Real-Time CCSL: Application to the Mechanical Lung Ventilator

Pavlo Tokariev, Frédéric Mallet

▶ **To cite this version:**

Pavlo Tokariev, Frédéric Mallet. Real-Time CCSL: Application to the Mechanical Lung Ventilator. ABZ 2024 – 10th International Conference on Rigorous State Based Methods, Jun 2024, Bergamo, Italy. pp.289-306, 10.1007/978-3-031-63790-2_24 . hal-04639949

HAL Id: hal-04639949

https://inria.hal.science/hal-04639949v1

Submitted on 9 Jul 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Real-Time CCSL: Application to the Mechanical Lung Ventilator

Pavlo Tokariev and Frédéric Mallet

Université Côte d'Azur, Inria, CNRS, i3S, Sophia Antipolis, France
Pavlo.Tokariev@inria.fr, Frederic.Mallet@univ-cotedazur.fr

Abstract. This case-study paper reports on our experience in modelling the mechanical lung ventilator using the Clock Constraint Specification Language (CCSL). CCSL captures the causal and temporal behaviour of a system by specifying constraints on logical clocks. Logical clocks are integer counters where the occurrence of an event, a *tick*, advances the counter and marks the advance in time. In this framework, chronometric clocks become logical clocks just with a special external meaning. Encoding chronometric clocks as counters may result in verification inefficiency and hard-to-read specifications.

The paper introduces in the language some real-time constructs to directly encode phenomena like clock drift, skew and jitter. This makes patterns explicit in turn enabling optimizations. To realize these optimizations, we alter the internal symbolic representation of clock constraints. We also introduce an explicit notion of parameters and intervals. While for some constraints it mainly consists of adding syntactic sugar and pre-processing facilities, we believe it improves the readability.

We illustrate the new constructs on the mechanical lung ventilator system. We start with a purely logical specification, we point at the sources of inefficiencies and then we discuss the benefits of the extensions on specific parts.

Keywords: Temporal requirements · Logical time · Real-Time

1 Introduction

As more and more systems become digital or include digital parts, the complexity of systems constantly increases. Thus, it requires revisiting the expressiveness of specification languages to cope with new features that need to be considered. Scientific challenges help compare the various expressiveness of solutions to identify pros and cons. We consider here the modelling of the mechanical lung ventilator provided as a challenge to the ABZ community and report on the experience.

The Clock Constraint Specification Language (CCSL) was defined as a specification language for capturing both timed and temporal aspects of safety-critical systems. Its formal semantics is used to conduct various kinds of analyses for the early detection of potential system flaws. In CCSL, recurring events are captured as logical clocks, i.e. (infinite) streams of ticks, where each tick stands for

an occurrence of this event. Logical clocks provide a generic mechanism to unify untimed events that are linked only by causal relationships and timed events usually related to physical phenomena associated with real-time. Chronometric clocks can measure timed events by associating a date with each occurrence. Logical clocks can both measure the progression of a system by observing the occurrence of an event, but also describe the activation conditions under which something good may happen or something bad must not happen. CCSL constraints serve both purposes, building probes to observe the system and activation conditions to control what should or should not happen.

While logical clocks give an elegant generic modelling framework, encoding everything as a logical clock may lead to some verification inefficiencies or difficult-to-read specifications. We use the example of the mechanical lung ventilator to identify such cases. We also provide new constructs that allow for optimization at the cost of a more complex internal symbolic representation.

Related work. The challenge of ABZ 2024 is about designing a mechanical lung ventilator [7] using formal methods. This usually includes, but not limited to, defining behaviour in state machines of various expressiveness and performing model checking of required liveness and safety properties (functional or operational).

Synchronous languages, such as Esterel [6] or Lustre [9] (and its industrial development, SCADE [10]), were devised as formal *programming* languages dedicated to the modelling of reactive systems. They rely on the synchronous hypothesis: the system consumes external actions, and may produce reactions in infinite loop, which happens when base clock ticks. This hypothesis is based on two assumptions. The loop body considered instantaneous regardless of actual computations and relation with the inputs and the state. And it requires to process the loop body at an adequate rhythm compared to the input rate so that no input is lost or aged.

Another way to approach the formal description of a system, called *declarative*, is to specify the expected behaviour without necessarily giving a unique operational way to execute it. In that sense, a specification describes possible valid or forbidden behaviours, but not necessary how to achieve actual valid executions. It is also a way to abstract the aspects important to requirement engineers or their customers from implementation details. The examples of specification languages or related formalisms include Timed Automata [2] (with UPPAAL model checker), Z specification language [24], AltaRica [5]. In this paper we shall focus on logical clock specifications, or more specifically, the Clock Constraint Specification Language (CCSL) [16].

Contributions. This paper proposes several extensions to the base language of CCSL and illustrates their application on the use case of the mechanical lung ventilator. We add real-time constraints to the language, which not only extend the expressiveness, but also add the possibility to provide better optimizations during the analysis. We discuss some elements of using this new information to build the analysis using abstract interpretation [11]. Other extensions are mostly

syntactic yet important to make the language suitable to describe large systems. An important addition regarding proof relation does require a specialized theoretical work to be implemented. Here we only present the intention and idea but leave the solution for future work.

Structure of the paper: Section 2 shortly explains the use case from our point of view and classify the requirements by the usage intention and/or nature. Section 3 briefly presents semantics of CCSL followed with the modelling of the system using only logical clocks. Section 4 introduces the real-time extensions to CCSL and show how they enrich the model. Section 5 discusses other extensions, but due to space limitations, only in a short form. Section 6 sums up the ideas we see promising in implementing the desired analyses. Section 7 concludes this exploration paper and discusses the ongoing and future work in supporting the full language with algorithmic solving capabilities.

2 Requirements classification

A mechanical lung ventilator is a complex interdependent system consisting of several cyber-physical components like mechanical parts, computer-human interfaces and a control. The description of mechanical parts includes oxygen and compressed air lines, their valves, pressure and flow sensors. The computer-human interface consists of a touch screen, buttons, a speaker and visual indicators. The embedded software has to coordinate the other parts according to the safety and functionality requirements.

