Real-Time CCSL: Application to the Mechanical Lung Ventilator

Pavlo Tokariev Frédéric Mallet

Team KAIROS
Université Côte d'Azur, Inria, CNRS, i3S
Sophia Antipolis, France

Wed, 26 June 2024, ABZ2024

Content

General background

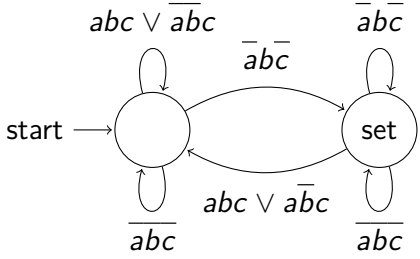
- We target safety critical **reactive systems** with concurrent interacting agents/parts
- To prove safety one can use testing, formal methods, etc.
- We choose to focus on formal descriptions of **temporal** relationships
- And Clock Constraint Specification Language (CCSL) is our current method

Clock Constraint Specification Language [11]

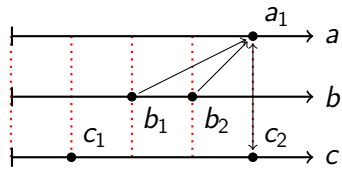
- **Logical clocks** are (infinite) sequences of event occurrences (ticks)
- **Constraints** use clocks as variables and define which sequences are allowed
- A **specification** expresses sequences that satisfy all constraints
- A **schedule** is an assignment of clock ticks to steps
- Problems of interest:
  - Existence of schedules
  - Finiteness of representation
  - Clock liveness

Examples of constraints (1/2)

Sampling constraint $\mathbf{a} = \mathbf{b}$ `sampled on` $\mathbf{c}$.



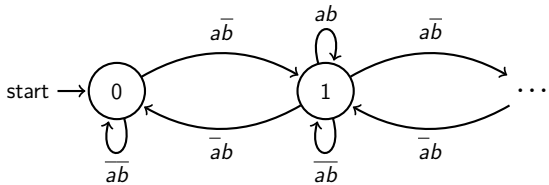
(a) Finite automaton



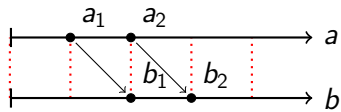
(b) Possible schedule

Examples of constraints (2/2)

Precedence constraint $\mathbf{a} \prec \mathbf{b}$.



(a) Infinite automaton (unbounded integer counter)



(b) Possible schedule

Comparison with other methods

- Reactive synchronous languages (Lustre [6], Esterel [3], Prelude [9]):
  - Synchronous assumption
  - Inspiration for CCSL and multiform logical time
- Timed Automata [2]:
  - Event synchronization
  - Uniform time
- State-based formal methods, like Event-B [1], ASM [5], Alloy [10]

Why extend to real-time?

- CCSL does support chronological clocks
- But not really real-time relations
- We can define them by discretizing, but it is:
  - Imprecise
  - Blows up the state space
- Thus, we add syntactic and semantic extension to the language

New constraints

- Real-time delay:

$$\textbf{out} = \texttt{delay}\ \textbf{arg}\ \text{by}\ [1s, 2s]$$

- Cumulative periodic:

$$\textbf{out} = \texttt{repeat each}\ 5s\ \textit{relative}\ \texttt{error}\ \pm 1\%\ \texttt{offset}\ 10s$$

- Absolute periodic:

$$\textbf{out} = \texttt{repeat each}\ 5s\ \textit{absolute}\ \texttt{error}\ \pm 1\%\ \texttt{offset}\ 10s$$

Illustration on PCV mode [4]:

