a period of 1s. The input files for Use Case (1) and Use Case (2) can be found online [34] (config files "usecaseone" and "usecasetwo"), together with the step-by-step procedure for monitor generation using the rmtld3synth tool.

8 Discussion and future work

Synthesis of RMTL-$\int_3$ into classical *timed automata* (TA) appears to be unfeasible for RV due to the state explosion problem. Encoding time can only be possible if we make use of more expressive classes of automata, such as TA extended with stopwatches [35]. However, the reachability problem for these classes is undecidable, which implies that no gain should be expected from the point of view of either static analysis or of space complexity for runtime verification purposes. Based on this, we have decided to construct an RV framework building on previous work [16].

Another important point is the expressiveness of the logic that has been adopted for this work. Contrary to MTL, which is not sufficiently expressive to deal with explicit durations of propositions/events, our experimental results have revealed that adopting RMTL-$\int_3$ allows for properties to be specified at the abstraction level of counting time and to be efficiently synthesized for a platform as small as PixHawk, which is certainly impressive.

Yet, regarding the expressiveness and computing feasibility of timed temporal logics, the unbounded *Since* operator was not considered in this work, because it requires a full history of a trace. This is not feasible in the context of lightweight real-time embedded systems where resources are scarce. It is known from [36] that for each formula containing the *Since* operator there exists a corresponding formula making use of its dual *Until* operator, which further justifies our exclusive use of the latter operator in this work.

Our experiments have shown that the adoption of simplification techniques for synthesis should be encouraged, in order to reduce the execution time as well as the stack usage. This is important, as the number of monitors and formulas increases. Predicting the size of the traces is also important, and more clever solutions should be investigated, for instance along the lines of the idea proposed in [37]. Instead of estimating the best periods, we could formulate an optimization problem to find the smallest trace size that is sufficient for both the application and the monitor.

The overall conclusion of our work is that software monitoring techniques, which draw verdicts about timing software faults as well as hardware timing failures, are valid and may be extremely useful to complement the fault-tolerant mech-

anisms [1,38] that are used for the detection of abnormal mechanical failures.

A `rmtld3synth` tool User's Guide

The `rmtld3synth` synthesis tool is able to automatically generate monitors based on the formal specifications written in RMTL-$\int_3$. Polynomial inequalities are supported by this formalism as well as the most common operators of temporal logics. Furthermore, quantification is also considered in the language of RMTL-$\int_3$ as a means to facilitate the decomposition of the quantified formulas into several monitoring conditions.

We will now present an overview of the typical process for generating monitors for Ocaml and `C++11` languages using this tool, together with a running example of a simple monitoring case generation. We begin by the running example, present the generated monitors, and show how to configure the RV monitoring model to couple with the system.

Consider the formula

$$(a \rightarrow ((a \vee b)\, \mathrm{U}_{<10}\, c)) \wedge \int^{10} c < 4 \tag{1}$$

that intuitively describes that given an event a, b occurs until c and, at the same time, the duration of b shall be less than four time units over the next 10 time units. For instance, a trace that satisfies this formula is

$$(a, 2), (b, 2), (a, 1), (c, 3), (a, 3), (c, 10).$$

From `rmtld3synth2ocaml` tool, we have synthesized the formula's example into the code of Listing 4. For that, we have used the command in Listing 2.

```
./rmtld3synth --synth-ocaml --input-
    latexeq "(a \rightarrow ((a \lor b
    ) \until_{<10} c)) \land \int^{10}
     c < 4"
```

Listing 2 Utilized shell command for Equation 1

Next, we can also generate `C++11` monitors by replacing `-synth-ocaml` with `-synth-cpp11`. The outcome is the monitor illustrated in Listing 5. To use those monitors, we need to define a trace for Ocaml reference as in Listing 3.

```
module OneTrace : Trace = struct let
    trc = [("a",(0.,2.));("b",(2.,4.))
    ;("a",(4.,5.));("c",(5.,8.));("a
    ",(8.,11.));("c",(11.,21.))] end;;

module MonA = Mon0(OneTrace);;
```