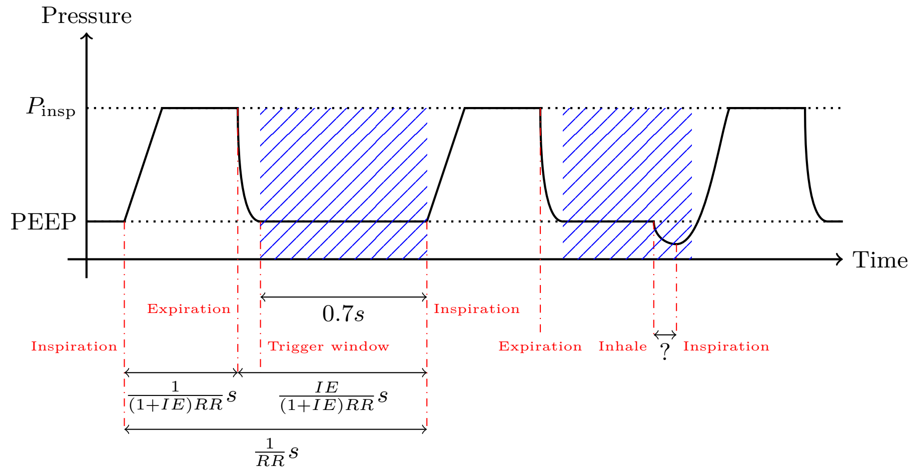

Fig. 1: PCV mode plot with events.

From the provided use case [7], we have built a formal model in RTCCSL (see [27]). We expect to check consistency on them, explore the possible solutions

to check what properties hold, and if the code of the system was provided, to check it against the requirements using so-called observers.

Our model refers directly to the identifiers provided (such as FUN.21). Overall we have encoded the following requirements:

- functional (FUN): 4, 19, 20, 21, 23, 24, 25, 27, 27.1, 27.2, 30, 31, 32;
- controller (CONT): 1, 2, 3, 4, 4.2, 5, 6, 8, 9, 10, 19, 21, 22, 25, 26, 30, 32, 36, 36.2, 36.3, 37, 40, 41.2, 56;
- parameter (PER): 4, 5, 11, 12, 13, 21.

We focused on requirements related to events and their time relations, sometimes with parameters. Timing parameters can vary in given intervals or be defined as expressions of other parameters. The subset of requirements is limited due to the functional and data-related expressiveness of CCSL. As such, we leave encoding and reasoning of the rest of the requirements to be complemented by other methods, like Event-B [1], Frama-C [15].

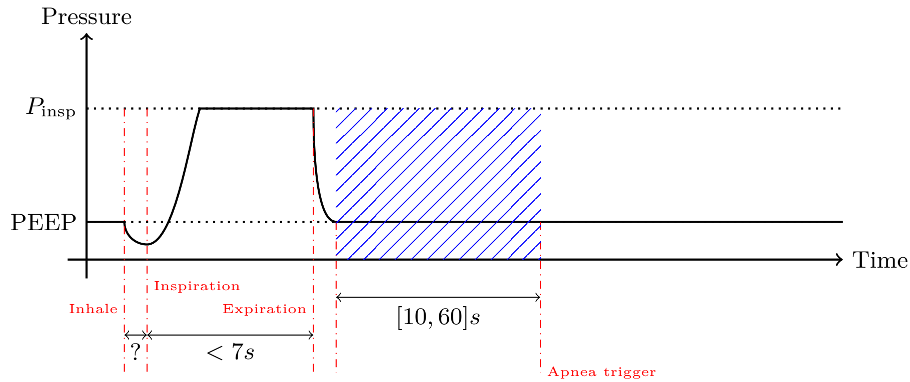

Fig. 2: PSV mode plot with events.

We start by describing the logical events and their relationships, like causality or other abstract time ordering. In Pressure Controlled Ventilation (PCV) mode (FUN.19 and Fig. 1), we have identified the following events: inspiration, expiration, trigger window deadline, detection of patient trying to inhale.

From the plot, we can establish some relationships among those events. For example, inspiration should alternate with expiration. Trigger window starts after expiration occurrence. The cycle should continue until stopped.

Pressure Support Ventilation (PSV) mode is similar (Fig. 2), inspiration and expiration events still alternate. The main difference is the reason to change the mode, it depends on the occurrence of apnea (FUN.27).

We also describe mechanical parts and some safety requirements. Valves can have 2 states: closed and open. When there is no ventilation, out valve should be open and in valve should be closed (CONT.38).

The second class of requirements is timing relations, mostly durations in the procedure of the ventilation. More precisely, some pairs of events have time relations between them: trigger window start and finish, inspiration and expiration, the whole ventilation cycle.

The third class is about ensuring that the specification, and so an implementation satisfying it, shall also satisfy some important properties by construction. Examples of this are the finiteness of memory needed to achieve the behaviour, absence of deadlocks or, to be more specific to the use case, the safety of the patient by ensuring that the exhalation valve is not closed for more than the required amount of time.

The fourth class is not represented widely, but there are some timing ranges that either can be selected by a user or to be computed from the behaviour of the ventilator. These parameters are then used in expressions, which in turn parametrize constraints. The uncertain nature of the parameters from the point of view of specification, requires checking the specification with all their combinations. Examples of such requirements would be recruitment maneuver duration (PER.3.2), PCV respiratory rate (PER.4), inspiration-expiration ratio (PER.5), PSV apnea lag (PER.11).

3 Logical modelling of the ventilator

This section shortly introduces the semantics of the Clock Constraint Specification Language and uses it to specify the use case. Refer to [20] for the full semantics and constraint definitions.

3.1 CCSL

The Clock Constraint Specification Language (CCSL) is a language operating with constraints as statements over logical clocks as variables. Each constraint binds ticks of its clocks to appear only in a certain order, effectively reducing the set of possible behaviours.

