Modular real-time clock constraint specification language

Pavlo Tokariev

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Langage Modulaire pour la Spécification de Contraintes d'Horloges Logiques et Temps-Réel

Pavlo TOKARIEV

Centre Inria d'Université Côte d'Azur / Laboratoire i3S / Équipe KAIROS

Présentée en vue de l'obtention du grade de docteur en INFORMATIQUE **d'**Université Côte d'Azur

Dirigée par : Frédéric MALLET, Professeur des Universités, Université Côte d'Azur
Soutenue le : 13 décembre 2024

Devant le jury, composé de :
Aurélie HURAULT, Professeure des Universités, ENSEEIHT, Université de Toulouse
Natalia KUSHIK, Maîtresse des Conférences, HDR, Telecom SudParis
Frédéric BOULANGER, Professeur, Centrale Supélec
Abdoulaye GAMATIÉ, Directeur de Recherche, LIRMM, CNRS

LANGAGE MODULAIRE POUR LA SPÉCIFICATION DE CONTRAINTES D'HORLOGES LOGIQUES ET TEMPS-RÉEL

Modular Real-Time Clock Constraint Specification Language

Pavlo TOKARIEV

Jury :

Rapporteurs

Aurélie HURAULT, Professeure des Universités, ENSEEIHT, Université de Toulouse
Natalia KUSHIK, Maîtresse des Conférences, HDR, Telecom SudParis

Examinateurs

Frédéric BOULANGER, Professeur, Centrale Supélec
Abdoulaye GAMATIÉ, Directeur de Recherche, LIRMM, CNRS

Directeur de thèse

Frédéric MALLET, Professeur des Universités, Université Côte d'Azur

Université Côte d'Azur

Pavlo TOKARIEV

Langage Modulaire pour la Spécification de Contraintes d'Horloges Logiques et Temps-Réel

xiii+192 p.

À l'Ukraine.

Langage Modulaire pour la Spécification de Contraintes d'Horloges Logiques et Temps-Réel

Résumé

Les systèmes en temps réel critiques (réactifs) sont des systèmes qui contrôlent des processus complexes et dont la faute n'est pas acceptable en raison des graves conséquences pour le système, l'infrastructure et les humains. Dans ces systèmes, le moment de la réaction est aussi critique que l'exécution de la bonne action. Dans ce travail, nous nous concentrons sur le premier. Pour ce faire, nous utilisons une abstraction du temps, connue sous le nom de temps logique. Il abstrait totalement les instants auxquels les événements se produisent par leur position relative. Le langage sur lequel nous basons notre travail, Clock Constraint Specification Language (CCSL), est purement basé sur la notion d'horloge logique et conçu pour décrire les exigences temporelles des systèmes. L'application du langage nous a permis de constater que l'approche purement logique n'est pas toujours adéquate. Le langage de spécification doit permettre de décrire des relations temps réel. Leur simulation purement avec des horloges logiques échoue en général en raison de la différence de complexité. Ceci nous incite à trouver des moyens d'abstraire ou de résoudre en utilisant des méthodes moins exactes.Ainsi, dans ce travail, nous proposons une série d'extensions du langage original, orthogonales mais se complétant. Celles-ci couvrent les contraintes en temps réel et les contraintes auxiliaires pour augmenter l'expressivité, la paramétrisation des contraintes et le cadre modulaire avec une fonction similaire au raffinement.Nous les définissons formellement et motivons leur conception à l'aide de plusieurs cas d'utilisation. Nous rapportons nos expériences avec l'interprétation abstraite dans l'analyse des spécifications et proposons plusieurs modifications pour la rendre plus précise. Enfin, nous introduisons notre propre solveur ad hoc utilisant une représentation polyédrique sur un fragment du langage.

Mots-clés : Systèmes Temps-Réel, Exigences Temporelles, Temps Logique, Temps Réel, Vérification Formelle, Interprétation Abstraite.

Modular Real-Time Clock Constraint Specification Language

Abstract

Safety-critical real-time (reactive) systems are systems in control of complex processes and which failure is not acceptable due to severe consequences for the system, infrastructure and people. In such systems, the timing of the reaction is as critical as doing the right action. In this work, we focus on the former. For this we use an abstraction of time, known as logical time. It completely abstracts away the instants at which events occur by their relation to each other. The language we base our work on, the Clock Constraint Specification Language (CCSL), is purely based on logical clocks and is designed to describe temporal requirements of systems. From the application of the language, we notice that pure logical approach is not always adequate or efficient, as specification languages for such systems do need to express real-time relations. And attempts to simulate using pure logical clocks fail in general for large systems due to the combinatorial complexity. Which in turn prompts us to find ways to abstract or solve the specifications using approximate methods and renounce exact solutions. Thus, in this work, we propose a series of extensions to the original language, orthogonal but complementing each other. These cover real-time and auxiliary constraints to increase expressiveness, parametrization of constraints and modular framework with a mechanism akin to refinement. We define them formally and motivate their design by using several use cases. We report our experiments with abstract interpretation in specification analyses, propose several modifications to make it more precise and demonstrate them on the mentioned use cases. Finally, we introduce our own polyhedra-based ad-hoc solver for a fragment of the language.

Keywords: Real-Time Systems, Temporal Requirements, Logical Time, Real-Time, Formal Verification, Abstract Interpretation.

Acknowledgements

First of all I thank my supervisor, Frédéric Mallet, professor at Université Côte d'Azur. I am grateful to him for providing me with the initial subject and his constant advice of how to approach it. He believed in my capacity to take on this task and continued to assert his belief in my work, even when I had doubts. Without it, I am not sure if I would have the motivation to finish this extremely complex journey, for which I am extremely grateful.

I thank the KAIROS team, on the premises of which this work was conducted. Everyone was really supportive of me, especially in the final months, and it matters a lot to me. Specifically, I cherish the discussions we had with Robert de Simone, his advices and suggestions that lead to discovery of crucial tools and techniques, used in this work.

This work would not be possible without professors of my alma mater, V. N. Karazin Kharkiv National University. It was due to their dedication and passion in teaching that I have become interested in the subject of theoretical computer science and later decided to pursue the doctorate degree.

I am thankful to my family, who despite being affected by the war in all-encompassing and deeply personal manner, still supported me with and despite everything. I am truly lucky that *you* are my family.

And lastly, I am grateful to the reviewers, Aurélie Hurault and Natalia Kushik, who spend considerable time and effort evaluating this work, and the examiners, Frédéric Boulanger and Abdoulaye Gamatié. Thanks to their feedback and discussions that followed, the work's presentation was further improved.

Table of contents

Appendix

CHAPTER 1

Introduction

1.1 Context

Safety-critical systems are the foundation of the modern society. We rely on them to operate continuously massive and complex machinery. There, the speed, scale and coordination of intervention needed to even attempt a recovery from a failure can be not even near comparable to the one possible for a human. Examples include various high-speed engines and aircraft in general, power plants, metallurgical and chemical factories, the high-speed railway network. And yet the consequences of failure there would not and cannot be tolerated by the society.

There are two ways to reduce the risks. First is through intensive and continuous testing. It is one of the most basic and used form of verification. To see if the system is safe, let it run and observe. Testing cannot and should not be avoided on large systems along the whole design cycle, in order to catch regressions and mistakes early and so cheaply. And in some cases a final test is required to pass in order for a system to become operational, usually mandated by government safety regulations. For instance, cars are crashed into obstacles to simulate possible collisions, before they are assigned a safety score. This is only possible if the definition of safety is bound in time and place, and the scale of destruction in case of something going wrong is manageable and acceptable. This is the case for cars, but not for a power plant, a satellite or when a life is at stake.

A second approach is analytical. It requires a model, a description of the real system in the area we are interested in, and of its operating environment and conditions. Such models can then be manipulated and studied, under various conditions without any physical experiment. A model can simply be a differential equation describing the evolution of a particular property only, solved on a piece of paper and then checked by few other people. Or it can be a set of interacting components, each having a complex and potentially non-deterministic definition of evolution, depending on other parts of the system. In this work, we focus on the latter.

More specifically, we are interested in such descriptions of reactive systems, which main requirement is to be proven bounded in their reaction time and memory footprint under some important events. Before computers became commonplace, the system would be fully made from mechanical components and simple signalling, while still complex to analyse, the already verified parts could be used as they are as long as the expected operating conditions hold. More importantly in this case, the scope of any bad behaviour would be severely limited by the laws of physics, and conversely the reaction would be also physically defined, by the speed of sound, for example. As modern systems implement much more sophisticated behaviour, they require much higher level of coordination among their parts. While this gives them the ability to be smart about the behaviour and so (potentially) efficient, it is also an opportunity to fail big. For example, if a component on an airplane would go into an infinite loop that sends out the messages on the communication bus, this will endanger the integrity of the whole system. If in this case the plane is controlled

"by-wire", i.e. the commands given to control surfaces are transmitted electronically, such local failure may make the whole plane uncontrollable.

From this it should be clear that time, coordination and coordination though time is important in reactive systems, as something true at one step is not necessary true in the next one and that the system is still a sum of its parts. One way to reason about time is to use logical clocks [Lam78]. In this paradigm, time is perceived as increasing count of discrete instants, with explicit or implicit relations to time of Physics. We would even argue that all observable time is logical, as there is no other way to distinguish time, other than by detecting discrete changes in something, meaning events, which are seen as individual progressions of time (instants). Then, a model that uses non-logical time is another view on the time, easier to analyse in some cases, but still have to be converted to events at some point in order to be actionable by humans or control devices which in the end are digital circuits orchestrated by (often periodic) *clocks*.

One way to provide a description of a system in logical time is to use so-called Multiform Logical Time [AMS07], also known as polychronous time model [LTL02; Gam+07], which means that a global view of time is reconstructed from individual, partially independent, clocks. The relations or interdependencies that may exist among clocks, restrict the global observable behaviour of the system. While this allows to approach time by decomposition of the system into subsystems, it does not solve the problem of many interacting parts. The Unified Modelling Language (UML) is widely used in the industry to build models. An extension, called a profile, was designed to address reactive systems. This extension, the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) (standard [Gro08]) has introduced a formal language, called the Clock Constraint Specification Language (CCSL [Mal08]) to annotate UML diagrams with precise time annotations so that such annotations allow for a precise orchestration of the separate diagrams into a model of the whole system. Those annotations are used for validation, verification and prediction of the emerging behaviour. Our work is based on CCSL and we make several propositions to extend it based on our experience with it over the years and on specific requirements from several case studies largely used by the real-time and formal languages research community.

1.2 Problem statement

While MARTE/CCSL is a good way to handle what it was designed for, it has several limitations.

First of all, using UML to describe a timing of the system is a roundabout way to obtain the final specification. While it can be nice to have one or several diagrams for a system, if only specification is needed, coding it is preferable and overall easier. But writing CCSL specifications is a tedious task for big systems, similar to writing assembly. There are tools to improve the experience, like TimeSquare [DM12a] and later the GEMOC initiative with its GEMOC Studio [Com+14]. GEMOC studio allows to create domain specific languages that then would expand, at least partially, into CCSL. It is a great tool for its purpose, but from the point of view of only temporal specifications is again too demanding on the user. So a push for a new way to write CCSL specifications is needed.

But even if we are able to define a specification in a way that we like, it is still too complex to analyse large CCSL specifications. The complexity grows exponentially with the number of clocks, which means systems with a hundred of clocks is completely impossible to handle. This can be managed either by decreasing the expressiveness, introducing more specialized and effi-

cient constraints, applying less precise analysis or by cutting the specifications into individually analysed parts and reconstruct the overall analysis from parts.

1.3 Contributions

The contribution of this thesis is divided into two parts, which exactly address the problems above. First, some evolutions to the language in order to improve user experience and change the expressiveness. The goal here is to specify more by writing less and to be able to optimize more by being more precise. Second, develop a more efficient symbolic analysis techniques for the new language. As the expressiveness grows, we allow approximations to gain tractability.

1.3.1 Language extensions

We introduce several language extensions to CCSL that add constraints and new structuring methods, specifically:

1. our real-time extension adds a set of constraints that link logical clocks to the physical (*real*) time; it is done to be able to specify relations between these real-time defined or related clocks easier from the user's perspective. This also leads to more performant analysis techniques that uses arithmetic and avoid enumeration. We name this language Real-Time CCSL (RTCCSL);

2. our module extension allows to define repetitive patterns for specific systems and share them between the projects if they are generic enough. Additionally, we define a notion of refinement that works with modules, and allows to conduct more scalable analysis, that was not possible before.

3. few additional constraints that are too helpful or important in our opinion not to be present and at the same time are not possible to express with the previous extensions in a compact way;

4. allowing the use of constrained parameters to express specification using templates, which later can be instantiated with concrete values.

Finally, we obtain what we call Modular Real-Time Clock Constraint Specification Language (MRTCCSL). In each of these extensions, we define formally their individual semantics and how the different extensions come together.

1.3.2 Symbolic model checking

Symbolic model checking looks for finite representations of infinite systems spaces. As we shall see, MRTCCSL uses variables that are non bounded, so using symbolic representations is mandatory. There we use two distinct techniques: inductive reasoning which is efficient but only works for a subset of the language due to its narrow assumptions, and abstract interpretation that handles the whole language. We describe how CCSL and each of the extensions would be handled in abstract interpretation, and propose methods to improve its precision specific to our case. These propositions include a new domain, its acceleration and a partitioning technique. Finally, we provide a partial implementation of the described analyses.

1.4 Thesis outline

The thesis consists of four main chapters and a conclusion. Linear order in reading is advised but not exactly required as we reference the definitions we use. The reader should refer to the notations page in case they have doubts about the typesetting, as we use a compact but not conventional syntax, especially for the automata.

1. Chapter 2 presents state of the art, where we give technical definitions that we need to explain the content of this work and an overview of modelling, requirement and programming languages that inspired us and additional motivations of the choices of how to extend MRTCCSL and how to do the analysis;

2. Chapter 3 provides motivation to the features we want in the language from an applied perspective. There we describe several examples and how they were or could be implemented in CCSL;

3. Chapter 4 defines the syntax and formal semantics of MRTCCSL and redefines the same examples of the previous chapter with the new language;

4. in Chapter 5 we explain the analysis techniques and illustrate them on parts of the examples introduced in previous chapters;

5. in Chapter 6 we summarize the work and give several perspectives that follow from it.

1.5 Publications and communications

This thesis has resulted in one publication in the international conference on rigorous state-based methods with the title “Real-Time CCSL: Application to the Mechanical Lung Ventilator” [TM24]. In this article we introduce shortly the language and apply it to the use case selected by the conference. This work is extended and covered by Sections 3.2 and 4.7.1.

We have also made a presentation in an international workshop without proceedings with the title “Real-time extension to clock constraint specification language” [Tok23b]. There we have presented the ideas that later became the basis for the analysis improvements in Section 5.2.5.

Notation

Sets

$\mathbb{B} = \{t, f\}$	set of Boolean values
$\mathbb{N}$	natural numbers with zero
$\mathbb{Q}$	rational numbers
$\mathbb{R}$	real numbers
$X_{>0}$	positive numbers without zero, $X \in \{\mathbb{N}, \mathbb{Q}, \mathbb{R}\}$
$X_{\geq 0}$	positive numbers with zero, $X \in \{\mathbb{N}, \mathbb{Q}, \mathbb{R}\}$
$\mathcal{P}(A)$	power set of some set A

Automata

The automata described in our work are always deterministic and only work with symbols as inputs. We use two styles of automata. In the first, transitions consist of three parts, usually vertically separated: an optional guard, a label and an optional variables' update:

- a guard is any condition, on the variables only, written as $[G(\vec{v})]$, where G is a predicate on current state of variables $\vec{v}$;
- a label is a set of symbols, and in our case these can only be clocks. We are being explicit with labels, so we specify both present and non-present clocks, even though only former is required. $\overline{a}b$ label means that clock a *does not* tick while b ticks, at the same instant. If we write $\overset{?}{a}\ldots$ in a label, then we mean two transitions, one is $a\ldots$, another is $\overline{a}\ldots$, written as one to save space. The special case, $\emptyset$ label, is equivalent to $\overline{c_1 \ldots c_n}$ for all clocks $c_i \in C$ of the constraint being described;
- the functions using which the variables are updated depend on what we describe and are introduced alongside it, but if given a function $f : \vec{V} \to \vec{V}$, the update is written as $\vec{v} := f(\vec{v})$.

An example of a simple automaton is shown on Figure 1.1. We use it to define CCSL and simple constraints in our work.

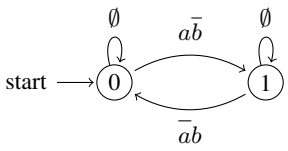


Figure 1.1: a `alternates` b

The second style of automata defines the transitions as relations on state, input and next state, all in one Boolean formula. We use this style to describe real-time constraints, like in Figure 4.3.

CHAPTER 2

State of the art

In this chapter we present some relevant existing methods used to model, verify and program reactive systems. We also talk about more basic and general aspects, like encodings of formal semantics of languages, proof systems and their properties. We start our overview of the system specifications with temporal logic. We then follow with various modeling languages, like Timed Automata, CCSL, TESL, and synchronous languages. We then move to model checking methods and abstract interpretation. And finish with a comparison between the approaches and conclude that there is a need for a (new) hybrid language.

In this chapter we introduce important languages of reactive systems' specification, modeling and programming. We define them formally and informally, including how to use them and solutions they utilize to provide solutions to their specific challenges. We follow it by the general argument about the analysis and verification. It consists of some exact methods like (symbolic) model checking. There, we briefly discuss data structures of BDD and MTBDD, and the principle of SMT. Then we discuss the approximate methods and the abstract interpretation in particular. The description consists of its fundamental theory as well as various subparts that one can tweak in order to obtain better precision in analysis of a specific problem. There we mainly concentrate on dynamic partitioning and acceleration techniques. In the end, we conclude with a summary and comparison the languages and methods, motivating our choice of developing a new language presented later in this work.

2.1 Basics

This section gives an overview on basic notions of logic, proofs and language semantics. We opted in this section as a reminder to the reader, but also to keep the definitions and their interpretation constant through the rest of the work.

2.1.1 Logic

Logic is a method of reasoning in mathematics. It is called formal because the assumptions and rules with which it is performed are strictly defined and expressed as special languages. Generally speaking, these languages are used to state facts about the objects under the study and subsequently deduce, by applying a set of rules, interesting facts about them. Different languages have different expressiveness and can therefore reason about objects of different complexity. Usually, the more expressive the language, the less the amount of properties that can be automatically established (decided) by an algorithm. So depending on the domain, one needs to find the right balance between expressiveness, to deal with interesting systems, and decidability, to be able to verify such systems. We start with basic explanation about a very simple logic, called *propositional logic*.

Propositional logic *Propositional logic* is a language defined with the following syntax:

$$\phi = t \mid f \mid x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid (\phi)$$

where:

- symbols t and f represent constants of true and false;
- symbol $x \in X$ for unknown variables;
- conjunction $\wedge$ and disjunction $\vee$, which allows for compound propositions, with conjunction having priority;
- $\neg\phi$ negating a proposition ϕ;
- parentheses (ϕ) to prioritize expression ϕ regardless of with which operators ϕ is defined.

Given a statement $\phi \in P(X)$, i.e. propositional formula over set of variables X, and a total function $v : X \to \{t, f\}$, assigning true or false value to each variable, the statement is interpreted as the following recursive function $e_v : P(X) \to \{t, f\}$:

$$\begin{aligned} e_v(t) &\stackrel{\text{def}}{=} t \\ e_v(f) &\stackrel{\text{def}}{=} f \\ e_v(x) &\stackrel{\text{def}}{=} v(x) \\ e_v(\phi \wedge \psi) &\stackrel{\text{def}}{=} e_v(\phi) * e_v(\psi) \\ e_v(\phi \vee \psi) &\stackrel{\text{def}}{=} e_v(\phi) + e_v(\psi) \\ e_v(\neg\psi) &\stackrel{\text{def}}{=} -e_v(\psi) \\ e_v((\psi)) &\stackrel{\text{def}}{=} e_v(\psi) \end{aligned}$$

with $*, -, +$ defined as Boolean algebra operators on constants t and f. This function provides meaning to the statements and so is called a language *semantics*, which we explain in more details in Section 2.1.3.

The inverse problem, i.e. finding the assignment v given a formula ϕ and result of $e_v(\phi)$, is called satisfiability problem, or shortened as SAT. SAT problem is decidable but has also been proven NP-complete. The algorithms to solve it include conflict-driven clause learning (CDCL) and Davis-Putnam-Logemann-Loveland (DPLL) algorithm (more about modern CDCL SAT solvers can be found in [MLM21]). Some data structures, like Binary Decision Diagrams (more in Section 2.8.3), provide an efficient symbolic representation and handling of Boolean formulas.

But, SAT assumes that the atoms or variables are independent. When they are not, a modification to the syntax and language semantics should be performed. For example, if the atoms are relations on numbers and numerical variables, then the variables also need a consistent assignment over the whole formula. The solving of such formula is harder, but still decidable as the combination of relations would be still finite and as long as the individual atoms and their conjunction are decidable.

First-order logic *First-order logic* introduces quantifiers over the universe of objects, namely universal $\forall$ and existential $\exists$ quantifiers. With them it is possible to express properties that hold on infinite, even not countable, sets or demand an existence of something, without providing a method to construct it. The quantifiers itself bind variables that later occur in the body of the predicate it describes, with variables being anything, not only Boolean values as with the propositional logic.

And thus, the problem of satisfiability of formulas in first-order logic is not decidable in general. The exception is the fragment where we remove one of the quantifiers and disallow negation, as $\forall i : P(i) \equiv \nexists i : \neg P(i)$. And in the case of number theory, Presburger arithmetics is a fragment of first-order logic with natural numbers and is decidable because multiplication and division between numerical variables is not allowed.

Usually and in this work, we stick to a version of first-order logic, where the domains of variables are defined explicitly, i.e. $\forall i \in \mathbb{N} : P(i)$ and not $\forall i : P(i)$.

2.1.2 Proof systems

A *proof system* is a formal language with a collection of axioms, tautologies, inference and replacement rules used to construct proofs of theorems. In this case, a *theorem* is a word or statement in the language and *axioms* are statements assumed to always hold. It is assumed, that a theorem is either proved or should be proved. *Tautologies* are formulas that are true in every possible interpretation of whatever the language calls atoms and variables, like $x = x$. An *inference rule* is a "transition" between statements: the rule rewrites the statement given, if it satisfies the conditions expressed in the precondition to the rule (upper part in notation) and substitutes with the lower part. A *proof* is either a sequence of the rules, using axioms as starting points and effectively constructing the target theorem, or a sequence that transforms the theorem to a subset of the axioms, or a mix of both, depending if inference, replacement or elimination rules are allowed, defined and used.

The main rule of inference, used in (*deductive*) logic and proof systems in general is modus ponens:

$$\frac{P \to Q \qquad P}{Q}$$

Which is read as: if from P follows Q and P is true, then Q is also true. Other well-known axioms include principle of excluded middle:

$$\frac{}{A \vee \neg A}$$

Elimination of double negation:

$$\frac{\neg\neg A}{A}$$

And proof by contradiction:

$$\frac{\neg P \implies f}{P}$$

Constructive logic does not include these two axioms, so the proofs cannot be written by providing non-existence of negation, i.e. the proof by contradiction is not allowed.

Another important proof technique is induction.

$$P \in \mathbb{N} \to \mathbb{B} : \frac{(P(0) \wedge \forall i \in \mathbb{N} : P(i)) \implies P(i+1)}{\forall i \in \mathbb{N} : P(i)}$$

It reads as: for any predicate of natural numbers, if predicate is true for zero and for all $i + 1$ assuming i is true, then it is true for all numbers. The $P(0)$ case is called a base step and $P(i) \implies P(i + 1)$ an inductive step, where $P(i)$ is an inductive hypothesis.

When a formal system can *prove* a theorem T from a set of proved statements Γ, we write $\Gamma \vdash T$. When an *interpretation* of statements Γ in a target model implies T being true, we write $\Gamma \models T$. Then using these notations, we define two properties of formal systems, which are to be sound and complete.

Definition 2.1.1 (Soundness)**.** *Soundness* is a property of a theory and it states that all the proofs given in the theory are semantically correct, i.e. $A_1, \ldots, A_n \vdash T \implies A_1, \ldots, A_n \models T$.

Definition 2.1.2 (Completeness)**.** *Completeness* is a property of a theory and it states that all correct things in the theory scope can be proven in this theory, i.e. $A_1, \ldots, A_n \vdash T \iff A_1, \ldots, A_n \models T$.

It is obvious from the direction of the implication, that these notions are converse of each other. Ideally, both of them should be satisfied. But depending on the purpose of the system, one is preferable to another. For example, propositional logic with algorithm to solve SAT problem is sound and complete proof system, while abstract interpretation is only sound (more in Section 2.9). While not ideal, it is nonetheless still better than no automatic analysis for a language with undecidable properties.

2.1.3 Semantics

In theory of formal languages, the interpretation we give to the syntax of the language is called *semantics*. It is important to give precise interpretation to a language to evade misinterpretation but also bugs, and to facilitate development of tools, as others would not need to guess what is meant by a given sentence accepted by the language.

There are several styles to encode the semantics. Even though, the semantics are equivalent, at least in the important details for the designer, with each written in a different style, there are advantages and disadvantages to each style when it comes to reasoning about it. These style are: denotational semantics, operational big-step and small-step semantics, and automata semantics.

Denotational semantics consists of a function (or a relation) that translates (resp. equalizes) a syntactic construct to a mathematical model, for example, a predicate on a state space or a set of traces. This places the denotational semantics on the abstract side, as really the only requirement on it, is to use first-order logic and some sort of set theory. And when using sets, it is quite easy to combine, i.e. make a composition of various parts of the original language. Sometimes it is enough to make an intersection of the individual statements. In case of languages that operate over languages, such semantics is straight to the point and so easier to write and understand. But because it is abstract and the parts may be combined in various ways, commonly only restricted by first-order logic, it makes their manipulation and executable implementation hard.

A description closer to executable semantics is *small-step semantics* (also known as structural operational semantics). This semantics defines reduction rules in terms of previous and next machine state for each expression of the language, one step at a time (thus small-*step*), until the state reaches some irreducible point, usually a specific value. It has an advantage that it is straightforward to implement a non-optimizing interpreter with it and simple to allow debug features.

Big-step semantics is somewhere in between denotational and small-step semantics. Instead of reduction to next expression, big-step reduction reduces directly to the value. Depending on the terminal value, the big-step semantics may not be able to distinguish, for example, different errors, while in small-step we can look at what lead to an error exactly. It is possible to convert small-step semantics into big-steps semantics automatically, though only if some assumptions hold [Cio13].

Another way is to define an automaton for each language feature and rules by which they will combine with each other into an automaton of the whole expression. But more importantly it gives a representation to express languages operationally and explicitly, which is not the case with other, rule-based semantics. In this work, we treat automata semantics as a alternative and more readable way to express small-step semantics. The better readability is achieved by explicit statement of transition and their labeling, in each individual state of an automaton, which is not the case when using inference rules or their combination.

The general relation between the semantics is the following, from abstract to concrete:

$$\text{Abstract} \leq \text{Denotational} < \text{Big-step} < \text{Small-step} \equiv \text{Automaton} \leq \text{Concrete}$$

2.2 Temporal logic

Temporal logic denotes a family of languages, that defines modal statements in time and is used extensively to define properties in reactive systems to be later formally verified. *Temporal* is usually distinguished from *timed* as the former usually denotes discrete evolution in successive steps, while the latter refers to chronometric time (as in laws of physics) and dense representations. Compared to propositional logic, the same formula may evaluate into different values depending on what has already happened in the past or may happen in the future, i.e. what is the actual sequence of system states. It is also possible for two seemingly opposite formula to execute to true, because both are possible in concept or in different point in time. Depending on the view point, the logic reasons about one or several timelines, or alternatively, a temporal formula defines what the timelines are and what the behaviour of the system should be a subset of. Still, the actual sequence of events is always single and definitive, and the logic allows us to express only at most countable sequences in finite way.

The family of temporal logics include a lot of languages, but we only cover the following ones:

- Linear Temporal Logic (LTL);
- Computation Tree Logic (CTL);
- CTL*;
- Metric Temporal Logic (MTL);
- Metric Interval Temporal Logic (MITL);
- Signal Temporal Logic (STL).

2.2.1 LTL, CTL, CTL*

We mostly follow the descriptions given by [PP18] here.

Linear Temporal Logic [Pnu77] (LTL) is a language with a linear view on the time. Informally, the language defines a formula that is evaluated on every (infinite) sequence of states, produced by some system. By being able to only check states in the individual sequence, means that from the point of view of LTL as a language, it cannot distinguish different sequences, thus events appear in the same sequence, which explains the name of linear time.

Syntactically the LTL formulas are defined as:

$$\varphi = t \mid f \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \texttt{next}\ \varphi \mid \texttt{prev}\ \varphi \mid \varphi_1\ \texttt{until}\ \varphi_2 \mid \varphi_1\ \texttt{since}\ \varphi_2$$

Propositional part is interpreted as defined above, the meaning of `next` is that it is true if its subformula is true at the next time point (step), `past` if it was true at the previous step. Formally, we define satisfiability relation $\sigma, i \models \phi$, which means that some formula ϕ is true on step i for the state σ, as the following recursive relation:

- $\forall p \in P : \sigma, i \models p \iff p \in \sigma$;
- $\sigma, i \models \neg\varphi \iff \sigma, i \not\models \varphi$;
- $\sigma, i \models \phi \vee \psi \iff \sigma, i \models \varphi \vee \sigma, i \models \psi$;
- $\sigma, i \models \texttt{next}\ \varphi \iff \sigma, i+1 \models \varphi$;
- $\sigma, i \models \varphi\ \texttt{until}\ \psi \iff \exists k \geq i, \forall i \leq j < k : \sigma, k \models \psi \wedge \sigma, j \models \varphi$;
- $\sigma, i \models \texttt{prev}\ \varphi \iff i > 0 \wedge \sigma, i-1 \models \varphi$;
- $\sigma, i \models \varphi\ \texttt{since}\ \psi \iff \exists 0 \leq k \leq i, \forall k < j \leq i : \sigma, k \models \psi \wedge \sigma, j \models \varphi$.

From these usually more natural expressions are derived:

- eventually (finally) $F\phi \stackrel{\text{def}}{=} \Diamond\phi = t\ \texttt{until}\ \phi$;
- always (globally) $G\psi \stackrel{\text{def}}{=} \Box\psi \equiv \neg\Diamond\neg\psi$;
- weak-until $\varphi W \psi \stackrel{\text{def}}{=} (\varphi\ \texttt{until}\ \psi) \vee \Box\varphi$;
- $\texttt{once}\ \phi \stackrel{\text{def}}{=} t\ \texttt{since}\ \phi$;
- $\texttt{historically}\ \psi \stackrel{\text{def}}{=} \neg\ \texttt{once}\ \neg\psi$ (predicate does not hold until this point in time);
- $\varphi\ \texttt{back-to}\ \psi \stackrel{\text{def}}{=} (\varphi\ \texttt{since}\ \phi) \vee\ \texttt{historically}\ \varphi$.

In LTL, the formulas can be divided into several classes, with their names reflecting a property of a reactive system, one would want to check. For a past LTL formula ϕ, these are:

- safety $\Box\phi$;
- liveness $\Box\Diamond\phi$;
- inevitability $\Diamond\Box\phi$;
- progress/reactivity $\Box(\phi_1 \implies \Diamond\phi_2)$.

Any LTL formula has a negation normal form, i.e. a formula can be rewritten into an equivalent, where negation only occurs directly at atoms, similarly to propositional and first-order logic.

For the language of Computation Tree Logic [CE81] (CTL), the same propositional core is present, but the set of temporal operators is different. The language expresses a different subset of behaviour from LTL: certain expressions are impossible to encode in one while possible in another and vice-versa. The temporal operators are strictly pairs of symbols with first being A or E, for universal and existential, and the second being either F, G, X or $[\phi\ \mathrm{U}\ \phi]$, meaning finally, globally, next and until respectively. The meaning of universal and existential quantifiers is not the same as in first-order logic, as instead of objects they quantify possible paths at the point in time they are checked. For example, a universal operator instructs to check that every path starting from the current satisfies its subformula. Because paths are distinguished on the language level, it makes a big conceptual difference with LTL: CTL is able to assert that something is possible in principle in the system but not necessary has to be reached now, and it is what makes the time branching.

For example, an important hardware property of reset is only expressible with CTL and not LTL. We obviously do not want to demand the system to reach reset all the time, only when necessary, which is exactly $AG(EF$ reset$)$.

By combining the two, LTL and CTL, we obtain a new language called CTL* [EH83], where one can freely mix the operators of both. Such language is strictly more expressive than the union of two.

A *Kripke structure* is a state-transition system, with states containing labeled by a subset of atomic propositions and transitions are a subset of directed edges between these states. Each of these languages shall check if a given Kripke structure satisfies a temporal formula. It can be done by recursively checking at what state each subformula is satisfied until the original formula is satisfied in the initial state or until the analysis fully saturates the structure, thus proving that it is not satisfied.

The Kripke structures are universal structures and can be generated from other languages, but may not be necessary convenient to reason about because of how explicit they are. One of the solutions is to use fair discrete systems (FDS) which are symbolic discrete transition systems with additional conditions of justice and compassion requirements: a property has to be infinitely often satisfied by a fair run and if a property is satisfied infinite amount of times, another one has to be satisfied infinitely often too, respectively. Then we would translate both the system under study and the property to check, expressed as a temporal formula, to FDS. With an exception that the formula is translated as its *complement*, so that when the two of them are synchronized, it is enough to check that the overall system has no solutions. If it does, it means, from the contrary, that the system does not satisfy the property and the result is a counter example.

2.2.2 MTL, MITL and STL

Metric Temporal Logic [Koy90] is a modification to LTL where operators are constrained in time. This means that the states are additionally annotated with dense time, increasing as the discrete evolution of the system progresses. Thus, the language of metric logic is able to express conditions on time depending on the logical state of the system and vice-versa.