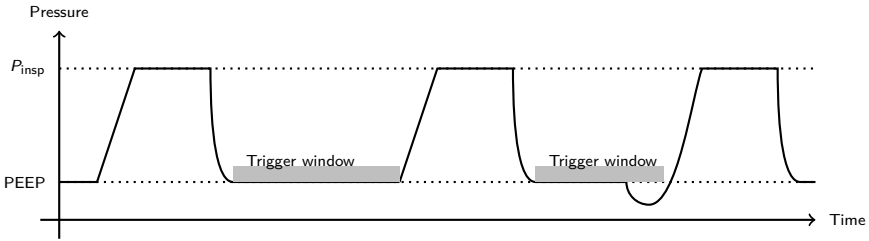


Illustration on PCV mode [4]: mapping to clocks

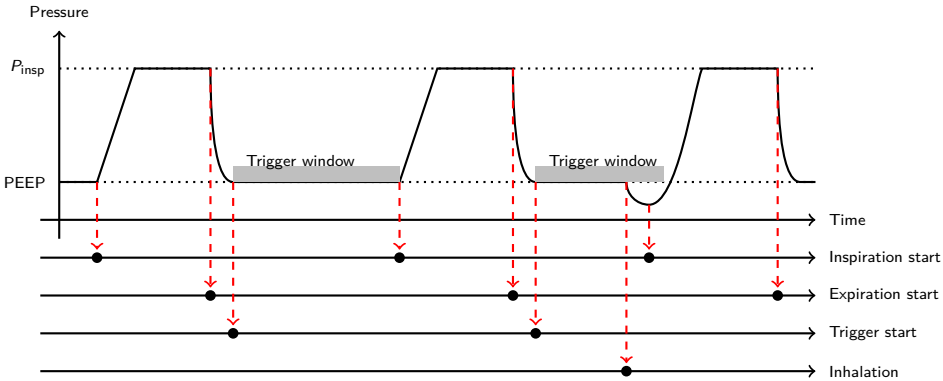


Illustration on PCV mode [4]: pure CCSL

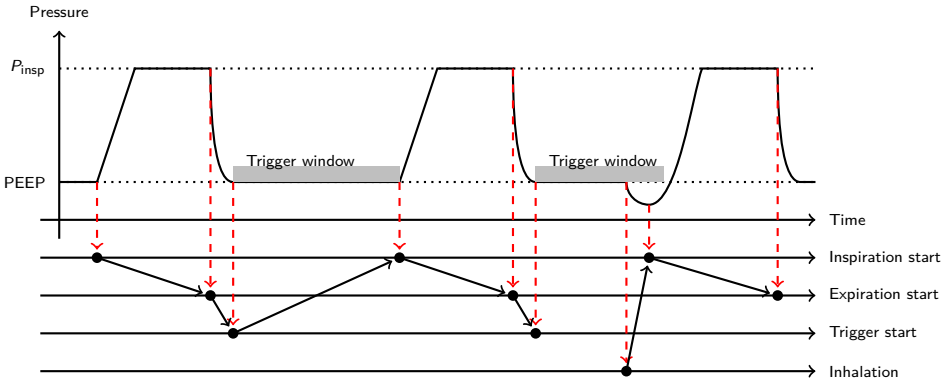


Illustration on PCV mode [4]: pure CCSL