Tooling. Over the years, CCSL was implemented with an extensive collection of approaches and tools. These include translation into other languages and theories, and then simulation or verification using native tools: VHDL [3], Esterel, Signal and Time Petri Nets [17], Timed Automata [25]. Simulation and some model checking, specific to CCSL, is implemented in TimeSquare [12], checking finiteness of state using graphs [19], finding bounded periodic schedules with SMT [28]. The modelling using CCSL was demonstrated in the following use cases: brake-by-wire subsystem [13], spark ignition control [22], CPU interference [21].

Definition 1. *A specification Spec is defined as a tuple $\langle C, R \rangle$: logical clocks C and constraints R, both are a finite sets.*

Definition 2. *A logical clock c is a finite or infinite sequence of ticks $(c_n)_{n=1}^{\leq\infty}$, where $c_i \prec c_{i+1}$, i.e totally ordered.*

For example, a clock can be chronometric, like the movement of the second's arm in a wall clock, an electric circuit oscillating and outputting signal at a certain frequency, or sporadic, like user request or start of communication. The real-time difference between two successive ticks of the same clock is not defined and only the time causality between the ticks has to be preserved.

Definition 3. *A schedule is a function $\sigma : \mathbb{N} \to 2^C$. Given an execution step $n \in \mathbb{N}$ and a schedule $\sigma, \sigma(n)$ denotes a set of clocks that tick at step n.*

Fig. 3 illustrates the relation between logical clocks and a schedule.

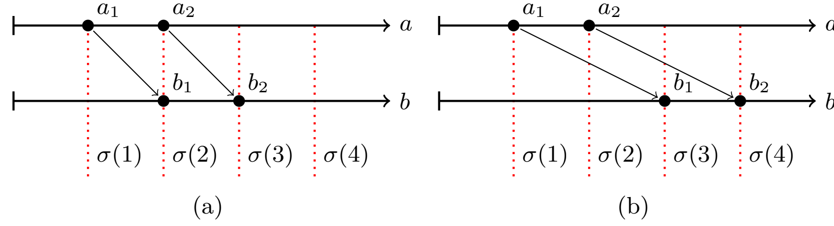

Fig. 3: Some valid schedules for $a \prec b$ constraint, arrows represent order.

Definition 4. *Given a schedule σ, a history over a set of clocks C is a function $H_\sigma : C \times \mathbb{N} \to \mathbb{N}$ defined inductively for all clocks $c \in C$:*

$$
\begin{aligned}
&H_\sigma(c, 0) = 0 \\
\forall n \in \mathbb{N} : c \notin \sigma(n) &\implies H_\sigma(c, n+1) = H_\sigma(c, n) \\
\forall n \in \mathbb{N} : c \in \sigma(n) &\implies H_\sigma(c, n+1) = H_\sigma(c, n) + 1
\end{aligned}
$$

Informally, for a clock $c \in C$, and step $n \in \mathbb{N}, H_\sigma(c, n)$ denotes the number of times clock c has ticked *before* step n within schedule σ.

CCSL constraints are divided into two groups: relations and expressions. Relations are intended to bound two clocks by some condition, while expressions are seen as a way to combine two clocks into a new clock. Some of the constraints are parametrized with integers. The notations and definitions of the common relations and expressions are defined in Table 1.

Definition 5. *Given a set Φ of constraints, the scheduling problem of CCSL is to compute whether there exists a schedule σ such that σ satisfies all constraints in Φ.*

In CCSL, "nothing happened" or $\sigma(i) = \emptyset$ is always a valid step in any schedule for any specification, so we only consider schedules without such steps, otherwise scheduling problem becomes trivial.

	Constraint	Notation	Definition, $\forall n \in \mathbb{N}$
Relation	Causality	$a \preccurlyeq b$	$H_\sigma(a,n) \geq H_\sigma(b,n)$
	Precedence	$a \prec b$	$(H_\sigma(a,n) = H_\sigma(b,n)) \Rightarrow b \notin \sigma(n+1)$
	Exclusion	$a \# b$	$a \notin \sigma(n) \vee b \notin \sigma(n)$
	Coincidence	$a = b$	$a \in \sigma(n) \Leftrightarrow b \in \sigma(n)$
	Subclocking	$a \subseteq b$	$a \in \sigma(n) \Rightarrow b \in \sigma(n)$
Expression	Delay	$c = a \mathbin{\$} d$	$H_\sigma(c,n) = \max(H_\sigma(a,n) - d, 0)$
	Supremum	$c = \sup(a,b)$	$H_\sigma(c,n) = \min(H_\sigma(a,n), H_\sigma(b,n))$
	Infimum	$c = \inf(a,b)$	$H_\sigma(c,n) = \max(H_\sigma(a,n), H_\sigma(b,n))$
	Intersection	$c = a * b$	$c \in \sigma(n) \Leftrightarrow (a \in \sigma(n) \wedge b \in \sigma(n))$
	Union	$c = a + b$	$c \in \sigma(n) \Leftrightarrow (a \in \sigma(n) \vee b \in \sigma(n))$
	Periodic	$c = a \propto p$	$c \in \sigma(n) \Leftrightarrow (H_\sigma(a,n) = p \cdot H_\sigma(c,n) \wedge a \in \sigma(n))$
	Sampling	$c = \texttt{sample}\ a\ \texttt{on}\ b$	$c \in \sigma(n) \Leftrightarrow \begin{pmatrix} b \in \sigma(n) \wedge \\ \exists 0 < j \leq n : a \in \sigma(j) \wedge \\ \forall j \leq k < n : b \notin \sigma(k) \end{pmatrix}$

Table 1: Definitions of common CCSL constraints [18,20]. Variables are $a, b, c \in C,\ d, p \in \mathbb{N}$, a schedule σ and its history H_σ.