When an interval $I = [l, r], l \leq r$ is added to temporal operators until or since, their semantics changes the following way:

- until operator:

$$\begin{aligned} \text{before: } \sigma, i \models \varphi \ \texttt{until} \ \psi &\iff \exists k \geq i, \forall i \leq j < k : \sigma, i \models \psi \wedge \sigma, i \models \varphi \\ \text{after: } \gamma, t \models \phi \ U_I \ \psi &\iff \exists t' \in I + t : \gamma, t' \models \psi \wedge \forall t < t'' < t' : \gamma, t'' \models \phi \end{aligned}$$

- since operator:

$$\begin{aligned} \text{before: } \sigma, i \models \varphi \ \texttt{since} \ \psi &\iff \exists 0 \leq k \leq i, \forall k < j \leq i : \sigma, k \models \psi \wedge \sigma, j \models \varphi \\ \text{after: } \gamma, t \models \phi \ S_I \ \psi &\iff \exists t' \in I - t : \gamma, t' \models \psi \wedge \forall t < t'' < t' : \gamma, t'' \models \phi \end{aligned}$$

where $\gamma : T \to A, T \subseteq \mathbb{R}_{\geq 0}$, with A being atomic propositions. Because MTL was claimed undecidable (by [AH94], but the work [OW05] claims otherwise), a subset of the language, Metric Interval Temporal Logic (MITL) was proposed [AFH96]. It features a restriction on intervals: they cannot be singletons, i.e. intervals of form $[a, a]$.

A further specialization, Signal Temporal Logic (STL) was proposed later [MN04], as the name suggests, specifically to monitor continuous signals. There, the MITL language is further restricted to finite rational bound intervals. Additionally, signals under analysis are finite in time and of finite variability to evade the zeno-effect (infinite number of events in finite time).

The signals are checked to satisfy the formulas in two steps: first, the continuous signals are split into intervals by atomic propositions, where inside the interval the signals are continuously true or false. Thanks to finite variability, the cardinality of such interval covering is finite too. We show an example of this on Figure 2.1. Next, the intervals of each formula are recursively derived from its subformulas, as demonstrated on Figure 2.2.

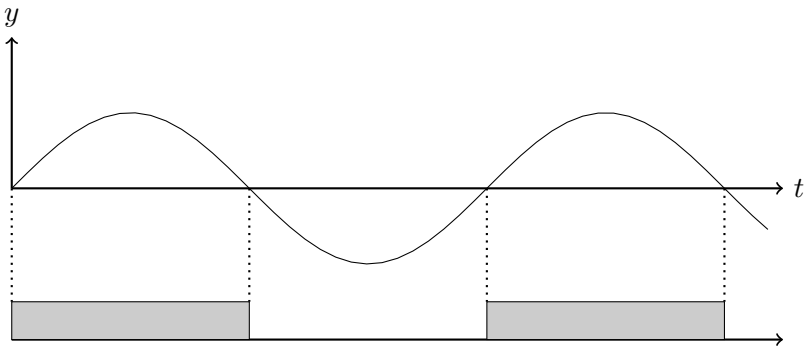


Figure 2.1: Transformation of signal $y = \sin(t)$ to intervals by condition $y \geq 0$

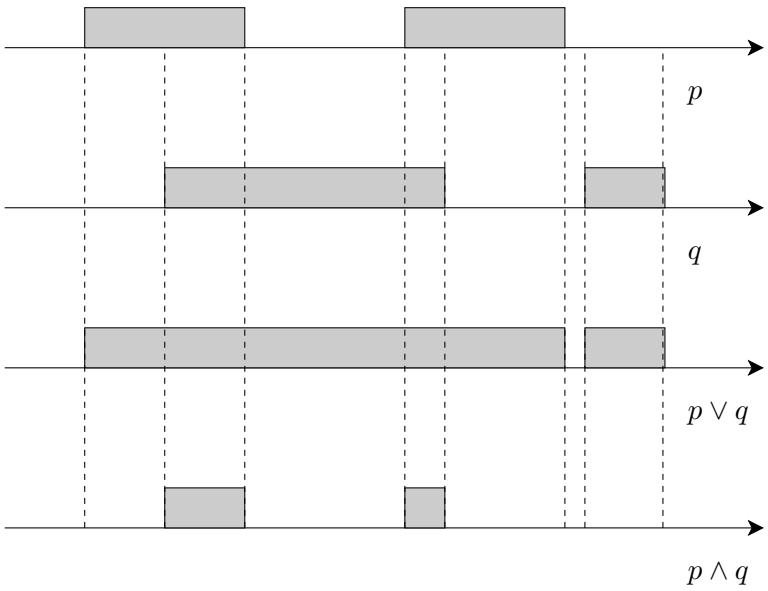


Figure 2.2: Point-wise semantics of $p \vee q$ and $p \wedge q$ for signals p, q

While for $\vee$ and $\wedge$ the semantics is point-wise, `until` operator is a bit more involved and requires a shift into the past, denoted as $\ominus$, and which we demonstrate on Figure 2.3. As every signal starts at zero, if the original formula is true at this instant, then the signal satisfies the desired property. To construct an interval for an STL formula, we need to go though the signals

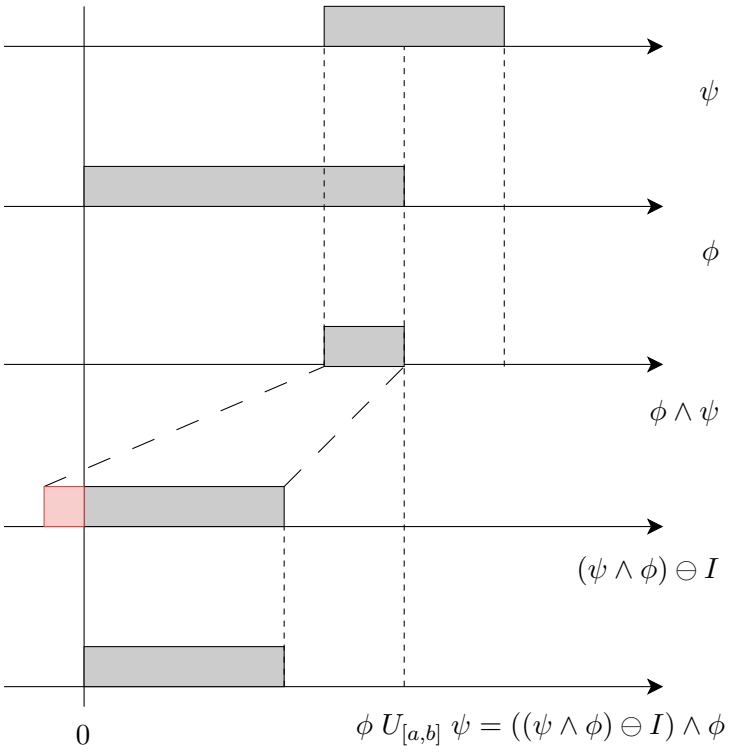


Figure 2.3: Semantics of $\phi\ U_{[a,b]}\ \psi$

and extract the intervals, and recursively apply the definitions above to construct intervals for each subformula. This algorithm is $O(k \cdot n)$ complex, with k being number of subformulas and n being maximal number of intervals in the signals. An disadvantage of the method is that we have to have a finite signal of long-enough length.

2.3 Synchronous languages

Synchronous languages is a family of languages with the common feature of synchronous hypothesis. Programs written in these languages are generally used to write software for reactive systems as they are literally reaction loops: read all inputs, react accordingly, write all outputs. No input is lost, as all inputs as considered synchronous to compute the current reaction. Thus, the so called *synchronous hypothesis* or *assumption* means that the reaction is fast enough (pre-computed and verified by the compiler) so that its execution time is negligible compared to the arrival speed of new inputs. An *instant* then is the period of time from reading the inputs to writing the outputs, and therefore inputs/outputs are said to be *instantaneous*. A *logical clock* denotes an (infinite) sequence of instants at which variables change their values (inputs, outputs, or local variables). Then the superset of instants of all clocks would define the base clock, the rate at which the program reacts. If one can determine the worst-case time of the reaction d_{WCET}, then an easy implementation consists of using a periodic clock of period p, such that $p > d_{\text{WCET}}$. Thus, from the point of view of synchronous languages, they use the logical time (execution step) while implementation conforms to physical time by the mentioned condition.

Obviously, when something is developed in such a language, it is imperative to check that the assumption holds. To help with this, the languages feature checks for termination. Infinite memory is disallowed by finite types, recursion should always end and checks are in place so that any variable does not depend on itself at the same instant. As reactive system implementation in synchronous language is its control, it has to be deployed on a computation unit, usually a microcontroller. Interaction with the environment from a microcontroller requires a specific code, including specific for the device memory addresses, and is called a hardware abstraction level (HAL). This code is usually written in C and is precompiled before linking into the synchronous language program. Meaning that the check of termination or other important properties of these external parts cannot be handled by the synchronous language alone and should be handled in conjunction with other tools.

The (historical) synchronous languages include Signal, Esterel, Lustre, SyncCharts, Zelus. As with more general languages, these languages represent different programming styles. Esterel is imperative, Lustre and Zelus are declarational and functional, SyncCharts [And95] uses hierarchical state-machines, Signal is both programming and specification language [DB03].

We do not go into the details of languages other than Lustre and derivatives, as they are less relevant to the content of this work, but we still want to give them credit and explain what their main features are, as these *are* important to later languages, more specifically CCSL (Section 2.6).

Signal [Gue+91] is able to describe programs as relations, which on one hand allows it to be easily compositional. On another hand it can encode systems that are either deadlocking (no next state) or non-deterministic (several next states) from the point of view of the clocks that we consider to be input. Thus, a proof should be made that establishes the determinism of the system before producing any code.

Esterel [BR83] (see [Ber02] for the latest open version) is similar in a sense that it defines reactions to signals which produce other signals. A signal is present or not at some instant (this is defined by its clock) and has a value (in a given domain). A given signal can have only one value at each instant. The compiler is in charge of computing (by using a fixpoint) this value or reject the program. We talk about fixpoints in more detail in Section 2.9, but shortly, a fixpoint in this case requires an application of the reactions to the state and the inputs some number of times. As it is not given that the number of applications is finite, Esterel demands the reactions to not form a loop. It is a sufficient condition for the fixpoint to converge in finite time.

2.3.1 Lustre

Lustre is a synchronous dataflow language designed for programming reactive systems [Cas+87]. Its industrial version is called SCADE [CPP17] and has been successfully used for the design of numerous industrial problems in avionics, for example, by Airbus [BBP23], but also on transportation systems and vehicles in general [AVR19].

It is inspired by the control theory, where the nodes are defined in compositional way with feedback loops. Thus, the core of the language consists of streams and nodes. Streams are sequences of values that are present (or not) on ticks of the global reference clock (steps). This global clock is then assumed to satisfy the synchronous hypothesis with respect to the reactions, defined with nodes. Thus, in a sense, the streams themself are subclocks of the reference clock. A node is a function that transforms one stream into another. The (arithmetic or logical) operators are defined point wise on the streams. Temporal operators, like `pre` and `->` allows the programmer to use previous values in a stream and to define the initial value in the beginning of the stream.

```
1 node Counter() returns ( OK : bool );
2     var C : int;
3 let
4     C = 0 -> pre C + 1;
5     OK = C >= 0;
6 tel
```

Listing 2.1: Example of Lustre code

Other two operators are `when`, allows to sample or simply remove values in a stream depending on a Boolean condition, and `current`, which reverts the sampling by interpolating the values when the original stream is empty by using the last known value. An example of a simple Lustre node is given in the Listing 2.1.

As the nodes are declarative, the order of computation is defined by the dependencies between expressions in compilation. For that, it should be non-ambiguous and finite. Thus, every variable is checked to not be used in defining itself, as it creates a dependency loop. To break such loop, one usually uses the defer operator `pre`, i.e. use previous value.

2.3.2 Zelus

Zelus [Ben+18] is a derivative of Lustre and features Ordinary Differential Equations (ODE) as additional nodes. This allows to describe reactive system control and continuous behaviour in the same language. An example of such system is shown in Listing 2.2. While ODEs are a subset of differential equations, they do cover substantial part of physics: laws of motion, resonance and oscillation in circuits. In control, we react to changes in the environment, usually modeled as an event when certain condition on properties is met. This in turn can be redefined as a point when a differential equation intersects the condition, or in other words, when a condition on ODEs' variables changes from false to true or vice-versa, and is known as a *zero-crossing event*. But, as it is not possible to analytically find such point to every differential equation, a numerical approximation has to be used. The general approach of the simulation of Zelus is then the following: the simulation oscillates between solving of differential equations and the control part execution. Every time a zero-crossing is detected, the simulation returns to control, and when control needs new approximation of the environment it returns to the equations.

A more conservative subset of Zelus, which uses linear relations instead of differential equations, is discussed in [BBP17]. While less expressive, it allows to analyse the evolution of the system in symbolic manner, similar to Timed Automata.

```
(** Bouncing ball. *)

(* [ground x] returns the position in [y] *)
let ground x = Flatworld.ground(x)
let ground_abs x = Flatworld.ground_abs(x)

let x_0 = 5.0
let y_0 = 10.0
let g = 9.81
let loose = 0.8

(* The bouncing ball *)
let hybrid ball(x, y_0) = (y, y_v, z) where
  rec
      der y = y_v init y_0
  and
      der y_v = -. g init 0.0 reset z -> (-. loose *. last y_v)
  and z = up(ground(x) -. y)

(* Main entry point *)
let hybrid main () =
  let rec (y, _, z) = ball(x_0, y_0) in
  present (period (0.04)) | z -> Showball.show x_0 (y fby y) x_0 y;
  ()
```

Listing 2.2: Example of Zelus code

2.4 Timed Automata

Timed Automata, as its name suggests, is a timed variant of automata. It is extensively used in development, proving correctness and simulation of real-time reactive systems. The tooling for Timed Automata or its variants include UppAaL [LPW95], TChecker [HPS] and IMITATOR[And21]. It was first proposed in [AD94], and this original version is presented here, mainly because it is decidable. Version of Timed Automata in UppAaL adds manipulation of integers and so is not decidable in general. The work on Timed Automata model checking lead to development of efficient data structures such as Clock Difference Diagrams [Lar+98] (equivalent to Difference Bound Matrices [BM83] and later Zone abstract domain, described in Section 2.9.4).

2.4.1 Preliminaries

X is a set of clocks, $v \in X \to \mathbb{R}_{\geq 0}$ time assignment of clock to a non-negative real number or a vector from $\mathbb{R}^X_{\geq 0}$ is a particular state of the clocks. For some $\delta \in \mathbb{R}_{\geq 0}$, $(v+\delta)(x) = v(x)+\delta$. Given $Y \subseteq X$, $[Y]v$ is a reset, such that $\forall x \in Y : ([Y]v)(x) = 0 \wedge \forall x \in X \setminus Y : ([Y]v)(x) = v(x)$.

Clock constraints over X, $C(X)$, are defined as the following language:

$$g = x \bowtie c \mid x - y \bowtie c \mid g \wedge g \mid true$$

where $x, y \in X, c \in \mathbb{Z}, \bowtie \in \{<, \leq, =, \geq, >\}$. $x - y \bowtie c$ are so-called diagonal constraints and conditions without them are called diagonal-free and denoted as $C_{df}(X)$.

Clock assignment v satisfies guard g, written as $v \models g$, is true when replacing clock with values given by v evaluates to true. Then the whole set of solutions to a formula g is $[\![g]\!] = \{v \mid v \in X \to \mathbb{R}_{>0} : v \models g\}$.

2.4.2 Definition

A timed automaton is a tuple $A = (Q, X, q_0, T, F)$ where Q is a finite set of states, X is a finite set of clocks, $q_0 \in Q$ is the initial state, $T \subseteq Q \times C(X) \times \mathcal{P}(X) \times Q$ is a finite set of transitions, and $F \subseteq Q$ is a set of final states. A timed automaton is said to be diagonal free if $C(X)$ is replaced by $C_{df}(X)$ in the definition of the transition relation.

Configuration is a tuple $(q, v), q \in Q, v \in X \to \mathbb{R}_{>0}$. The initial configuration is $(q_0, 0)$, the evolution of the configuration is given with the following *alternating* rules:

- time elapse: $(q, v) \xrightarrow{\delta} (q, v + \delta), \delta \in \mathbb{R}_{>0}$
- discrete transition: $(q, v) \xrightarrow{t} (q', v')$ if there is a transition $t = (q, g, R, q') \in T$ such that $v \models g$ and $v' = [R]v$.

A run of A is a finite or infinite alternating sequence of these rules starting from the initial: $(q_0, 0) \xrightarrow{\delta_1} (q_0, 0 + \delta_1) \xrightarrow{t_1} (q', v_1) \xrightarrow{\delta_2} (q', v_1 + \delta_2) \xrightarrow{t_2} \ldots$.

2.4.3 Analysis

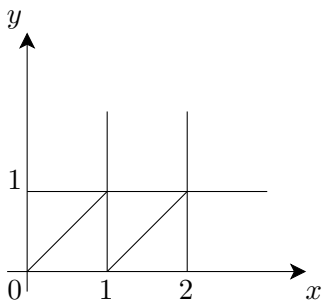


Figure 2.4: Clock regions

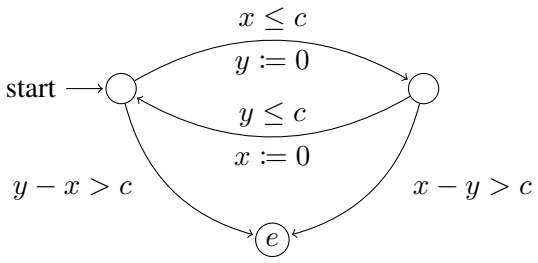


Figure 2.5: Alternating Timed Automaton with bad state e

The model checking is decidable for pure timed automata, without extensions with arbitrary integer state and other expressions, like in UppAaL. The original reachability algorithm used region graph, a directed graph with vertices containing diagonals $c_1 < x = y < c_2$, inner triangles $c_1 < x < y < c_2$ and corners $(x, y) = (c_1, c_2)$, where c_1 and c_2 are constants (Figure 2.4). Because the number of regions is exponential, other methods are used. If timed automata is bounded

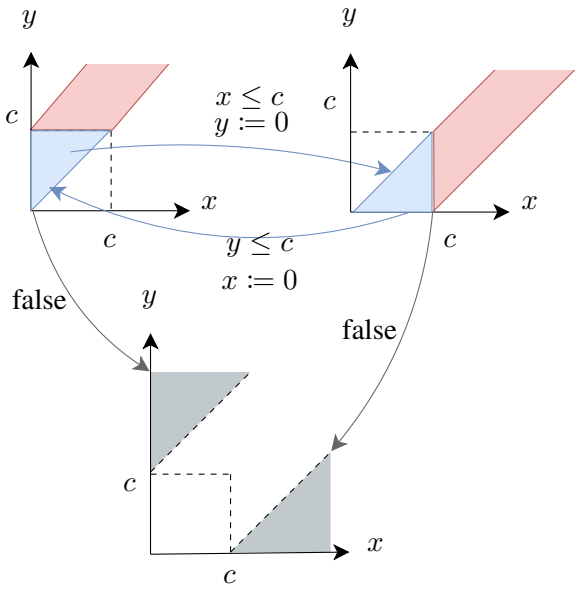


Figure 2.6: Symbolic analysis of Figure 2.5, blue regions are live, red are valid non-live, gray are for the error location and are not reachable

and diagonal-free, an algorithm based on least fixpoint and zone abstract domain (bounded diagonal matrices) can be used, otherwise an approach using simulation relation. We show a simple automaton on Figure 2.5 and its analysis on Figure 2.6. More about the latest advances in the model checking of Timed Automata is written in [Bou+22].

2.5 Event-B

Event-B [Abr10] is a framework which evolved from the B-method [Abr+91] and features discrete systems triggered with parametrized events. We base our explanation of Event-B mainly on [Hoa13].

As with B-method, refinement is one of the most important features which extends to the concept of context, instead of only machines. Contexts are static knowledge including sets, constants and properties on them. These properties divide into axioms and theorems, with the usual meaning. The framework then helps to prove the theorems using the axioms, either by automatically using heuristics or by stating what cases are not covered. Machines specify dynamic behaviour and contain variables, invariants, theorems, events and a variant. Variables define the state, invariants properties maintained by the state and theorems are additional properties derived from the invariants (under some context). Events (thus the name Event-B) define conditions for the system to evolve from one state to another one. Invariants over states must still be preserved. The methodology forces the design to prove that each event will preserve those invariants by generating so-called proof obligations. An event may feature a guard that defines, an enabling condition, a set of relevant parameters, and modifications to the state variables. A special event is used to initialize the state. If update to the state is a relation, such transition is non-deterministic and should be further refined. To prove that an invariant holds, it is usually inductively proved. For that in-

variant is proved to hold after the initialization event and is preserved every time something else occurs (and so changes). For most of these features, proofs have to be provided. These include proofs of theorems in contexts and machines, correctness of refinements, consistency, or in other words, that all updates preserve invariants. Usually though, the user does not see the full proof statements, but only the parts that the framework could not prove on its own.

To summarize, in Event-B one first would define an abstract machine defining the goal of the system as generally as possible with the least specialized assumptions. The assumptions (context) are then expanded if needed, when the machine is refined. The hierarchy of the contexts does not have to be a tree, just any directed graph with no loops. The refinement is a relation, that introduces more variables and events, or completely replaces them (see Figure 2.7). The method still requires the developer to prove that the changes in internal structure of the implementation provide the same guarantees, but it helps in this with a structured approach to generation of proof obligations.

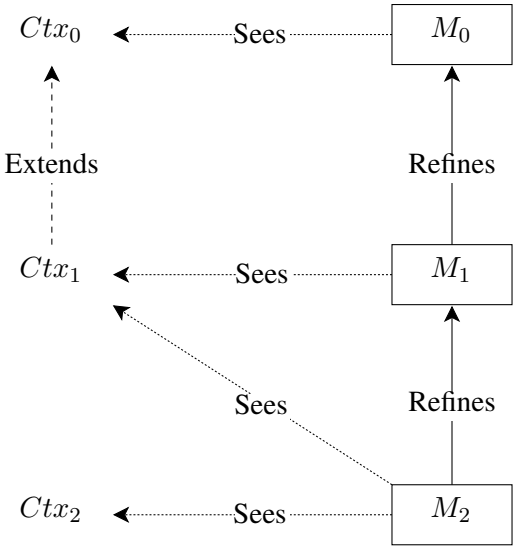


Figure 2.7: Hierarchy of refinement, extension and dependencies in development with Event-B

2.6 CCSL

The Clock Constraint Specification Language (CCSL) is a declarative language used to specify behaviour of reactive systems in composable way, first defined in [And09]. It consists of relations, called constraints, which define what clocks can and not be, separately. This means that the overall behaviour has to be solved and its properties found, which is the main task of the analysis developed.

CCSL is a language developed to fill the gap between the requirements, the formal specification of these requirements and their operationalization. Thanks to its composable nature, it allows the translation of requirements, usually given in plain English, one by one, while constantly discovering if the specification as a whole still makes sense, i.e. contains solutions, and that our own understanding of the requirements is valid. As the focus of the language is on time and causal-

ity, which is usually an afterthought in other languages, not all even simple behaviour in general purpose languages can be expressed easily, and is a conscious design choice.

CCSL was inspired by the synchronous languages and in a sense is one: the ticks of clocks are synchronous on a global but unknown clock. The difference though, is that usually (Signal is the exception) synchronous languages are *programming* languages, i.e. supposed to define operationally the behaviour, while CCSL is a specification language. CCSL also disregards the values in the instants itself, unlike Lustre or Esterel, and so as a language cannot express control on anything other than number of occurrences of ticks and only in a specific way. The second source of inspiration is real-time scheduling, with its rates, periodic and sporadic tasks, deadlines and data-dependencies.

In this section we introduce the language itself and its equivalent semantics in different styles. Following by properties that one would like to check on CCSL specifications and tools that help to achieve it.

To see how modeling with CCSL is done, refer to Chapter 3 and Section 3.1 specifically, or other use cases.

2.6.1 Language description

CCSL is a language operating with constraints as statements over logical clocks as variables, collected in a specification. Each constraint binds ticks of its clocks to appear only in a certain order, effectively reducing the set of possible behaviours.

Definition 2.6.1 (Logical clock)**.** A logical clock c is a finite or infinite sequence of ticks (instants) $(c_i)_{i=0}^{\leq\infty}$, 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 causality between the ticks can be specified.

Definition 2.6.2 (Constraint)**.** A constraint r is an n-ary relation on set of clocks C and defines a set of allowed schedules which can be seen as language. Language is denoted as $\Sigma(r)$.

The set of all possible constraints on C will be denoted as $\mathcal{R}(C)$.

As a language, CCSL operates on variables of logical clocks, and an assignment of each variable to its logical clock needs to be made in order to satisfy it. Such assignment is called a schedule. Not every assignment is valid though.

Definition 2.6.3 (Schedule)**.** A schedule is a function $\sigma : \mathbb{N} \to \mathcal{P}(C)$. Given an execution step $n \in \mathbb{N}$ and a schedule $\sigma, \sigma(n)$ denotes a set of clocks that tick at step n.

Definition 2.6.4 (Valid schedule)**.** A schedule σ satisfies some constraint r if it is contained in the set of its allowed schedules and is defined as $\sigma \models r = \sigma \in \Sigma(r)$.

The Figure 2.8 illustrates the relation between logical clocks and a schedule.

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. In principle, the distinction is superficial, as both are mathematically relations and so restrictions on clocks defined by the expressions also influence what the argument

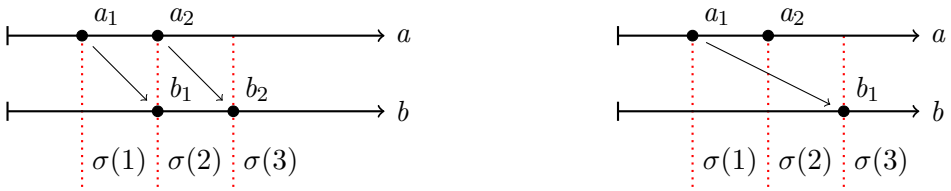


Figure 2.8: Some valid schedules for $a \prec b$ constraint, arrows represent strict order

clocks can be. Some of the constraints are parametrized with integers, thus acting like a meta-constraint $\mathbb{N} \to \mathcal{R}(C)$. The notations and definitions of the common relations and expressions are defined later in Table 2.1 and Table 2.2.

Then, as the specification is a flat program, it is only enough to define what synchronization is between two constraints, the definition of what specification as a whole represents is a reduction of the list of constraints with synchronization, and is universal across the semantics.

Definition 2.6.5 (Constraint synchronization)**.** A synchronization of constraints represents all schedules that are satisfied by both constraints and is defined as

$$r_1 \parallel r_2 \stackrel{\text{def}}{=} \Sigma(r_2) \cap \Sigma(r_2)$$

.

Definition 2.6.6 (Specification interpretation)**.** A specification Φ is a finite set of constraints. It is interpreted as a synchronization of all constraints in it:

$$\begin{aligned} \sigma \models \Phi &\stackrel{\text{def}}{=} \sigma \models r_1, \ldots, \sigma \models r_n \\ &\equiv \sigma \models \Big\|_{r \in \Phi} r \\ &\equiv \sigma \in \bigcap_{r \in \Phi} \Sigma(r) \end{aligned}$$

This allows us to follow mostly the same pattern across all the semantics defined later: definitions of intermediate structures, translation of constraints into the structures and their synchronization.

A note: in this work we rename constraints previously known as supremum and infimum into slowest and fastest respectively, as these names provide better intuition behind the name and the behaviour of these constraints. Also, we use $c = \texttt{next}\ a$ as a shortcut for $c = a\ \$\ 1$ and $a\ \texttt{alternates}\ b$ for $a < b < \texttt{next}\ a$.

In this work we write specifications in a few different ways, depending on their size and if it is important to emphasize on synchronization or not. For example, specification $a \prec b \prec c$ is equivalent to $a \prec b \parallel b \prec c$, can also be written as:

$$\begin{aligned} a &\prec b \\ b &\prec c \end{aligned}$$

2.6.2 Denotational semantics

We start the definition of the formal semantics of CCSL with denotational semantics. As we explained before in Section 2.1.3, denotational semantics provides an interpretation function into some mathematical model, transforming syntactic constructs into operations on model objects. In our case, it is an operator $[\![\ ]\!] : \mathcal{R}(C) \to \mathcal{P}(\mathbb{N} \to \mathcal{P}(C))$ translating constraints to their sets of valid schedules. As the result is a set, the definition of the synchronization between constraints is somewhat trivial and consists of an intersection.

Definition 2.6.7 (Denotational synchronization)**.** Synchronization of denotational constraints is defined as an intersection of languages. For a given constraints ϕ, ψ on clocks $C(\phi)$ and $C(\psi)$ respectively and their denotational semantics $[\![\ ]\!]$, the interpretation of their synchronization is defined as:

$$[\![\phi \parallel \psi]\!] \stackrel{\text{def}}{=} \text{extend}_{C(\psi)\setminus C(\phi)}([\![\phi]\!]) \cap \text{extend}_{C(\phi)\setminus C(\psi)}([\![\psi]\!])$$

where $\text{extend}_{C_e}(X) \stackrel{\text{def}}{=} \{\lambda i \to \sigma(i) \cup e_i \mid \sigma \in X, e_i \in \mathcal{P}(C_e)\}$ adds all combinations of non-covered clocks $C_e, C_e \cap C(X) = \emptyset$, to the schedules X, and thus extends the domain of definition of each constraint to the common clock set $C \equiv C(\phi \parallel \psi) \equiv C(\phi) \cup C(\psi)$ before making synchronizing intersection.

Alternatively, if each constraint rewrites into first-order formula with some function FO, and there is a solver SC that finds schedules from such formula, then $[\![\phi \parallel \psi]\!] \equiv \text{SC}(\text{FO}(\phi) \wedge \text{FO}(\psi))$, i.e. the synchronization is defined as the logical conjunction.

Next we present two alternative ways to describe the interpretation function: index-based and history-based. And the definition of synchronization does not depend if it is history or delta-counter based, as it is a style of definition, the type of the result is the same. Both ways based on first-order logic with both existential and universal quantifiers, which makes finding solutions to the constraints undecidable. In fact, the synchronization then is a conjunction of the propositions. But while the two translate constraints into different models, their expressiveness is equivalent.

2.6.2.1 Index-based

First variant is index-based and was first described in [DAG14] and formalized in [Mon20]. We base the explanation here on the former with modifications. The distinct feature of index-based denotational semantics is that the relations are defined by querying the objects of clocks using indices, variables bound by quantifiers, and setting relations between such instants.

First of all we assume there exists a partially ordered set of instants I, the tuple $\langle I, \equiv_I, \prec_I \rangle$; $a \preccurlyeq_I b$ is a shortcut for $a \prec_I b \vee a \equiv_I b$. Assuming such set, then a clock is its subset. To make the definition stricter, we actually define it as indices first.

Definition 2.6.8 (Denotational clock)**.** A clock c is the tuple $\langle i^{\max}, [\_] \rangle$, where $i^{\max} \in \mathbb{N}$ is the maximum allowed index, and a *dependent function* $[\_] : \forall i \in \mathbb{N} : i < i^{\max} \to I$, which means that it is defined only on indices before $i^{\max}$. Additionally, the function is strictly monotonic on integers to maintain the total order. Then an indexing of a clock is the function $\_[\_] : C \to \mathbb{N} \to I$, where C is a set of clocks.

To check, that a tick with index i exists in clock a we will write $i \in a$.

To construct schedule σ, one assigns an order between the instants following the rules defined in Table 2.1, but so that a total order is achieved. While this mostly results in a whole set of possibilities, it is the point. When it is not possible, the schedule does not exists and so specification is empty.

	Constraint	Notation	Definition, $\forall i \in \mathbb{N}$
Relation	Causality	$a \preccurlyeq b$	$a[i] \preccurlyeq_I b[i] \wedge i \in b \implies i \in a$
	Precedence	$a \prec b$	$a[i] \prec_I b[i] \wedge i \in b \implies i \in a$
	Exclusion	$a \# b$	$\forall j \in \mathbb{N} : a[i] \not\equiv_I b[j]$
	Coincidence	$a = b$	$a[i] \equiv_I b[i]$
	Subclocking	$a \subseteq b$	$\exists j \in \mathbb{N} : a[i] \equiv_I b[j]$
	Alternation	a `alternates` b	$a[i] < b[i] \wedge b[i] < a[i+1]$
Expression	Delay	$b = a \$ d$	$b[i] \equiv_I a[k+d]$
	Ternary delay	$b = a \$ d$ `on` r	$\exists j \in \mathbb{N} : b[i] \equiv_I r[j] \iff \exists k : r[j-1-d] \prec_I a[k] \preccurlyeq_I r[j-d])$
	Slowest	$c =$ `slowest`(a,b)	$c[i] \equiv_I \max(a[i], b[i])$
	Fastest	$c =$ `fastest`(a,b)	$c[i] \equiv_I \min(a[i], b[i])$
	Intersection	$c = a * b$	$\forall j \in \mathbb{N} : a[i] \equiv_I b[j] \iff \exists k \in \mathbb{N} : a[i] \equiv_I b[j] \equiv_I c[k]$
	Union	$c = a + b$	$\exists k \in \mathbb{N} : c[i] \equiv_I a[k] \vee c[i] \equiv b[k]$
	Minus	$c = a - b$	$\exists k \in \mathbb{N} : c[i] \equiv_I a[k] \wedge \forall j \in \mathbb{N} : a[k] \not\equiv_I b[j]$
	Periodic	$c =$ `skip` φ `every` p a	$c[i] \equiv_I a[p * i + \varphi]$
	Sampling	$c =$ `sample` a `on` b	$(\exists k \in \mathbb{N} : (b[i-1] \prec_I a[x] \preccurlyeq_I b[i]) \iff \exists j \in \mathbb{N} : c[j] \equiv_I b[i])$

Table 2.1: Definitions of CCSL constraints, $a, b, c, r \in C,\ d \in \mathbb{N}, p \in \mathbb{N}_{>0}$

Something that this form somewhat hides is that when $a[i]$ is written it is supposed that this tick exists. When it does not, the relation is skipped, i.e. $a[i] \bowtie b[j]$ rewrites into $(i \in a \wedge j \in b) \implies (a[i] \bowtie b[j])$. In some constraints it is expected behaviour, while in others it is not that symmetric, like precedence. For such constraints we write the additional condition explicitly. Also, to simplify, whenever equivalence is specified, both of the ticks has to exist, i.e. $a[i] \equiv_I b[j] \implies (i \in a \iff j \in b)$.

2.6.2.2 History-based

An alternative definition of denotational semantics uses an auxiliary function that we call history: a function that remembers how many times a clock ticked before the step in the schedule. The original definitions can be found in [MMR13; MdS15].

Definition 2.6.9 (History)**.** 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 recursively and 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.

By using the history and what is present in the step, we define a proposition that asserts if the schedule satisfies the constraint. The propositions for all constraints are defined in Table 2.2. An important note: the definitions are shortened to fit and the propositions in cells are universally quantified by step n.