Listing 3 Ocaml's reference code for monitor instantiation

For the Ocaml language, experimental integration with RTMLib is available. However, we do not describe it here, but refer

```
open List
open Rmtld3

module type Trace = sig val trc : trace end
module Mon0 ( T : Trace ) = struct
let compute_uless gamma f1 f2 k u t =
 let m = (k,u,t) in
 let eval_i b1 b2 =
   if b2 <> False then b3_to_b4 b2 else if b1 <> True && b2 = False then b3_to_b4 b1 else Symbol
 in

 let eval_b (k,u,t) f1 f2 v =
   if v <> Symbol then v else eval_i (f1 k u t) (f2 k u t)
 in

 let eval_fold (k,u,t) f1 f2 x =
   fst (fold_left (fun (v,t') (prop,(ii1,ii2)) -> (eval_b (k, u, t') f1 f2 v, ii2)) (Symbol,t) x)
 in

 if not (gamma >= 0.) then
   raise (Failure "Gamma_of_U_operator_is_a_nonnegative_value")
 else
 begin
   let k,_,t = m in
   let subk = sub_k m gamma in
   let eval_c = eval_fold m f1 f2 subk in
   if eval_c = Symbol then
     if k.duration_of_trace <= (t +. gamma) then Unknown else ( False ) else b4_to_b3 eval_c
 end

let compute_tm_duration tm fm k u t =
 let dt = (t,tm k u t) in

 let indicator_function (k,u) t phi = if fm k u t = True then 1. else 0. in
 let riemann_sum m dt (i,i') phi =
   (* dt=(t,t') and t in ]i,i'] or t' in ]i,i'] *)
   count_duration {:=} !count_duration + 1 ;
   let t,t' = dt in
   if i <= t && t < i' then
     (* lower bound *)
     (i'-.t) *. (indicator_function m t phi)
   else (
     if i <= t' && t' < i' then
       (* upper bound *)
       (t'-.i) *. (indicator_function m t' phi)
     else
       (i'-.i) *. (indicator_function m i phi)
   ) in
 let eval_eta m dt phi x = fold_left (fun s (prop,(i,t')) -> (riemann_sum
 m dt (i,t') phi) +. s) 0. x in
 let t,t' = dt in
 eval_eta (k,u) dt fm (sub_k (k,u,t) t')

 let env = environment T.trc
 let lg_env = logical_environment
 let t = 0.
 let mon = (fun k s t -> b3_not ((fun k s t -> b3_or ((fun k s t -> b3_not ((fun k s t -> b3_or ((
         fun k s t -> b3_not ((fun k s t -> k.evaluate k.trace "a" t) k s t)) k s t) ((
         compute_uless 10. (fun k s t -> b3_or ((fun k s t -> k.evaluate k.trace "a" t) k s t) ((
         fun k s t -> k.evaluate k.trace "b" t) k s t)) (fun k s t -> k.evaluate k.trace "c" t))
         k s t)) k s t)) k s t) ((fun k s t -> b3_not ((fun k s t -> b3_lessthan ((
         compute_tm_duration (fun k s t -> 10.) (fun k s t -> b3_or ((fun k s t -> k.evaluate k.
         trace "c" t) k s t) ((fun k s t -> k.evaluate k.trace "d" t) k s t))) k s t) ((fun k s t
          -> 4.) k s t)) k s t)) k s t)) env lg_env t
end
```

Listing 4 Generated Ocaml monitor

the reader for the examples in `rmtld3synth`'s repository [3]. For `C++11` we will now briefly describe how it is performed. Given the verbosity of the generated code, we have removed the conjunction including the duration inequality and used instead the simple formula

$$\int^{10} c < 4.$$

Now, we describe the settings for constructing the RV monitoring model.