3.2 Usage in the use case

First of all, to better express the intention, we make some changes to the way we use the basic language presented above:

- $c = \sup(a,b)$ is $c = \texttt{slowest}(a,b)$ (resp. inf is `fastest`);
- $c = a \mathbin{\$} 1$ is $c = \texttt{next}\ a$;
- $a \prec \texttt{next}\ b$ expands into $a \prec \alpha \wedge \alpha = \texttt{next}\ b$, where α is a unique anonymous clock name;
- the dot . in *prefix.c1* $\prec$ *prefix.c2* is part of the name of the clock, and is used only for convenience to name and structure clocks, the semantics remains identical;
- a `alternates` b rewrites into $a \prec b \prec \texttt{next}\ a$.

From the provided plot and the requirements (Fig. 2, FUN.19), it is obvious that some of the events are causally related: expiration cannot begin without inspiration, trigger window is activated only after expiration starts. The next cycle, which starts with inspiration, can only begin after the trigger deadline or with inhale detection (whichever is faster; FUN.21):

$$\begin{aligned} \textit{inspiration} \prec \textit{expiration} \prec \textit{window.start} \prec \textit{window.finish} \\ \texttt{fastest}(\textit{window.finish}, \textit{sensor.inhale}) \preccurlyeq \texttt{next}\ \textit{inspiration} \end{aligned}$$

Then we describe the relevant physical parts, including valves and sensors. Valves are devices which are supposed to open and close, and so have only 2 states, which is precisely what the alternation constraint represents. In the specification, we have decided to alternate close with open. It is so to force the valve to close as soon as possible, which clearly defines the initial state as closed.

Next we define a safety check, which is not present in the requirements, that the valves should not be open at the same time. For this, we define an equivalent of a mutex. This mutex mediates the access of valves to the shared resource of the patient mask.

$$\begin{gathered} \mathit{in.close}\ \texttt{alternates}\ \mathit{in.open} \\ \mathit{out.close}\ \texttt{alternates}\ \mathit{out.open} \\ (\mathit{in.open} + \mathit{out.open})\ \texttt{alternates}\ (\texttt{next}\ \mathit{in.close} + \texttt{next}\ \mathit{out.close}) \end{gathered}$$

For sensors, we model only the detection signalling and not the whole collection and processing of sensor data. For example, inhalation is detected when the pressure drops below the set value (FUN.21.1). The resulting clock is named *sensor.inhale*.

3.3 Gaps in the requirements

There are two places (Fig. 1 and Fig. 2; marked by "?") for which we could not find a timing requirement. This missing requirements prevent the specification from being fully time-deterministic while ventilating. This means that some phases are permitted to be executed indefinitely, which is against the purpose of the device and will prevent proving functional safety.

The missing requirement should specify what is the maximum latency between actual pressure dropping below target value and the controller making the decision that it *is* an inhalation attempt. How wide is the interval when the decision can be made will influence the solution cost as tighter requirements may imply costlier approaches. For example, the latency depends on the sensor itself, data acquisition architecture or simply the processor frequency.

3.4 Real-time as logical time

Additionally to having logical relations, some events are time-bounded and require a special treatment to be expressed in CCSL. These mostly cover PCV and PSV mode cycle durations and their parts. The basic trick to write real-time relations in logical time is to introduce a clock that is interpreted externally as the progress of physical time with a given period. Let us assume that the precision of one nanosecond (ns) is enough for the ventilator. Then to express the time difference of d seconds, we use the following template:

$$\mathit{right} = \mathit{left}\ \$\ n_d\ \texttt{on}\ \mathit{ns}$$

where $n_d = \frac{d\,\mathrm{s}}{1\,\mathrm{ns}}$, i.e. the number of nanoseconds in d. If clock *left* always coincides with *ns*, the delay is exact, otherwise it is approximate. Then, this template should be read as: clock *right* should tick after counting n_d number of nanoseconds.

As it is, there are several problems with this approach. First, one needs to decide on the precision beforehand, and in the case the precision should change, all real-time constraints have to be rewritten or regenerated.

Second, the state grows too fast in the case of using the classic automata approach. Ternary delay constraint $a = b \$ d$ `on` *base* is a combination of delay and sampling constraints and requires 2^d of states in the automata representation, in other words, $\lceil \frac{d}{8} \rceil$ bytes. If we would encode this way just 3 constraints and synchronize them into a single automaton, in order to do model checking, we would need $2^{\frac{7\,\mathrm{s}}{1\,\mathrm{ns}}} \times 2^{\frac{10\,\mathrm{s}}{1\,\mathrm{ns}}} \times 2^{\frac{60\,\mathrm{s}}{1\,\mathrm{ns}}} = 2^{77\times 10^9}$ states, i.e. 9.625 GB of memory.

To tackle it where possible, we need to detect specific patterns of behaviour (see examples in Section 6). To simplify this process we opted in to make the behaviour syntactically explicit by introducing new *real-time* constraints.

4 Real-time constraints

This section introduces new constraints, explains their meaning and gives semantics as a modified labelled transition system. Just as in most languages dealing with real-time, we assume the existence of a global notion of (physical) time and we use a *special* clock s (for second) to refer to it. This is a major evolution as otherwise all clocks in CCSL are assumed to be independent unless explicitly constrained. With this assumption, all clocks referring to physical time become indeed (implicitly) related. Part of our work to get efficient analysis is therefore to compute this implicit relation and use it whenever possible.

To capture the semantics of these real-time constraints, we introduce a new form of transition systems. We then describe how these transition systems are composed with *classical* CCSL.

4.1 Definition

Syntax This extension adds three new constraints to the language to capture frequently used timing patterns.

$o_1 =$ `repeat each` p `relative error` e `offset` φ defines a periodic clock o_1 with period p, cumulative error e and offset φ (a clock with jitter). In short, a periodic clock with cumulative error is best suitable for events like ticks of oscillator inside of a computer. Declaring 2 periodic clocks with exactly the same parameters *will* result in desynchronization;

$o_2 =$ `repeat each` p `absolute error` e `offset` φ defines a periodic clock o_2 with absolute or non-cumulative error (a clock with skew), other parameters have the same meaning as with cumulative error. A periodic clock with absolute error can be used to describe ideal processes that are observed thought other non-ideal processes or with unknown influences. For example, it could model the sampling of shaft turns with a sensor, which produces signal earlier or later depending on vibration or temperature. In other words, assuming constant speed of the shaft, frequency of turning *will not* drift indefinitely;

$b =$ `delay` a `by` d defines a clock b that is delayed between d_1 and d_2 seconds relative to the time when a clock ticks.