	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	$a \subseteq b$	$a \in \sigma(n) \Rightarrow b \in \sigma(n)$
	Alternation	a `alternates` b	$(H_\sigma(a,n) = H_\sigma(b,n) \implies b \notin \sigma(n+1)) \wedge$ $(H_\sigma(a,n) = H_\sigma(b,n)+1 \implies a \notin \sigma(n+1))$
Expression	Delay	$b = a$ \$ d	$H_\sigma(b,n) = \max(H_\sigma(a,n) - d, 0)$
	Ternary delay	$b = a$ \$ d `on` r	$b \in \sigma(n) \Leftrightarrow r \in \sigma(n) \wedge$ $(\exists m \leq n : a \in \sigma(m) \wedge H_\sigma(r,n) - H_\sigma(r,m) = d)$
	Slowest	$c =$ `slowest`(a,b)	$H_\sigma(c,n) = \min(H_\sigma(a,n), H_\sigma(b,n))$
	Fastest	$c =$ `fastest`(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	$c = a + b$	$c \in \sigma(n) \Leftrightarrow (a \in \sigma(n) \vee b \in \sigma(n))$
	Minus	$c = a - b$	$c \in \sigma(n) \Leftrightarrow (a \in \sigma(n) \wedge b \notin \sigma(n))$
	Periodic	$c =$ `skip` φ `every` $p\ a$	$c \in \sigma(n) \Leftrightarrow (H_\sigma(a,n) - \varphi = p \cdot H_\sigma(c,n) \wedge a \in \sigma(n))$
	Sampling	$c =$ `sample` a `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 2.2: Definitions of CCSL constraints, $a, b, c, r \in C,\ d \in \mathbb{N}, p \in \mathbb{N}_{>0}$, a schedule σ and its history H_σ

2.6.3 Automata semantics

Automata are not ambiguous and readable way to formalize operational semantics, and the suitable to define languages that define languages, like CCSL. Here we present two equivalent versions of the automata semantics, the first featuring unbounded or potentially infinite state explicit automata, and the second, symbolic automata with integer variables and transition guards.

2.6.3.1 Unbounded automata

The unbounded automata as CCSL definition were first presented in [MMR13]. We start with the definition of the automaton, or more precisely, transition system, called Clock-Labeled Transition System (cLTS).

Definition 2.6.10 (Clock-Labeled Transition System)**.** A Clock-Labeled Transition System (cLTS) is a tuple $\mathcal{A} = \langle S, T, s_0, C \rangle$ where:

- S is a countable set of states;
- $s_0 \in S$ is the initial state;
- C is a finite set of clock symbols;
- $T \subseteq S \times \mathcal{P}(C) \times S$ is a set of transitions, with $(s, Y, s') \in T$ meaning that all the clocks in $Y \subseteq C$ tick when the transition $s \to s'$ is fired.

cLTS is distinct from other definitions of other automata by fact that the states are not necessary finite and that the alphabet, usually an opaque set of symbols, is always a power set of clock symbols. Additionally, the automaton contains the clocks on which it is defined C, which is important when synchronization is performed. Without it, we would have to define at each transition, which clocks should and should not tick.

Definition 2.6.11 (cLTS synchronization)**.** The synchronized product of cLTSes $\mathcal{A} \parallel \mathcal{B}$ is a cLTS $\langle S, T, s_0, C \rangle$ constructed as:

- $S \stackrel{\text{def}}{=} S_A \times S_B$;
- $T \stackrel{\text{def}}{=} \{((s_a, s_b), Y_a \cup Y_b, (s'_a, s'_b)) \mid \forall (s_a, T_a, s'_a) \in T_A, (s_b, T_b, s'_b) \in T_B : (C_A \cap Y_B) = (C_B \cap Y_A)\}$;
- $s_0 \stackrel{\text{def}}{=} (s_{0A}, s_{0B})$;
- $C \stackrel{\text{def}}{=} C_A \cup C_B$.

It is obvious that such synchronization procedure is computationally intensive and exponentially complex thanks to Cartesian product in states and transitions, and in case of infinite state does not terminate.

We present only some of the constraints here in Figure 2.9 in order to save space.

2.6.3.2 Symbolic automata

Symbolic automata for CCSL or extended finite state machines as proposed in [MdS15] are automata that hide or compress previously defined infinite state automata by the usage of integer variables.

In order to hope keeping the variables bounded, the variables only record the difference between the specific clocks. We call these variables delta-counters.

Definition 2.6.12 (Delta-counter)**.** Delta-counter is a difference between number of ticks of two clocks a and b at step of schedule i:

$$\delta(a, b, i) \stackrel{\text{def}}{=} H_\sigma(a, i) - H_\sigma(b, i)$$

We hide the i when we use delta-counter as a variable, as they are in actual state for the current step.

Given the semantics of the variables, we define the automaton.

Definition 2.6.13 (Symbolic automaton)**.** Symbolic automaton for CCSL is defined as a tuple $\langle L, l_0, C, V, T \rangle$, where

- L is a finite set of locations;
- $l_0 \in L$ is the initial location;
- C is a set of clock symbols;
- $V \subseteq C \times C$ is a set of delta counters;

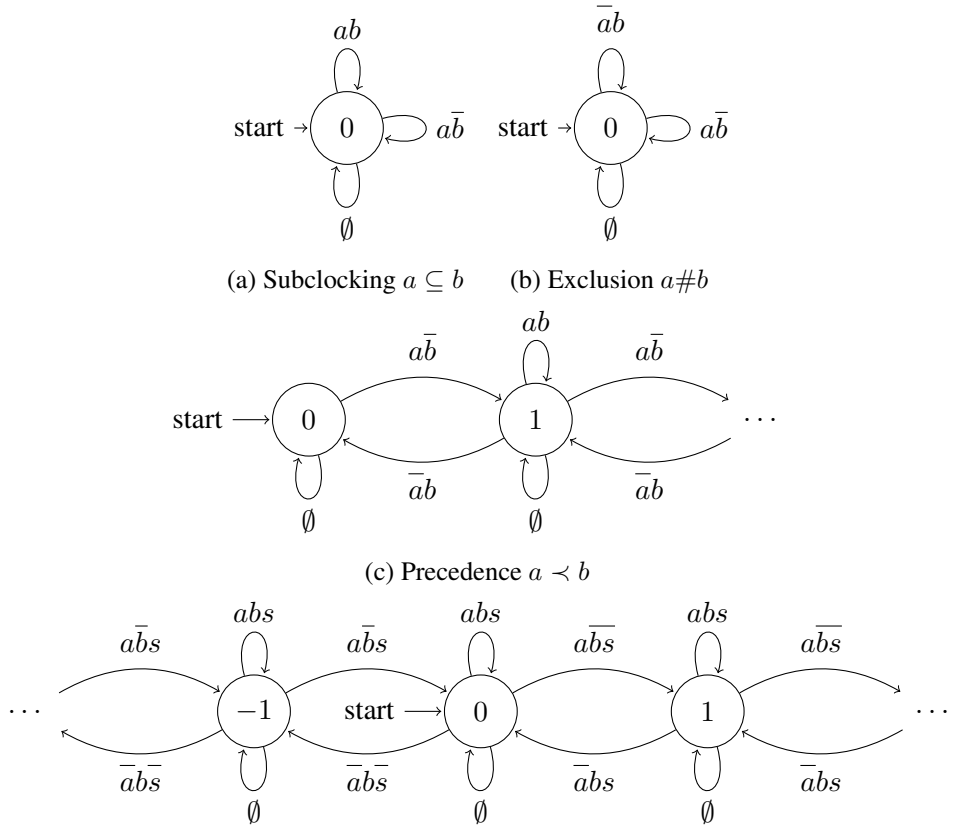


Figure 2.9: Labeled transition system (finite and infinite)

- $T \in L \times S \times \mathcal{P}(C) \to L$ transition function, given an evaluation $S \in V \to \mathbb{Z}$ of the delta-counters V.

In the automata, the explicit state consists of a location and variable values, but when defining actual constraints' automata, we use guards which in turn covering whole subsets of the state in specific locations. It is not hard to see there that these guards are mutually exclusive, meaning that the determinism required by the transition function is respected.

One can notice that the transition function does not provide next state assignment, and it is done so to keep the delta-counters updated the same way. The update is provided in the following definition of the automata execution.

Definition 2.6.14 (Symbolic automaton execution)**.** A valid trace in automaton $A = \langle L, l_0, C, V, T \rangle$ is a potentially infinite sequence $t = (a_i \subseteq C)_{i=1}^{\leq \infty}$ with the supporting sequence of transitions $(l_i, s_i) \xrightarrow{a_i} (T(l_i, s_i, a_i), s_{i+1})$, where:

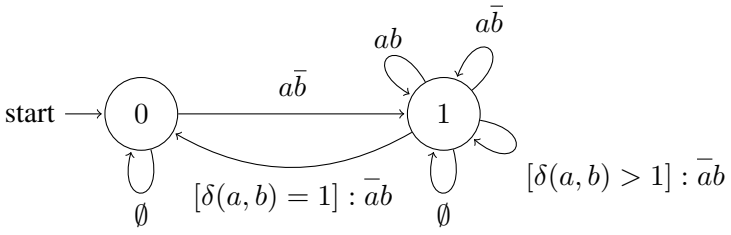
- $s_0 = \lambda\_ \to 0$, i.e. every variable starts at zero in the beginning of each trace;
- $s_{i+1} = \bigcup_{(c_1,c_2)\in V} \lambda\delta \to s_i(\delta) + (c_1 \in a_i) - (c_2 \in a_i)$, where $c_x \in a_i$ is a check that the clock ticked at that transition and implicitly is converted to 1 if true and 0 if false.

In contrast to denotational semantics, where restrictions on behaviour for the whole specification are expressed using essentially a conjunction of the formulas, in automata we have to perform an explicit synchronization as defined in [Arn94].

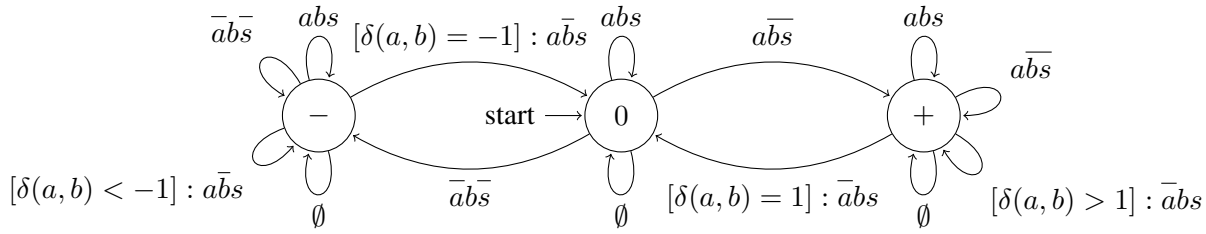
Definition 2.6.15 (Synchronization)**.** Given A_1, A_2 symbolic automata, their synchronized automaton is $A = \langle L, l_0, C, V, T \rangle$, where:

- $L \stackrel{\text{def}}{=} L_1 \times L_2$;
- $l_0 \stackrel{\text{def}}{=} (l_{0,1}, l_{0,2})$;
- $C \stackrel{\text{def}}{=} C_1 \cup C_2$;
- $V \stackrel{\text{def}}{=} V_1 \cup V_2$;
- $T \stackrel{\text{def}}{=} E_{C_2}(T_1) \cap E_{C_1}(T_2)$, where E_C extends transition domain to $T \subseteq L \times S \times A \times L$ by cartesian product, with important detail that because state S and actions A may share the definition domain (the whole point of having same clocks in different constraints) they are multiplied *only* with actions and variables not defined sorely by their respective clocks. The resulting relation is not minimal though, there could be states and so transitions that can never be reached from the initial.

The constraints in symbolic automata are defined on Figure 2.10. Same as with unbounded automata, we only present a subset of the constraints.



(a) Precedence $a \prec b$



(b) Slowest $s \coloneqq \texttt{slowest}(a, b)$

Figure 2.10: Extended state machines

2.6.4 Operational semantics

Operational small-step semantics [ZDM18] is a rule-based definition of CCSL interpretation.

Clock system $\langle X, \Phi \rangle$, where X is a configuration and Φ is a set of constraints. $X : C_\Phi \to \mathbb{N}$, $X(c)$ is a number of ticks for clock c. Then $F \subseteq C_\Phi, F \neq \emptyset$. If F satisfies the set of constraints then transition $\langle X, \Phi \rangle \xrightarrow{F} \langle X', \Phi \rangle$ is performed, where

$$\forall c \in C_\Phi, X'(c) = \begin{cases} X(c) + 1 & \text{if } c \in F \\ X(c) & \text{otherwise} \end{cases}$$

Each of the constraints defines inference rules that change the state depending on what clock ticks at the current step. If no rule is enabled then such behaviour is not allowed. In a specification, individual rules are synchronized by making an inference step in all constraints at once, which is only possible if the current configuration and set of ticking clocks satisfy the rules' preconditions. Here we show only the rules of some constraints, refer to the original work for all the definitions.

$$\frac{X(c_1) = X(c_2) \implies c_2 \notin F}{\langle X, c_1 \prec c_2 \rangle \xrightarrow{F} \langle X', c_1 \prec c_2 \rangle} \text{ precedence}$$

$$\frac{X(c_2) \geq d \qquad c_1 \in F \iff c_2 \in F}{\langle X, c_1 = c_2 \ \$ \ d \rangle \xrightarrow{F} \langle X', c_1 = c_2 \ \$ \ d \rangle} \text{ delay-I} \qquad \frac{X(c_2) < d \qquad c_1 \notin F}{\langle X, c_1 = c_2 \ \$ \ d \rangle \xrightarrow{F} \langle X', c_1 = c_2 \ \$ \ d \rangle} \text{ delay-II}$$

2.6.5 Refinement

In CCSL the refinement is usually expressed as iterative addition of constraints to the specification. As constraints are sets of possible behaviour and putting them into one specification makes an intersection of the individual behaviours, adding more constraints reduces it further. Then the new specification is checked for the same desired properties as the previous one. If some properties fail to be checked then it is clear which constraints may be responsible for it. While certainly an approach to go from an abstract to more concrete description, using this method may unnecessary reduce the set of solutions without triggering non satisfaction of its properties.

Another notion of refinement, an instant refinement, first proposed in [MP18a] and then in [MP21]. It features the idea that a refinement is a relation between the instants of the clocks of different specifications, abstract and concrete respectively. An intuition for the relation is the following: a set of events from one point of view, may contribute to the occurrence of a singular event from another point of view. While computation can be involved and require a complicated coordination, its result (or absence) can be observed regardless of these low-level details. Another example is provided on Figure 2.11 (taken from [MP21]): driving nails into a plank is "refined" by striking the nail with a hammer. Then depending on what refinement that is, 1–1 or 1–N, constraints set in the abstract level are proven to imply constraints in the concrete level, and vice-versa.

2.6.6 Properties of interest

CCSL features several properties that a designer may want to check to hold on a specification. These properties either involve the set of schedules that a specification defines (scheduling, liveness, deadlock) or internal representation (finiteness).

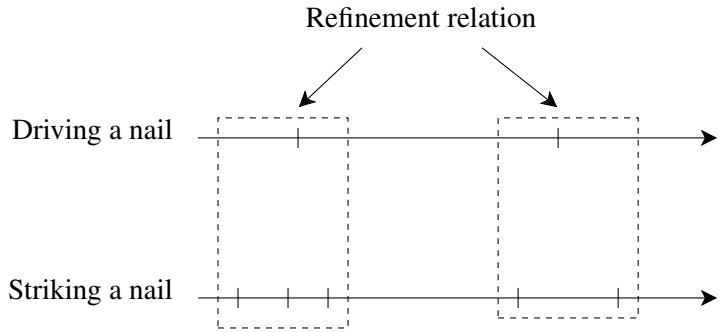


Figure 2.11: An example of the 1–N refinement

2.6.6.1 Scheduling problem

Existence of schedule, or scheduling problem, is the most basic property to hold. If a specification does not have *any* valid schedule, it means that at least some of the constraints contradict each other. In design phase, failure to schedule should be considered a bug and be fixed before moving to the implementation.

Definition 2.6.16 (Scheduling problem)**.** Given a specification Φ, the scheduling problem of CCSL is to compute whether the language $\Sigma(\Phi)/\sim$ is empty, where $\sim$ is an equivalence relation that compares schedules modulo $\emptyset$ meaning that the equivalence class is effectively $\emptyset$-less.

In CCSL, "nothing happened" or $\sigma(i) = \emptyset$ is always a valid step in any schedule for any specification, so we are only interested in schedules without such steps, otherwise scheduling problem becomes trivial. But we keep it in the definition for composability. As some of the constraints have to stutter to allow others to contribute to the common schedule.

2.6.6.2 Existence of periodic schedules

A periodic schedule is a word $a(b^*)$, where a and b are arbitrary but finite schedules. The prefix word a is an initialization sequence and the word b is a reaction loop. Such schedule is memory efficient and deterministic, and so can be used as a basis for the implementation.

2.6.6.3 Deadlock

Deadlock is a condition at which the system cannot progress further in one its functionality or at all. Such condition is obviously should be avoided in a reactive system. In case of deadlock free specification, it means there are no finite schedules that are not prefixes of some infinite schedule.

$$\Sigma(\Phi) \subseteq \bigcup_{w}^{L^\omega(\mathcal{P}(C))} \{w, \text{prefix}_{\leq 1}(w), \text{prefix}_{\leq 2}(w), \dots\}$$

where $\text{prefix}_{\leq n}(w)$ cuts the infinite word to length $n \in \mathbb{N}$.

2.6.6.4 Liveness

Another classic property is liveness. Liveness property for CCSL specifications requires all clocks to be live, i.e. to tick infinitely often in *each* valid schedule of the specification. It may seem that it is equivalent to the inverse of the deadlock problem, but in this case, only one clock has to be live in a schedule and this clock is not preselected, it is found.

Definition 2.6.17 (Liveness)**.** Schedule is live if every clock ticks infinitely often in every schedule.

$$\forall \sigma \in \Sigma(\Phi), c \in C : \exists f \in \mathbb{N} \to \mathbb{N} : \forall i \in \mathbb{N} : c \in \sigma(f(i))$$

where f is total and strictly increasing.

For some systems it is too strong of a requirement, it could be only enough to check that the specific subset of clocks is live. In this case, the liveness becomes something in between nonexistence of deadlocks and full liveness.

In worth noting that he property is almost exactly the same as in LTL, named recurrence and written as $\bigwedge_{c \in C} \square\Diamond c_i$ticks, but with the additional requirement that the path has to be infinite.

2.6.6.5 Finiteness of representation/Safety

Finiteness or safety is a property of the representation itself. We often require it in order to do analysis in tools that use finite methods or if we intend to use the specification as part of the implementation. As with liveness, it is possible to select only the part, where it is important to be finite. For example, if we model reactive system with its environment, finiteness of the environment constraints is not important for the implementation, and even should be avoided if cannot be realistically guaranteed.

It is easily expressed in automata terms.

Definition 2.6.18 (Finiteness (automata semantics))**.** Given automaton $A = \langle L, l_0, C, V, T \rangle$, it is safe or finite iff

$$\forall \delta \in V, \exists n \in \mathbb{N}, \forall t \in \Sigma(A), \forall i \in t : |s_i(\delta)| < n$$

where $s_i(\delta)$ is the value of the variable at step i of the trace t, as defined in execution of the symbolic automaton Definition 2.6.14.

Alternatively, we define it using the history-based denotational semantics.

Definition 2.6.19 (Finiteness (denotational semantics))**.** When history is used in a constraint, difference between values of clocks is bounded by a know value:

$$\forall \sigma \in \Sigma(\Phi), \varphi \in \Phi : \exists N \in \mathbb{N} : \forall c_1 \neq c_2 \in C(\Phi) : \forall i \in \mathbb{N} : |H_\sigma(c_1, i) - H_\sigma(c_2, i)| \leq N$$

The only known method, other than trying to unfold the state space, to solve this problem was proposed in [MMS13b] and involves a marked graph. Unfortunately, the result is only sound as CCSL is undecidable.

2.6.7 Tooling

Over the years, CCSL was implemented using an extensive collection of approaches and tools. These include translation into other languages and theories, and then simulation or verification using native tools: VHDL [AMD10], Esterel, Signal and Time Petri Nets [MA08], Timed Automata [Sur+13a]. Simulation and some model checking, specific to CCSL, is implemented in TimeSquare [DM12b], checking finiteness of state using graphs [MMS13b], finding bounded periodic schedules with SMT [Zha+19].

2.7 TESL

Tagged Events Specification Language or TESL [Bou+14] is a declarative language of specifications of reactive systems, derived from CCSL. The constraint language consists of implications between ticks, delays and tag relations. The tag relation can be arbitrary or not exist between the clocks. What we cannot do in a relation is to skip some of the ticks. I.e. the relations are "total", but they not necessarily bijective. The only restriction on the relation, or better say, a pair of conversion functions (d, r) is that d and r are monotonic, $d \circ r \circ d = d$ and $r \circ d \circ r = r$. In other languages such relations would have to be simulated or only approximated, while TESL supports them natively. Each clock is a sequence of instants each having a tag and may be ticking or not. Then the schedule or solution to a specification is a sequence with tag values and ticking for each of the clocks. As for the tooling, TESL has a simulation engine, generated from mechanized operational semantics, written in Isabelle/HOL [Ngu18].

2.8 Exact methods of analysis

In this section we discuss some of the generic methods of exact analysis. The word exact here means that the analysis performed does not allow any losses of precision. It is regardless if it terminates or not (like SMT). More specifically, we are presenting classic model checking and its symbolic version using Binary Decision Diagrams (BDD), a really important data structure used extensively in the tools and in this work to represent Boolean formula. Another symbolic approach is Satisfiability Modulo Theories (SMT) that combines solvers in specific domains by using propositional logic and algorithms of SAT in a modular way.

2.8.1 Model checking

Model checking consists of finding out whatever a system under development (the model) satisfies the property (the specification). This method is fully automatic or heavily machine-assisted in order to reduce errors in case of critical systems, speed as algorithms can be scaled with the technology development in computing or reproducibility, important for troubleshooting and certification.

The basic assumption of the method is that the state space is finite. It is a reasonable assumption, considering that the hardware and by extension software are finite. And in a lot of cases we can and do pretend that we have as much memory as we need, but in case of critical systems it is too risky. With this we can use the explicit state space and check the specification on it.

But only because the system is finite as so analysis theoretically terminates, does not mean we would not want it to finish faster or to have more realistic requirements to processing units on

which the analysis runs. For this, symbolic model checking was developed. One of the data structures to make it possible is Binary Decision Diagram (BDD, Section 2.8.3). Any type computer can represent is possible to encode in BDDs and so is possible to symbolically represent. Then the contribution of the method is that the transition function is also a BDD. Then by applying one to another, and because the state is finite, in finite amount of applications, one can obtain the reachable state space, the requirement to check safety properties.

Bounded model-checking operates on infinite state spaces by bounding the reaction to a number of steps in the future.

2.8.2 SMT

Satisfiability Modulo Theories is an approach that allows to prove that a formula containing atoms in non-Boolean theories has solutions, while staying modular. Thus *modulo* theories: the only requirement to the theory solver is to be able to answer if a proposition is satisfiable or not, the coordination and exploration of the solutions is done by the SMT algorithm itself, via clause learning or other approaches. Its general principle is demonstrated on Figure 2.12. The known tools based on this principle are (not exhaustive) Z3, CVC4 and CVC5, PONO and SMT-LIB.

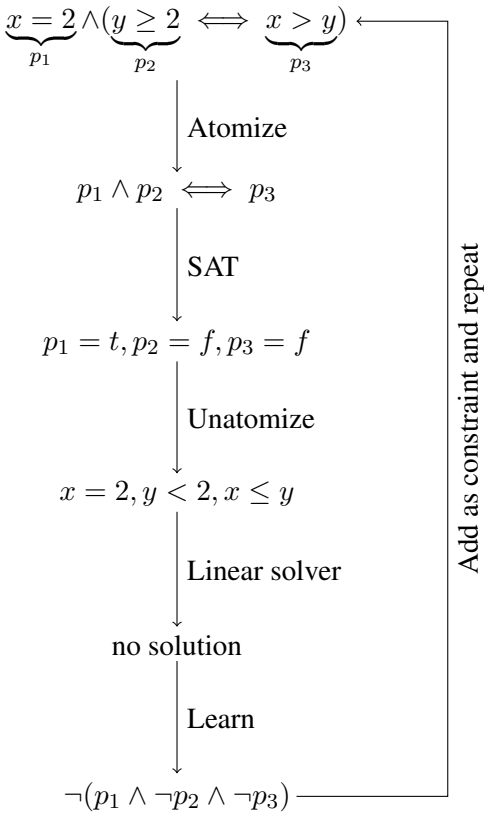


Figure 2.12: Principle of SMT

2.8.3 Binary Decision Diagrams

Binary decision diagram is a rooted, directed, acyclic graph which represents a Boolean formula. In other words, it is a data structure for relations.

In this graph, vertices are variables and edges are their assignments. There are two special vertices, called terminals, representing 0 (false) and 1 (true). If an edge reaches one of them, it means that any path from the root to that terminal is true (or false, respectively). There are two terminals, despite the fact that if it is known when formula is true it is known when it is false, because depending on the formula, it could be more beneficial for the size of the graph to represent the formula as true or false paths.

BDDs are constructed iteratively from the basic operations. If we added the vertices and edges to the graph, it is not beneficial as a data structure, as it would be a truth table. For that a special rule is applied after any operation. The most used variant of BDD is Reduced Ordered Binary Decision Diagram [Bry86]. It makes two changes: fixes order in which variables occur in the paths to terminals, and applies special reduction rules on the graph. The result is a canonical and compact representation of a Boolean formula. This structure, in most of the cases, is able to *not* grow exponentially in number of variables. And its canonical form allows to check equivalence of the formulas as equivalence of trees.

2.8.3.1 MTBDD

Multi-terminal Binary Decision Diagrams [FM97] refer to a modification to BDDs that instead of binary terminals, 0 and 1, have some other finite number of terminals. It may also use another domain for variables, like $I_n = \mathbb{N}_{\leq n}$, making it possible to represent the functions of form $I_n^m \to I_k$ instead of original $\mathbb{B}^n \to \mathbb{B}$. The original intention for developing the structure was to use it for sparse matrices and the authors claim that it is efficient in this regard. It was later used in tool NBac to encode more efficiently logico-numerical partitioning (more in Section 2.9.5).

2.9 Abstract interpretation

Abstract interpretation is a theory of program abstraction and uses approximations in the form of abstract domains to achieve said abstraction from the formal semantics of the studied language. It allows us to perform checks of properties that otherwise would be undecidable. In short, abstract interpretation trades accuracy of this analysis with speed and decidability, while still maintaining its soundness with respect to the property.

In this section, we first present the notion of approximation, following by the definitions and the theoretical foundation of abstract interpretation, with all its assumptions. We then describe some of the abstract domains available and what is expected of a domain. This is important, as this way abstract interpretation becomes really flexible with the analysis, as it is then possible to substitute one domain with another, even on the fly. We follow with more operational explanation of the method at the end, when we provide the descriptions of the tools of abstract interpretation and descriptions of their specific features.

2.9.1 Approximations

In model checking or in solving of problems, we always prefer precise and fast results. Unfortunately, it is not always possible, either because the problem is undecidable or complexity of the best algorithm is still too large. In this case, a compromise between the precision and speed (or termination at all) is needed. But even by doing this, we still should be sure that the results are correct with respect to the properties asked. This led to a definition of sound approximations.

Definition 2.9.1 (Property checking)**.** Given a program and its reachable state space S and a safety property reaching states P, a program satisfies the property if and only if $S \subseteq P$.

Definition 2.9.2 (Sound overapproximation)**.** $A^{\uparrow}$ is sound overapproximation of a set A when $A \subseteq A^{\uparrow}$.

Then, the best case, given program's reachable state space S and property state space P, is when sound overapproximation $S^{\uparrow}$ of S exists and the P itself can be exactly represented. Then if $S^{\uparrow} \subseteq P$ holds true, $S \subseteq P$ by definition of sound overapproximation is true and so means that the program satisfies the property. At the same time, if $S^{\uparrow} \not\subseteq P$, that does not mean that the property is not satisfied, as we have reached the limit of the available overapproximation. For the cases when property is not satisfied or when better precision for this state is not available, the answer is the same and it is "do not know".

To improve on this, we need to introduce a sound *under*approximation.

Definition 2.9.3 (Sound underapproximation)**.** $A^{\downarrow}$ is sound underapproximation of A when $A^{\downarrow} \subseteq A$.

With sound underapproximation, we can narrow the cases when we do not know what the result of the analysis is, i.e. the false alarms, by checking $S^{\downarrow} \subseteq P^{\uparrow}$. If this inclusion is false, the property is surely *not* satisfied.

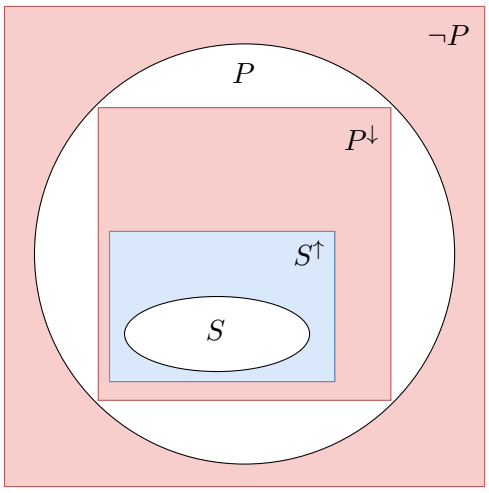


Figure 2.13: Property check by using overapproximations

Theorem 2.9.1 (On proving properties with approximations)**.** *By checking if overapproximation of the program is inside an underapproximation of the property, $S^{\uparrow} \subseteq P^{\downarrow}$, we can prove that the property $S \subseteq P$ holds.*

Proof. The proof is trivial because $S \subseteq S^{\uparrow}, P^{\downarrow} \subseteq P$ (visually on Figure 2.13). □

Alternatively, if only overapproximation is available to approximate reachable state space in both cases, inverting the property, overapproximating and inverting it again, is a form of underapproximation of the original property, and can be used as above.

Theorem 2.9.2. *$\neg((\neg P)^{\uparrow})$ is an underapproximation of P, i.e. $\neg((\neg P)^{\uparrow}) \subseteq P$.*

Proof.

$(\neg P)^{\uparrow}$	$\vdash$by definition
$\neg P \subseteq (\neg P)^{\uparrow}$	$\vdash$axiom of excl.middle on $(\neg P)^{\uparrow}$
$\neg P \not\subseteq \neg((\neg P)^{\uparrow})$	$\vdash$axiom of excl.middle on $(\neg P)$
$\neg\neg P \supseteq \neg((\neg P)^{\uparrow})$	$\vdash$double negation elimination
$\neg((\neg P)^{\uparrow}) \subseteq P$	□

The axiom of excluded middle and double negation are reasonable assumptions for simple types and logic used in abstract interpretation later.

2.9.2 Collecting semantics

Collecting semantics is a version of the semantics of the analysed language. While abstract interpretation is language agnostic, depending on specific properties to check, one interpretation of the language semantics is more efficient than another. As such, the collecting semantics can be reachable state-based or path/trace-based [Sch98; MR05].

Mostly used collecting semantics is the reachable state-based and it describes an overapproximation of the reachable state space in specific program location, starting in the initial one. The trace collecting semantics on another hand relates traces that can reach a location or can be produced from the location. The properties then expressed in terms of this reachable state or path space.

2.9.3 Theory of abstract interpretation

Here we present our summary of the theory of abstract interpretation. We heavily rely on explanations given in [Min17].

We start with the definitions of lattices and complete lattices. Abstract and concrete domains rely on these notions in state collecting semantics for correct collection of state though joins (unions).

Definition 2.9.4 (Lattice)**.** A lattice $(X, \sqsubseteq, \sqcup, \sqcap)$ is a partially ordered set such that $\forall a, b \in X$: $a \sqcup b$(greatest lower bound) and $a \sqcap b$ exist (least upper bound).

Definition 2.9.5 (Complete lattice)**.** A complete lattice $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a partially ordered set such that:

1. a join (union) of all elements in a subset exists $\forall A \subseteq X : \bigsqcup A \in X$;
2. a meet (intersection) of all elements in a subset exists $\forall A \subseteq X : \bigsqcap A \in X$;

3. then the smallest and the biggest elements are $\bot = \bigsqcap X, \top = \bigsqcup X$.

Alternatively, a complete lattice is a lattice in which *every* subset has greatest and lowest element.

Important note on lattices: it may be tempting to say that given a lattice, it should be possible to construct the biggest or smallest element. The tricky detail is that it would be only true for finite subsets. For example, the set of integers $\mathbb{Z}$: it is a lattice, but not complete, as there is no biggest element. Another example would be rationals $\mathbb{Q}$: they do have sometimes biggest and smallest elements for even an infinite subset (the series $\sum_{i=1} \frac{1}{2^i}$), but again, not for the whole set, so not every subset. But, a power set $\mathcal{P}(A)$ of any set A is a complete lattice if ordered by inclusion.

If we represent our program as a function $f : X \to X$ that given initial state set, gives the set of next states. Then we assume that the state is finite, after some number of iterations we would obtain the reachable state space, on which we could check a desired property. In other words, to find the final set, we need to find a fixpoint, i.e. a point in execution when our reachable space stabilizes/no longer grows.

Definition 2.9.6 (Fixpoint)**.** Fixpoint is any value x that satisfies $f(x) = x$ for any operator (1-ary function on the same domain) $f : X \to X$.

The set of fixpoints for partially ordered set is $\mathrm{fp}(f) \stackrel{\text{def}}{=} \{x \in X \mid f(x) = x\}$.

It is even better if this set is the smallest (least) fixpoint possible, as then such analysis does not include any useless states, which would make it less precise. Such set is possibly infinite or really big, so doing it by recording states explicitly would just repeat what model checking is doing already (see Section 2.8.1).

Definition 2.9.7 (Least fixpoint)**.** Given function $f : X \to X$, for partially ordered set $(X, \sqsubseteq)$, least fixpoint that contains x is defined as

$$\operatorname*{lfp}_{x}(f) \stackrel{\text{def}}{=} \min\{y \mid y \in \mathrm{fp}(f) : x \sqsubseteq y\}$$

Least fixpoint of a function is a least fixpoint to contain bottom element $\bot$, defined as $\mathrm{lfp}(f) = \mathrm{lfp}_{\bot}(f)$.