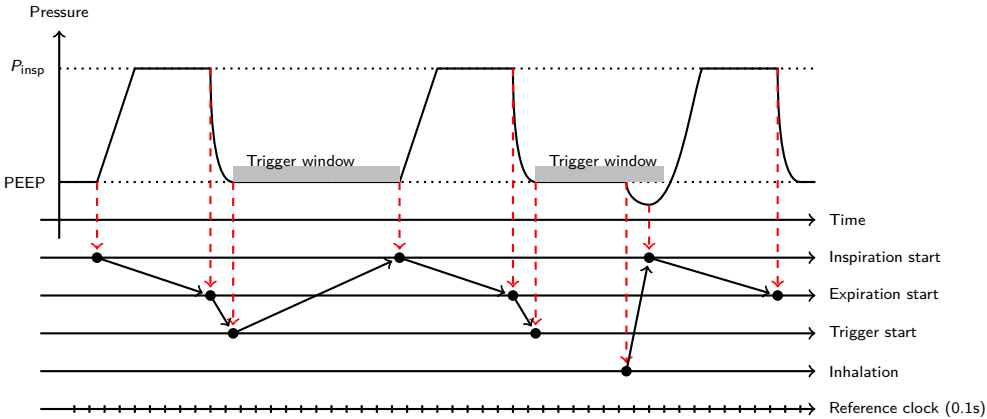


Illustration on PCV mode [4]: pure CCSL

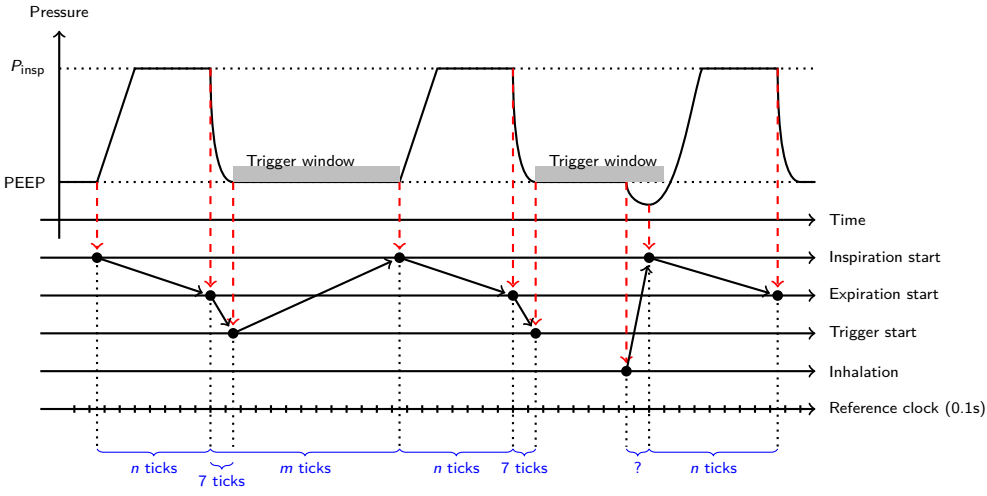


Illustration on PCV mode [4]: pure CCSL

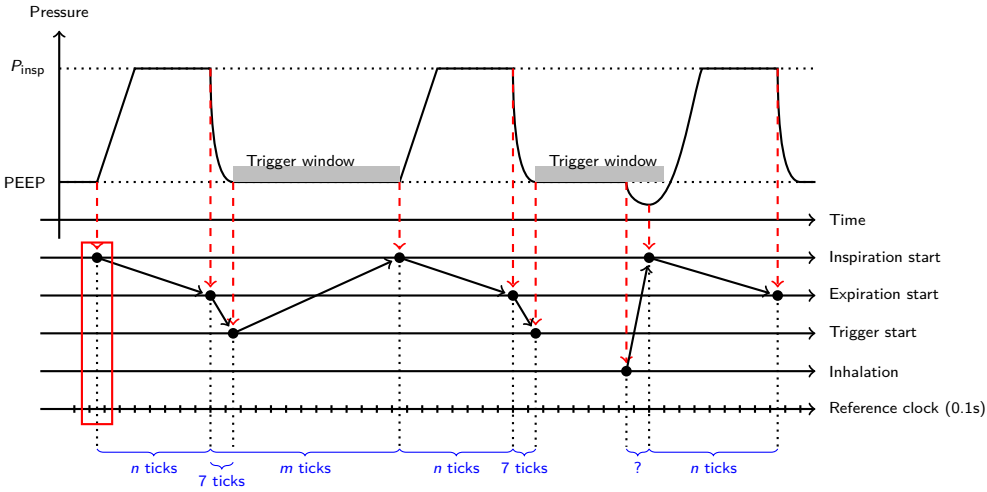
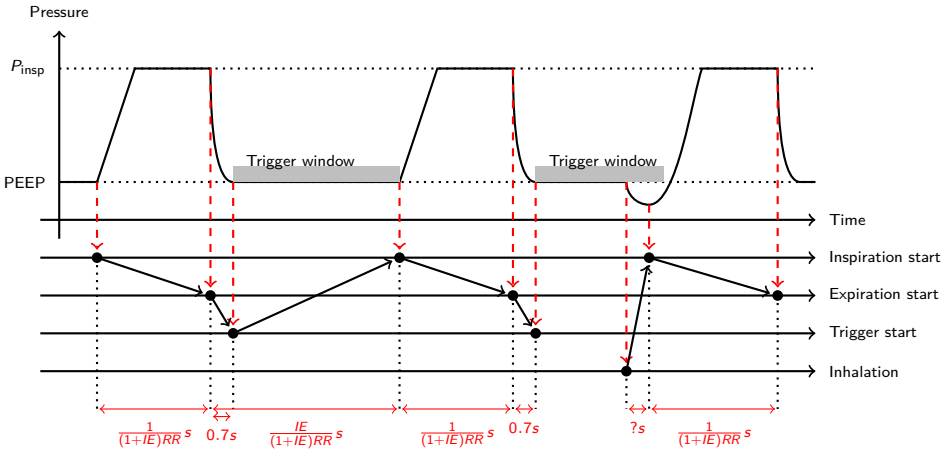