The parameters $p, \varphi \in \mathbb{R}_{\geq 0}$ are in seconds, error d and e are intervals $[d_1, d_2], [e_1, e_2]$, where $0 \leq d_1 \leq d_2, e_1 \leq e_2 \in \mathbb{R}$ and are in seconds too.

Automata Each constraint is defined as the corresponding real-time augmented CCSL automaton.

Definition 6. *Real-time augmented automaton A is a tuple $\langle P, p_0 \in P, p_e \in P, C, V, Q, T\rangle$, where:*

- *locations P, clocks C, variables V, queues Q are sets of symbols;*
- *S is a set of possible variable evaluations in a state, $S = (V \to \mathbb{R}_\perp) \times (Q \to \mathbb{N} \to [\mathbb{R}, \mathbb{R}] \cup \{\perp\}))$, where $\mathbb{R}_\perp = \mathbb{R} \cup \{\perp\}$ with $\perp$ as a symbol for nothing;*
- *$p_0 \in P$ is the initial location, $s_0 = (\lambda v.\ \perp, \lambda q.\lambda i.\ \perp)$ is the initial evaluation;*
- *p_e is a state that is reachable only if an invariant in some state is violated (for example, time passed certain value);*
- *$T \in P \times S \times \mathbb{R} \times 2^C \to P \times S$ is a transition function from location, its variable evaluation, clock label, current time into the next location and new evaluation.*

We put some syntactic restrictions on *transition functions*. The *guard* (valid arguments) is any boolean expression with atoms being a clock, an inclusion test or a test of queue emptiness.

If clock atom is true in the expression then the related clock has to tick, and vice-versa.

Inclusion tests are expressed as $a \in i$ or equivalently $i_1 \leq a \leq i_1$. In that case, atom a can be either current time η, a variable $v \in V$ or the first value from the queue $\text{head}(q), q \in Q$. i is an interval that can be expressed as a linear combination of atoms. If an expression evaluates to a scalar r we interpret it as an interval $[r, r]$. A queue emptiness test is $\text{head}(q) = \perp, q \in Q$.

The *assignment* is a set of variable assignments and queue updates. A variable assignment $v \coloneqq e$ contains an expression e, which is a linear combination of atoms described above. A queue update $\text{pop}(q)$ removes the head value and returns a new queue. $\text{push}(q, e)$ adds evaluation of e at the end of the queue and returns the new queue. These operations can be combined, i.e it is possible to remove and add a value to the same queue within a single transition.

We describe the new constraints as automata in Fig. 4, Fig. 5 and Fig. 6. The guard of the transitions is given above the arrow and the assignment is below.

Definition 7. *A* run *is an alternating sequence of rules (similarly to Timed Automata [8]):*

- *time elapse: $(p, s, \eta) \xrightarrow{\delta} (p, s, \eta + \delta)$;*
- *transition: $(p, s, \eta) \xrightarrow{l} (p', s', \eta)$ such that $\exists l \in 2^C : T(p, s, \eta, l) = (p', s')$.*

A run $(p_0, s_0, \eta = 0) \xrightarrow{\delta} (p_0, s_0, \eta' = \eta + \delta) \xrightarrow{l} (p', s', \eta) \xrightarrow{\delta'} (p', s', \eta' = \eta + \delta') \xrightarrow{l'} \dots$ is *valid* if p_e is never visited, and where $\eta, \eta' \in \mathbb{R}$ are the current times, $\delta, \delta' \in \mathbb{R}_{>0}$ are the time evolutions, $l, l' \in 2^C$ are labels or sets of clock ticks.

We then consider a *trace* to be a sequence of clock labels with preceding values of real-time η in a run. Thus we extend the original CCSL schedules (traces) from $\sigma : \mathbb{N} \to 2^C$ to $\mathbb{N} \to 2^C \times \mathbb{R}$.

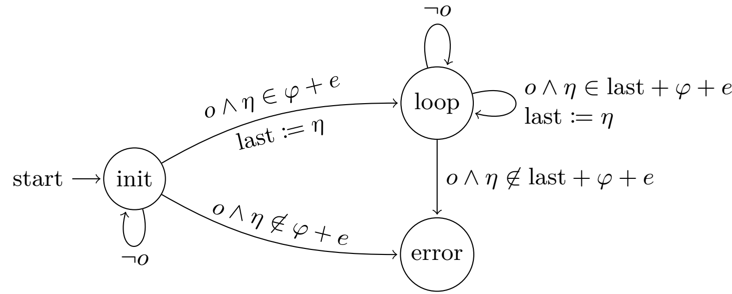

Fig. 4: Relative periodic $o =$ `repeat each` p `relative error` e `offset` φ.

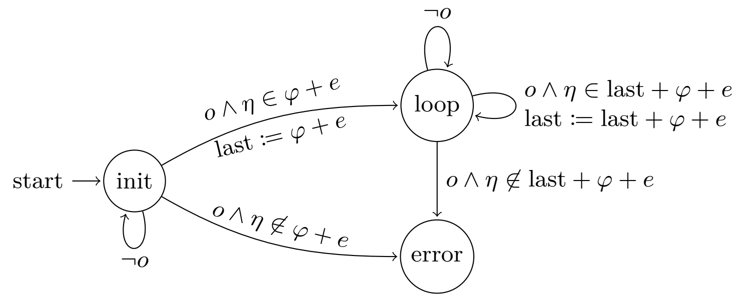

Fig. 5: Absolute periodic $o =$ `repeat each` p `absolute error` e `offset` φ.

Synchronization We use the classic synchronized product [4] for most of the states and transitions. The only exception is the error states as they should stay unique, thus they are fused across the network of automata and all transitions, meaning pointing to some error state translates to pointing to the new global one.