Theorem 2.9.3 (Tarski's fixpoint theorem)**.** *If $f \in X \to X$ is a monotonic operator in a complete lattice $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, then the set of fixpoints $\mathrm{fp}(f)$ is a non-empty complete lattice. In particular, $\mathrm{lfp}(f)$ exists and $\mathrm{lfp}(f) = \sqcap\{x \in X \mid f(x) \sqsubseteq x\}$.*

Theorem 2.9.4 (Kleene's fixpoint theorem)**.** *If $f \in X \to X$ is a continuous operator in a complete partial order $(X, \sqsubseteq, \sqcup, \top)$, then $\mathrm{lfp}(f)$ exists and $\mathrm{lfp}(f) = \sqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$.*

Practically, Theorem 2.9.3 and Theorem 2.9.4 mean that given the program f, a set that includes its state exists. Depending on the conditions, monotonicity (weak) vs continuity (strong), complete lattice (strong) vs complete partial order (weak), one may be preferential to other. The second, Kleene's theorem is more constructive as it features the starting point and the exact sequence to perform to obtain the least fixpoint. The problem though is that the sequence not necessarily converges in finite time. So convergence accelerating techniques are required, namely widening. While widening is a function with specific properties which are always the same, each domain has to define its own, because the widening depends on the relations that can be represented by the domain and the implementation details.

Next we introduce the relation between concrete and abstract domains. It is important to reference the concrete domain, as the semantics of the analysed language is defined in terms of the concrete values and the properties to be verified are defined on the concrete values too. Job of the abstract interpretation is to transform the operations and properties into their faster but still sound analogues.

Definition 2.9.8 (Minimal abstract-concrete structure)**.** Minimal structure for abstract interpretation consists of a concrete partially ordered set $(C, \leq)$ and abstract partially ordered set $(A, \sqsubseteq)$ with concretization function $\gamma \in A \to C$. γ is total and monotonic with respect to $\leq$ and $\sqsubseteq$ and is supposed to interpret abstract domain in terms of concrete domain.

Example of such minimal structure is $(C, \leq) = (\mathcal{P}(\mathbb{Z}), \subseteq)$ and $(A, \sqsubseteq)$ being interval domain (defined later in Section 2.9.4). Then an interval $[a, b]$ is interpreted as a set of every value between the bounds $\gamma([a, b]) = \{x \mid \forall a \leq x \leq b\}$.

Definition 2.9.9 (Sound operator abstraction)**.** $g : A \to A$ is a sounds abstraction of f if $\forall a \in A : f(\gamma(a)) \leq \gamma(g(a))$, for a concretization γ of an abstract domain $(A, \sqsubseteq)$ to a concrete domain $(C, \leq)$ and a concrete operator $f : C \to C$.

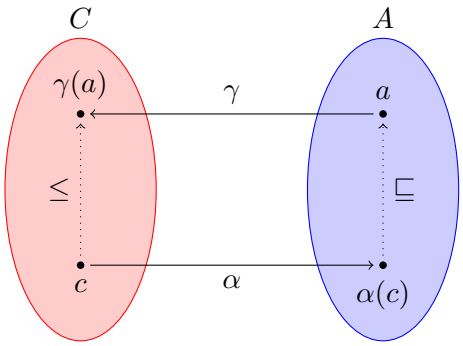


Figure 2.14: Galois connection

An important property for an abstract domain is to have a Galois connection(Figure 2.14) with the concrete domain. It is based on the minimal abstract-concrete structure and provides the best abstraction.

Definition 2.9.10 (Galois connection)**.** Given two partially ordered sets $(C, \leq)$ and $(A, \sqsubseteq)$, the pair $(\alpha : C \to A, \gamma : A \to C)$ is a Galois connection when:

$$\forall a \in A, c \in C : c \leq \gamma(a) \iff \alpha(c) \sqsubseteq a$$

which is denoted as $(C, \leq) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (A, \sqsubseteq)$. α and γ are said to be adjoint functions, where the abstracting α is the upper adjoint and the concretizing γ is the lower adjoint.

Some of the properties of Galois connection:

- $\gamma \circ \alpha \circ \gamma = \gamma$ and $\alpha \circ \gamma \circ \alpha = \alpha$;
- $\alpha \circ \gamma$ and $\gamma \circ \alpha$ are idempotent;

- $\forall c \in C : \alpha(c) = \sqcap\{a \mid c \leq \gamma(a)\}$;
- $\forall a \in A : \gamma(a) = \vee\{c \mid \alpha(c) \sqsubseteq a\}$
- α maps concrete lower upper bounds to abstract lower upper bounds: $\forall X \subseteq C$: if $\vee X$ exists, then $\alpha(\vee X) = \sqcup\{\alpha(x) \mid x \in X\}$;
- γ maps abstract greatest lower bounds to concrete greatest lower bounds: $\forall X \subseteq A$: if $\sqcap X$ exists, then $\gamma(\sqcap X) = \wedge\{\gamma(x) \mid x \in X\}$;

But, Galois connection does not exist for all abstract domains. And in simpler words, absence of Galois connection means that there is no *function* that can take a set of values in concrete domain and make the best *single* abstraction. There are possibly several or infinite variants that can represent the set though, just not exactly one. If best abstraction does not always exist, the best abstraction of some operators does not exist, meaning they may lose precision.

The meaning of widening is essentially extrapolation, a pessimistic guess of what the trend is, given several observed points. It is used to help converge to the fixpoint. First of all because it is an overapproximation, it is sound, but also we prefer terminating analysis over sometimes more precise not always terminating.

Definition 2.9.11 (Widening operator)**.** A binary operator $\triangledown : A \times A \to A$ is a widening operator in an abstract domain $(A, \sqsubseteq)$ when:

- it computes upper bounds: $\forall x, y \in A : x \sqsubseteq x \triangledown y \wedge y \sqsubseteq x \triangledown y$;
- and it enforces convergence: for any sequence $(y^i)_{i\in\mathbb{N}}$ in A, the sequence $(x^i)_{i\in\mathbb{N}}$ computed as $x^0 = y^0, x^{i+1} = x^i \triangledown y^{i+1}$ stabilizes in finite time: $\exists k \geq 0 : x^{k+1} = x^k$.

Theorem 2.9.5 (Convergence with widening)**.** *If f is a monotonic operator in a concrete complete lattice and g is a sound abstraction of f, then the following iteration:*

$$x^0 = \bot$$
$$x^{i+1} = x^i \triangledown g(x^i)$$

converges in finite time, and its limit x is a sound abstraction of the least fixpoint $\mathrm{lfp}(f) : \mathrm{lfp}(f) \leq \gamma(x)$.

There is also an operator used to make the analysis more precise after the widening. It is possible because the result of the converging sequence contains the least fixpoint, but not necessary is it exactly, i.e. a *post*fixpoint. Thus it is possible to approach it from the approximated side. But in domains that feature infinite decreasing chains, such iteration may not converge, thus requiring a special operator, called narrowing. The meet $\sqcap$ is the sound default for narrowing, but may not always terminate, same as with widening and $\sqcup$.

Definition 2.9.12 (Narrowing operator)**.** A binary operator $\triangle$: $A \times A \to A$ is a narrowing operator in an abstract domain A when:

- $\forall x, y \in A : (x \sqcap y) \sqsubseteq (x \triangle y) \sqsubseteq x$;
- for any sequence $(y^i)_{i\in\mathbb{N}}$ in A, the sequence $(x^i)_{i\in\mathbb{N}}$ computed as $x^0 = y^0, x^{i+1} = x^i \triangle y^{i+1}$ stabilizes in finite time: $\exists k \geq 0 : x^{k+1} = x^k$.

The intuition for narrowing is the following: the loop heads are the target for widening and the place where the bounds are most relaxed. On one hand it makes discovery of all possible loop behaviour faster, on another hand, if the loop body and loop initialization define well bounded domains, the widening will result in too pessimistic approximation. Iterating more will not improve precision, as it is additive. Thus, with narrowing we can surgically revert the over relaxed bounds, making the analysis more precise while still maintaining soundness. An illustration of a process of finding a fixpoint and roles of widening and narrowing in it is shown on Figure 2.15.

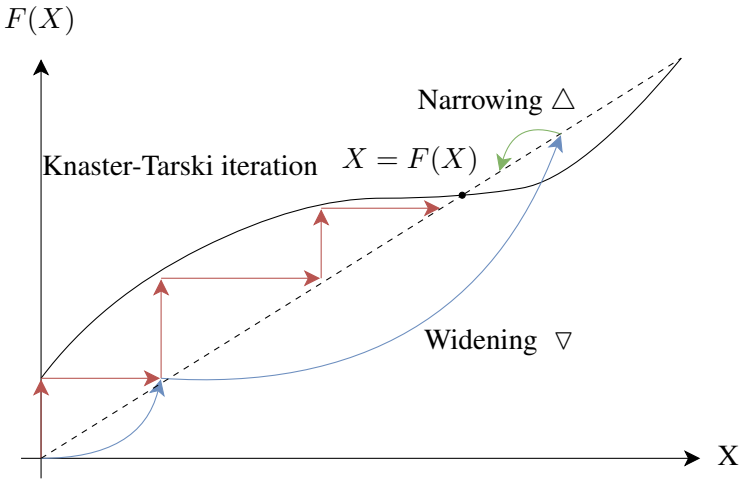


Figure 2.15: Demonstration of acceleration with widening and narrowing

2.9.4 Domains

Abstract interpretation features a lot of domains, with different level of precision. Most notably, it has several domains to express numerical spaces, which we list from the least to the most expressive:

- *sign domain* splits numbers into positive, negative, zero, every and no number;
- *interval domain* expresses continuous subsets of a numerical set;
- *octagon domain* defines a set that conforms to $\pm V_j \pm V_i \leq c$;
- *zone domain* refines zone domain with relations of form $\bigwedge_{i,j} V_j - V_i \leq m_{ij}$;
- *congruence domain* encodes the set of solutions to some $a \equiv x \mod b$;
- *affine equalities* and *inequalities* (also known as polyhedra) domains.

There are domains for other types of course, like lattice automata for queues.

Next we introduce the concept of a domain transformer. Domain transformer is a function that accepts one or several domains and returns a new one. The reasons to use one may vary: either it is to combine various domains due to the program being defined on different types, or to obtain a domain with better characteristic. The domain transformers include the non, partially and fully reduced product domain and disjunctive completion. The concept of the product domain

is to use several different domains for the same variables and so do the analysis on both at the same time. So while we end up with more computations, if we use reduced product, the resulting domain is more expressive due to the exchange of information between the domains it consists of. As for disjunctive completion, it is a domain that simply saves every argument when join is performed. This allows the abstract element to represent non-convex sets, which is not possible in the numerical domains. A special case of disjunctive completion is a partition, which we describe in Section 2.9.5.

The domains can be additionally characterized with how relational they are. The distinction is dictated by the fact that in some domains, values of variables can be interdependent or not. For example, polyhedra domain is relational, because $a + b \leq 1$ is expressible in it, so depending on what a is, b can be different. While in interval domain $a = [1, 2], b = [3, 4]$ choice of a does not depend on choice of b.

For a domain to be a valid structure it has at least be a lattice with correct concretization function, and preferentially have a Galois connection. Basic operations defined on domains are:

1. order $\sqsubseteq$, implies subset relation;
2. meet (intersection) $\sqcap$;
3. join (union) $\sqcup$;
4. Π_V projection to variables V; obviously only if domain constraints variables;
5. $e[v_1 \to v_2]$ renaming variable v_1 into v_2;
6. widening $\triangledown$ and narrowing $\triangle$.

Then, depending on the nature of the domain, one would also define its other operators. For numerical domains, these would be arithmetic operations $(+, -, *, /)$, for queues push, pop, head, etc. Also, a domain should be able to be constructed from its native relations, $a \leq x \leq b$ for relational domains, or simply values, $x = [a, b]$ for interval domain. This way the relational domains may encode transitions itself, and by using combination of renaming, intersection and projection, one implements a symbolic reachability algorithm. More about that in description of NBac Section 2.9.6.1.

Interval domain Interval domain is one of the simplest domains. It is non-relational as it only defines an interval per variable. As an object, the element of the domain is a pair of bounds, which are either ordered numbers or infinity, or nothing. Technically, every incorrectly ordered pair could represent nothing, but then the representation is not canonical.

$$\mathbb{I} \stackrel{\text{def}}{=} \{[a, b] \mid a \in \mathbb{N} \cup \{-\infty\}, b \in \mathbb{N} \cup \{+\infty\}, a \leq b\} \cup \{\bot\}$$

Lattice structure of interval domain is defined as:

- $\top \stackrel{\text{def}}{=} [-\infty, +\infty]$;
- $[a, b] \sqsubseteq [c, d] \stackrel{\text{def}}{=} (c \leq a) \wedge (b \leq d)$;
- $[a, b] \sqcup [c, d] \stackrel{\text{def}}{=} [\min(a, c), \max(b, d)]$, is a sound abstraction but not exact;

- $[a,b] \sqcap [c,d] \stackrel{\text{def}}{=} \begin{cases} [\max(a,c), \min(b,d)] & \text{if } \max(a,c) \leq \min(b,d) \\ \bot & \text{otherwise} \end{cases}$ is exact abstraction.

Additionally, the Hasse diagram on Figure 2.16 illustrates the logic of inclusion relation of the domain.

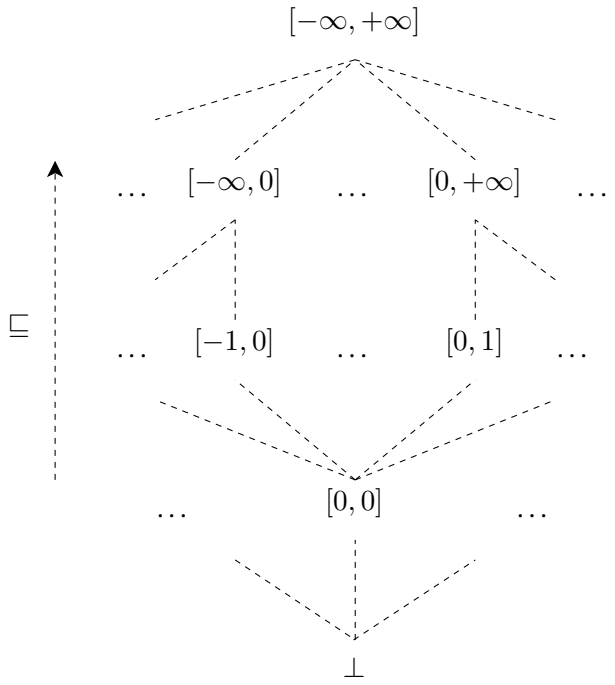


Figure 2.16: Hasse diagram of interval domain

Galois connection is defined as:

- $\gamma(\bot) \stackrel{\text{def}}{=} \emptyset$;
- $\gamma([a,b]) \stackrel{\text{def}}{=} \{x \in \mathbb{I} \mid a \leq x \leq b\}$;
- $\alpha(X) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } X = \emptyset \\ [\min X, \max X] & \text{otherwise} \end{cases}$.

Widening operator is

$$[a,b] \triangledown [c,d] \stackrel{\text{def}}{=} \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right]$$

Narrowing operator is

$$[a,b] \triangle [c,d] \stackrel{\text{def}}{=} \left[\begin{cases} c & \text{if } a = -\infty \\ a & \text{otherwise} \end{cases}, \begin{cases} d & \text{if } b = +\infty \\ b & \text{otherwise} \end{cases} \right]$$

In its definition, the interval domain does not binds by itself the values to any variable symbol. Thus this domain is actually a value domain and to be used in usual way abstract interpretation is performed, should be transformed into a "variable domain". A multi-variable interval domain then is a product of several domains and variables can be accessed with a total function $V \to \mathfrak{I}$.

Zone domain Zone domain [Min01] represents two sets of constraints: $\bigwedge_{i \neq j} V_j - V_i \leq m_{ij}$, called potential constraints, and $\bigwedge_i a_i \leq V_i \leq b_i$, called zone constraints (bounding intervals). The domain can represent any domain of numbers, i.e. $N \in \{\mathbb{I}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. The implementation can be done using either potential graph or difference bound matrix, which are both implementations used in analysis of Timed Automata Section 2.4. The zone constraints can be represented with potential constraints and a special variable V_0 that is always kept as 0. Octagon domain [Min06] is an extension of Zone domain, where instead of V_j and $-V_i$ in the potential constraints, we can write any constant.

The lattice structure of the domain is:

- $D \stackrel{\text{def}}{=} (N \cup \{+\infty\})^{(n+1)\times(n+1)} \cup \{\bot\}$;
- $\forall i, j : [\top]_{ij} \stackrel{\text{def}}{=} +\infty$;
- $m \sqsubseteq n \stackrel{\text{def}}{=} \forall i, j : m_{ij} \leq n_{ij}$;
- $\forall i, j : [m \sqcup n]_{ij} \stackrel{\text{def}}{=} \max(m_{ij}, n_{ij})$;
- $\forall i, j : [m \sqcap n]_{ij} \stackrel{\text{def}}{=} \min(m_{ij}, n_{ij})$.

Galois connection of zone domain exists only for integers and reals, as for some rationals there is no rational maximum. Then concretization function is $\gamma(m) \stackrel{\text{def}}{=} \{(v_1, \ldots, v_n) \in \mathbb{P} \mid \forall i, j \in [0, n] : v_j - v_i \leq m_{ij} \wedge v_0 = 0\}$ and abstraction function $\alpha(a) \stackrel{\text{def}}{=} \max\{v_j - v_i \mid (v_1, \ldots, v_n) \in a \wedge v_0 = 0\}$.

Zone domain features a normal form (see Figure 2.17), and is defined as:

$$\forall i \neq j : m^*_{ij} \stackrel{\text{def}}{=} \min_{\forall N : \langle i = i_1, \ldots, i_N = j \rangle} \sum_{k=1}^{N-1} m_{i_k i_{k+1}}$$

$$\forall i : m^*_{ii} \stackrel{\text{def}}{=} 0$$

From a logical point of view, this operation is a saturation. From a graph point of view, we construct the shortest-path closure. Such closure can be computed using the Floyd-Warshall algorithm [Flo62].

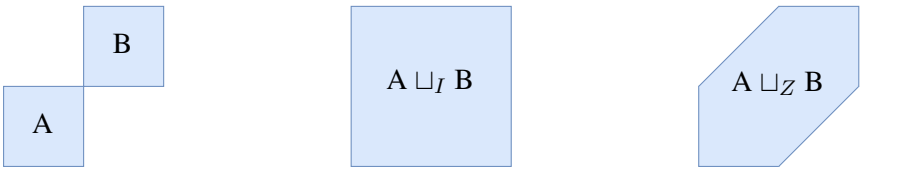
The operators include:

- union: $m \cup n \stackrel{\text{def}}{=} m^* \sqcup n^*$, where n^* means normalized;
- intersection: $m \cap n \stackrel{\text{def}}{=} m \sqcap n$;
- widening:

$$\forall i, j : [m \triangledown n]_{ij} \stackrel{\text{def}}{=} \begin{cases} m_{ij} & \text{if } n_{ij} \leq m_{ij} \\ +\infty & \text{otherwise} \end{cases}$$

- narrowing:

$$\forall i, j : [m \triangle n]_{ij} \stackrel{\text{def}}{=} \begin{cases} n_{ij} & \text{if } m_{ij} = +\infty \\ m_{ij} & \text{othewise} \end{cases}$$



(a) Exact union (b) Interval union (c) Zone domain union

Figure 2.17: Comparison of union operation on sets A and B

Affine inequalities domain Affine inequalities domain [CH78] or more colloquially, polyhedra, is a domain of intersections of half-spaces. It is strictly more expressive than intervals and affine equalities domains and is a relational abstract domain. It relies on theory of convex polyhedra and linear programming and represents the following constraints:

$$\bigwedge_{j=1}^{m} \sum_{i=1}^{|\mathbb{V}|} \alpha_{ij} V_i \geq \beta_j, \alpha_{i,j}, \beta_j \in \mathbb{I}$$

Polyhedra domain can be represented via two different methods: constraints and generators. The two representations have different theoretical complexities for different operations, so in most of the implementations [JM09; SPV17], the representation change depending on which operation is performed. This conversion between the two is possible thanks to Chernikova's algorithm [Che68], but it is EXPTIME complex.

Constraint representation is analogous to affine equalities domain representation, but with inequality instead of equality.

$$\langle M, \vec{C} \rangle, M \in \mathbb{I}^{m \times n}, \vec{C} \in \mathbb{I}^m, n = |\mathbb{V}|$$
$$\gamma(\langle M, \vec{C} \rangle) \stackrel{\text{def}}{=} \left\{ \vec{V} \in \mathbb{P} \;\middle|\; M \times \vec{V} \geq \vec{C} \right\}$$

Generator representation lists edges of the space hull, with the edges represented either with vertices or rays. In other words, $[P, R]$ is a pair of sets of vertices P and rays R, which then interpreted as:

$$\gamma([P, R]) = \left\{ \left(\sum_{j=1}^{p} \alpha_j \vec{P}_j \right) + \left(\sum_{j=1}^{r} \beta_j \vec{R}_j \right) \;\middle|\; \forall j : \alpha_j, \beta_j \geq 0, \sum_{j=1}^{p} \alpha_j = 1 \right\}$$

There some restrictions related to the domain:

- the representations are not unique;
- the minimal representation, i.e the one from which impossible to remove constraints without changing the set it represents, can be not minimal in number of constraints;
- there are infinite number of empty set constraint combinations, but only one such generator: $[\emptyset, \emptyset]$;
- Galois connection does not exist, as there is no best abstraction for some shapes (a circle for example).

Definition of operations on polyhedra:

- intersection $\sqcap$ is a union of all constraints and is an exact abstraction;
- union $\sqcup$ is a convex hull of the argument polyhedra and so is only an optimal/sound abstraction, so is one of the sources of imprecision;
- widening operator is called a semantic widening:

$$X \triangledown Y \stackrel{\text{def}}{=} \{c \in X \mid Y \sqsubseteq\in \{c\}\} \cup \{c \in Y \mid \exists c' \in X : X = (X \setminus \{c'\}) \cup \{c\}\}$$

  Intuitively, the widening chooses, among the possible equivalent constraint representations of the first argument, the one that maximizes the number of constraints that are kept, based on the second argument.

- no native narrowing.

Lattice automata Lattice automata [Gal08] is a domain that forms a regular language from elements of another domain. It requires the parameter domain to be atomic lattice, a lattice with atoms directly bigger than bottom element $\bot$.

Definition 2.9.13 (Atomic lattice)**.** Atomic lattice $(\Lambda, \sqsubseteq)$ is a lattice with a set of atoms $At(\Lambda) \subseteq \Lambda$ such that:

- $\forall a \in At(\Lambda) : \lambda \sqsubseteq a \implies \lambda = a \vee \lambda = \bot$ (if $\bot$ exists it should be a least upper bound-Complete lattice), i.e atoms are only one step bigger than "nothing";
- $\forall \lambda \neq \bot \in \Lambda : \lambda = \bigsqcup\{a \in At(\Lambda) \mid a \sqsubseteq \lambda\}$, i.e everything is exactly upper bound of set of atoms.

Then the lattice elements are considered as labels in a finite automaton. Thus, a potentially infinite (but regular) language can be expressed, for example, to simulate a queue. Such domains as polyhedra and intervals can be used as the parameter domain. The automaton can be deterministic or not, and the two are *not* equivalent.

Definition 2.9.14 (Lattice automaton)**.** Lattice automaton is a tuple $\langle \Lambda, Q, Q_0, Q_f, \delta \rangle$, where:

- Λ is from an atomic lattice $(\Lambda, \sqsubseteq)$;
- Q is a finite set of states;
- $Q_0 \subseteq Q$ and $Q_f \subseteq Q$ are sets of initial and finite states respectfully;
- $\delta \subseteq Q \times (\Lambda \setminus \{\bot\}) \times Q$ is a finite translation relation.

A finite word $w = a_0 \dots a_n \in At(\Lambda)^*$ is accepted by the lattice automaton if there exists a sequence $q_0, q_1, \dots, q_{n+1}$ such that $q_0 \in Q_0, q_{n+1} \in Q_f$, and $\forall i \leq n, \exists (q_i, \lambda_i, q_{i+1}) \in \delta : a_i \sqsubseteq \lambda_i$. Thus the language defined by the automaton A is L_A. A lattice-based regular language $Reg(\Lambda)$ is a language recognized by lattice automaton $A = \langle \Lambda, Q, Q_0, Q_f, \delta \rangle$.

2.9.4.1 Transformers

Transformers are special domains parametrized by other domains. The ones we discuss here make the analysis more precise while still using the original domains unchanged. These include disjunctive completion, product domain and partitioning.

Disjunctive completion Disjunctive completion is a domain consisting of a finite subset of a power set of the basis (parameter) domain elements: $\hat{\mathcal{D}} \stackrel{\text{def}}{=} \{A \in \mathcal{P}(A) \mid \forall X \neq Y \in A : X \not\sqsubseteq Y\}$. Important detail, is that the new domain does not allow redundant elements. Then the concretization function is $\hat{\gamma}(A) \stackrel{\text{def}}{=} \bigcup_{X \in A} \gamma(X)$.

It is hard to compare the new elements, so a relaxed order is used: $A \sqsubseteq B \iff \forall X \in A : \exists Y \in B : X \sqsubseteq Y$. Without redundancy elimination it would be only preorder and not partial order. $A \hat{\sqsubseteq} B \implies \hat{\gamma}(A) \subseteq \hat{\gamma}(B)$.

Number of disjunctions may grow exponentially, thus we define a simplification operation that is triggered when a certain threshold is reached: $\text{collapse}(A) = \{\sqcup\{X \in A\}\}$, which means that the disjunctive elements are summed in abstract domain into one element (potentially big loss of precision).

In order to guarantee finite increasing chains in widening, the simplification function is used before applying base domain widening: $A \hat{\triangledown} B = \{\text{collapse}(A) \triangledown \text{collapse}(B)\}$

Example of how differently encoded domains can be valid and represent the same concrete set, yet be mostly incomparable is on Figure 2.18. Galois connection is not guaranteed even if it exists in the base domain (Figure 2.19).

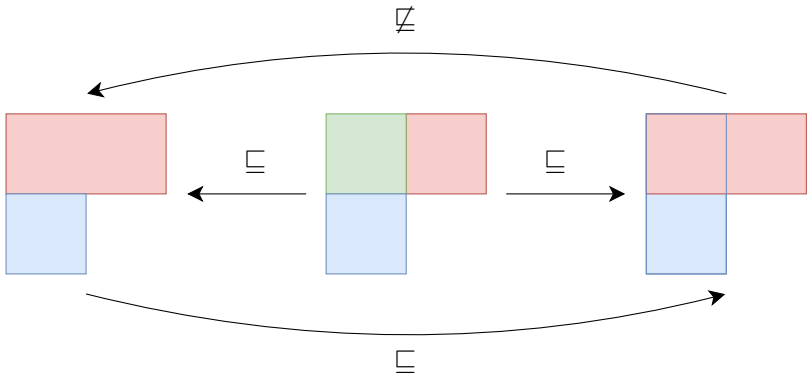


Figure 2.18: Examples of disjunctive completion for the same concrete set and their order

Product domain Product domain puts two abstract domains side by side and executes corresponding operations on both of them. This alone does not provide any improvements to precision, as the domains do not share information, and essentially runs two (or more) analyses in parallel. For improvement in precision, a reduction operation should be defined. The reduction in turn can be partial or full, which depends on if Galois connection exits for the involved domains or not.

(a) With intervals (b) With disjunctive completion

Figure 2.19: Best abstraction is not guaranteed by disjunctive completion

An operator $\rho : D_1 \times D_2 \to D_1 \times D_2$ is a partial reduction between domains D_1 and D_2 if:

$$(Y_1, Y_2) = \rho(X_1, X_2) \implies \begin{array}{l} \wedge\, \gamma_1(Y_1) \cap \gamma_2(Y_2) = \gamma_1(X_1) \cap \gamma_2(X_2) \\ \wedge\, \gamma_1(Y_1) \subseteq \gamma_1(X_1) \\ \wedge\, \gamma_2(Y_2) \subseteq \gamma_2(X_2) \end{array}$$

The first condition, i.e., the equality up to $\gamma_{1\times 2}$, states the soundness. The two inclusions state that, although the product concretization is the same, each element has been strengthened in its respective domain.

In fully reduced domain, one executes the reduction after applying specific version of the operation on each domain (except widening). As an example, we define the reduction between interval domain and congruence domain:

$$\rho([a,b], c\mathbb{Z}+d) \stackrel{\text{def}}{=} \begin{cases} (\bot, \bot) & \text{if } a' > b' \\ ([a,a'], 0\mathbb{Z}+a') & \text{if } a' = b' \\ ([a',b'], c\mathbb{Z}+d) & \text{if } a' < b' \end{cases}$$

where

$$\begin{aligned} a' &= \min\{x \geq a \mid x \mod c \equiv d\} \\ b' &= \max\{x \leq b \mid x \mod c \equiv d\} \end{aligned}$$

Also, we can define a universal simple reduction, which reduces only when one of the domains is empty. Given abstract domains X, Y the reduction is a function $X \to Y \to X \times Y$:

$$\rho(x,y) \stackrel{\text{def}}{=} \begin{cases} (\bot_X, \bot_Y) & \text{if } x = \bot_X \vee y = \bot_Y \\ (x,y) & \text{otherwise} \end{cases}$$

2.9.5 Partitioning

Partitioning is a domain used to improve precision by allowing to represent non-convex sets, similar to disjunctive completion. But unlike disjunctive completion it allows only finite amount of disjunctions, with strictly defined definition regions to cover.

Formally, a partitioning of abstract domain A of concrete domain D is a set $\widetilde{P} \in \mathcal{P}(A)$ for which $D = \bigcup\{\gamma(X) \mid X \in \widetilde{P}\}$ holds. In other words, the set of the abstract elements should

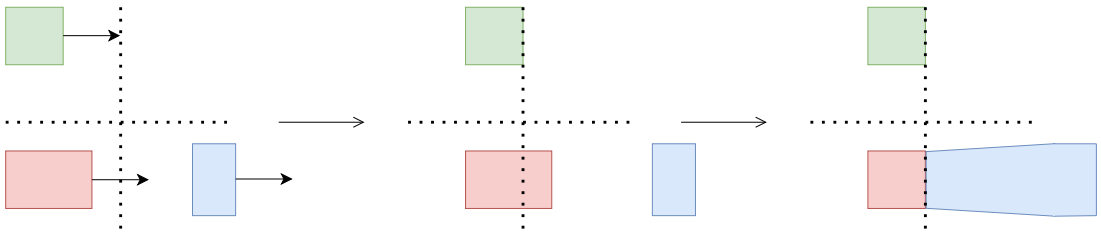


Figure 2.20: Choice of partitioning boundaries is important

add up to the whole concrete domain via concretization. Technically abstract partitioning is a set covering, and *not* partitioning, as does not require elements to be disjunctive.

Then, the partitioning *domain* is $\widetilde{D} \stackrel{\text{def}}{=} \widetilde{P} \to A$ with concretization function $\widetilde{\gamma}(p) \stackrel{\text{def}}{=} \bigcup\{\gamma(p(x)) \cap \gamma(x) \mid x \in \widetilde{P}\}$. In this definition, the function returns an abstract value for each part and makes sure to restrict its interpretation to the part definition area.

The partial order $\widetilde{\sqsubseteq}$ is defined part-wise $A \widetilde{\sqsubseteq} B \stackrel{\text{def}}{=} \forall X \in \widetilde{P} : A(X) \sqsubseteq B(X)$. If Galois connection exists for the basis domain, then it exists for the partitioning too: $\widetilde{\alpha}(S) \stackrel{\text{def}}{=} \lambda X \in \widetilde{P} \to \alpha(S \cap \gamma(X))$.

Operators are defined part-wise:

- intersection $A \widetilde{\cap} B \stackrel{\text{def}}{=} \lambda X : \widetilde{P}.A(X) \cap B(X)$;
- union $A \widetilde{\cup} B \stackrel{\text{def}}{=} \lambda X : \widetilde{P}.A(X) \cup B(X)$. Union is not exact in general as inside the partition the elements are abstracted as usual. Then the exactness of the union depends on how good the partition separates the elements we add to it; this way we can control the precision;
- widening operator $A \widetilde{\triangledown} B \stackrel{\text{def}}{=} \lambda X : \widetilde{P}.A(X) \triangledown B(X)$; exists if widening in base domain exists;
- same for narrowing operator $A \widetilde{\triangle} B \stackrel{\text{def}}{=} \lambda X : \widetilde{P}.A(X) \triangle B(X)$.

It is quite important how the partition boundaries are chosen, as it may lead to a loss of precision, as demonstrated on Figure 2.20.

Decision tree domain When a program contains Boolean variables, it is a good idea to partition by them first, as they behave non-linearly and so are the cause of non-convexity. But a partitioning with respect to n Boolean variables produces 2^n partitions which is too much. Thus a technique similar to MTBDD was proposed [Ber+10]: instead of numerical or other leaves, the abstract elements are used. This way the equal subtrees are shared, reducing the costs of the partitioning. But it also acts as a reduced product domain, as the values of Boolean variables are able to be related to other domains. We give an example on Figure 2.21.

2.9.6 Tools

In this section, we would like to talk about some of the tools that use abstract interpretation. There are quite a few, including Astree [Käs+10], Frama-C [Kir+15], NBac [Jea03], ReaVer [GS14], InterProc [Jea13], MCSCM [HLS12], MOPSA [Jou+20]. But because most focus on general purpose languages, we cover only the ones important to us, namely NBac and ReaVer.

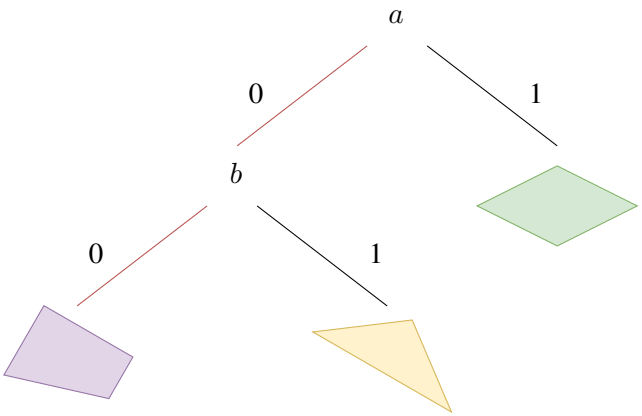


Figure 2.21: Decision tree domain

2.9.6.1 NBac

NBac is a verification tool for symbolic transition systems. The tool accepts a specific format describing the system, including the property to be checked. The format was designed to express reactive systems, as the main purpose of the tools was to verify Lustre programs. As such, the tool analyses transition systems, expressed declaratively, containing combination of Boolean and numerical variables, which may be modified by interaction with non-deterministic environment.

Format The input file describing the system consists of several sections. Roughly these are declarations, definitions and assertions.

First are variables declarations. NBac splits them into 3 categories: state, input and local variables. While names of state and input variables talk for themself, local variables are not that oblivious. The role of local variables is to be shortcuts for long expressions.