Overview of the configuration settings. The settings for `rmtld3synth` tool are defined using the syntax

```
(<setting_id> <bool_type | integer_type | string_type>)
```

[3] Available at https://github.com/anmaped/rmtld3synth/tree/v0.3-alpha, version 0.3-alpha.

```
#ifndef _MON0_COMPUTE_H_
#define _MON0_COMPUTE_H_
#include "rmtld3.h"

auto _mon0_compute = [](struct Environment &env, timespan t) mutable -> three_valued_type { return [](struct Environment env, timespan t) -> three_valued_type { auto tr1 = [](struct Environment env, timespan
        t) -> duration {

 auto eval_eta =  [](struct Environment env, timespan t, timespan t_upper, TraceIterator< int > iter) -> duration
 {
   auto indicator_function = [](struct Environment env, timespan t) -> duration {
     auto formula = [](struct Environment &env, timespan t) mutable -> three_valued_type { auto sf1 = [](struct Environment &env, timespan t) mutable -> three_valued_type { return env.evaluate(env, 2, t); }(
             env,t); auto sf2 = [](struct Environment &env, timespan t) mutable -> three_valued_type { return env.evaluate(env, 1, t); }(env,t); return b3_or (sf1, sf2); }(env, t);

     return (formula == T_TRUE)? std{:}{:}make_pair (1,false) : ( (formula == T_FALSE)? std{:}{:}make_pair (0,false) : std{:}{:}make_pair (0,true)) ;

   };

   // compare if t is equal to the lower bound
   auto lower = iter.getLowerAbsoluteTime();
   // compare if t is equal to the upper bound
   auto upper = iter.getUpperAbsoluteTime();

   timespan val1 = ( t == lower )? 0 : t - lower;
   timespan val2 = ( t_upper == upper )? 0 : t_upper - upper;

   auto cum = lower;

   // lets do the fold over the trace
   return std{::}accumulate(
     iter.begin(),
     iter.end(),
     std{::}make_pair (make_duration (0, false), (timespan)lower), // initial fold data (duration starts at 0)
     [&env, val1, val2, &cum, t, t_upper, indicator_function]( const std{::}pair<duration,timespan> p, Event< int > e )
     {
       auto d = p.first;
       auto t_begin = cum;
       auto t_end = t_begin + e.getTime();
       cum = t_end;
       auto cond1 = t_begin <= t && t < t_end;
       auto cond2 = t_begin <= t_upper && t_upper < t_end;
       auto valx = ((cond1)? val1 : 0 ) + ((cond2)? val2 : 0);
       auto x = indicator_function(env, p.second);

       return std{::}make_pair (make_duration (d.first + (x.first * ( e.getTime() - valx )), d.second || x.second), p.second + e.getTime());
     }
   ).first;
 };

 // sub_k function defines a sub-trace
 auto sub_k = []( struct Environment env, timespan t, timespan t_upper) -> TraceIterator< int >
 {
   // use env.state to speedup the calculation of the new bounds
   TraceIterator< int > iter = TraceIterator< int > ( env.trace, env.state.first, 0, env.state.first, env.state.second, 0, env.state.second );
   // to use the iterator for both searches we use one reference
   TraceIterator< int > &it = iter;

   ASSERT_RMTLD3( t == iter.getLowerAbsoluteTime() );
   auto lower = env.trace->searchIndexForwardUntil( it, t);
   auto upper = env.trace->searchIndexForwardUntil( it, t_upper - 1 );

   // set TraceIterator for interval [t, t + dif
   it.setBound(lower, upper);

   // return iterator ... interval length may be zero
   return it;
 };

 auto t_upper = t + make_duration(10.,false).first;

 return eval_eta(env, t, t_upper, sub_k(env, t, t_upper));

 }(env,t);

 auto tr2 = make_duration(4.,false);
 return b3_lessthan (tr1, tr2);
}(env,t);};
#endif //_MON0_COMPUTE_H_

#ifndef MONITOR_MON0_H
#define MONITOR_MON0_H
#include "Rmtld3_reader.h"
#include "RTML_monitor.h"
#include "mon0_compute.h"
#include "mon1.h"

class Mon0 : public RTML_monitor {

private:
 RMTLD3_reader< int > trace = RMTLD3_reader< int >( __buffer_mon1.getBuffer(), 0. );

 struct Environment env;

protected:
 void run(){
    three_valued_type _out = _mon0_compute(env,0);
   DEBUG_RTEMLD3("Veredict:
 }

public:
 Mon0(useconds_t p):RTML_monitor(p,SCHED_FIFO,50), env(std{::}make_pair(0,0),&trace, __observation) {}
};
#endif //MONITOR_MON0_H
```