Definition 8. *Synchronization of transition functions T_1 and T_2 is the new transition function T defined as*

$$T = T_1 \circ T_2 = \lambda((p_1, p_2), (s_1, s_2), \eta, l). \\ E \begin{pmatrix} (T_1(p_1, s_1, \eta, \Pi_{C_1}(l)) \times P_2 \times S_2) \cap \\ (P_1 \times S_1 \times T_2(p_2, s_2, \eta, \Pi_{C_2}(l))) \end{pmatrix}$$

where $\Pi_{C_i} : 2^C \to 2^{C_i}$ is a projection to labels on clocks $C_i \subseteq C$, E is a mapping of compound and partial error locations to the new error location p_e.

Definition 9. *A synchronized real-time augmented automaton A of automata A_1 and A_2 is the tuple $\langle P \subseteq P_1 \times P_2 \cup \{p_e\}, p_0 = (p_{0_1}, p_{0_2}), p_e \in P, C = C_1 \cup C_2, V = V_1 \cup V_2, Q = Q_1 \cup Q_2, T = T_1 \circ T_2 \rangle$.*

To synchronize with the regular CCSL, the automata of CCSL constraints need to be translated into the new representation. The automata stays mostly

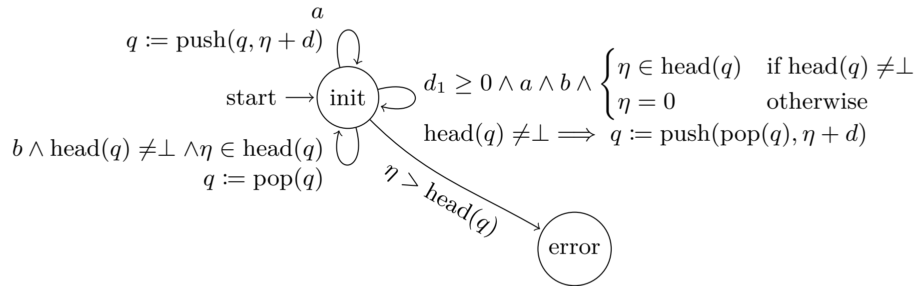

Fig. 6: Real-time delay $b = \texttt{delay}\ a\ \texttt{by}\ d$.

the same as described in [18], with the addition of a disconnected error state, empty sets of variables and queue symbols. The transitions are a subset of real-time transition functions and so can be synchronized as described above.

4.2 Rewriting real-time constraints in the ventilator

Going back to the timing constraints of the ventilator, we replace previously defined template (Subsec. 3.4) with the real-time delay constraint. Instead of specifying number of ticks on some base clock, we write durations in seconds directly. This way we let the solver figure out the necessary precision, if needed, or reason without discretization at all, when possible.

While there is no precision specified for any values in the requirements, to make the system more realistic we may add some imprecision to the delays. As multiplication is distributive under addition, adding the same precision to all constraints, at least in the PCV mode, will not result in any interesting constraint interaction. Thus we assume that the PCV cycle should be precise up to 1%, inspiration and expiration phases up to 5% and trigger window delay and duration up to 10%.

With such a specification, the controller has more freedom in implementing the exact timings. For example, it could use a precise dedicated timer to measure the cycle and an interrupt to switch execution to prepare for a new cycle, and a general coarse timer and cooperative preemption to execute intermediate tasks.

Additionally, it is possible to model the controller with more details if we know the frequency of the CPU base clock, for instance by assigning a frequency of some MHz plus-minus some error. One then could describe the control as conditions when to sample the inputs and to schedule tasks. As any computation should happen on the base clock, real-time constraints will induce causal relations between real-time related logical clocks. Then the task is to reverse search for conditions when the control would satisfy the higher-level constraints. We discuss this idea more in Subsec. 5.1.

5 Other extensions

This section briefly discusses other CCSL extensions intended to better convey user intentions. These include modular specifications and their proofs, parameters and constraints, and new logical constraints.

5.1 Modularity

The need for modularity arises from the demand to design and analyse complex specifications, including the one for the use case. We propose several interdependent extensions to make it easier: macros, name binding, specification structuring and proofs.

The central unit of the modular system is a macro. A macro is a list of constraints that has a name, for example `m1(){a < b}`. When the macro name is called in another macro, all constraints and clocks are added to it, but with a specified prefix, for example `m2(){prefix = m1()}` is equivalent to `m2(){prefix.a < prefix.b}`. In the use case, the macros are used to reuse specifications and to separate functionally different parts: environment interaction, modes and control.

As such, it is just a build system and does not make the analysis modular. For that we allow macros to contain not only the body of specification but sections of assumptions, assertions and interface, all of which can be specified as constraints. The interpretation is the following: under the given assumptions, does the specification violate the assertions and if not, does the interface include the specification behaviour? To make an analogy to functions, assumptions are argument types, an interface is a return type and a body is an algorithm. Making such a distinction allows separation of proofs between the body interface, and the specification properties. Proved properties on interfaces become "theorems" later on. This pattern allows scaling of the overall system analysis.

There are several properties that we are interested in, but which cannot be expressed as regular specifications and are only allowed for the top level macro. These include:

- check for deadlocks: deadlock-free specifications do not have a trace prefix that leads to traces that exclude some clocks indefinitely;
- safety: safe specifications have finite memory representation;
- existence of periodic or k-periodic schedules: a periodic schedule is a regular schedule where each clock ticks with a period defined on some global clock, high regularity of periodic schedules provides a compact and performant implementation of the system;
- condition for *some* liveness: being deadlock-free is a strong property, it may be easier to identify which subpart of the specification is live instead of proving that the specification is *exactly* live;
- and any combination of these properties.

For the ventilator, physical and control parts, as well as modes, are described as separate macros which are then connected together in the top level macro.

The top level macro should be checked to satisfy safety and liveness. We add the following general functional safety property: the out valve cannot be closed for too long, otherwise it may result in the death of the patient.