Second section is about transition relation definition. It consists of a list of statements about next values of each *state* variable. There we can write pretty much any expression, involving Boolean conditionals, arithmetic expressions, using any previously defined variables, state, input or local. Then product of the definitions makes the transition relation of the system.

Third section defines initial and final states. NBac is flexible in this regard, so we can specify a whole set of initial states, as it is defined with a Boolean formula on the state variables, i.e. a relation. As for the final state it is the same and the final state is something that should not be reached. While the property can be defined in its positive form, i.e. when it is not violated, internally it will be inverted and checked for unreachability.

Finally, the assertion condition specifies restrictions and assumptions of the system. As the only requirement to the assertion is to be a Boolean formula that evaluates to true on variables defined above, we can use it as a way to restrict what input variables can be in the relation to the state and each other. In other words, it is a way to specify guards or labels of the transition system symbolically.

Dynamic partitioning Unique feature of NBac is dynamic partitioning. The fine details of the approach are explained in [JHR99; Jea03], but we describe the main principle.

In general, the technique consists of splitting (partition) the state space of a program and refining the partition following a heuristic, which uses conditions in the program. If a property is never violated in any explored this way state, then it is proved to always hold.

Formally, we obtain the initial S_I, final S_F and "everything els" $S \setminus S_F \setminus S_I$ partition of the program's state space. Function $\mathrm{def} : L \to \mathcal{P}(S)$ is definition or domain of a location $l \in L$, set as some element $a \in A$ of the abstract domain A. $\bigcup_{l \in L} \gamma(\mathrm{def}(l)) = S$ has to be satisfied for the state space S to be a partition, as previously explained in Section 2.9.5. Additionally, the initial state set S_I and initial locations L_I should satisfy $S_I \subseteq \bigcup_{l \in L_I} \gamma(\mathrm{def}(l))$, same for final states S_F and locations L_F. Here and later we treat the partition as equivalent to the locations, as defined by the def. We check that the initial and final locations are mutually exclusive, otherwise the invariant represented by the final location is trivially violated.

Definition 2.9.15 (Transition relation)**.** Between the locations, a transition relation $\rightsquigarrow \subseteq L \times L$ is defined, where E is a set of input events (labels) of the original program:

$$\exists s \in \gamma(\mathrm{def}(l)), \exists s' \in \gamma(\mathrm{def}(l')), \exists e \in E : s \overset{e}{\mapsto} s' \implies k \rightsquigarrow k'$$

We use it later to construct reachable state space.

$\rightsquigarrow$ is a complete relation at the beginning of the analysis, except for initial and final states: they are sources and sinks respectively.

The main loop of the analysis consists of two functions, defining reachable and coreachable (backwards reachable from the final locations) state space. Formally, the target of the analysis is to prove that the initial does not reach final, i.e. $\mathrm{reach}(l \in L_F) = \emptyset$. Or if defined differently, if for all locations intersection of reachable and coreachable states is empty, then it cannot lead to the final state. Using this fact, any such location can be safely removed from the analysis and so simplify the next cycle of the analysis. When the intersection is not empty, we call such states dangerous, and continue to refine the partition until convergence and the further refinement is not possible.

Definition 2.9.16 (Forward reachability)**.**

$$l \in L_I : \mathrm{reach}(l) \overset{\mathrm{def}}{=} \mathrm{def}(l)$$
$$l \notin L_I : \mathrm{reach}(l) \overset{\mathrm{def}}{=} \bigsqcup_{l' \rightsquigarrow l} \mathrm{next}(\mathrm{reach}(l')) \sqcap \mathrm{def}(l)$$

Definition 2.9.17 (Backward reachability)**.**

$$l \in L_F : \mathrm{coreach}(l) \overset{\mathrm{def}}{=} \mathrm{def}(l)$$
$$l \notin L_F : \mathrm{coreach}(l) \overset{\mathrm{def}}{=} \bigsqcup_{l \rightsquigarrow l'} \mathrm{prev}(\mathrm{coreach}(l')) \sqcap \mathrm{def}(l)$$

Transitions, or next and prev functions, are obtained by compiling the syntax of the original transition system into diagrams [Jea02]. These transitions are forward and backward abstract interpreted using reachability (resp. coreachability) information of locations they lead to. This results in specialized and so more precise than before representation of the transition. If this analysis concludes that the post or pre condition of a transition is not part of the source or target location, such transition can also be safely removed.

After forward and backward analysis reachability relation $\rightsquigarrow$ and def are updated as:

- forward:

$$l \rightsquigarrow l' \iff \text{next}(\text{reach}(l)) \wedge \text{reach}(l') \neq \bot$$
$$\text{def}(l) \coloneqq \text{reach}(l)$$

- backward:

$$l \rightsquigarrow l' \iff \text{prev}(\text{coreach}(l')) \wedge \text{coreach}(l) \neq \bot$$
$$\text{def}(l) \coloneqq \text{coreach}(l)$$

When analysis has concluded but dangerous states still present, it means that refining of locations is needed. The intuition is that the increase in precision will remove a path to the final location. For this, the tool employs several levels of heuristics. First, the intention of the refinement is to remove the part of the state that both is the result of an overapproximation and leads to a location, not reachable otherwise. More specifically, the refinement splits transitions and related

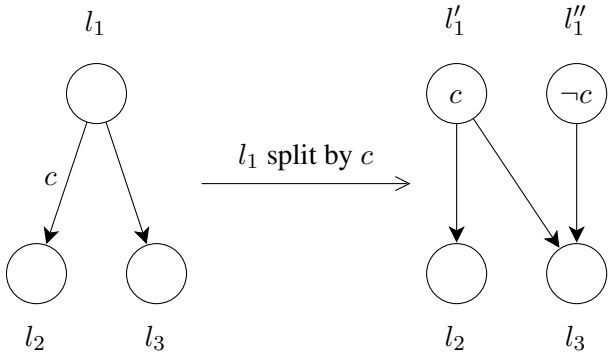


Figure 2.22: NBac location refinement

locations by atomic propositions, either Boolean or numerical conditions, present in the transition itself (Figure 2.22). It is important point because the transition is defined from finite number of conditions, so the choice of atoms is finite, making the process converging. If there are several atoms in a precondition, they can be ranked by how important they are: in $(a \vee b) \wedge c$ atom c is "more important" as it is a singular necessary condition to violate the formula. Second, as there could be several locations to split, a strategy to choose is employed. The locations and the transition relation are represented as a graph, and later grouped into strongly connected components. Then the job of the refinement is to try to disconnect the components from each other. The necessary condition to leave a component is to satisfy precondition of at least one transition that leaves the component, i.e. $\forall l \in L_s, l' \in L_t : \bigvee_{\text{pre}(l \rightsquigarrow l')}$, where L_s are source component locations, L_t are target component locations, $\text{pre}(l \rightsquigarrow l')$ returns precondition to the transition $l \rightsquigarrow l'$. The components are hierarchical, so the process starts on the highest (more abstract) level and goes lower, smaller components. It is natural to start big, as disconnection of a bigger component removes more work to do than a smaller one. The process continues until all possible conditions are found and split.

2.9.6.2 ReaVer

ReaVer is a further development of NBac and accepts the same language. Its unique feature is to use numerical acceleration of loops. Unfortunately, it is not a superset of NBac, as the dynamic partitioning is not present and neither partitioning by numerical bounds. The reason is that the acceleration would not be necessary compatible with dynamic partitioning. But it provides a lot of different partition and analysis strategies to choose and experiment with. The strategies to the engine are provided as a string of instructions, for example, "aB;aB:b", which means to perform forward and then backward Boolean analysis.

Logico-numerical acceleration We start with the explanation of acceleration itself, followed by logico-numerical specifics and how it is implemented in the tool.

Numerical acceleration is a part of linear relation analysis. Roughly speaking, the effect of a simple loop, guarded by a linear condition on integer variables, and consisting of constant increments/decrements of these variables can be computed exactly [Bar+08; GH06; GS14].

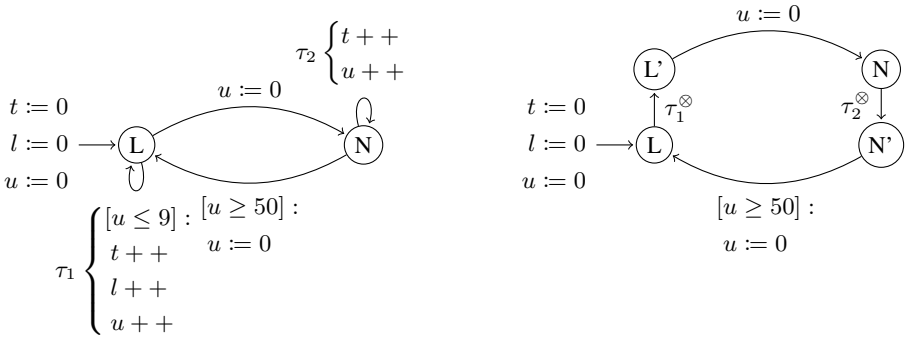


Figure 2.23: Automaton of the gas burner [CHR91] and its accelerated version

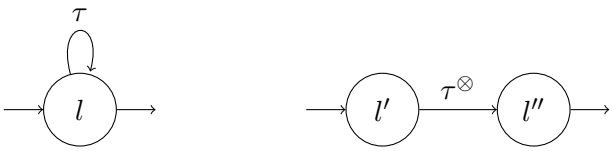


Figure 2.24: Acceleration of a simple loop

More formally, given loop transition $l \overset{\tau}{\rightsquigarrow} l$, acceleration produces $l' \overset{\tau^{\otimes}}{\rightsquigarrow} l''$ (Figure 2.24). Note that the locations are new. If before the location l would contain values $0 \leq x \leq N$, with transition $\tau^{\otimes} = [x := x+N]$, location l' contains only $x = 0$ and l'' contains only $x = N$. Another example of acceleration is shown on Figure 2.23.

Definition 2.9.18 (Extending polyhedra with a ray)**.** Given polyhedra P, its extension by ray vectors $R \in \mathbb{Q}^n$ is written as $P \nearrow R$ and defined as

$$\{x + \sum_{\vec{r}_j \in R} \mu_j \vec{r}_j \mid x \in P, \mu_j \in \mathbb{Q}_{>0}\}$$

There are several case that can be handled with the acceleration. These include: single or double loop, with complete or partial rest. We describe only one loop and the complete reset.

Case of one loop

$$\tau^{\otimes}(P_0) \stackrel{\text{def}}{=} \text{hull}(\{x \mid \exists i \in \mathbb{Q}_{>0} \exists x_0 \in P_0 : g(x_0) \wedge g(x - D) \wedge x = x_0 + iD\}) \sqcup P_0$$

where τ is original loop relation, $\tau^{\otimes}$ is its accelerated version, hull is a function that approximates the set in used abstract domain (for example, polyhedra). The approximation computation of the accelerated loop is defined as:

$$\tau^{\otimes}(P_0) \stackrel{\text{def}}{=} ((P_0 \cap Ax \leq \vec{B}) \nearrow \{\vec{D}\}) \cap \{A(x - \vec{D}) \leq \vec{B}\}$$

where $Ax \leq \vec{B}$ is a loop guard g, loop function is $Cx + \vec{D}$, C is an identity matrix, $P \nearrow \{V_1, V_2\}$ means that polyhedron P is widened with rays V_1 and V_2.

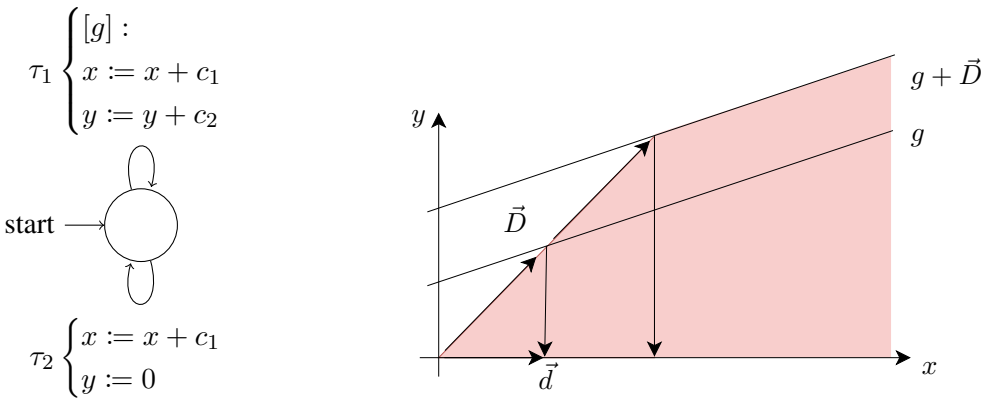
The intuition here is that, given the linear transformation, the affine translation is performed on the input state. This translation is only stopped when the guard is encountered. But, it does not take into account the last application, thus $A(x - D)$ is used as the limit.

Complete reset In the case of loops with resets, we are given two loops, one that does constant iteration $\tau_1 : [g] : x := x + c_1, y := y + c_2$ and another doing a reset to one of the variables $\tau_2 : y := 0$. The reset transition is assumed to be always enabled, thus no guard. Then the acceleration of this loop is computed as:

$$(\tau_1 + \tau_2)^{\otimes}(P_0) \stackrel{\text{def}}{=} P_0 \nearrow \{\vec{D}, \vec{d}\} \cap g(x - \vec{D})$$

where $\vec{d} = \vec{D} \downarrow [y = 0] = \begin{pmatrix} c_1 \\ 0 \end{pmatrix}$ is a projection of $D = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$ to $y = 0$ plane.

Here, the idea is that the acceleration does the same interpolation as before, but it is modified with a shift to the right, thus projecting two ways at the same time. This results in an abstraction with the shape as on Figure 2.25.



(a) Location with the loops

(b) Deduced state space

Figure 2.25: Acceleration of two loops with reset

In the method of logico-numerical abstract acceleration [SJ11a], acceleration is the most efficient for pure numerical constraints, like $x' = f(x, \xi)$. Because of this, they try to decompose

logico-numerical transitions into Boolean and numerical parts when possible or at least such that the Boolean part does not influence the numerical value. These transitions then can be separately accelerated and combined again as a Cartesian product. But, when a transition can be only partially accelerated, the convergence of the Kleene iteration (self-looping iteration) is not guaranteed.

Additionally, the partitioning is specially constructed so that the transition functions have more chance to be accelerable: the numerical part is cleaned from Boolean dependencies by grouping them into the same locations.* The assumption is that this makes the transitions not depend on the Boolean variables and so makes them accelerable. But in the worst case, every Boolean combination is enumerated or there is only single state, both supposed to be unlikely. The end result should reduce the size of the partitioning, improve the precision as the transitions are exact and simpler, both because the guards in the accelerated transitions are ignored and because the effect of the transition is simpler.

2.10 Conclusion

To summarize, we have shown different approaches to describe and verify reactive systems. We compare them in Table 2.3 by the following points:

- decidability of validity or formula non-contradiction: we want to be sure that the solutions are possible at all before trying to check that the implementation satisfies it;
- metric relations: reactive systems do no exist in a vacuum and are executed along with some physical process they control or depend on. And so the reaction time is important and so we would like to specify the deadlines and delays as properties or as part of specifications;
- notion of time: the view on evolution of the systems. For that we outline three attributes: discrete vs continuous, branching and multiformity. Difference between discrete and continuous time lies in the definition of progression: do we consider system progressing when something distinct happens (an event, change of state) or not, i.e. the evolution is "smooth". By branching of time we mean, that the language can express properties to be true while considering the alternative possible futures or alternative past timelines. Here only CTL and CTL* are able to do it. As for multiform time, while linear, it defines time as progression of individual events and not by observation of state. Then some of them allow simultaneous events and some do not. This makes the language constructs compositional, as then there is a possibility to fit one more event between others. The exception here are synchronous languages, they do have a limit and it is defined by how slow the reaction can be;
- language style, with imperative, declarative and functional variants. We define language as imperative if its statements explicitly say what the reaction is. A language is purely declarative, when a language specifies temporal possibilities. Functional language defines the reaction purely as a deterministic transformation from input and state variables to output, but where . Thus temporal logics are imperative because they can express simultaneous or delayed satisfaction of conditions using next and implication with eventually, and are declarative because of temporally permissive eventually. Metric temporal logics lose declarativity as we define it because the eventually is bounded.

*Word "accelerable" was used in [SJ11b]

	Expressiveness		Time		Programming style
	Decidable validity	Metric relations	Branching	Multiform	
LTL	✓	×	×	×	imp./dec.
CTL	✓	×	✓	×	imp./dec.
CTL*	×	×	✓	×	imp./dec.
MTL	×	✓	×	×	imp.
STL	✓	✓	×	×	imp.
TESL	✓†	✓	×	✓	dec.
Event-B	×	✓	×	×	imp.
CCSL	✓†	×	×	✓	dec.
Timed Automata	✓	✓	×	×	imp.
Lustre	✓†	×	×	✓	fun.

Table 2.3: Comparison of the approaches

Something not applicable to every language but important is refinement: from developer point of view, it is much easier to go from more generic to less generic description, step by step, than to get the system right in the first try. But, it has a problem of consistency that needs to be solved: a prove should be constructed that refined description represents a subset of the abstract behaviour. Here, only Event-B features refinement as a conscious feature.

Another note is that while CCSL, TESL and Lustre are decidable to find out whether a finite trace exists, the infinite ones are not.

In this comparison we want to note that there is no language which is declarative, can express metric constraints, like delays and deadlines, allows iterative development with refinement and is "friendly" to symbolic model checking. We also think, while there are plenty of the languages to do the implementation, the description of the requirements is lacking in case of time. The closest to our description is TESL, but decision to remove the asynchronous constraints of CCSL made it less useful to describe systems, while really permissive relations between tags are too relaxed to be efficiently analysed. For this reason TESL only provides a simulation support. In contrast, Timed Automata has nice results in abstract model checking, but because simultaneous events are not allowed, emulating the product of automata is too complex. In Event-B, it is possible to express a limited metric constraints by using parametrized events, but it struggles from the same problem as Timed Automata. Then, while there is CCSL, in its current form, it lacks exact metric relations, only approximation can be expressed. Additionally, the refinement of CCSL was not investigated enough, while we think it is an important feature that provides great improvement to the development.

Thus, we have decided to develop a limited case of metric relations and a model checking for CCSL as a compromise solution between TESL, Timed Automata and original CCSL, named Real-Time CCSL. By using only selected patterns, inspired by Timed Automata, we intend to avoid generality of relations of TESL and take advantage of results of Timed Automata. Using

these results directly is not possible, as CCSL is not equivalent, we have to implement a similar analysis. Due to fact that CCSL is essentially defined using Boolean and integer variables and their interleaving expressions, abstract interpretation seems to be a good choice. As we have seen in this chapter, there are works, namely on NBac and ReaVer, with which we are able to express CCSL itself, but not the new language nor it can do all the analysis we would want to. An orthogonal modification to enable refinement and specification reuse is what we define as the modular CCSL. The combination of both is then called Modular Real-time CCSL (MRTCCSL).

We provide further motivation to the exact features of the language in Chapter 3, followed by the language definition in Chapter 4 and implemented and proposed analyses in Chapter 5.

CHAPTER 3

Motivational examples

This chapter describes some examples that are representative of typical systems we would like to describe. They also contain elements that have proven to be difficult to either be modelled or be analysed. We give an informal description of those examples and describe anterior works that was already done to describe them in CCSL. We emphasize the problems encountered related to CCSL analysis and describe the changes needed to tackle them.

CCSL has been used over the years on numerous examples from various domains: brake-by-wire system [Gok+13], spark ignition control system [PD11a], CPU interference [Oue+19] and temperature control system [Sur+13b]. In this chapter we describe some of them (Sections 3.2 to 3.4), but also an example of a drone complex (Section 3.1) as an introduction. We include the system requirements as well as their logical specifications. We also discuss the problems encountered and summarize them at the end of the chapter as they are the main driver for this work and largely explain where we have focused our attention.

3.1 Drone complex

We start with an easy to understand illustration of how CCSL can be used, an agriculture drone and its supporting devices.

In this setting, a drone usually flies with a certain mission, either predefined and automatic or manually controlled. The drone monitors the state of the crops or the fields with cameras, reports the exact geometry of the crops so that the harvesting could be optimized later. An operator controls the drone with a control device, setting up the waypoints or correcting the automatic path. The feedback consists of the video feed and telemetry, like positioning and characteristics of the environment around the drone. The video is then further processed to derive the information by a processing unit, either located in the control device itself or in the datacenter. In both cases the data is ultimately collected into a datacenter, where the data is aggregated and analysed over periods of time for each plot of land. For example, this allows finding a correlation between biomass growth, crop species, final harvest and weather history at that location. If bad harvest cannot be explained by factors outside our control, like bad weather, then investigating the case and fixing the cause should improve the total harvest. The Figure 3.1 reflects this description by showing components of the system, environment and actors, and the information flow between them.

3.1.1 Modeling

As a reactive system, drone and its control consists of various interacting parts, including local ones like tasks executing on a CPU, or not so local, like the wireless communication between the drone and the base station. We describe some of the usage cases of the system as a composite diagram of several activity flowcharts on Figure 3.2. The hardware in turn is described on Figure 3.3 as a structure diagram a-la SysML and features physical connections between different components (resources).

In a CCSL specification, we would define the behaviour and the resources usage as clocks. These include user clicking a specific button, moving the control over the joystick arm, or even just a general “user input” event. Depending on input, like submitting waypoint plan, the preparation of the message to the drone may require non-negligible amount of processing, and so such task is modelled as a pair of clocks, start and end respectively, with the duration relation defined with respect to some reference clock. The communication itself is similar, with transmission and receiving of a message encoded as a pair of clocks, with the content of the message disregarded. The relation between clocks can be either simple causality or an approximation of the duration that the message should take to be communicated. Then, the drone would receive the message and do some processing on its part, with interacting tasks, either because of data dependency or interference due to periodic processes that keep the drone in air and the occasional communication having to share resources.

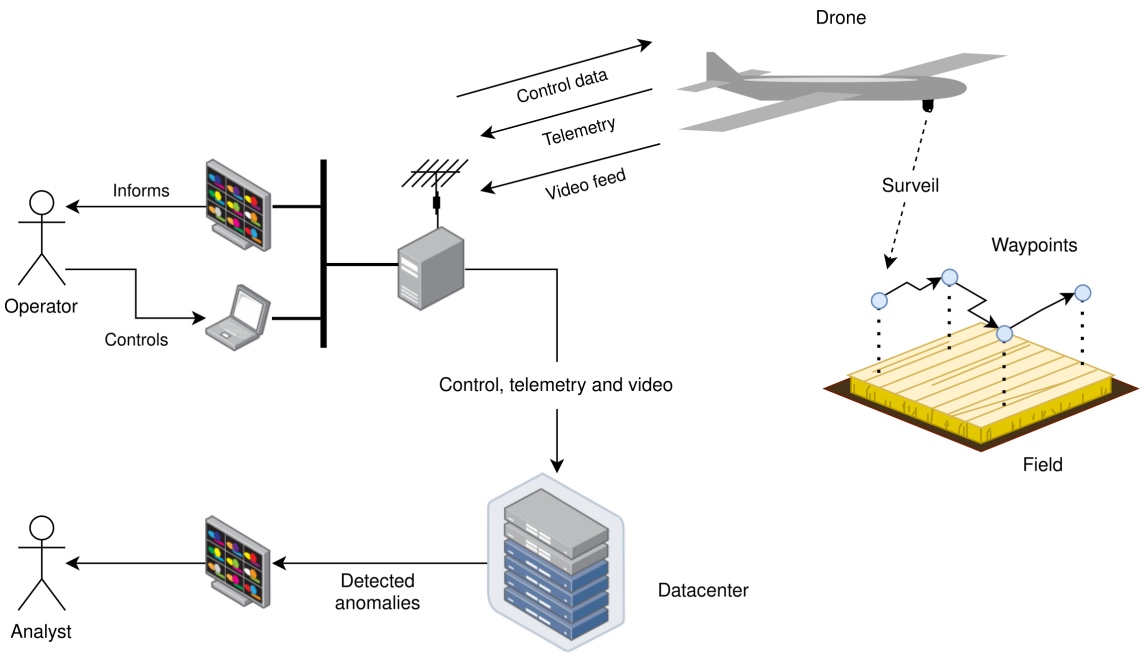


Figure 3.1: Overview of the drone complex and its information flow

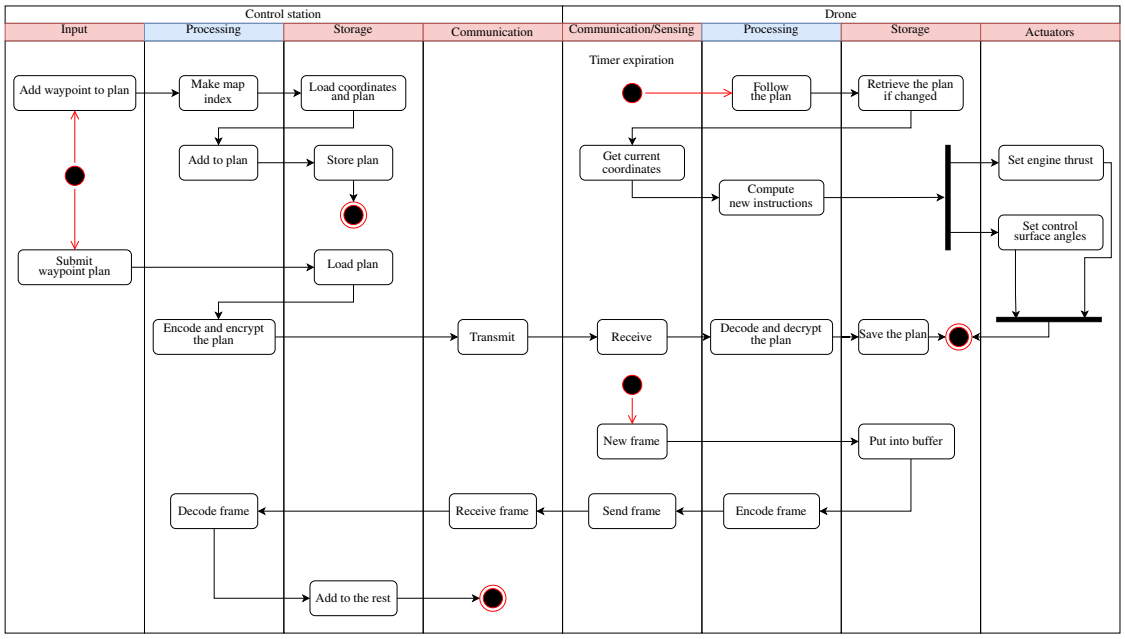


Figure 3.2: Composite activity diagram (cyber part is blue, physical is red)

3.1.2 Discussion

Currently, the only option to organize the constraints is to arrange them sequentially, maybe rearranging into groups. In any case, such specification, consisting sometimes or tens and hundreds of constraints, would be really difficult to process for people without any abstractions. Additionally,

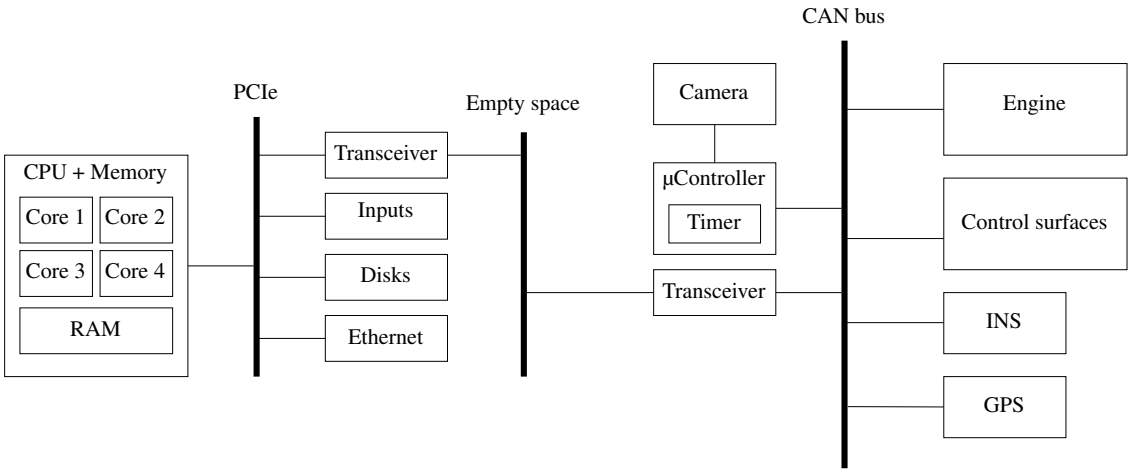


Figure 3.3: Composite structure diagram of hardware and its connections

some patterns, like to describe tasks, resources or modes of the system, would reappear everywhere as essentially same constraints, just with different parameters. Thus we envision a better organization.

In total, the system would consists of three main specifications: activity, hardware connections and its independent of activity restrictions, and deployment (mapping) of the activity on the hardware. Each main specification then could be split for each individual component, if it is too complicated. Within the deployment specification we would also specify constraint that relate both physical assumptions, hardware limitations and control. For example, transmission time depends simultaneously on speed of light, hardware speed, message encoding and size, encoding algorithm complexity, all of which do not fall under the description of activity or components.

The strength of CCSL here, is that the designer can concentrate on a single case and on a subsystem at the time without too much thought about the other parts. With the exception that the different subsystems still need to share names of the same logical clocks, otherwise no connection can be made. The job of CCSL itself then is to figure out if the cases overall and so the complete system makes sense, i.e. can produce some behaviour. The developer then can further investigate if this behaviour corresponds to their understanding of the system and correct respectively. If the analysis concludes that there is no behaviour, it is either the developer misunderstood what they are trying to do, or the requirements are inconsistent with each other, both requiring fixes to the specification. If the specification development process has finished with success, then the development team ends up with something really nice: each constraint can be transformed into a test or an objective in implementation, giving the team (hopefully) a well-defined goals to achieve. And if all of them are implemented, it is guaranteed by the previous analysis to result in a correct system. It is important to note, that this does not guarantee that the implementation actually runs because the implementation defines a subset of the behaviour and thus not equivalent. But what runs is conforming to the specification.

3.2 Mechanical Lung Ventilator

In this section we present the mechanical lung ventilator use case [BG24] as an extended version of our work [TM24].

The 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.

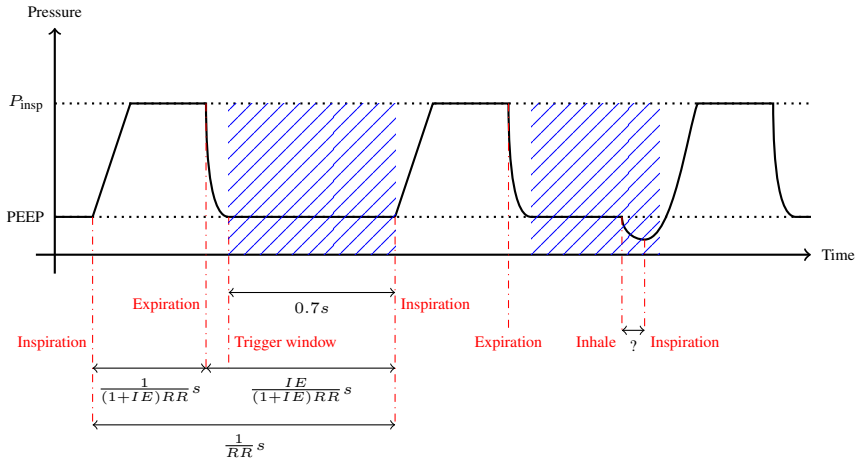
The purpose of the system is to support the patient with oxygenated air in case of breathing issues. This use case was developed as a challenge to describe, formalize and model a real ventilator that was quickly introduced in response to COVID pandemic. Obviously and ideally, medical anything, including hardware, should be thoroughly tested and verified before it is deployed in the field. But sometimes in case of massive challenges, like pandemics or war, the standards are lowered. Another reason for this is our inability to provide cost and time effective techniques and frameworks to develop such systems, and such challenge helps to develop this capability as a sort of retrospective.

We start by introducing the main requirements of the ventilator. First of all, the ventilator should go though a procedure of initialization and self-testing, before it can be used. Then the user can choose ventilation in two different modes, pressure controlled ventilation (PCV) and pressure support ventilation (PSV). The idea of PCV mode (Figure 3.4a) is that it supports breathing of patience that mostly cannot breath at all. Thus it consists of pressurizing and depressurizing cycles, with variable duration. But, mostly does not mean cannot breath at all, thus it allows and detects the attempts in the expiration cycle, starting a new cycle as soon as possible to support the attempt. The second mode, PSV, does exactly this, but without fallback to forced breathing. In case if patient does not breath before a set deadline, reset after each cycle, the system will sound an alarm. But if the patient does breath, the pressure will support it. Medically, it allows for a quicker recovery, as the muscles can recover some strength after the infection before doing it alone.

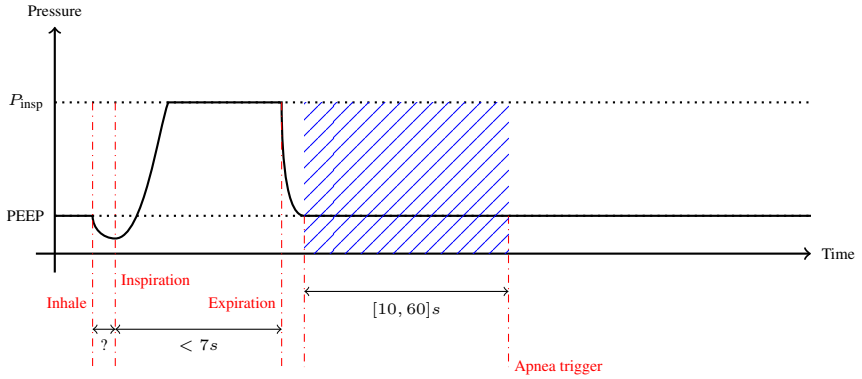
3.2.1 Modeling

From the provided use case we can build a formal model in CCSL. We focus 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 (Section 2.5), Frama-C [Kir+15] and synchronous languages (Section 2.3).

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. 3.4a), 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.



(a) PCV mode plot with events



(b) PSV mode plot with events

Figure 3.4: Ventilator modes

Pressure Support Ventilation (PSV) mode is similar (Fig. 3.4b), 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).

Next are the mechanical parts and some safety requirements related to them. 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).

A big class of requirements is timing relations, mostly durations in 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. These time relations are durations expressed in seconds and can be modelled in CCSL by using a global reference clock with some period and express the time in terms of this clock.