Illustration on PCV mode [4]: with RTCCSL



PCV code

```
pcv_mode(mode: struct, sensor: struct) where { //FUN.19
    IE in [1, 4]; //PER.5, includes PER.13
    RR in [4,50]/1 min; //PER.4, includes PER.12
    trigger_window_delay = 0.7s; //CONT.45
    trigger_window.start ≼ fastest(sensor.inhale, trigger_window.finish)
        ≼ next inspiration.start; //FUN.21
    between(trigger_window.start, trigger_window.finish, sensor.inhale)
} {
    expiration = inspiration delayed by 1/RR/(1+IE); //FUN.20
    trigger_window = {
        start ≺ finish;
        start = .expiration delayed by trigger_window_delay; //CONT.45
        finish = .inspiration delayed by 1/RR; //FUN.20
    };
    inspiration_condition = sensor.inhale || trigger_window.finish
        \ ((sensor.inhale || mode.pcv.finish) sampled on trigger_window.finish)
        \ (mode.pcv.finish sampled on sensor.inhale); //CONT.25
    next inspiration = first sampled inspiration_condition on trigger_window.finish;
} assert {
    trigger_window.finish ≼ expiration delayed by IE/RR/(1+IE); //FUN.20
    inspiration alternates expiration;
}
```

PCV code: assumption and assertion

```
pcv_mode(mode: struct, sensor: struct) where { //FUN.19
    IE in [1, 4]; //PER.5, includes PER.13
    RR in [4,50]/1 min; //PER.4, includes PER.12
    trigger_window_delay = 0.7s; //CONT.45
    trigger_window.start ≼ fastest(sensor.inhale, trigger_window.finish)
        ≼ next inspiration.start; //FUN.21
    between(trigger_window.start, trigger_window.finish, sensor.inhale)
} {
    expiration = inspiration delayed by 1/RR/(1+IE); //FUN.20
    trigger_window = {
        start ≺ finish;
        start = .expiration delayed by trigger_window_delay; //CONT.45
        finish = .inspiration delayed by 1/RR; //FUN.20
    };
    inspiration_condition = sensor.inhale || trigger_window.finish
        \ ((sensor.inhale || mode.pcv.finish) sampled on trigger_window.finish)
        \ (mode.pcv.finish sampled on sensor.inhale); //CONT.25
    next inspiration = first sampled inspiration_condition on trigger_window.finish;
} assert {
    trigger_window.finish ≼ expiration delayed by IE/RR/(1+IE); //FUN.20
    inspiration alternates expiration;
}
```

PCV code: parameters

```
pcv_mode(mode: struct, sensor: struct) where { //FUN.19
    IE in [1, 4]; //PER.5, includes PER.13
    RR in [4,50]/1 min; //PER.4, includes PER.12
    trigger_window_delay = 0.7s; //CONT.45
    trigger_window.start ≼ fastest(sensor.inhale, trigger_window.finish)
        ≼ next inspiration.start; //FUN.21
    between(trigger_window.start, trigger_window.finish, sensor.inhale)
} {
    expiration = inspiration delayed by 1/RR/(1+IE); //FUN.20
    trigger_window = {
        start ≺ finish;
        start = .expiration delayed by trigger_window_delay; //CONT.45
        finish = .inspiration delayed by 1/RR; //FUN.20
    };
    inspiration_condition = sensor.inhale || trigger_window.finish
        \ ((sensor.inhale || mode.pcv.finish) sampled on trigger_window.finish)
        \ (mode.pcv.finish sampled on sensor.inhale); //CONT.25
    next inspiration = first sampled inspiration_condition on trigger_window.finish;
} assert {
    trigger_window.finish ≼ expiration delayed by IE/RR/(1+IE); //FUN.20
    inspiration alternates expiration;
}
```