Listing 5 Generated C++11 monitor

```
(gen_tests true)
(minimum_inter_arrival_time 102)
(maximum_period 2000000)
(event_subtype uint_8)
(cluster_name monitor_set1)

(m_simple 1000000 (Or (Until 200000 (Prop A) (Prop C))
     (Prop B)))
(m_morecomplex 500000 (Or (Until 200000 (Prop set_off)
     (Or (Until 200 (Prop A) (Prop C)) (Prop B))) (
    Prop B)))
```

Listing 6 The default configuration file.

where | distinguishes between the supported types of arguments such as Boolean, integer or string, and `setting_id` is a string containing the name of the setting to which values are assigned. An example of a set of possible settings for the tool is given in the first five lines of Listing 6.

We now briefly describe the purpose of each of the setting entries present in Listing 6:

- `gen_tests` sets the automatic generations of test cases (to be used as a demo for testing monitor's execution).
- `gen_concurrency_tests` constructs tests for testing lock- and wait-free monitors executing concurrently.
- `gen_unit_tests` constructs tests for `C++11` synthesis using the Ocaml source code as an oracle.
- `buffer_size` sets the static size of the buffer to be used (`rmtld3synth` tool can change it if required by some constraints).
- `minimum_inter_arrival_time` establishes the minimum inter-arrival time that the events can have. It is a very pessimistic setting but provides some information for static checking.
- `maximum_period` sets the maximum interval between two consecutive releases of a task's job. It has a correlation between the periodic monitor and the minimum inter-arrival time. It provides static checks according to the size of time stamps of events.
- `event_type` provides the type for dealing with events (commonly is a class parameter).
- `event_subtype` provides the type for the event data. In that case, it is an identifier that can distinct 255 events. However, if more events are required, the type should be modified to *uint32_t* or greater. The number of different events versus the available size for the identifier is also statically checked.
- `cluster_name` identifies the set of monitors. It acts as a label for grouping monitor specifications.

Writing formulas in RMTLD3 The formulas "m_simple" and "m_morecomplex" follow the same syntax defined in this document. For setting a periodic monitor, we use (m_usecase1 <period> (<monitor sexpr>)). They are formatted as a symbolic expression. The type of monitors in Ocaml is according to Listing 7.

```
type var_id = string with sexp
type prop = string with sexp
type time = float with sexp
type value = float with sexp

type formula =
        True of unit
      | Prop of prop
      | Not of formula
      | Or of formula * formula
      | Until of time * formula * formula
      | Exists of var_id * formula
      | LessThan of term * term
and term =
        Constant of value
      | Variable of var_id
      | FPlus of term * term
      | FTimes of term * term
      | Duration of term * formula
with sexp

type rmtld3_fm = formula with sexp
type rmtld3_tm = term with sexp
type tm = rmtld3_tm with sexp
type fm = rmtld3_fm with sexp
```

Listing 7 The inductive type.

References

1. Ranjbaran, M., Khorasani, K.: Fault recovery of an under-actuated quadrotor aerial vehicle. In: CDC, pp. 4385–4392 (2010)
2. Meier, L., Honegger, D., Pollefeys, M.: Px4: a node-based multithreaded open source robotics framework for deeply embedded platforms. In: ICRA, pp. 6235–6240 (2015)
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
5. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods, volume 54 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (2001)
6. Harrison, John: Handbook of Practical Logic and Automated Reasoning, 1st edn. Cambridge University Press, New York (2009)
7. Shin, I., Lee, I.: Periodic resource model for compositional real-time guarantees. In: RTSS, pp. 2–13 (2003)
8. Liu, J.W.S., Shih, W.-K., Lin, K.-J., Bettati, R., Chung, J.-Y.: Imprecise computations. Proc. IEEE **82**(1), 83–94 (1994)
9. Mizotani, K., Hatori, Y., Kumura, Y., Takasu, M., Chishiro, H., Yamasaki, N.: An integration of imprecise computation model and real-time voltage and frequency scaling. In: CATA, pp. 63–70 (2015)
10. Lakhneche, Y., Hooman, J.: Metric temporal logic with durations. Theor. Comput. Sci. **138**(1), 169–199 (1995)
11. Ouaknine, J., Worrell, J.: Some recent results in metric temporal logic. In: FORMATS '08, pp. 1–13, Springer, Berlin (2008)
12. Pike, L.: Modeling time-triggered protocols and verifying their real-time schedules. In: FMCAD, pp. 231–238, (2007)