Additionally, some of the requirements are parametric, while others specify in what ranges these parameters are and how they should change. The parameters cannot be set with a specification itself, because there is no state as concept of the *language*, so the logic of update is handled by something else. This uncertain nature of the parameters from the point of view of a 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).

Lastly, we want to ensure 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 (Section 2.6.6.5), absence of deadlocks (Section 2.6.6.3) or, specific to this case, the safety of the patient. It is expressed by ensuring that the exhalation valve is not closed for more than the required amount of time, and so a *specification* on itself.

CCSL description From the provided plot and the requirements (Fig. 3.4b, 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):

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

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 two 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.

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

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 $\mathit{sensor.inhale}$.

3.2.2 Discussion

While we can identify a lot of places, where we repeat the same definitions (all sorts of phases and modes), we cannot describe it in CCSL. Neither we can use parameters natively and ask a solver to find combinations that have solutions. Additionally, some events are time-bounded and require a special treatment to be expressed in CCSL. These are mostly concentrated in PCV and PSV modes.

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 s, 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. Then, this template should be read as: clock $right$ should tick after counting n_d number of nanoseconds. Meaning that we are sure that n_d have ticked and so at least $d\,\mathrm{s}$ have passed. If clock $left$ always coincides with ns, the delay is exact, otherwise it is approximate. Additionally, some relations should not be exact, as they represent physical processes which we cannot model exactly. Their evolution may change greatly from only a slightly change in starting or operating conditions. For example, oscillators change frequency in different temperature and temperature stabilized ones are more expensive. As well as the fact that these processes are measured by other non-ideal processes, introducing their own imprecision. Thus, 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 somehow regenerated. But also the relation will not be exact, unless it is infinite precision reference clock.

Lastly, the state grows too fast in the case of using the automata approach (Section 2.6.3). Ternary delay constraint $a = b\ \$\ d\ \texttt{on}\ base$ 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 3 constraints and synchronize them into a single automaton, in order to do model checking (Section 2.8.1), 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. $9.625\,\mathrm{GB}$ of memory.

3.3 Spark ignition control system

Spark ignition control system [AMP07] is a reactive system which main objective is to light up spark in cylinders of an engine at the “right time”.

First, we need to explain how engine works. Engine consists of two main parts: cylinders and crankshaft. Inside the cylinders there are pistons that are attached to the crankshaft. An air-fuel mixture is ignited with a spark plug inside a cylinder and this moves the piston which moves the crankshaft in circular motion. This provides the power to whatever load is, wheels on a road or something else.

Secondly, we describe how the four stroke engine works. In there the piston does two pack and fourth movement, each corresponding to full rotation of the crankshaft. This motion splits the cycle into four stages: exhaust, fuel and air injection, combustion and work. More specifically, when the piston moves up, it creates pressure on the air inside. If valves are open, it will push out the burned fuel from previous stage. Then the output valves close, the air valves and fuel is inserted. This happens when the piston moves down, dispersing and mixing the inputs. Next, the piston returns to compression. There the mixture is heated from the pressure and somewhere at the top position for the piston (top dead center), the spark will ignite the fuel. And the cycle repeats.

The timing of the spark is important as too early or too late ignition will make engine lose power. One way to detect the right moment is to mechanically couple the crankshaft to a rotating mechanism that closes the spark plug circuit every $720°$ (two rotations), with each such mechanism shifted $180°$ degrees for each cylinder. But nowadays, this control is made electronically as it is less expensive and there are less moving parts. Additionally, it is possible to optimize the efficiency further by recomputing the ignition point, which is the system under study here.

Given a sensor that detects potential knock phenomenon, the temperature of the engine and the mode of the motor (warm-up phase or not), position of the crankshaft in degrees the ignition instant can be advanced or delayed. To be more specific, we are concerned with the following requirements:

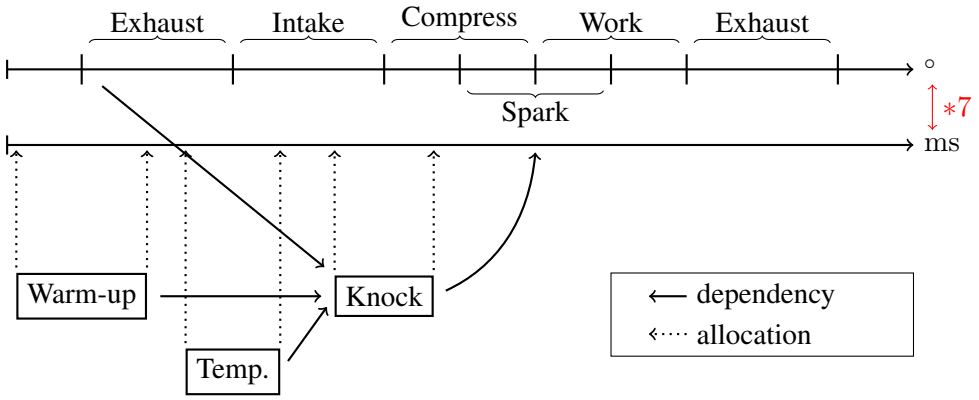


Figure 3.5: General relation between events in the engine, tasks and time

- there three tasks to compute: over temperature correction (TO), warm-up correction (TW) and knock control (TK);
- they need to be computed before computing the knock control;
- knock control need to finish $10\,\mathrm{ms}$ before the piston reaches the position where it can fire, and it should be computed every full cycle of a cylinder;
- the knock control data should not be more than two cycles old;
- the spark correction should be executed in the interval $[-15°; +15°]$ from the top dead center;
- the same process is repeated for each cylinder, and all the tasks execute on the same controller.

We also show some of the requirements on Figure 3.5.

Lastly, we define the relationship between the speed of crankshaft rotation RPM in revolutions per minute to the duration it takes for the crankshaft to turn by one degree as $p_{crk} = \frac{1}{6*\mathrm{RPM}}\,\mathrm{rpm}$.

3.3.1 CCSL specification

Thus the authors of [PD11b] make the following specification of this system. It is additionally fixed to the parameter $\mathrm{RPM} = 1166\,\mathrm{rpm}$ which implies $p_{crk} = 7\,\mathrm{ms}$ and $p_k = 720°/n_c$ implies

$p_k = 180°$ as the engine has 4 cylinders.

$$TK_F = TK_S \ \$ \ 14 \ \texttt{on} \ PK \tag{3.1}$$
$$TO_F = TO_S \ \$ \ 6 \ \texttt{on} \ PO \tag{3.2}$$
$$TW_F = TW_S \ \$ \ 6 \ \texttt{on} \ PW \tag{3.3}$$
$$TK_S = \texttt{skip} \ 6 \ \texttt{every} \ 180 \ crk \tag{3.4}$$
$$TK_D = TK_S \ \$ \ 60 \ \texttt{on} \ crk \tag{3.5}$$
$$TK_F \prec TK_D \tag{3.6}$$
$$TO_D = TO_S \ \$ \ 60 \ \texttt{on} \ crk \tag{3.7}$$
$$TO_F \prec TO_D \tag{3.8}$$
$$TW_D = TW_S \ \$ \ 60 \ \texttt{on} \ crk \tag{3.9}$$
$$TW_F \prec TW_D \tag{3.10}$$
$$TW_F \prec TK_S \tag{3.11}$$
$$TO_F \prec TK_S \tag{3.12}$$
$$TW_S \ \texttt{alternates} \ TK_S \tag{3.13}$$
$$TO_S \ \texttt{alternates} \ TK_S \tag{3.14}$$
$$PK \# PO \tag{3.15}$$
$$PK \# PW \tag{3.16}$$
$$PO \# PW \tag{3.17}$$
$$ms = (PK = PW) = PO \tag{3.18}$$
$$crl = \texttt{every} \ 7 \ ms \tag{3.19}$$

We show how they relate to the requirements above:

- Equations (3.1) to (3.3) specify worst case execution time (WCET) of the tasks of knock control, over temperature correct and warm up correction respectively;
- Equation (3.4) sets the start of the knock control on the degree scale;
- Equations (3.5), (3.7) and (3.9) specify the deadlines for each of the tasks, and Equations (3.6), (3.8) and (3.10) constrain the the tasks cannot violate the deadlines;
- Equations (3.11) to (3.14) defines the dependencies between the tasks;
- Equations (3.15) to (3.17) specifies that the tasks cannot execute at the same time and Equation (3.18) says that the processor is always busy with the tasks, but also that they can preempt each other; thus it was important to set deadlines for tasks, as the completion is not guaranteed;
- lastly, Equation (3.19) specifies the relation between the time periods of on degree crankshaft advance and controller millisecond clock.

3.3.2 Discussion

First of all, we can see in the specification, that there a several patterns that repeat for different clocks and that could be automated with a better language. Second, the milliseconds and

crankshaft turns are strongly coupled together. Which is certainly not the case in reality. Even if we set the periods, there still should be an error related to both parts. To model that, we could introduce a more refine clock, say of a nanosecond. Then define both in its terms, including the errors. But if we would need to introduce more moving parts, with another independent, but real-time related clocks, the reference clocks would have to be made even smaller, which is a problem computationally. Third, the specification definitely contains parameters, at least speed of micro-controller and the speed of the engine. Everything else is derived from them or some constants, like WCET. But it is not possible to specify and change these easily inside the language itself. Going further, we would also want a solver to find out if the specification is correct for the parameters in some nominal ranges.

3.4 Brake-by-Wire

Braking in any type of vehicle is a safety-critical feature. Ensuring that brakes always work as expected is a priority. Thus, while we want to implement more features in a car, like anti-blocking in brakes, it is mandatory to verify that it is correct and safe.

Thus, brake-by-wire [Gok+13; MPA09] is a system consisting of a pedal, controller, sensors and actuators (brakes) inside the wheels. They then form a network where everything connects to the controller.

Additionally, part of it implements an anti-blocking system (ABS) which goal is to make sure that the wheels are not blocked. The problem with wheels blocking is that the car then starts uncontrollably drift. Obviously, this is dangerous on the road and should be avoided. The reason is not enough friction and the solution then is to temporarily allow the blocked wheel to spin and so maintain the trajectory, until better surface is reached and nominal braking can resume.

More technically, the controller collects speeds of individual wheels, speed of the vehicle and pedal position. From the pedal position, it determines the desired braking torque. The blocking in turn is detected by the controller by comparing wheel speeds with vehicle speed. No wheel should be too different, otherwise it is considered blocked. Then the braking force is reduced proportionally to the speed difference until the difference reduced.

3.4.1 Modeling

Next we show how the previous authors have described part of this system. In the Listing 3.1, there are several constructs, called TADL2 [Per+12] constraints, which are then further translated into CCSL constraints. We do not give their exact definition but explain informally:

- `ReactionConstraint` places a restriction on the difference in time between arrival of stimulus event to reaction event;
- `PeriodicConstraints` defined a periodic event with desired period in seconds, which is given by some other clock;
- `SynchronizationConstraint` demands that the n^{th} occurrences of the specified events occur not far from each other: the difference between first arrival and the last one should be bounded by some fixed time duration.

To summarize, the specification demands that:

```
1  var reactionTimeMin ms on universaltime := 0.0
2  var reactionTimeMax ms on universaltime := 330.0
3  var X3 ms on universaltime := 10.0
4
5  ReactionConstraint tc1a {
6    source startBraking
7    target firstWheelBrakeActuation
8    lower = reactionTimeMin
9    upper = reactionTimeMax
10   scope pedalPositionWrite, pedalPositionRead,
11         GlobalTorqueWrite, globalTorqueRead,
12         torqFirstWheel, requested TorqFL,
13         FLABSRead, firstWheelTorqCmd
14 }
15
16 PeriodicConstraint tc3a {
17   event firstWheelSensorAcquisition
18   period = X3
19 }
20 ReactionConstraint tc5a {
21   source globalTorqueRead
22   target torqFirstWheel
23   lower = (reactionTimeMin*0.275)
24   upper = (reactionTimeMax*0.275)
25 }
26 ReactionConstraint tc8a {
27   source firstWheelTorqCmd
28   target firstWheelBrakeActuation
29   lower = 0
30   upper = (10 ms on ecu1)
31 }
32
33 SynchronizationConstraint tc10 {
34   events firstWheelBrakeActuation,
35          secondWheelBrakeActuation,
36          thirdWheelBrakeActuation,
37          fourthWheelBrakeActuation
38   tolerance = (5.0 ms on universaltime)
39 }
```

Listing 3.1: TADL2 specification of brake-by-wire requirements (from [Gok+13])

- the reaction to braking command as well as reading reply is time bounded;
- the reading of sensors is periodically recurring;
- the braking also actually happens when the command arrives, and the difference between the actuations is bounded too.

3.4.2 Discussion

A domain specific language (DSL) is a step in the right direction to cope with the complexity of describing the complex systems. The only disadvantage is that constructing one is a difficult endeavour. Tools, like GEMOC [Com+14], certainly help with this, but it is much better to not have to develop it at all. Allowing to automate most of CCSL pain points, like simple repeating patterns of the same constraints, could be an intermediate solution before committing to a DSL.

Finally, the reaction constraint is an inherently real-time related constraint. While it is defined here to use derived a logical clock, it is only a simulation and the actual relation should involve some sort of real-time, which is not possible to describe with CCSL.

3.5 Conclusion

In summary, CCSL's strong points are that it can abstract away from the exact instants when the events occur and is highly compositional, thanks to it declarative nature. But, in our opinion, it is not enough for successful description of complex systems, with a lot of details. As we saw in each, the actual specification end up pretty small. Surely, they represent the core idea, but there much more details to be specified and checked, before it could be used to aid the implementation. Thus, the next iteration of the specification language should include the following:

- features of abstraction, structuring and "macros" to save the developer's time editing and redoing what was already defined, potentially introducing errors;
- native real-time constraints: while we can describe logically the observations of real-time relations (not the relations itself), we have to do it manually, so should be solved as a part of the language;
- parameters, their manipulation and constraining as part of specification should be part of the specification itself as long as it is not too complicated. This helps with self-documenting of the specifications, but also would allow the future tools to attempt parametric verification.

CHAPTER 4

MRTCCSL

In this chapter we present the Modular Real-Time Clock Constraint Specification Language. The language is based on the Clock Constraint Specification Language and adds several extensions. The first extension is to syntactically distinguish constraints related to real-time from the others. We add the ability to relate clocks by distance in real-time and for this we refine the original formal semantics. The second extension is about how the specifications is written and composed. We introduce a notion of module and a way to capitalize on them for better scalability of the specifications. Importantly, the extensions are orthogonal to each other and can be implemented separately. We also introduce other constraints and constructs, which are required to describe the use cases.

4.1 Motivation

We draw our motivation for the language development from the use cases presented in Chapter 3. We reiterate them here, but also add a few other*. More specifically, the specification language and its tools should additionally address the following points:

- relations to real-time: in reactive systems and embedded systems in general, it is mandatory to interpret some events with connection to real-time, as they reflect changes in the environment the system is supposed to control. And changes in the environment are related to their consequences by the evolution models of the environment, which in turn are driven by time. To prevent bad consequences from happening, the system has to react before some deadline and the guarantee that it will always happen has to be checked. But it first needs to be expressed, which CCSL can only simulate;
- modularity and syntactic sugar: the same as writing individual functions is easier to reason about than plain assembly, having modules or at least some sort of parametric macros simplifies the construction of a specification by allowing to abstract and aggregate, at least mentally. Then, a well crafted and abstract enough component of one system and with a good name, can be successfully reused in other projects. But more common patterns are better to be expressed as new constructs, both for the user convenience and better analysis results as optimizations can be matched to more specific combination of constraints;
- refinement: while, to a degree, specifications can already be refined when more constraints are being added, a structural approach which provides better guarantees is needed;
- other specification properties: in a lot of cases, liveness is too strong of a property to demand, and so we think should be relaxed.

4.2 Real-time extension

The first extension we propose concerns the real-time relations between clocks. Real-time here means a unique continuous and dense dimension. As regular CCSL constraints express relations between instants of the clocks, we want to express their relation to this unique dimension. By doing so, we would not only be able to specify some of the reactive system requirements directly as they are defined, but also do it in a way that removes complexity that otherwise is needed to encode them.

In order to explain why CCSL encoding of such relations is complex, we provide the following example. We discuss the same point in Mechanical Lung Ventilator example in Section 3.2.2.

Given instant a and b, we may want to say that they happen exactly d seconds apart. From CCSL's point of view, this does not mean anything unless defined by a constraint. To define one, we first have to say that there is a reference clock r, that ticks with a period of 1 second. To express the duration of d seconds, we simply start to count d times on this reference clock when a ticks, and make b tick when these d ticks are counted. But in this setting, the duration between a and b is $d\,\mathrm{s}$, if and only if, instants a and b align on r, as demonstrated on Figure 4.1. Because from an individual point of view time can *only* be logical, it is usually not a problem. But when several

*They do not appear in the previous chapter because it is not the main point of the work to explain use cases this detailed.

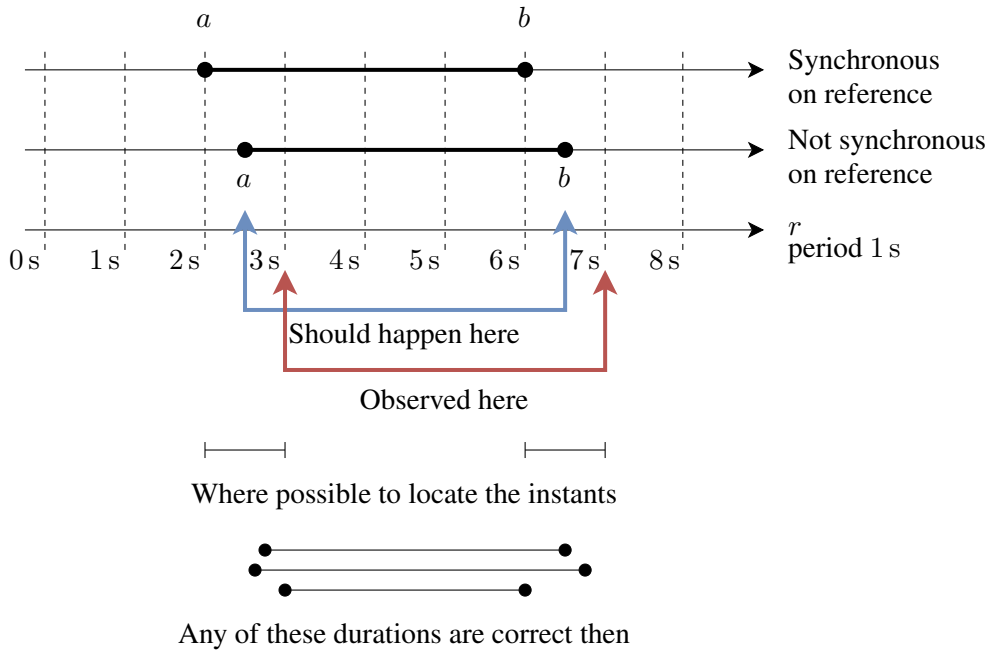


Figure 4.1: Demonstration of non-exactness of real-time in CCSL

components need to be combined and their mutual reference to the real-time is significant for the system to work, it becomes complicated. When the components rely on different and independent reference clocks, a new global reference clock with sufficiently fast period should be defined, so that each individual reference clock can be redefined in its terms. But then again it is only an exact relation when the ticks align on the global reference. This and the fact that we have to choose the precision of the reference clocks early, we consider to be suboptimal.

Thus we propose an extension (mathematically, a refinement) to the existing model in order to support real-time requirements. For this, each instant of a clock is related to that unique dimension via a tag. As the instants in a clock are totally ordered to indicate the time passing, the tags have to be strictly increasing too. When ticks of different clocks coincide, as their instant is the same, so is the tag. While this extension is heavily inspired by Timed Automata and TESL, it is not equivalent to neither of them.

It is similar to Timed Automata in regard to the unique real-time dimension. But it is different in how this time is perceived to evolve. As logical clocks are not required to keep the rates the same, the order between the instants can vary and so the values of tags, unless specifically constrained. And as we will see later, the expressiveness is not at all the same, as constraints like precedence cannot be handled by Timed Automata at all.

In relation to TESL, real-time extension can be seen as a particular case, when the tag relations are simply an identity on one domain, which is the real-time. So with this extension, we simplify its main feature of separate time scales and revert the simplifications it made to CCSL with removal of asynchronous constraints.

Next we define the concrete changes we make to CCSL under the name of Real-Time CCSL (RTCCSL). These include new constraints and backwards compatible modifications to the original semantics to support these constraints. The backwards compatibility is possible thanks to the fact

that the assumption of the set of instants in CCSL should be, mathematically speaking, is strictly strengthened by real-time tags. We start with the introduction of the new constraints and their meaning. After that we show 2 versions of the semantics: CCSL-like clock-local, named base semantics, in Section 4.2.2, and complementary time-triggered semantics in Section 4.2.3. To expand on that: in CCSL, adding a constraint on clocks $c_1, \ldots, c_n$ to a specification that already has the same clocks, means that the overall behaviour will shrink as it has to satisfy more conditions. But the conditions only act with respect to *shared clocks* between the constraints. Thus, if one would look at the contribution of each constraint to overall set of behaviour, adding a new constraint will decrease it radially, as a ripple, by propagation from a constraint to a constraint via shared clocks. In case of time-triggered mode it is no longer true. Progression in real-time caused by a time-triggered or not clock, may trigger ticking of another, time-triggered clock. Thus decisions taken in one constraint, make consequences appear in "random places" of the specification, if using the same ripple analogy. We decided to split the semantics because ultimately it depends on what we describe. If some process should be positioned to happen when time reaches the point it is set to happen, then it is time-triggered. For example, it is reasonable to assume that an electrical signal will appear after some fraction of a second on another end of a PCB trace. But in case if this clock denotes a wireless communication in noisy environment, that may be no longer true.

4.2.1 Syntax and intuitive interpretation

This extension adds four new constraints to capture patterns of timing relations that we often see in the real systems. These are real-time delay, absolute and relative periodic, and sporadic constraints. The last three are special in terms of CCSL because they define a relation on *one* clock. From a logical point of view, there is no other relation possible, as every clock is already a totally ordered sequence. But from real-time perspective, totally ordered ticks can still have different tags, and thus are further constrained, just in real-time domain.

For those new constraints, the same parameters are used: $p, d, \varphi \in \mathbb{T} = \mathbb{Q}_{\geq 0}$. Where $\mathbb{T}$ is positive set of rationals with zero and are interpreted as seconds, and $\mathbb{T}_{>0}$ is the time without zero. Others are closed intervals $[d_1, d_2]$, where $0 \leq d_1 \leq d_2 \in \mathbb{T}$. A scalar x in position, where interval is expected, is interpreted as $[x, x]$.

Real-time delay $b = \texttt{delay}\ a\ \texttt{by}\ [d_1, d_2]$ is the first constraint we add. This constraint relates i^{th} instant of the clock b, $b[i]$, to be at a distance from d_1 to d_2 from instant with the same indexed instant i^{th} on clock a, $a[i]$. In other words, the delay constrains those tags to be such that $b[i] - a[i] \in [d_1, d_2]$. Thus the delay constraint is a specialization of precedence (or causality when $d_1 = 0$) for real-time. For example, by using real-time delay we can specify an end-to-end latency of a reactive system regardless of how it defines the behaviour that leads to a reaction.

Relative periodic and **absolute periodic** constraints are the second and the third constraints added. They cover periodic relations defined in the real-time domain, in contrast to the periodic constraint of the original CCSL. With these constraints, we can explicitly define which clock is supposed to be chronometric and with which parameters. This in turn puts implicitly a relation between such clocks, which otherwise would have to be written manually and for each pair is neither perfect nor scalable to write and then analyse. Relative periodic constraint, written as $o_1 = p \cdot i^{\text{th}} \pm \texttt{rel}\,.\,[e_1, e_2] + [\varphi_1, \varphi_2]$, defines a clock o_1 with real-time period $p \in \mathbb{T}_{>0}$ and error $[e_1, e_2]$ with an uncertain offset $[\varphi_1, \varphi_2]$. In other words, it is a clock with jitter. A periodic clock with cumulative error describes the general physical periodic phenomena, like ticks of base clock inside a computer or other regular yet non-ideal processes. Declaring two relative periodic clocks

with exactly the same parameters *will* not result in identically tagged instants, unless there is no error and the offset is the same. Meaning, they desynchronize, the ticks with the same index do not occur in a constant temporal vicinity of each other.

On the contrary, **absolute periodic** constraint $o_2 = p \cdot i^{\text{th}} \pm \texttt{abs} \,.\, [e_1, e_2] + [\varphi_1, \varphi_2]$ defines a periodic clock o_2 with absolute or non-cumulative error. Thus, it is a clock with skew. The parameters of the constraint have the same meaning as in the periodic constraint with cumulative error. A periodic clock with absolute error can be used to describe ideal processes. Obviously, it is the case in reality, but we keep this constraint available for the case when only one chronometric clock is used, as it has a definition slightly simpler for analysis than with the cumulative error.

In both cases, by replacing offset with symbol ?, we can indicate that we do not care about which value the offset really has.

Sporadic constraint $a = \varepsilon$ `[non-]strict sporadic` expresses that the successive ticks of clock a have to be spaced in real-time with at least the distance of ε. The distance can be strict or not, thus there are two versions of this constraint. For example, it is reasonable to assume that a button cannot be pressed faster than with a certain frequency, yet it is not periodic and is not a result of another action, delayed by some time. It is not really a "core" constraint as others, because it can be rewritten in terms of previously defined real-time delay and CCSL itself:

$$(a \$ 1) \prec \texttt{delay}\ a\ \texttt{by}\ \varepsilon$$

We include it with its own dedicated syntax because it is commonly used in real-time reactive systems and real-time scheduling, and the definition is simpler than if defined by synchronization of the specification above.

To indicate the time-triggered mode for a constraint, we use `[tt]` suffix.

4.2.2 Base semantics

In this section we define formally the base of real-time extension, in different styles, including its integration with the original CCSL. These being denotational semantics, that expresses condition between clock objects, operational semantics, encoding progression as inference rules, and automata semantics, with automata similar to Timed Automata, and with a synchronization algorithm. The denotational semantics is basis for inductive reasoning, operational semantics for simulation and automata for abstract interpretation, all presented later in Chapter 5.

The different semantics share some notations and sets, specifically:

- $\mathbb{T} \stackrel{\text{def}}{=} \mathbb{Q}_{\geq 0}$ is the real-time domain, positive rational numbers with zero;
- then $\mathbb{T}_{>0}$ is the time domain *without* zero;
- $\mathbb{T}^{\infty} = \mathbb{T} \cup \{\infty\}$ set with ∞ symbol to express an unreachable positive number with the interpretation $\forall x \in \mathbb{T} : x < \infty$.

Additionally, we use an unusual set of intervals:

$$\mathbb{I} \stackrel{\text{def}}{=} [a, b] \mid (a, b] \mid (a, b) \mid [a, b) \mid [a, \infty) \mid (a, \infty) \mid \bot \quad \text{where } a \leq b \in \mathbb{T}$$

It is similar to the interval domain of abstract interpretation presented in Section 2.9.4, with the difference that it represents both strict and non-strict bounds and their combinations, and lacks

negative infinity side as the used time domain is only positive. As we use rationals and not integers as the basis of time, we cannot use the same trick abstract interpretation usually uses to have strict bounds in non-strict domain: previous and next number to the actual bound. The reason is of course the fact that in rationals no natural definition of next or previous number exists, thus we need the explicit encoding. Other than that, the intervals behave as one would expect open and closed intervals to behave:

$$\begin{aligned} [a,b] \sqcap (a,b) &= (a,b) & \forall i \in I : \bot \cap i &= \bot \\ [1,2) \sqcup (2,3) &= [1,3) & \forall i \in I : \bot &\subseteq i \\ (a,b) &\sqsubset [a,b] & \forall i \in I : \bot \sqcup i &= i \\ a &\in [a,b] & \forall i \in I : i &\sqsubseteq [0,+\infty) \\ a &\notin (a,b) & \forall i \in I : i \sqcap [0,+\infty) &= i \\ \forall x \in \mathbb{T} : (a,a) &= \bot & \forall i \in I : i \sqcup [0,+\infty) &= [0,+\infty) \end{aligned}$$

4.2.2.1 Denotational semantics

We base this semantics on the index-based semantics of CCSL as described in Section 2.6.2.1. The core modification to this semantics is that the set of instants I is now $\mathbb{T}$, positive rational numbers. Meaning that the partial order is subsumed by the total order $\leq_{\mathbb{Q}}$ and instants across different clocks can always be compared in this domain.

The universal property of clocks being totally ordered is augmented with a condition, that the first tick has to be bigger or equal 0:

$$\begin{aligned} \forall c \in C, i \in \mathbb{N} : & \quad c[i] < c[i+1] \\ \forall c \in C : & \quad c[0] \geq 0 \end{aligned}$$

As it was defined in Definition 2.6.8, the relation is implicitly guarded by max index $i^{\max}$ when the clock is finite. Meaning that when $i \geq i^{\max}$ there is no $c[i] < c[i+1]$ relation. We provide explicit conditions in the constraints itself when the relation disappearance is not desired.

Sporadic constraint is a derived constraint, defined through a parallel product of other constraints. Instead of a 2-level process of rewriting first into CCSL constraints and then translating into formula, we write its definition directly, given it is simple.

Thus we define the denotational semantics using the following first-order formulas. As before, these formulas are a way to encode the denotational function from constraint into the set of solutions. In the definition, the notation $< / \leq$ is used to indicate the choice between strict and non-strict sporadic constraint. The expression $i \in c$ checks existence of tick with an index i in clock c.

Definition 4.2.1 (Denotational semantics of real-time constraints)**.**

$$\begin{aligned} c_2 = \texttt{delay}\ c_1\ \texttt{by}\ d &\stackrel{\text{def}}{=} \forall i \in \mathbb{N} : (c_2[i] - c_1[i]) \in d \wedge i \in c_2 \implies i \in c_1 \\ c = p \cdot i^{\texttt{th}} \pm \texttt{rel}.e + \varphi &\stackrel{\text{def}}{=} \begin{cases} c[0] & \in \varphi \\ \forall i & \in \mathbb{N}_{>0} : c[i] \in c[i-1] + p + e \end{cases} \\ c = p \cdot i^{\texttt{th}} \pm \texttt{abs}.e + \varphi &\stackrel{\text{def}}{=} \forall i \in \mathbb{N} : c[i] \in p \cdot i + e + \varphi \\ c = \varepsilon\ \texttt{[non-]strict sporadic} &\stackrel{\text{def}}{=} \forall i \in \mathbb{N} : c[i] + \varepsilon < / \leq c[i+1] \end{aligned}$$

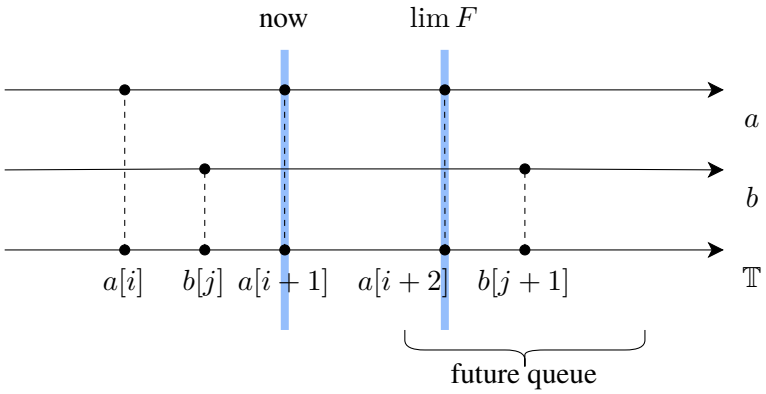


Figure 4.2: Greatest time progress to $\lim F$

4.2.2.2 Operational semantics

We base the definition of operational semantics on CCSL semantics as defined in Section 2.6.4. We cannot keep the same configuration, nor the inference rules, thus we introduce new versions first, followed with translation from old configuration and rules.

Definition 4.2.2 (Configuration)**.** Configuration Γ is a tuple $\langle H, \xi, F, \eta \rangle$, where:

- history $H : C \to \mathbb{N}$, where C is a set of clocks;
- a tag function $\xi : C \to \mathbb{N} \to \mathbb{T}$, an assignment of real-time tags to ticks of the clocks;
- current time $\eta \in \mathbb{T}$;
- bounds for future ticks, expressed as function $F : C \to \mathbb{N} \to \mathbb{I}$, indexed by clocks, with ∞ being the default value. One may think of it as a queue, and so to manipulate the queue in a simpler way, the usual operations on queues are defined:
  - $\lim F : \mathbb{T}^\infty$ is the smallest upper bound among clocks in the queue F, as demonstrated on Figure 4.2;
  - $\text{peek} : C \to F \to \mathbb{I}$ returns currently relevant bound for a clock c and is defined as $\text{peek}(c, F) = F(c, H(c))$;
  - $\text{push} : C \to \mathbb{N} \to \mathbb{I} \to F \to F$ puts a value v for index n in the queue F and is defined as $\text{push}(c, n, v, F) = \lambda(c' : C, n' : \mathbb{N}).\begin{cases} v & \text{if } (c', n') = (c, n) \\ F(c', n') & \text{otherwise} \end{cases}$;
  - there is no need to define pop of this "queue" explicitly, the operation can be done in implementation every time the pointer $H(c)$ goes up, as essentially every $i < H(c)$ in $F(c, i)$ goes out of scope given the definition of any other operation.

Then, $\Gamma \models \phi$ means that a configuration Γ is valid for the constraint ϕ.

Definition 4.2.3 (History update)**.** History update function $T : (C \to \mathbb{N}) \to \mathcal{P}(C) \to (C \to \mathbb{N})$ is a higher-order function defined as

$$H' = T(H, P) \iff \forall c \in C : H'(c) = \begin{cases} H(c) + 1 & \text{if } c \in P \\ H(c) & \text{otherwise} \end{cases}$$

Definition 4.2.4 (Partial transition between configurations)**.** Configuration transition $\Gamma \overset{(P,\Delta)}{\rightarrow} \Gamma'$, between $\Gamma = \langle H, \xi, F, \eta \rangle$ and $\Gamma' = \langle H', \xi', F', \eta' \rangle$, for a set of ticks $P \in \mathcal{P}(C)$ and $\Delta > 0$ is defined as:

- $H' = T(H, P)$;
- $\eta' = \eta + \Delta$;
- $\xi'(c, i) = \begin{cases} \eta' & \text{if } i = H(c) \wedge c \in P \\ \xi(c, i) & \text{otherwise} \end{cases}$ i.e. an update to ξ, with the tags of ticked clocks set to new current time η';
- $\forall c \in P : \eta' \in \texttt{peek}(c, F)$, a check that tags of ticked clocks appear in the right interval.