5.2 Parameters and constants

An orthogonal feature to modules is macro parameters and constants. In the ventilator, the modes have different allowed ranges of parameters and some of them are used in the constraints. In this case, we consider all of them to be constant as they are local to the modes and are not subject to the search for optimization. The constant parameters are declared as scalars or intervals inside the macros and can be algebraically manipulated with usual numerical operators, parametrize constraints, and specify or additionally constrain other parameters. The example of a non-constant parameter is the frequency of the CPU in the ventilator. The intention then is to find a value of this parameter that allows the specification to satisfy the desired properties. Such a scheme is known as parametric verification.

The parametric verification in CCSL is undecidable, but we believe that the specification and its parameters should be kept together, even if it is not possible to solve in general and a strategy to preselect values of parameters has to be provided.

5.3 Extension to logical constraints

The final extension introduces some convenient logical constraints to alleviate the reading of the specification.

The first extension is to allow intervals instead of single values to express delays. This makes the delay itself uncertain, but bounded. The proposed syntax is $b = a \ \$ \ [d_1, d_2]$, where $d_1 \leq d_2 \in \mathbb{N}$.

We introduce also other extensions. `first sampled` and `last sampled`, which produce a new clock when sampling occurs the first or last time in the sampling period. `allow` and `forbid`, respectively allows and forbids clocks to occur within a time interval defined by 2 other clocks.

6 Implementation

Any language is hardly useful without the appropriate tooling. This section discusses the already available implementation and several ways to achieve the required analysis for the extended CCSL.

6.1 Simulation

We have implemented a preliminary version of a simulator [26] for the extended language. Not all constraints and features are implemented at this moment, meaning the whole use case is not feasible to express and simulate. Thus we have implemented a fragment that describes the PCV mode. It is able to produce traces of this mode, an example being Fig. 7.

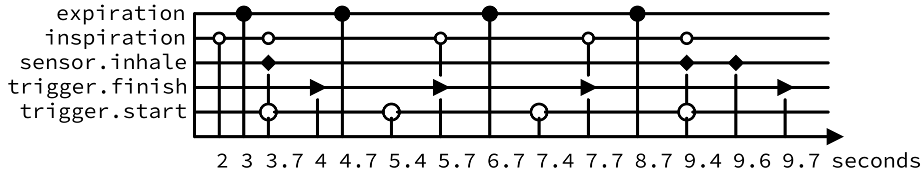

Fig. 7: Trace of specification for PCV mode.

6.2 Specialization

Specialization for Real-time CCSL means that depending on the shape of the specification, it may be beneficial to use or not real-time constraints, or a whole different representation. For example, in a specification with a real-time delay, if at no point it is checked when the ticks occur, the real-time information is irrelevant, and the constraint can be simplified to the causality relation. An example of such specification simplification is:

$$\begin{aligned} \mathit{base} &= \texttt{repeat each } 20\,\mathrm{MHz} \texttt{ relative error } \pm 1\% \texttt{ offset } 0 \\ \mathit{delayed} &= \texttt{delay } \mathit{external} \texttt{ by } 5\,\mathrm{ns} \\ \mathit{detected} &= \texttt{sample } \mathit{delayed} \texttt{ on } \mathit{base} \\ &\Downarrow \texttt{(erasure of real-time information)} \\ \mathit{external} &\prec \mathit{delayed} \\ \mathit{detected} &= \texttt{sample } \mathit{delayed} \texttt{ on } \mathit{base} \end{aligned}$$

Another variant of the specialization is algebraic reasoning. For certain type of constraints, it is possible to conclude results without building any automata, only by manipulating relations related to real-time values. An example would be checking that neither of the ventilation modes allows for the valves to be closed or open for too long. Just making sure that the cycle has real-time delays defined or implied should be enough for the check.

6.3 Abstract interpretation

CCSL is undecidable as it is Turing-complete, thus it is not possible to build a generic model checking. A way around it would be to use abstract interpretation that would trade precision for decidability and efficiency.

It is possible to express CCSL specifications as symbolic transition systems (sort of a program with constrained inputs) which then would be symbolically run by an abstract interpretation engine with a goal to prove that a bad condition is never reached. Generally, it includes partitioning the state of the transition system into locations, which later are complemented by an abstract domain, and specializations of the symbolic transition function from and to these locations. The specialization is made by propagating the reached state in the given location

though the transition function and remembering to which locations it leads. Then by iterating the transition application and sometimes refinement of the partitioning, the engine saturates the locations with the reachable state until a fixpoint is obtained, after which the property is checked.

The current development in the analysis seems to be promising as we managed to describe the core of CCSL with it. More precisely, we made the translation of CCSL into transition systems of NBac ([14], later used in [23]). Unfortunately, important optimizations and features are lacking due to the generic nature of the analysis. It is related both to the domains, optimizations in the application of the transition function and the properties we want to check. We intend to develop and implement the missing components as future work.

7 Conclusion

In this paper, we define a part of the specification that describes timing and causality relations between events of the mechanical lung ventilator, between its software and hardware parts. Missing requirements of the specification are either derived from pressure plots or common sense (different valves should not be opened at the same time) and influenced by the way of modelling in CCSL. We also include the patient safety check, formalized as a proof that under some conditions, the system specification satisfies the safety specification.

To describe the use case, we introduce several new constraints that express real-time relations directly and in turn allow easier reasoning and more consistent usage of real-time in specifications. We added other extensions, like modularity and separation of concern, parameters, as well as expressions and restrictions on the parameters. While this addition may not be theoretically significant, it allows users to better structure and scale the systems. It also includes the possibility to reuse generic specifications in different subsystems or even projects. We conclude the contribution with the addition of logical constraints. Some are just syntactic sugar, some are not. All are of equal importance if the goal is to avoid accidental complexity or cumbersome constructs.

The future work will consist of implementing the proposed extensions and further explore solutions sketched in this paper. We expect that execution of proofs in a proof tree can be produced by dedicated tools. Some work of using abstract interpretation for relevant subsets has been done already, but it needs to be implemented for the new constructs and integrated with relevant optimization rules to produce the main solving engine of CCSL.