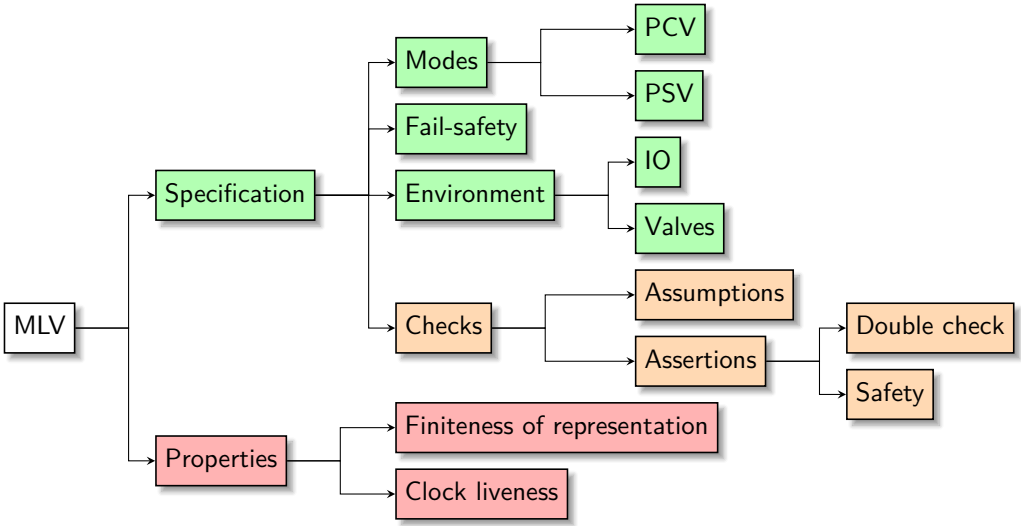
PCV code: purely logical

```
pcv_mode(mode: struct, sensor: struct) where { //FUN.19
    IE in [1, 4]; //PER.5, includes PER.13
    RR in [4,50]/1 min; //PER.4, includes PER.12
    trigger_window_delay = 0.7s; //CONT.45
    trigger_window.start ≼ fastest(sensor.inhale, trigger_window.finish)
        ≼ next inspiration.start; //FUN.21
    between(trigger_window.start, trigger_window.finish, sensor.inhale)
} {
    expiration = inspiration delayed by 1/RR/(1+IE); //FUN.20
    trigger_window = {
        start ≺ finish;
        start = .expiration delayed by trigger_window_delay; //CONT.45
        finish = .inspiration delayed by 1/RR; //FUN.20
    };
    inspiration_condition = sensor.inhale || trigger_window.finish
        \ ((sensor.inhale || mode.pcv.finish) sampled on trigger_window.finish)
        \ (mode.pcv.finish sampled on sensor.inhale); //CONT.25
    next inspiration = first sampled inspiration_condition on trigger_window.finish;
} assert {
    trigger_window.finish ≼ expiration delayed by IE/RR/(1+IE); //FUN.20
    inspiration alternates expiration;
}
```

PCV code: real-time constraints

```
pcv_mode(mode: struct, sensor: struct) where { //FUN.19
    IE in [1, 4]; //PER.5, includes PER.13
    RR in [4,50]/1 min; //PER.4, includes PER.12
    trigger_window_delay = 0.7s; //CONT.45
    trigger_window.start ≼ fastest(sensor.inhale, trigger_window.finish)
        ≼ next inspiration.start; //FUN.21
    between(trigger_window.start, trigger_window.finish, sensor.inhale)
} {
    expiration = inspiration delayed by 1/RR/(1+IE); //FUN.20
    trigger_window = {
        start ≺ finish;
        start = .expiration delayed by trigger_window_delay; //CONT.45
        finish = .inspiration delayed by 1/RR; //FUN.20
    };
    inspiration_condition = sensor.inhale || trigger_window.finish
        \ ((sensor.inhale || mode.pcv.finish) sampled on trigger_window.finish)
        \ (mode.pcv.finish sampled on sensor.inhale); //CONT.25
    next inspiration = first sampled inspiration_condition on trigger_window.finish;
} assert {
    trigger_window.finish ≼ expiration delayed by IE/RR/(1+IE); //FUN.20
    inspiration alternates expiration;
}
```

MLV specification summary



Lessons from the modelling

- Going from natural language into formal specification helps remove ambiguities
- Examples of ambiguities:
  - Reaction latencies for:
    - Inhalation
    - Valves
    - Fail-safe
  - Precision
- Parametric verification is really desired

Existing CCSL tooling

- TimeSquare [8]:
  - Exhaustive state model checking
  - Simulation
  - Observer code generation
- MyCCSL [7]:
  - Existence of schedules
  - Clock liveness
  - LTL model checking
  - Uses SMT

RTCCSL tooling

- Simulation:
  - Can produce or check a schedule/trace for a specification
  - Can use different strategies to choose the steps in schedules

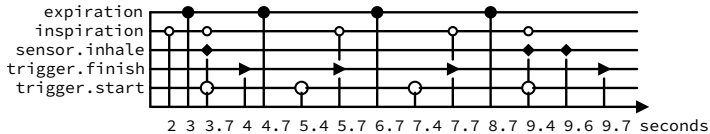


Figure 3: Generated schedule for PCV mode

- Symbolic:
  - Inductive reasoning about existence of certain type of infinite schedules, checking assertions and assumptions, with some parametric verification
  - (WIP) state-based abstract interpretation to check finiteness of representation

Conclusion

- Language extensions with new syntax and semantics
- Described MLV use case in this language
- Both another iteration on language
- And new perspective on the use case
- Implemented simulation and the first symbolic tool
- Working on a tool that uses abstract interpretation

Questions?