Because we did not specify what F' should be updated to, it is a partial definition. This part is the job of the constraints themselves, as well as defining when the transition can be applied, i.e. which subset of $\mathcal{P}(C)$ is allowed.

Definition 4.2.5 (Initial default configuration)**.**

$$\Gamma_0 = \langle H = \{(c, 0) \mid \forall c \in C(\phi)\}, \xi = \lambda c\, i \rightarrow \bot, F_\phi = \lambda c\, i \rightarrow \top, 0 \rangle$$

Definition 4.2.6 (Valid configuration)**.** Configuration Γ is valid if it can be constructed from the initial configuration with valid steps.

With this, we can define how the original semantics translates to the new definition. It is important for us to maintain the compatibility so that we do not have to rewrite the constraints. And because previous semantics does not use queues, the only valid configurations are possible in real-time interpretation.

Definition 4.2.7 (Mapping of CCSL semantics to RTCCSL semantics)**.** Transition between configurations $\langle X, \Phi \rangle \overset{P}{\rightarrow} \langle X', \Phi \rangle$ translates into $\Gamma \overset{(P,\Delta)}{\rightarrow} \Gamma'$ for a set of ticks $P \in \mathcal{P}(C)$ for any $\Delta > 0$.

Inference rules The rules specify how the configuration change depending on the constraint, the configuration itself and what clocks were selected to tick. For that we use transitions, partially defined before, between configurations and updates to the future assignments. Because the transition already checks for the ticks to happen in the right intervals, only the assignments differ between the constraints. As such, the generic rule is:

$$P \subseteq \mathcal{P}(C) \wedge \Delta > 0 : \frac{\Gamma \models \phi \wedge \Gamma \overset{(P,\Delta)}{\rightarrow} \Gamma' \wedge U(\Gamma, P, \Delta, \Gamma')}{\Gamma' \models \phi}$$

where predicate $U(\Gamma, P, \Delta, \Gamma')$ is specific to the constraint ϕ.

Delay For delay constraint $c_2 = \texttt{delay}\ c_1\ \texttt{by}\ d$, if parameter d is $[0, 0]$, the constraint becomes coincidence constraint $c_1 = c_2$, otherwise the update is defined as:

$$U(\Gamma, P, \Delta, \Gamma') \overset{\text{def}}{=} c_1 \in P \iff (F' = \text{push}(c_2, H(c_1), \eta' + d, F)$$

The causality of delay is then implied by two facts: the future assignment is only done given c_1 ticking and so the tag of c_2 can only be equal or strictly bigger than the one of c_1, depending on the d_1 value.

Absolute periodic Absolute periodic constraint $c = p \cdot i^{\text{th}} \pm \texttt{abs}.e + \varphi$ requires two modifications to the generic rules. In case when the clock has to start at the specific time φ, the initial configuration has to be modified:

$$F = \text{push}(c, 0, \varphi, F)$$

Otherwise, the assignment stays empty by default, meaning there are no restrictions on when the first tick can occur. As for the update, it pushes a restriction on the next tick of the same clock that triggered the update in the first place. The condition of the tag is in turn defined as a relation on the number of tick itself, meaning that the error does not grow with progression, giving the name absolute to the constraint.

$$U(\Gamma, P, \Delta, \Gamma') \stackrel{\text{def}}{=} c \in P \iff \text{push}(c, H'(c), p \cdot H'(c) + e + \varphi, F)$$

Relative periodic Relative periodic constraint $c = p \cdot i^{\text{th}} \pm \texttt{rel}.e + \varphi$ is similar to absolute, as it features the same initial configuration change:

$$F = \text{push}(c, 0, \varphi, F)$$

But the update is slightly different, as instead of defining the restriction as relation to the index of the tick, the tag is *relative* to the previous one.

$$U(\Gamma, P, \Delta, \Gamma') \stackrel{\text{def}}{=} c \in P \iff \text{push}(c, H'(c), \eta' + p + e, F)$$

Sporadic constraint As for sporadic constraint $c = \varepsilon$ `[non-]strict sporadic`, it is really similar to relative periodic constraint, with an exception that it does not have an upper bound on the next tick. Thus, the update is defined as, where $[\varepsilon, +\infty)$ is replaced by $(\varepsilon, +\infty)$ for the strict variant of the constraint:

$$U(\Gamma, P, \Delta, \Gamma') \stackrel{\text{def}}{=} c \in P \iff \text{push}(c, H'(c), \eta' + [\varepsilon, +\infty), F)$$

Synchronisation Now that we know how individual constraints can be rewritten, we present how to make a step on the whole specification Φ. For that both precondition to the rules and their results should result in non-empty sets, formally defined as:

$$\frac{\begin{cases} \forall \phi \in \Phi : & \Gamma_\phi \models \phi \\ \forall \phi \neq \psi \in \Phi : & \eta_\psi = \eta_\phi = \eta \\ \forall \phi \neq \psi \in \Phi, c \in C(\psi) \cap C(\phi) : & \begin{cases} H_\psi(c) = H_\phi(c) \\ \forall i \in \mathbb{N} : \begin{cases} \xi_\psi(c,i) = \xi_\phi(c,i) \\ F_\psi(c,i) \cap F_\phi(c,i) \neq \emptyset \end{cases} \end{cases} \end{cases}}{\langle \bigcup_{\phi \in \Phi} H_\phi, \bigcup_{\phi \in \Phi} \xi_\phi, \{\bigcap_{\phi \in \Phi} F_\phi(c,i) \mid \forall c \in C(\Phi), \forall i \in \mathbb{N}\}, \eta \rangle \models \Phi}$$

The interpretation of the formula is the following: for constraints to be eligible for synchronization, they need to:

- to have a valid configuration Γ;
- be at the exactly same point in real-time η;

- and for constraints that share clocks, their histories in these clocks needs to be the same, up to the assignments of tags ξ, as well as the bounds on the future ticks F should be compatible.

Then the synchronization is performed as union of histories and tag assignments. To do it, we interpret the function as a relation just in this case. Because we checked before for conflicts, the result will again be a function. The future bounds are unified similarly, with exception that the bounds are intersected to find a bound that satisfies all the constraints. Again, because we checked before that this will not result in empty set, which is not permitted, the function stays correct.

4.2.2.3 Automata

In case of automata semantics, each constraint is defined as the corresponding real-time augmented CCSL automaton.

Definition 4.2.8 (Real-time augmented automaton)**.** Real-time augmented automaton A is a tuple $\langle P, C, V, Q, T\rangle$, where:

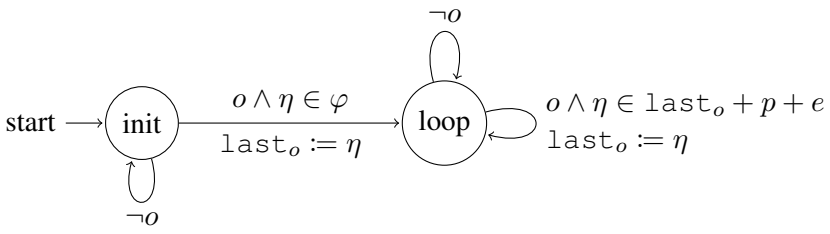
- locations L, clocks C, variables V, queues Q are sets of symbols;
- S is a set of variable evaluations in a location, $S = (V \to \mathbb{T}_\perp) \times (Q \to \mathbb{N} \to \mathbb{I}))$;
- $l_0 \in L$ is the initial location, $s_0 = (\lambda v.\ \perp, \lambda q\ i \to \perp)$ is the initial state;
- $T \in L \times S \times \mathbb{T} \times \mathcal{P}(C) \to L \times S$ is a transition function from current time, location, its variable evaluation, clock label and into the next location with new evaluation of state and current time.

While pretty general, the *transition functions* in our case are defined in two parts and with specific languages.

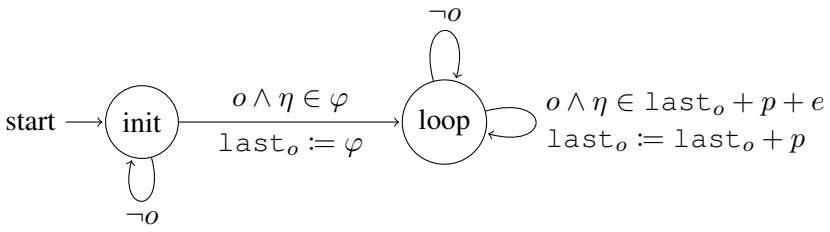
First is the *guard* that describes the valid label, and is a Boolean expression with atoms being either clocks, inclusion tests, tests of queue emptiness or their propositional combination. If clock atom is set to be true then the related clock ticks, and vice-versa. The inclusion tests are expressed as $a \in i$, where i is an interval defined before and can be expressed as constant or a linear combination of other expressions, like head of the queue $\text{head}(q), q \in Q$. A constant or a variable e is implicitly transformed into an interval $[e, e]$ if used in interval context. Then the atom a can be either current time η or a variable $v \in V$. If an expression evaluates to a scalar r, but we expect an interval, we interpret it as an interval $[r, r]$. A queue emptiness test is $\text{head}(q) = \perp$ for some $q \in Q$.

The *assignment* is a set of real-time variable and queue updates. A variable assignment $v := e$ 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. While operational semantics is not required to have pop operator because old values go out of scope automatically, here we have to use it. $\text{push}(q, e)$ 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.

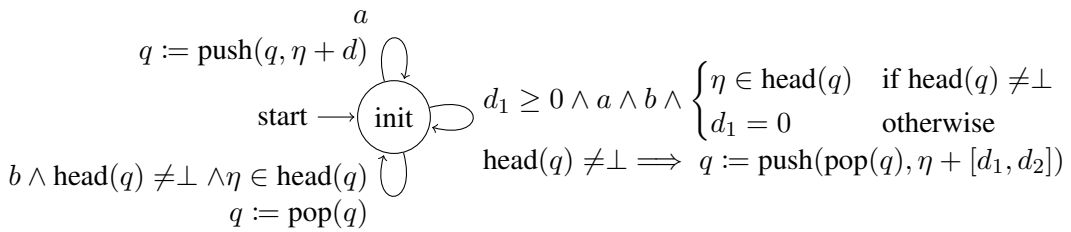
We describe the new constraints as automata on Figure 4.3. The guard of the transitions is given above the arrow and the assignment is below.



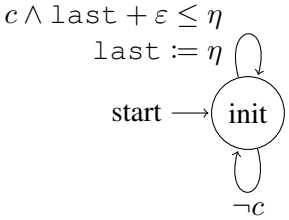
(a) Relative periodic $o = p \cdot i^{\text{th}} \pm \texttt{rel}.e + \varphi$



(b) Absolute periodic $o = p \cdot i^{\text{th}} \pm \texttt{abs}.e + \varphi$



(c) Real-time delay $b = \texttt{delay}\ a\ \texttt{by}\ [d_1, d_2]$



(d) Sporadic constraint $c = \varepsilon$ `[non-]strict sporadic`

Figure 4.3: Real-time augmented automata

Definition 4.2.9 (Real-time augmented automaton run)**.** A *run* is an alternating sequence of rules (similar to Timed Automata, more in Section 2.4):

- time elapse: $(l, s, \eta) \xrightarrow{\delta} (l, s, \eta + \delta)$;

- transition: $(l, s, \eta) \xrightarrow{P} (l', s', \eta)$ such that $\exists P \in \mathcal{P}(C) : T(p, s, \eta, P) = (p', s')$.

We then consider a *schedule* 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 $\sigma : \mathbb{N} \to \mathcal{P}(C)$ to $\mathbb{N} \to \mathcal{P}(C) \times \mathbb{T}$.

Definition 4.2.10 (Automata synchronization)**.** A synchronized real-time augmented automaton A of automata A_1 and A_2 is the tuple $\langle P \subseteq P_1 \times P_2, p_0 = (p_{01}, p_{02}), C = C_1 \cup C_2, V = V_1 \cup V_2, Q = Q_1 \cup Q_2, T = T_1 \otimes T_2 \rangle$. Synchronization of transition functions T_1 and T_2 is defined as

$$T_1 \otimes T_2 \stackrel{\text{def}}{=} \lambda((l_1, l_2), (s_1, s_2), \eta, P) \to$$
$$\text{extend}(T_1(l_1, s_1, \eta, P), L_2 \times S_2) \cap \text{extend}(L_1 \times S_2, T_2(l_2, s_2, \eta, P))$$
$$\text{extend}(D_1, D_2) \stackrel{\text{def}}{=} \{((l_1, l_2), (s_1, s_2)) \mid (l_1, s_1) \in D_1, (l_2, s_2) \in D_2\}$$

where $C_i \subseteq C$ and extend makes a point-wise Cartesian product of the two transition sets.

The essence of the synchronization is that only transitions from both automata that are left, are the common ones, i.e. valid for both constraints. This is achieved by first obtaining the destination location and variable evaluation, then with Cartesian product we bring both sets to the same domain and intersect to satisfy both.

To synchronize with the regular CCSL, the automata need to be translated into the new representation. Automata stay mostly the same as described for unbound automata in Section 2.6.3, but with addition of empty sets for real-time variables and queues. Transitions are a subset of real-time transition functions and so can be synchronized as described above. Symbolic CCSL automata and real-time augmented automata are orthogonal to each other, because while symbolic ones try to encode the integer domain, real-time ones encode the real-time domain. In the final version, where we try to do the analysis on the whole language, we use both as demonstrated on Figure 4.4.

4.2.3 Time-triggered mode semantics

As it was explained in introduction of the extension (Section 4.2), the purpose of time-triggered semantics is to bind tag relation on clocks both ways. Practically, it means that real-time progress made with one clock may not succeed because of constraint placed on another clock. For example, relatively periodic constraint defines a clock, ticks of which occur with a certain frequency. If we use two of them, then by observing one we can predict how many ticks has occurred on the other one, because in this particular case, the progression of time itself implies the occurrence of the other clock. In other words, a real-time clock is live as long as the real-time progresses.

Yet it is not the same as the liveness (Definition 2.6.17). Liveness says that certain clocks have to be infinite in all possible schedules, while such time-triggered semantics expects ticks just not later than some time. For example, a real-time delay only requires that the delayed clock eventually ticks, not later than the deadline, and it does not mean it has to be infinitely occurring.

We define formally what it means in the case of real-time CCSL by putting additional conditions on the base semantics presented before in Section 4.2.2.

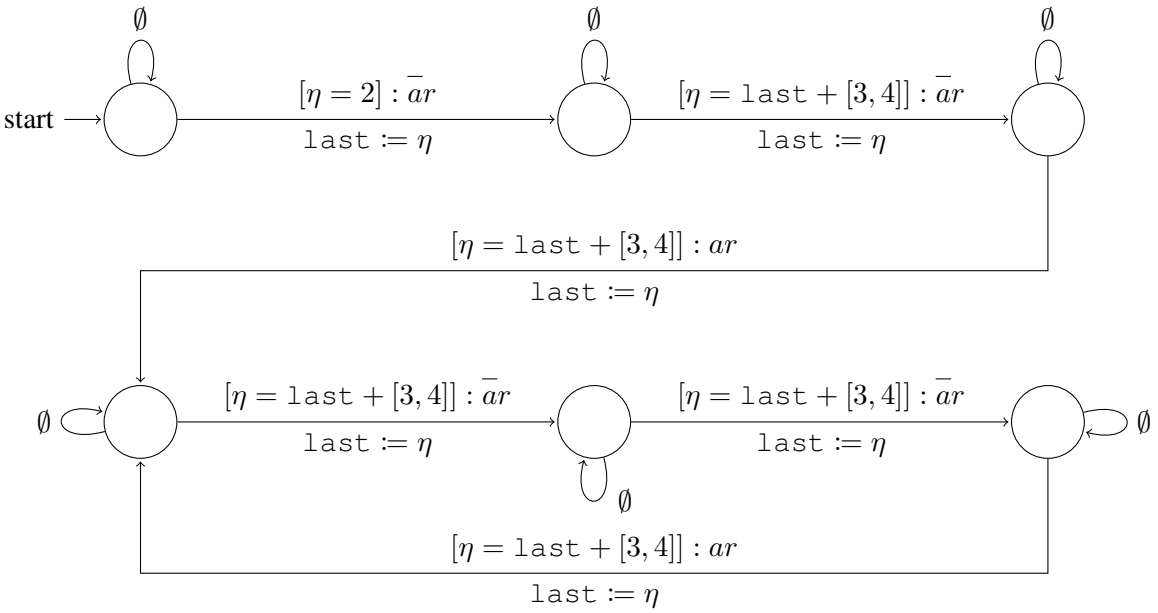


Figure 4.4: Synchronized automaton of $a = \texttt{every}\ 3\ r$ and $r = 3.5 \cdot i^{\text{th}} \pm \texttt{rel}\,.\,[-0.5, 0.5] + 2$

4.2.3.1 Denotational semantics

In the following sections we provide the semantics of real-time constraints with real-time to tick implications. We additionally define a function $\max I : \mathbb{T}^\infty$ i.e. the maximum element in the set if exists. In the modified semantics, we only list additional condition, previous definition is denoted as O (old).

$$c_2 = \texttt{delay}\ c_1\ \texttt{by}\ [d_1, d_2]\texttt{[tt]} \quad \stackrel{\text{def}}{=} \quad O \wedge \forall i \in \mathbb{N} : \max I \geq c_1[i] + d_2 \implies i \in c_2$$

$$c = p \cdot i^{\text{th}} \pm \texttt{rel}\,.\,e + \varphi\texttt{[tt]} \quad \stackrel{\text{def}}{=} \quad O \wedge \forall i \in \mathbb{N} : \begin{cases} x = \begin{cases} \varphi_2 & \text{if } i = 0 \\ c[i-1] + p + e_2 \end{cases} \\ \max I \geq x \implies i \in c \end{cases}$$

$$c = p \cdot i^{\text{th}} \pm \texttt{abs}\,.\,e + \varphi\texttt{[tt]} \quad \stackrel{\text{def}}{=} \quad O \wedge \forall i \in \mathbb{N} : \max I \geq p \cdot i + e + \varphi \implies i \in c$$

$$c = \varepsilon\ \texttt{[non-]strict sporadic[tt]} \quad \stackrel{\text{def}}{=} \quad O$$

We do not modify the sporadic constraint because it only constraints the lower bound of the difference.

4.2.3.2 Operational semantics

It is enough to update the partial transition $\Gamma \to \Gamma'$ (defined in Definition 4.2.4) with the condition:

$$\Gamma \to \Gamma' \coloneqq \Gamma \to \Gamma' \wedge \forall \phi \in \Phi : \eta + \Delta \leq \lim F_\phi$$

It means that the time can progress at most to the smallest upper bound on tag among all bounds of ticks in all constraints, for which there is a restriction.

4.2.3.3 Automata semantics

Because in automata we do not have a central place where we store the relation on tags, as these are split into state in the location itself and an expression with a relation in transitions from the location, it is a better idea to modify the automata itself to check the condition. For this we have to add to the automaton a special location, say $l_e \in L$, which should not be visited. We call it error location. The condition when it should not be visited is defined with the respective transition to it. Then the requirement becomes local to the constraint, and propagates to the whole of specification via synchronization. Synchronization though, has to be modified to accommodate this special error location. In the translation from regular CCSL automata, the error state is never reachable.

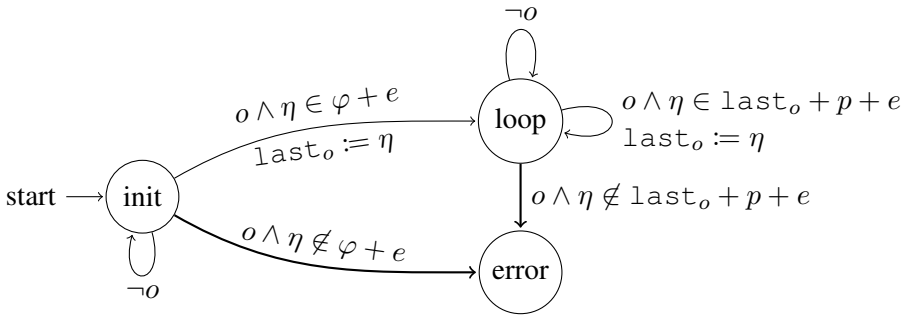
Definition 4.2.11 (Real-time augmented automaton synchronization (with error))**.** A synchronized real-time augmented automaton with error A of automata A_1 and A_2 is the tuple $\langle L \subseteq L_1 \times L_2 \cup \{l_e\}, l_0 = (l_{01}, l_{02}), l_e \in L, C = C_1 \cup C_2, V = V_1 \cup V_2, Q = Q_1 \cup Q_2, T\rangle$, where T is synchronized as was defined before with only modification that transitions leading to any of error locations from A_1 and A_2 lead to the new error location regardless:

$$T(l, s, \eta, P) \stackrel{\text{def}}{=} W((T_1 \otimes T_2)(l, s, \eta, P))$$

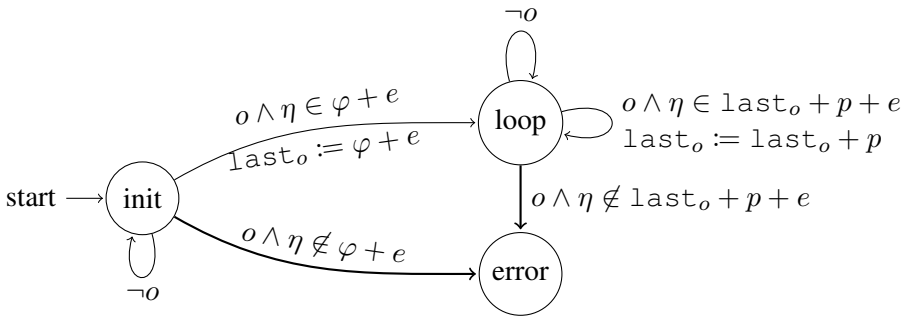
$$W(D) \stackrel{\text{def}}{=} \left\{ \left(\begin{cases} l_e & \text{if } l_1 = l_{e1} \vee l_2 = l_{e2} \\ (l_1, l_2) & \text{otherwise} \end{cases}, s \right) \;\middle|\; ((l_1, l_2), s) \in D \right\}$$

Then 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 l_e is never visited.

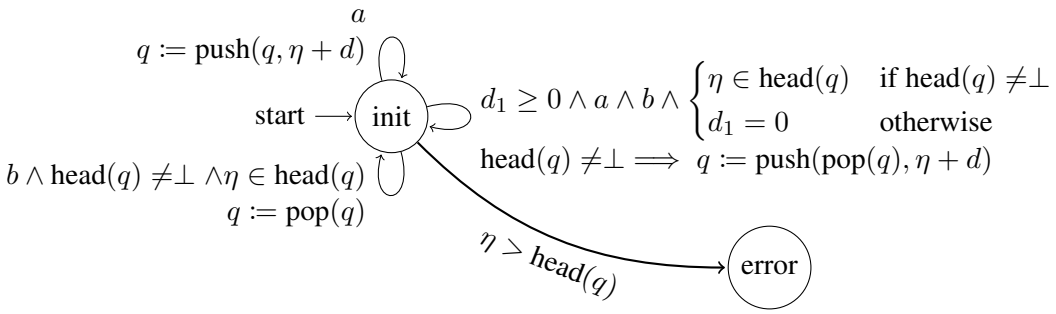
The respective automata for constraints are presented on Figure 4.5. It is important to understand, that the error location is only for violations of time-triggered behaviour.



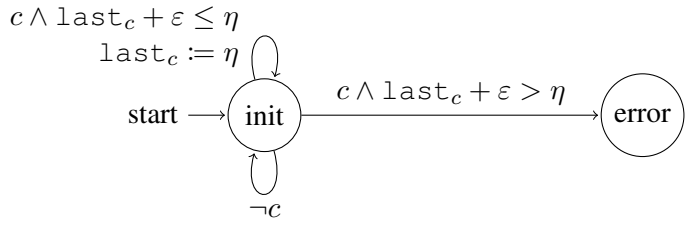
(a) Relative periodic $o = p \cdot i^{\text{th}} \pm \texttt{rel}.e + \varphi\texttt{[tt]}$



(b) Absolute periodic $o = p \cdot i^{\text{th}} \pm \texttt{abs}.e + \varphi\texttt{[tt]}$



(c) Real-time delay $b = \texttt{delay}\ a\ \texttt{by}\ [d_1, d_2]\texttt{[tt]}$



(d) Sporadic constraint $c = \varepsilon$ `non-strict sporadic[tt]` (flip comparisons $\leq \leftrightarrows <$ in strict sporadic)

Figure 4.5: Real-time augmented automata with error location

4.3 Parameters and their constraints

It is a common situation for systems to not operate under the same assumptions in all situations. It is also normal that during the design phase, developers are not aware of all values for all the constraints. This also quite natural to expect a language with analysis tools to provide some of these values, their relationships or at least some acceptable ranges. To be able to express these, we propose an extension about parameters and their constraints.

First of all, parameters are additional variables from the point of view of the language (*not* semantically, more in Definition 4.3.1). Before, each variable had to be a *clock*, thus this extension brings more types to the language of CCSL, more specifically, now it is the set $Sort$ containing:

- clocks $Clock$;
- $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$, i.e. domains of naturals, integers and rationals;
- the domain of real-time $\mathbb{T}$, internally positive rationals $\mathbb{Q}_{\geq 0}$;
- Hz is a domain of frequencies, inverse of time.

If parameter of another type has to be used, first it has to be converted into appropriate type with a conversion:

- integers and natural numbers can be freely used in rational expressions;
- rationals can be converted to time using SI prefixes for second i.e. s, ms, us, etc., plus common minutes m, hours h, years y.
- rationals can convert to frequency with Hz, MHz, GHz, etc. or by division of frequency variables;
- rationals translated to integers with ceil, floor or near functions;
- integers translated into naturals with absolute $|x|$.

As for the language of the numerical expressions, full arithmetics is permitted. Same with the relations, any of $<, \leq, =, \neq, >, \geq$ are allowed. Additionally, for the sake of convenience, the implementation of the language should allow to pass the frequencies in the periodic real-time constraints, as well as specify error as parts per million (ppm), the general characteristic of an oscillator. The conversion is simple: frequency f translates into period as $\frac{1s}{f}$, $n$ppm for frequency f to the error $e = \frac{1s}{f} \cdot [-n \cdot 10^{-6}, n \cdot 10^{-6}]$.

As we allow full arithmetics, we are presented with the problem of undecidability of arithmetics with multiplication and division. While it is true, we do not want to syntactically limit the expressiveness because some of the cases do have a solution or can be emulated to have an analytical solution. For example, in the system $1 \leq a \leq 2 \wedge 3 \leq b \leq 4 \wedge c = a * b$, if parameters a, b are only used in the last relation, they can be inlined, leading to $3 \leq c \leq 8$, while other parameters can be reconstructed later. Thus the result depends on what expressions are used and which backend is available. Same goes with the conversions and what types are available in the underlying solver.

Additionally, the scheduling problem for such specifications is defined differently.

Definition 4.3.1 (Schedulability with parameters)**.** For a specification $S : (V \rightarrow \cup Sort) \rightarrow R(C)$, a function that returns a specification using an assignment of parameters $V \rightarrow \cup Sort$, where $\cup Sort$ is any value in any type, the schedulability problem is defined as:

$$\exists v : \Sigma(S(v)) \neq \emptyset$$

I.e. it is a question of finding the assignments to parameters for which a regular CCSL specification has schedules. We use $\cup Sort$ only to simplify the definition, assignments that return values of different types for the same parameters are not allowed. Thus, it could be replaced by a tuple of disjoint functions, each mapping parameters to values of different and strictly specified types.

Thus, a specification is schedulable if it is possible to find one assignment of parameters and for which there exist some valid schedule. From the point of view of an individual schedule, constraints have only constant arguments.

4.4 Modular framework

Regular CCSL specifications are really "flat": it is a list of constraints. Not only it is difficult to read, but it is difficult to reuse as names of clocks have to be unique across the whole specification. Additionally, long lists of statements are not easy to understand, neither in programming languages, nor in CCSL. We would even argue that it is much more complicated to write CCSL, as the behaviour emerges from the combination of constraints as a whole and not from the sequence of actions. Meaning, that limiting the size of specifications and making their interfaces as concise and simple as possible is critical.

With this in mind, we have developed a second big, after real-time, extension to CCSL: modules. In this section we explain what modules are and how everything to this point integrates with them. We also define how they should be interpreted standalone and connected to each other. Also, we defined rules when a property analysis can be scaled and what properties are supported and why.

4.4.1 Syntax

It is important to understand that the modules defined below are not what is seen in the code. From the point of view of the code, a module is a function from parameter names to lists of constraints. When such function is called it creates a module instance, which is what we describe here.

The code itself can consist of standalone signatures (interface) definitions and modules without any interface. Then, it is possible to reuse the same interface for different "implementations", by calling a module under the signature.

4.4.2 Modules

Before modules, we want to talk about more basic construct: a block. The purpose of a block is to separate constraints from each other and clocks declared alongside them. It does it by prefixing each clock with its name. Blocks and clocks defined outside the block can be accessed with

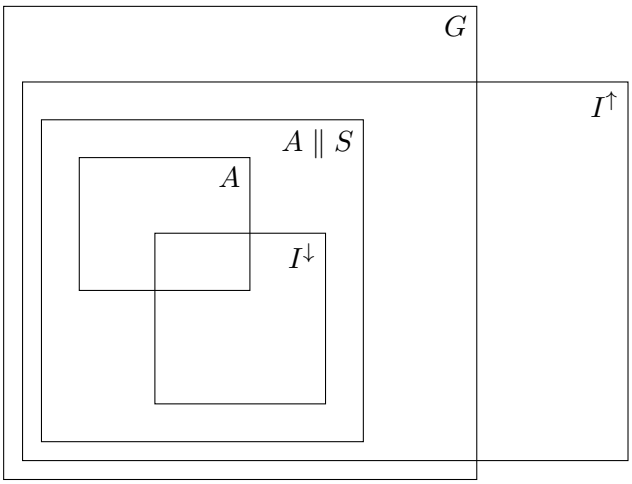


Figure 4.6: Set-based illustration of relations between module parts

previous scope operator \$ and delimited with a dot. A basic block then is interpreted as such:

$$\begin{array}{rcl} a = c \sqcup b & & a = c \sqcup b \\ \text{prefix1} = \{\$.a \prec d\} & \equiv & a \prec \text{prefix1}.d \\ \text{prefix1}.d \prec e & & \text{prefix1}.d \prec e \end{array}$$

In the translated specification, dot becomes part of the name and \$ disappears. Alternating dot with scope operator allows to access increasingly wider scope, up to the root one.

A module instance is a tuple $M = \langle A, S, G, I \rangle$ meaning assumption, structure, assertion (guarantee) and interface respectively. The interface consists of upper $I^{\uparrow}$ and lower $I^{\downarrow}$ parts. We use module and module instance interchangeably, but make it precise when it matters. Modules body B is at least the specification $A \parallel S$. We redefine it later when we build connections with other modules.

The meaning of each part is defined as (alternatively, see Figure 4.6):

- structure S is the behaviour of the desired system;
- assumption A specifies something that should hold regardless of the system, example: oscillator ticks infinitely often, with defined period and error, the behaviour of the system should not constraint any of these parameters;
- assertion G checks that the module does not violate it;
- interface I checks that the module contains at least the specified behavior to occur or satisfies another specification.

In other words, assertion is used to double check the behaviour of the specification, assumption is something that user cannot control and so change, while interfaces is under and overapproximation of its behaviour. All the parts, except of structure can be omitted. In that case it is the same as regular CCSL.

Subspecification relation is the key to the definition of the modules.

Definition 4.4.1 (Subspecification relation)**.** For specifications S and P, with common clocks $X = C(S) \cap C(P)$, subspecification relation $S \Subset P$ is defined as:

$$S \Subset P \iff \Pi_X(\Sigma(S)) \subseteq \Pi_X(\Sigma(P))$$

Meaning that it is a simulation relation, with transition systems from the specifications projected over common clocks only. If the set of common clocks is empty, the relation is trivially true. Additionally, the relation does not change its meaning for real-time extension as real-time is only a refinement of logical-only time, from the perspective of language and instant set it is defined on.

Another way to see subspecification relation is as a simulation. Just as a reminder, for transition systems $\mathcal{S} = \langle S, \Sigma, \rightarrow \rangle, \mathcal{T} = \langle T, \Sigma, \rightarrow \rangle$, where S, T are states, Σ is an alphabet of actions, relation $R \subseteq S \times T$ is a simulation when $\forall (s,t) \in R, \forall s \xrightarrow{a} s', \exists t \xrightarrow{a} t' : (s', t') \in R$ (we borrow the definition from [KM02]). Thus subspecification relation is then a simulation between the transitions systems representing the specifications S, P in $S \Subset P$ while projected onto "actions" $C(S) \cap C(P)$.

The semantics of a single module is to check the following relationship holding (we do not care about the relation if part is not defined, i.e. the set of constraints is empty), B is body defined as $A \parallel S$:

- assumption A:
$$|A| \neq 0 \implies A \Subset B$$
- assertion G:
$$|G| \neq 0 \implies B \Subset G$$
- interface $I = (I^{\uparrow}, I^{\downarrow})$:
$$|I^{\uparrow}| \neq 0 \implies B \Subset I^{\uparrow}$$
$$|I^{\downarrow}| \neq 0 \implies I^{\downarrow} \Subset B$$

Plus, usually we would want $(|X| \neq 0 \implies \Sigma(X) \neq \emptyset)$ to hold, for all $X = \{A, S, G, I, B\}$, as most probably the intention of the user was not to define an empty behaviour. If these conditions are satisfied, then the module is called correct.

Definition 4.4.2 (Subspecification relation with parameters)**.** For a specification $S : (V \rightarrow \cup Sort) \rightarrow R(C)$, a function that returns a specification using an assignment of parameters $V \rightarrow \cup Sort$, where $\cup Sort$ is any value in any type, the subspecification relation is defined as:

$$S \Subset P \iff \forall v : S(v) \neq \emptyset \implies S(v) \Subset P(v)$$

Additionally, a module can be parametrized and so be called from other modules with expressions involving other parameters. Further, the language is extended with module calls denoted as $M(a_1, a_2 \dots)$ where a_i are arguments of type $T_i \in Sort$. Of course, from the point of view of the specification inside, these are parameters. The type of types $Sort$ is additionally extended with type of blocks $Block$. This type then allows us to do structural polymorphism, similar to OCaml's polymorphic variants except not explicitly typed. Another way to think of, is as it is a *prefix* to some clocks. The access to them can be performed by using the variable of type $Block$ and dot operator we saw before. Additionally, if a modules is passed as a parameter, it is treated

as another block. Parameters of $Block$ type can be dereferenced with . operator. The right side is then either block type again or a clock, depending if it is the last application of the operator. The correctness of dereferencing is checked sequentially though the specification listing, i.e. if a module is passed to another module, extended with new clocks after, the called module will not see those new clocks.