References

1. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer **12**(6), 447–466 (Nov 2010). https://doi.org/10.1007/s10009-010-0145-y

2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (Apr 1994). https://doi.org/10/bn332s, https://www.sciencedirect.com/science/article/pii/0304397594900108
3. André, C., Mallet, F., Deantoni, J.: VHDL Observers for Clock Constraint Checking. IEEE computer society (Jul 2010). https://doi.org/10/bf3jng, https://hal.inria.fr/inria-00587107
4. Arnold, A.: Finite transition systems - semantics of communicating systems. International Series in Computer Science, Prentice Hall (1994)
5. Arnold, A., Point, G., Griffault, A., Rauzy, A.: The AltaRica Formalism for Describing Concurrent Systems. Fundamenta Informaticae **40**(2-3), 109–124 (Jan 1999). https://doi.org/10/gpb5x8, https://content.iospress.com/articles/fundamenta-informaticae/fi40-2-3-02, publisher: IOS Press
6. Berry, G.: The Esterel v5 Language Primer (Dec 2002)
7. Bonfanti, S., Gargantini, A.: The Mechanical Lung Ventilator Case Study. In: Rigorous State-Based Methods 10th International Conference, ABZ 2024, Bergamo, Italy, June 2528, 2024, Proceedings, Lecture Notes in Computer Science, vol. 14759. Springer (2024)
8. Bouyer, P., Gastin, P., Herbreteau, F., Sankur, O., Srivathsan, B.: Zone-based verification of timed automata: extrapolations, simulations and what next? (Jul 2022). https://doi.org/10.48550/arXiv.2207.07479, http://arxiv.org/abs/2207.07479, arXiv:2207.07479 [cs] version: 1
9. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: LUSTRE: A declarative language for programming synchronous systems* (1987), https://www.semanticscholar.org/paper/LUSTRE%3A-A-declarative-language-for-programming-Caspi-Pilaud/893b9e21f01df1f14a922d2e4eb863be9ecb25d2
10. Colaço, J., Pagano, B., Pouzet, M.: SCADE 6: A formal language for embedded critical software development (invited paper). In: 11th International Symposium on Theoretical Aspects of Software Engineering, TASE. pp. 1–11. IEEE Computer Society (2017). https://doi.org/10.1109/TASE.2017.8285623
11. Cousot, P., Cousot, R.: Abstract interpretation: past, present and future. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–10. ACM, Vienna Austria (Jul 2014). https://doi.org/10.1145/2603088.2603165, https://dl.acm.org/doi/10.1145/2603088.2603165
12. DeAntoni, J., Mallet, F.: TimeSquare: Treat Your Models with Logical Time. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Furia, C.A., Nanz, S. (eds.) Objects, Models, Components, Patterns, vol. 7304, pp. 34–41. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30561-0_4, http://link.springer.com/10.1007/978-3-642-30561-0_4, series Title: Lecture Notes in Computer Science
13. Goknil, A., DeAntoni, J., Peraldi-Frati, M.A., Mallet, F.: Tool Support for the Analysis of TADL2 Timing Constraints Using TimeSquare. In: 2013 18th International Conference on Engineering of Complex Computer Systems. pp. 145–154. IEEE, Singapore, Singapore (Jul 2013). https://doi.org/10.1109/ICECCS.2013.28, http://ieeexplore.ieee.org/document/6601815/
14. Jeannet, B.: Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Reactive Systems. Formal Methods in System Design **23**(1), 5–37

(Jul 2003). https://doi.org/10.1023/A:1024480913162, https://doi.org/10.1023/A:1024480913162
15. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (May 2015). https://doi.org/10.1007/s00165-014-0326-7
16. Mallet, F.: Clock constraint specification language: specifying clock constraints with UML/MARTE. Innovations in Systems and Software Engineering **4**, 309–314 (Oct 2008). https://doi.org/10/dn4ptd
17. Mallet, F., André, C.: UML/MARTE CCSL, Signal and Petri nets. report, INRIA (2008), https://hal.inria.fr/inria-00283077
18. Mallet, F., Millo, J.V., Romenska, Y.: State-based representation of CCSL operators. Tech. rep., INRIA
19. Mallet, F., Millo, J.V., Simone, R.d.: Safe CCSL Specifications and Marked Graphs. p. 157. IEEE CS (Oct 2013), https://hal.inria.fr/hal-00913962
20. Mallet, F., de Simone, R.: Correctness issues on MARTE/CCSL constraints. Science of Computer Programming **106**, 78–92. https://doi.org/10/f7qbxg
21. Oueslati, A., Cuenot, P., Deantoni, J., Moreno, C.: System Based Interference Analysis in Capella. The Journal of Object Technology **18**(2), 14:1 (2019). https://doi.org/10.5381/jot.2019.18.2.a14, https://hal.inria.fr/hal-02182902
22. Peraldi-Frati, M.A., DeAntoni, J.: Scheduling Multi Clock Real Time Systems: From Requirements to Implementation. In: 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. pp. 50–57 (Mar 2011). https://doi.org/10.1109/ISORC.2011.16, iSSN: 2375-5261
23. Schrammel, P., Jeannet, B.: Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs. In: Yahav, E. (ed.) Static Analysis. pp. 233–248. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_19
24. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall international series in computer science, Prentice Hall, New York, 2nd ed edn. (1992)
25. Suryadevara, J., Seceleanu, C., Mallet, F., Pettersson, P.: Verifying MARTE/CCSL Mode Behaviors Using UPPAAL. vol. 8137, p. 1. Springer (Sep 2013). https://doi.org/10/f3rnq6, https://hal.inria.fr/hal-00866477
26. Tokariev, P.: Implementation of MRTCCSL. https://github.com/PaulRaUnite/mrtccsl
27. Tokariev, P.: Mechanical lung ventilator specification. https://github.com/PaulRaUnite/mlv_spec
28. Zhang, M., Song, F., Mallet, F., Xiaohong, C.: SMT-Based Bounded Schedulability Analysis of the Clock Constraint Specification Language (Apr 2019), https://hal.inria.fr/hal-02080763