References I

[1] Jean-Raymond Abrial. *Modeling in Event-B : system and software engineering*. eng. Cambridge ; New York : Cambridge University Press, 2010. ISBN: 978-0-521-89556-9. URL: http://archive.org/details/modelingineventb0000abri (visited on 05/30/2024).

[2] Rajeev Alur and David L. Dill. "A theory of timed automata". en. In: *Theoretical Computer Science* 126.2 (Apr. 1994), pp. 183–235. ISSN: 0304-3975. DOI: 10/bn332s. URL: https://www.sciencedirect.com/science/article/pii/0304397594900108 (visited on 12/07/2021).

References II

[3] Gerard Berry and Jean-Paul Rigault. “Esterel: Towards a synchronous and semantically sound high-level language for real-time applications”. In: 1983.

[4] Silvia Bonfanti and Angelo Gargantini. “The Mechanical Lung Ventilator Case Study”. In: *Rigorous State-Based Methods 10th International Conference, ABZ 2024, Bergamo, Italy, June 2528, 2024, Proceedings*. Vol. 14759. Lecture Notes in Computer Science. Springer, 2024.

References III

[5] Egon Börger. "The ASM Refinement Method". en. In: *Formal Aspects of Computing* 15.2 (Nov. 2003), pp. 237–257. ISSN: 1433-299X. DOI: 10.1007/s00165-003-0012-7. URL: https://doi.org/10.1007/s00165-003-0012-7 (visited on 05/30/2024).

[6] P. Caspi et al. "LUSTRE: A declarative language for programming synchronous systems*". In: 1987. URL: https://www.semanticscholar.org/paper/LUSTRE%3A-A-declarative-language-for-programming-Caspi-Pilaud/893b9e21f01df1f14a922d2e4eb863be9ecb25d2 (visited on 12/13/2022).

References IV

[7] Xiaohong Chen, Frédéric Mallet, and Xiaoshan Liu. "Formally Verifying Sequence Diagrams for Safety Critical Systems". en. In: Dec. 2020. URL: https://hal.inria.fr/hal-03121933 (visited on 12/07/2021).

[8] Julien DeAntoni and Frédéric Mallet. "TimeSquare: Treat Your Models with Logical Time". en. In: *Objects, Models, Components, Patterns*. Ed. by David Hutchison et al. Vol. 7304. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 34–41. ISBN: 978-3-642-30560-3 978-3-642-30561-0. DOI: 10.1007/978-3-642-30561-0_4. URL: http://link.springer.com/10.1007/978-3-642-30561-0_4 (visited on 02/02/2022).

References V

[9] Julien Forget et al. "A Multi-Periodic Synchronous Data-Flow Language". In: *11th IEEE High Assurance Systems Engineering Symposium*. Nanjing, China, Dec. 2008, pp. 251–260. URL: https://hal.archives-ouvertes.fr/hal-00802695 (visited on 11/03/2022).

[10] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Jan. 2012. ISBN: 978-0-262-01715-2.

[11] Frédéric Mallet. "Clock constraint specification language: specifying clock constraints with UML/MARTE". In: *Innovations in Systems and Software Engineering* 4 (Oct. 2008), pp. 309–314. DOI: 10/dn4ptd.

Bonus slides

Refinement

- The MLV specification can be made of 2 parts: high-level and low-level requirements
- High-level would contain the requirements $\pm$ precision
- Low-level will refine the behaviour using sampling and logical delays on real-time cumulative clock. This corresponds closer to how the actual system will work
- The high-level specification should include low-level one

General framework

$$A \sim (A \wedge S_{\mathtt{LL}}) \sim S_{\mathtt{HL}} \sim P_{\mathtt{safety}}$$

where

- A is for assumptions constraints
- $S_{\mathtt{LL}}$ for low-level specification
- $S_{\mathtt{HL}}$ for high-level
- $P_{\mathtt{safety}}$ is for general patient safety property
- $\sim$ is a simulation relation, meaning that solutions to the left are all present in the right specification