13. Pedro, A.M., Pereira, D., Pinho, L.M., Pinto, J.S.: Logic-based schedulability analysis for compositional hard real-time embedded systems. SIGBED Rev. **12**(1), 56–64 (2015)
14. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: CAV, pp. 176–189. Springer, New York (2008)
15. Cimatti, A., Roveri, M., Tonetta, S.: Requirements validation for hybrid systems. In: CAV, pp. 188–203. Springer, New York (2009)
16. Pedro, A.M., Pereira, D., Pinho, L.M., Pinto, J.S.: Monitoring for a decidable fragment of mtld. In: RV, pp. 169–184. Springer, New York (2015)
17. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition: a synopsis. SIGSAM Bull. **10**(1), 10–12 (1976)
18. Chen, Y., Chang, L., Kuo, T.I., Mok, A.K.: Real-time task scheduling anomaly: observations and prevention. In: SAC, pp. 897–898. ACM, New York (2005)
19. Barringer, H., Rydeheard, D., Havelund, K.: Rule Systems for Run-Time Monitoring: From Eagle to Ruler. Springer, Berlin (2007)
20. Sammapun, U., Lee, I., Sokolsky, O.: Rt-mac: runtime monitoring and checking of quantitative and probabilistic properties. In: RTCSA, pp. 147–153 (2005)
21. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: RV'10, pp. 345–359. Springer, Berlin (2010)
22. Bauer, A., Kuster, J., Vegliach, G.: From Propositional to First-order Monitoring. Springer, Berlin (2013)
23. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. In: TACAS, pp. 341–356. Springer, New York (2014)
24. Bouyer, P., Chevalier, F., Markey, N.: On the expressiveness of TPTL and MTL. Inf. Comput. **208**(2), 97–116 (2010)
25. Koymans, Ron: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990)
26. Pnueli, A.: The temporal logic of programs. In: SFCS '77, pp. 46–57. IEEE Computer Society, Washington (1977)
27. Pedro, A.M.: rtmlib Monitoring Library. https://anmaped.github.io/rtmlib/doc/ (2016), version 0.1-alpha
28. The OCaml Development Team. Ocaml programming language (2013)
29. Pedro, A.M.: rmtld3synth Synthesis Tool. https://github.com/cistergit/rmtld3synth/ (2016), version 0.1-alpha
30. Nelissen, G., Pereira, D., Pinho, L.M.: A novel run-time monitoring architecture for safe and efficient inline monitoring. Ada-Europe **2015**, 66–82 (2015)
31. Coombes, M., McAree, O., Chen, W. H., Render, P.: Development of an autopilot system for rapid prototyping of high level control algorithms. In: Proceedings of 2012 UKACC CONTROL, pp. 292–297 (2012)
32. Nutt, G.: NuttX Real-Time Operating System. http://nuttx.org/ (2007), version 7.11
33. Schumann, J., Moosbrugger, P., Rozier, K.Y.: R2u2: Monitoring and diagnosis of security threats for unmanned aerial systems. In: RV 2015, pp. 233–249 (2015)
34. Pedro, A.M.: Use Case (1) and Use Case (2) configure files with settings. https://github.com/anmaped/rmtld3synth/tree/dev/config (2016). version 1
35. Cassez, F., Larsen, K.G.: The impressive power of stopwatches. In: CONCUR, pp. 138–152. Springer, Berlin (2000)
36. Hunter, P., Ouaknine, J., Worrell, J.: Expressive completeness for metric temporal logic. In: IEEE Computer Society, pp. 349–357 (2013)
37. Navabpour, S., Bonakdarpour, B., Fischmeister, S.: Time-triggered runtime verification of component-based multi-core systems. In: RV. Springer, New York (2015)
38. Mueller, M.W., D'Andrea, R.: Stability and control of a quadrocopter despite the complete loss of one, two, or three propellers. In: IEEE international conference on robotics and automation (ICRA), pp. 45–52, (2014)