4.4.3 Intermodule semantics

The semantics boils down to simple embedding: when a module call another module, it is embedded into the body of the parent module. Thus, we redefine the definition of the body: now it is $A \parallel S \parallel X_1 \parallel \cdots \parallel X_n$, where X_i is some part of another module. Depending on if interface is available to that module and which property we are trying to prove, there are two ways to select X_i: one is exact and the other is abstracting. First is canonical and the second one is designed to help with scalability, by also being a way to do refinement.

It is also forbidden to call the modules recursively as the abstractions break in a cycle. Theoretically, if the recursive call is made with the same parameters the cycle can be broken if the recursive module is abstracted with an interface, i.e. it is a specification fixpoint. If an abstraction is not available, it would require all the specifications in the cycle and their dependencies to be merged into one. It is potentially a really big specification which is exactly what we try to avoid with modules. If there is a behaviour that relies on each other, then it should be placed in the same module, and such the recursive dependencies between modules are considered a bug.

4.4.3.1 Dependency tracking graph

Dependency tracking graph is an intermediate data structure to be used to provide iterative development with modules. Inclusion of modules defines a graph of proofs to be made in order for the verification to succeed. This can be done separately which makes the overall proof easy to execute in parallel and therefore scalable. Additionally, when a part of module changes, we regenerate part of the graph. If it does not have hard dependencies (inclusion into other modules), we have to prove only relations dependent on the changed part. It is still requires to traverse the source code though, but comparing to the complexity of the proofs it is still overall beneficial.

We define directed tree specifically because instantiation of modules always produces different modules unless they are called with the same parameters and have the same name in the parent module. But in that case they are literally the same. The parameters are passed to the modules and their definitions can be traced, which is the easiest way to differentiate their equivalence. Another way is to use abstract interpretation or some normalization, but because of how complex the expressions can be, we can only guarantee the syntax equivalence.

Definition 4.4.3 (Dependency tracking graph)**.** Dependency tracking graph G is a directed tree $\langle M, m_0 \in M, D, e\rangle$ with no restriction on arity, where vertices M are a set of module instances, m_0 is the root module, $D : M \times M$ is a set of directed edges representing module dependencies with labeling function $e : D \to Part$ specifying on which part it embeds, with $Part = \{A \parallel S, I\}$.

On change to any part of a module, reevaluation to the graph is propagated the following way: if part has changed, any subspecification relation it is in has to be reevaluated. If another module embeds modified part, its body is modified too and the propagation continues. Then, if a module

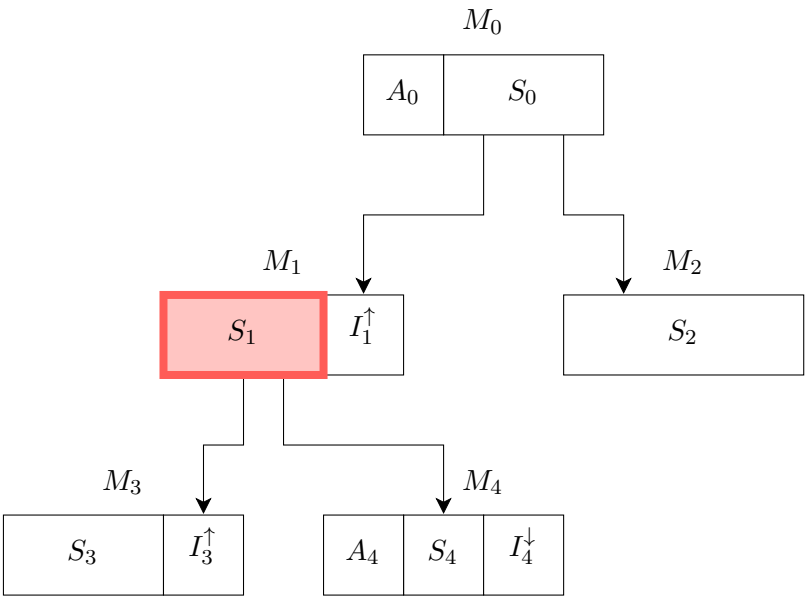


Figure 4.7: Dependency traking graph with a change in red

depends on interfaces of other modules, and not the body, the changes and so reevaluation are limited to two modules, the modified and the dependent if change is in the interface, and to one module if the changes is to any other part. The reevaluation progresses from the affected module to root in the worst case.

The propagation is not the same when parameters are used. Any module using a parameter has to be embedded. It has to do with the fact that the parameters are considered bidirectional and so constraints introduced in one part (in the whole specification) can make other part unschedulable. This makes the analysis non-local, thus triggering embedding. The embedded modules then solve the parameters as defined in Section 4.3.

An exception is when the parameter is a constant, then it can be essentially eliminated. But changing it still would trigger reevaluation, because the module instance itself changed.

An abstract example of change with non-propagation of reevaluation is demonstrated on Figure 4.7. The change on the figure results in the following relations being reevaluated, but not the whole system:

$$\begin{aligned} B_1 &= S_1 \parallel I_3 \parallel S_4 \\ \Sigma(B_1) &\neq \emptyset \\ B_1 &\Subset I_1^{\uparrow} \end{aligned}$$

4.4.3.2 Analysis semantics

Given the graph these are the rules to construct the result, the body of the root module, on which then a property can be checked. By following the rules from leaves to root in the graph, it should be possible to obtain the result in parallel or even iterative manner. It does depend on actual connections though, as only dependencies on interface do not require to go deep into the graph

on iterative changes. Checks for emptiness and subspecification relations of correctness can be parallelized inside the modules itself.

To do this, we define the following mutually recursive functions, that try to inference the module's body, the actual list of constraints for each module instance, with the end goal to construct root modules' body. $E : D \to R(C)$ is the embedding selected by the graph, $V : M \to \mathbb{B}$ is the predicate for the check of module correctness (validity), $B : M \to R(C)$ returns the body of the module with embedded parts, thus B_v is defined only if the module m is correct.

$$E(m_x, m_y) = [\texttt{raises error when } \neg V(m_y)] \begin{cases} I^{\uparrow}_{m_y} & \text{if } e(m_x, m_y) = I^{\uparrow} \\ I^{\downarrow}_{m_y} & \text{if } e(m_x, m_y) = I^{\downarrow} \\ B_v(m_y) & \text{otherwise} \end{cases}$$

$$B(m_x) = A_{m_x} \parallel S_{m_x} \quad \Big\| \quad E(m_x, m_y)$$

$$B_v(m) = B(m)[\texttt{raises error when } \neg V(m)]$$

$$\begin{aligned} V(m) = {} & |S_m| \neq 0 \\ \wedge \; & |A_m| \neq 0 \implies A_m \Subset B(m) \\ \wedge \; & |G_m| \neq 0 \implies B(m) \Subset G_m \\ \wedge \; & |I^{\uparrow}_m| \neq 0 \implies B(m) \Subset I^{\uparrow}_m \\ \wedge \; & |I^{\downarrow}_m| \neq 0 \implies I^{\downarrow}_m \Subset B(m) \end{aligned}$$

We have opted to use errors à-la exceptions here to simplify the definition and because they are not actionable, there is nothing we can do to fix them in the algorithm itself, and so the procedure stops when one error occurs.

4.4.3.3 Non-abstracting case

Let us look at specifications that do not contain any other part than structure. This is the case of classic CCSL specifications. When such specification is called from another one, the module is unpacked as a block in the one, then the block unpacks with renaming.

For example specification M_1 calls $m2 = M_2()$ and $m3 = M_3()$, the final constraints of first module would be $S_1 \parallel S_2 \parallel S_3$, minus variable manipulation or renaming to bring them into the same namespace. Such procedure is then repeated for every level from leaves to the top level module, producing a completely flat specification on which the properties are checked. The correctness of each module should still be checked before embedding. It would produce gigantic specifications though, so we propose another method. There is at least one advantage over regular CCSL though, as it is possible to check various things on each module separately.

4.4.3.4 Abstracting case

If module contains a fully defined (two part) interface, then the call to that module is replaced by it, depending on which property is supposed to be checked:

- schedulability: underapproximation part;
- emptiness (inverse of schedulability): overapproximation part;

- liveness: underapproximation, but it can only guarantee existence not exactness; which may be enough to prove weak-liveness, which is usually what one wants anyway. We define what weak-liveness is later in Section 4.6.1.

It is done this way to guarantee soundness of the analysis despite updating the specification iteratively. If the task is to check that the root module has the property, it is enough to check it on it only, given that the root body was returned without errors.

Proposition 4.4.1. *Property of emptiness is sound when using overapproximation of interface in modular analysis.*

Proof. As an overapproximation of interface is a sound approximation of the child's module structure with its dependencies, meaning it represents more solutions that there actually are in the module, if the synchronization between parent module and the child's interface results in empty set, then it would have resulted in empty set without the abstraction. □

The propositions and proofs for liveness and schedulability are analogous. Important detail is that the liveness needs an underapproximation and cannot be proved exactly with an abstraction, only its existence of live or weak-live schedules. As for schedulability, the abstraction depends on which one we are trying to prove, existence of schedules or non-existence, with underapproximation and overapproximation respectively.

4.4.3.5 Finiteness of representation

Unfortunately, in case of finiteness of representation, such scheme is not applicable. The reason for that is that the subspecification relation is not monotonic relative to finiteness. In simpler words, the subspecification has nothing (explicit) to do with the state that this property is concerned with, and so the check of finiteness is not sound with the abstraction interface provides. For example, schedules of periodic constraint are accepted by precedence, or c = fastest(a,b) is accepted by c subclocks (a + b), but replacing one by another would not be correct in proving finiteness.

Thus, to prove the finiteness, either we need to flatten the modules into one or go through each module separately. If we can check that every module is finite on its own, when every one is, their synchronization is too, but not vice-versa.

4.4.4 Discussion

We would like to address two general points on the usage of the modular framework. These are refinement and deduplication of code respectively.

4.4.4.1 Refinement

Modular framework allows us to use it as a form of refinement. By using interfaces one can make a hierarchy reminiscent of Event-B (Section 2.5). The difference is that the user does not have to prove anything, the subspecification relation is supposed to conclude if the refinement is correct. As we understand from the definition of the relation (Definition 4.4.1), it is not decidable in general case, so there is still things to do both for the solver as we show in the Chapter 5 and for the developer.

Next we compare this approach with other attempts of refinement in CCSL (Section 2.6.5). First, the refinement by adding more constraints is still supported and extended by splitting the specification into different parts, each representing different role and properties. For example, by splitting out assumptions from the rest, we verify that adding more constraints does not violate rules beyond our control as designers. This separation of concern can be propagated to the whole system, with individual components or different views on the system described in different specifications, later unified in one.

Second, the relation of subspecification can actually be checked, as we describe in Section 5.2.3, in contrast to instant refinement. On another hand, instant refinement is proved to be able to propagate some properties to and from concrete side of the refinement, as well as split the abstract instants. We cannot express neither. The subspecification relation verifies an inclusion of solutions on common clocks, meaning that abstract properties (constraints) are always present in the concrete by the nature of relation, and other way around means nothing because abstract part is a bigger set, nothing else is known.

4.4.4.2 Deduplication of code

Reusing code is obviously good, as no one wants to write same things over and over again, when they were already written by someone else. Additionally, rewriting a small portion to accommodate for a specific to our system change may introduce mistakes in the process. Proper encapsulation of code and parametrization allows to evade such problems, which we feel this extension achieves.

First level of code reuse comes from decoupling of signatures from modules itself. Second is parametrization itself. We did not allow the number of constraints change on the parameters, which would allow the ultimate freedom of writing modules, it is still much better than before. Third is the fact of modules and their instantiation itself. Calls of the same module always produce different instances, unless in the same module with the same assigned name for the resulting block and parameters. This allows to define simple patterns, like modes, in a module that is called when needed.

A good example of this is the definition of an execution platform. Such a module can be reused in a lot of systems, with different parameters: number of CPUs, with or without interrupts, the frequency of the processor. And it cannot be abstracted away, as the tasks to be executed are not known before using them. Such a platform can be easily defined with the pool primitive presented above in Section 4.5.3.

4.5 Additional constructs

When we were experimenting with the descriptions of the motivational examples, it became evident that there are some missing constraints that could help to build simpler specification. Those other extensions are described in this section. Some are somewhat trivial and adding them does not make the language theoretically more expressive, only nicer to use. Others bring more complexity and some may be even considered experimental as they were only useful in some parts of the use cases and not for others, and so their roles may be handled differently in the future language iterations.

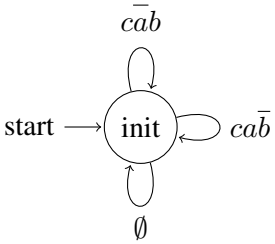


Figure 4.8: Automaton of disjunctive union $c = a \sqcup b$

4.5.1 Simple constraints

Simple constraints are the ones that can be defined with current CCSL constraints as patterns or macros, but their internal representation is much simpler if defined explicitly.

Disjunctive union Disjunctive union is a union of clocks which are also exclusive pairwise. The definition using other constraints is straightforward:

$$c = a \sqcup b \quad \equiv \quad c = a + b \ \| \ a \# b$$

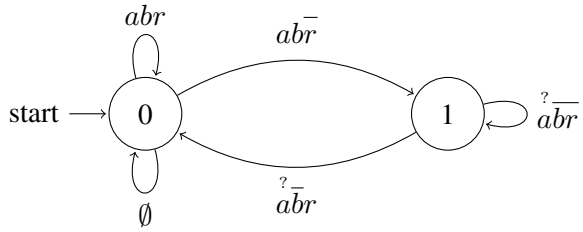
The automata representation is also pretty simple (Figure 4.8).

First and last sampled To express end-to-end delays, we need to know which clock has caused the reaction during a sampling. As in a lot of cases, the external event is the one causing and the reaction being performed in a loop, sampled by another clock, it is not that evident how to obtain it. Because a definition derived from other CCSL constraints is not possible or at least not obvious, we add such a constraint explicitly.

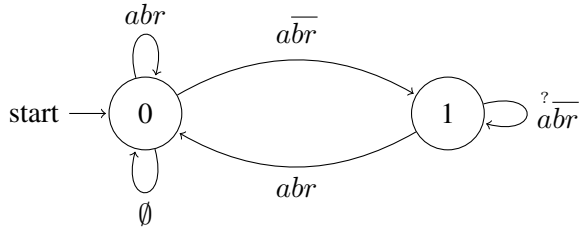
The constraint $b = \texttt{first sampled } a \texttt{ on } r$ defines clock b that ticks when first a ticks in the window between two successive ticks of r (the window's left bound is open and right is closed). Similarly, the dual constraint $c = \texttt{last sampled } a \texttt{ on } r$ defines clock c that ticks on the last a within the same window. The automata of the constraints are defined on Figure 4.9.

Allow and forbid One of the most used patterns in Mechanical Lung Ventilator (Section 3.2) are modes. To make behaviour change depending on which mode we are it, we need to disallow a given clock from ticking in the intervals between the respective *start* and *end* of a mode. For this, we propose `allow` and `forbid` constraints, two dual constraints doing exactly that: allowing or disabling ticking. We need two, because CCSL does not allow inversion of constraints as it would need a way to match on nothing happening which is not compositional. And it is not compositional because nothing is only relative to the current scope of a specification, adding a new constraint may fill the void, changing the behaviour significantly.

Thus we define the constraint manually. The constraint `allow`, written as $\texttt{allow } a, b \texttt{ in } [s, f]$, for clocks a, b, s, f, where s, f are start and finish respectively, allows clocks a, b to tick only after clock s and before f. The square brackets indicate that it can also be simultaneous. By changing the bound to parenthesis we can restrict to a stricter version of the constraint, when the synchronicity is not allowed. Similarly, the constraint `forbid`, written as $\texttt{forbid } a, b \texttt{ in } [s, f]$, disallows any ticks of clocks a, b if s ticked but not f. In both cases, s and f must alternate. The automata implementing those constraints is shown on Figure 4.10.

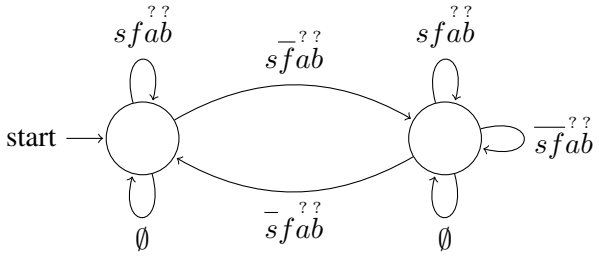


(a) $b =$ `first sampled` a `on` r

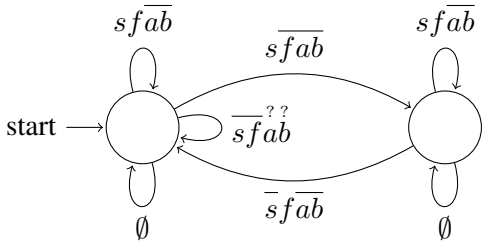


(b) $b =$ `last sampled` a `on` r

Figure 4.9: *First* and *last sampled* automata



(a) `allow` a, b `in` $[s, f]$



(b) `forbid` a, b `in` $[s, f]$

Figure 4.10: *Allow* and *forbid* automata

4.5.2 Build-level constraints

In contrast to simple constraints, these constraints cannot simply be expanded into a list of constraints and involve a collection of calls made in different parts of the specification to define the

actual behaviour. In other words, addition of the constraint is not local and requires some preprocessing before the actual behaviour is known.

Extensible constraints Extensible union allows user to declare the union though the module or across the modules. The idea comes from OCaml's extensible variants and its most used instance of exceptions. Exceptions can be declared anywhere, yet they act as an enum and so can be deconstructed as enum, so there is an unspecified number of the variants in it. In the end, the amount is known at compilation time, but not from the user's perspective. For example:

$$\begin{array}{l} o \mathrel{+}= a \\ o \mathrel{+}= b \\ o \mathrel{+}= c \end{array} \quad \equiv \quad o = a + b + c$$

Using this constraint we make new compound constraints, like extensible precedence. The idea is to allow the precedence to be more like implication: when several precedences are defined for one caused clock c, i.e. $c_1 \prec c, \ldots, c_n \prec c$, it requires each $c_1, \ldots, c_n$ to tick before c can, while for a lot of cases several *independent* things may define an event, and each can even be from different modules. An example of usage is different levels of alarms in Mechanical Lung Ventilator (specification described in Section 4.7.1). The reason to use this constraint for alarms is that the same type of alarm may appear in different, unrelated parts of the specification. If this constraint did not exist, we would have to manually declare the local alarms and them separately sum them together to obtain the global alarm. Then we can still use the modules to separate the concerns yet connect them easily. The only disadvantage being that it strongly binds the modules together, disallowing abstract analysis. But that would not be possible in any case.

The extensible union can also be used to define new constraints, like extensible causality and precedence. The definition is the following:

$$\begin{array}{l} a \prec_+ c \\ b \prec_+ c \end{array} \quad \equiv \quad (a + b) \prec c$$

We can also switch the union to disjunctive union $a| = b$ and causality derivatives $\prec_|, \preccurlyeq_|$, in order to keep the number of caused ticks equal to the number of causes.

4.5.3 Mutex and pool

Mutex is a classic synchronization primitive in programming languages used to allow only one thread to change data at a time. In this case, its function is more abstract as the object under protection is only in the mind of the user. Thus it can express giving sequential access not only to a piece of data, but to anything that is a *resource*: CPU cores, memory, communication medium. As a CCSL constraint, a mutex $\texttt{mutex}\{c_1 \mapsto c_2, c_3 \mapsto c_4, \ldots\}$ can be either derived from other constraints, which is shown in Specification 4.1 or represented as an automaton, like on Figure 4.11.

$$\texttt{mutex}\{c_1 \mapsto c_2, c_3 \mapsto c_4, \ldots\} \quad \equiv \quad \begin{array}{l} c_1 \texttt{ alternates } c_2 \\ c_3 \texttt{ alternates } c_4 \\ \ldots \\ (c_1 \sqcup c_3 \sqcup \ldots) \overset{\text{non-strict}}{\texttt{ alternates }} (c_2 \sqcup c_4 \sqcup \ldots) \end{array} \tag{4.1}$$

To note: $a \overset{\text{non-strict}}{\texttt{ alternates }} b$ is a version of a `alternates` b that allows schedules of form $a(ba)(ba)$ instead of usual $ababab$ ((ab) means a and b are at the same instant), thus allowing the mutex to unblock and lock immediately but only with different clocks. Alternation has to be defined for every pair of the clocks, locking (clocks before the arrow) and unlocking (after the arrow) should be alternated as groups, expressed by disjunctive unions $\sqcup$.

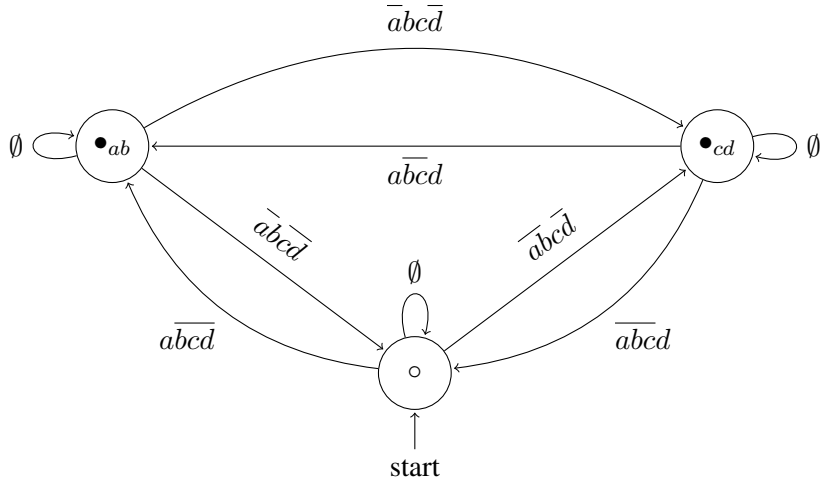


Figure 4.11: $\texttt{mutex}\{a \mapsto b, c \mapsto d\}$ automaton, empty circle means unlocked

On another hand, a pool is a set of resources that should be shared with tasks that do not distinguish the resources and do not care about which one is used: it could be a new one each time. An example is the tasks and their allocation to CPU cores. For a computer, individual tasks may not care on which core exactly they are executed, or it may be interesting to figure out if a dynamic assignment of tasks to cores is possible. In order to model it, we provide the constraint *pool*, written as $\texttt{pool}^n\{c_1 \mapsto c_2, \dots\}$. The syntax is similar to the mutex with the exception of the argument $n \in \mathbb{N}_{>0}$. This argument indicates how many homogeneous resources are in the pool. Thus, for example, given $\texttt{pool}^2\{a \mapsto b, c \mapsto d\}$, the schedule $(ac)(bd)$ is accepted, in contrast to mutex, as well as the schedule $(a)(a)(b)(b)$, with the obvious restriction that the same clock cannot allocate two resources instantly as it cannot tick twice in the same step.

The constraint can be defined the following way by using already defined constraints of disjunctive union and mutex:

$$\texttt{pool}^2\{a \mapsto b, c \mapsto d\} \quad \equiv \quad \begin{aligned} &\texttt{mutex}\{a_1 \mapsto b_1, c_1 \mapsto d_1\} \\ &\texttt{mutex}\{a_2 \mapsto b_2, c_2 \mapsto d_2\} \\ &a = a_1 \sqcup a_2 \\ &b = b_1 \sqcup b_2 \\ &c = c_1 \sqcup c_2 \\ &d = d_1 \sqcup d_2 \end{aligned}$$

There, the clocks a_1 and a_2 (same for b, c, d) are auxiliary clocks defined in order for the individual mutexes to be functional. From the point of view of clock a it defines a choice of resource to lock.

These are definitions, possible to be defined using previously defined constraints, as shown above. The problem with this is that it requires knowing all the uses of the pool beforehand, which

is limiting and does not follow the iterative spirit of CCSL. Much better option is to have the mutex and pool as special objects from the point of view of the language. We then supplement them with new locking/unlocking pairs from any point of the specification, by writing $\mathrm{add}(m, a, b)$. The build system's task is to collect these calls for us and rewrite into the standalone constraint. In this example, if $\mathrm{add}(m, a, b)$ is the only call, the mutex is defined as $m = \mathtt{mutex}\{a \mapsto b\}$.

It would be great, if the user could define constraints like macro and pool themselves, but we opted out of doing so, as it is too complicated. Such extension would require a whole scripting language, executed at build time. Thus, the current proposition is for the constraints to have a special syntax and the build-level logic. It also safeguards us from implementing too big language only to remove most of it later because it is too complicated to reason about.

4.6 New properties of interest

Because we have introduced new constructs and ideas, like modules and their parts, and because the available CCSL properties seem to be not always adequate, we define the new ones.

4.6.1 Weak-liveness

Property of liveness means that a clock ticks infinitely often in every schedule. This is usually not what we want in a system. For example, fault handling should always be able to execute, which does not mean be forced to execute. In order to relax the strong requirement of liveness, we introduce weak-liveness.

Definition 4.6.1 (Weak-liveness)**.** Weak-liveness is a property stating that for any schedule there is a possible path leading to schedule with infinite occurrence for every clock.

$$\forall c \in C, \forall \sigma \in \Sigma(\Phi), n \in \mathbb{N} : \exists \sigma_c \in \Sigma(\Phi) : \text{prefix}_{<n}(\sigma) = \text{prefix}_{<n}(\sigma_c) \wedge c \text{ live in } \sigma_c$$

Trivially, live specification is necessary weak-live but not the other way around.

Alternatively, we can define it using CTL. The difference with usual CTL is that we assert transitions and not states.

Definition 4.6.2 (Weak-liveness in CTL)**.** A clock c is weak-live when the following CTL formula is satisfied, where atom c means tick of the clock:

$$(EFc) \wedge AG(c \implies AX(EFc))$$

Which means that we want a loop with infinite c to exist in a specification and for it to be reachable. Then the specification is weak-live if every clock is weak-live.

4.6.2 Properties as assumptions

Previously in CCSL it only made sense for a specification to satisfy a property. With addition of time-triggered semantics, it is of interest for us to use properties as *assumptions*, in contrast to previous use as implications. For example, assuming that delayed clock is live implies that the time-triggered semantics is satisfied. Or, assuming time-triggered semantics for a system of two periodic clocks, allows us to derive more information. Additionally, the analysis with such assumption is more informative of the nominal case, i.e. when everything works.

```
basic_platform(frequency: Hz, error: interval<time>) {
  base_clock = periodic frequency rel.error error + ?;
  cpu = mutex{};

  task(wcet: int) {
    finish = start $ wcet on $.base_clock;
    add(cpu, start, finish);
  }
}
```

Listing 4.1: Basic platform macro

Zeno effect Property of progress states that it should not be possible to have infinite amount of ticks of a clock with another clock never ticking. In CCSL in general and in real-time extension in particular, there is a problem of assuring that the specification clocks are able to progress. Because of the choice in the domain of tags being rational numbers $\mathbb{Q}$ i.e. a dense set, if we do not explicitly constraint tag differences between the consequent ticks of clocks, they may tick infinitely often approaching but not reaching some value. This is usually known as zeno effect. This means that while schedule is infinite, it is not live in real-time sense.

By placing properties of liveness or time-triggered semantics on chronometric clocks as *assumptions*, we can obtain the progression of time. Liveness and time-triggered semantics are not equal though, as liveness does not allow the system to stop at all, while for time-triggered semantics is fine with that as long as stopping means stopping of the real-time too.

We do not provide solutions for these, except enforcement of the time-triggered semantics in Chapter 5.

4.7 Motivational examples in MRTCCSL

After we have introduced the language extensions and their semantics, we show how its actual code looks like, on the examples we have presented in Chapter 3.

In two of the example specifications, Sections 4.7.2 and 4.7.3, we use something called a basic platform. Its definition is given in Listing 4.1 and it is a module that implements a simple processor running with some frequency which defines its reference clock. The access of tasks to that processor is managed by a mutex (Section 4.5.3). The definition of a task in that case is continuous delay on the reference clock.

We are pushing the language a bit there, because in this case the module returns another module, which then can "spawn" the tasks. In actuality it replicates the defined set of constraints and returns it as a block (Section 4.4.2), which then is assigned some name, while capturing the mutex name. Thus it embed fully in the parent module.

Finally, we need to explain the differences with the syntax we use in the specifications. We try to use as compact but still unambiguous notation in this document, which involves usage of some special symbols. We did not want to demand the user to write expressions like this, thus some expressions do look slightly different:

- disjunctive union (see Section 4.5.1) $c = a \sqcup b$ is `c = a | b`;
- subclocking (see Table 2.1) $a \subseteq b$ is `a subclocks b`;

```
1  //FUN.19
2  pcv_mode(mode: struct, sensor: struct) assume {
3    //ratio of expiratory time to inspiratory time PER.5, includes PER.13
4    IE in [1, 4];
5    //respiratory rate, breath per minute PER.4, includes PER.12
6    RR in [4,50]/1 min;
7
8    trigger_window_delay = 0.7s; //CONT.45
9
10   //Check that nothing obstructs inspiration to start
11   //(if window is too small,the "faster" will not reset difference in time)
12   trigger_window.start <= fastest(sensor.inhale, trigger_window.finish) <=
         next inspiration.start; //FUN.21
13   //Rationale:  we should not allow inhale sensing outside of trigger window,
14   //otherwise it messes up the logic
15   allow sensor.inhale in [trigger_window.start, trigger_window.finish];
16 } {
17   expiration = delay inspiration by 1/RR/(1+IE); //FUN.20
18
19   trigger_window = {
20     start < finish;
21     start = delay $.expiration by trigger_window_delay; //CONT.45
22     finish = delay $.inspiration by 1/RR; //FUN.20
23   };
24   inspiration_condition = sensor.inhale + trigger_window.finish - (sample
         (sensor.inhale + mode.pcv.finish) on trigger_window.finish) - (sample
         mode.pcv.finish on sensor.inhale); //CONT.25
25   next inspiration = first sampled inspiration_condition on
         trigger_window.finish;
26 } assert {
27   //FUN.20, double check really
28   trigger_window.finish <= delay expiration by IE/RR/(1+IE);
29   inspiration alternates expiration; //same
30 }
```

Listing 4.2: PCV mode specification

- extensible precedence (see Section 4.5.2) $a \prec_+ b$ is `a +< b`.

As for the module syntax itself, the scope of a block is denoted as curly brackets `{}` and a module is declared as a function like in any C-like language, with parameters written as its arguments and types from $Sort$.

4.7.1 Mechanical Lung Ventilator

This specification is the deliverable part of our work on the Mechanical Lung Ventilator [TM24]. We do not show the full specification here as it is too big, its complete listing is available in Appendix A.1. The description of the use case is presented in Section 3.2.

The module we do show, Listing 4.2, describes the pressure controlled ventilation (PCV) mode. It is important to notice our use of the assumption and assertion functionality of MRTCCSL. There, we assume what the parameters to that mode can be, and that the sensor should appear only in the allowed trigger window, after expiration and before next inspiration. In the end, we also check that such complicated specification satisfies a simpler property of alternation. This is not

```
1  //cylinder timings
2  cylinder(shaft_degree: clock, offset: int) {
3    otdc = skip offset+0   every 720 shaft_degree;
4    fbdc = skip offset+180 every 720 shaft_degree;
5    itdc = skip offset+360 every 720 shaft_degree;
6    sbdc = skip offset+540 every 720 shaft_degree;
7
8    exhaust.start = sbdc $ -[45, 60] on shaft_degree;
9    exhaust.finish = (next otdc) $ [5, 20] on shaft_degree;
10   ignition_point = (next itdc) $ [-30, 10] on shaft_degree;
11   knock_window.start = (next itdc) $ [0, 55] on shaft_degree;
12   knock_window.finish = knock_window.start $ [0, 55] on shaft_degree;
13 }
```

Listing 4.3: MRTCCSL specification of Spark ignition control system

```
15 //timing constraints of cylinders on tasks
16 task_cylinder_rel(t: struct, c: struct, sensor_sampling: clock) {
17   c.exhaust.start |<= t.oxygen_sensing.start;
18   t.oxygen_sensing.finish <= c.exhaust.finish;
19   t.ignition_control.finish <= c.ignition_point;
20   t.knock_sensing.start <= sample c.knock_window.start on sensor_sampling;
21   t.knock_sensing.finish <= sample c.knock_window.finish on sensor_sampling;
22 }
```

Listing 4.4: MRTCCSL specification of Spark ignition control system

the requirement of the system, but an additional check that we wanted to make to validate our own understanding.

4.7.2 Spark ignition control system

For the spark ignition control we present the full specification. It is also more complete version than what was presented in Section 3.3. The general structure of the specification is the following:

- `cylinder` module (Listing 4.3) describes how the degree of the crankshaft connected to the piston movement and other important events, like ignition and knock window, where the controller monitors if phenomenon of an engine knock is going to happen;
- `task_cylinder_rel` module (Listing 4.4) specifies which timing relations the tasks has to satisfy to produce the initiation correction. Here we use the extensible *disjunctive* union constraint, as we need to specify that the exhaust of *this* cylinder is causing to start the task, but also that it leaves the possibility for something else to cause it. In this case, other cylinders;
- lastly, `engine_control` module (Listing 4.5) defines the relations to real-time of the base clock of the controller by using the `basic_platform` and of the crankshaft. For that we use the parameter of revolutions per meter, set externally. But we know that this engine cannot operate anywhere outside the specified range of $[600, 4500]\mathrm{rpm}$, which the parameter has to satisfy. It then defines the cylinders, tasks and their data dependencies, timings between the tasks and cylinders and the fact that there is only one filtering buffer.

```
engine_control(rpm: int) {
    rpm in [600, 4500];
  //constants
  shaft_period = 1s/(360*rpm/60);
  frequency = 20MHz;
  ms_scale = frequency/1kHz;
  sensor_scale = frequency/100kHz;

  //controller hardware
  platform = basic_platform(frequency,20ppm of frequency);

  internal_ms = every ms_scale base_clock;
  sensor_sampling = every sensor_scale base_clock;

  //engine generic timing
  shaft_degree = periodic shaft_period rel.error +-1% + ?;
  cylinder0 = cylinder(shaft_degree, 0);
  cylinder1 = cylinder(shaft_degree, 180);
  cylinder2 = cylinder(shaft_degree, 360);
  cylinder3 = cylinder(shaft_degree, 540);

  //processing
  task = {
        knock_sensing = task(?);
        oxygen_sensing = task(?);
        ignition_control = task(?);
        knock_filtering = task(?);
        knock_control = task(?);
    }
  //data dependencies
  task.oxygen_sensing.finish <= task.ignition_control.start;
  task.knock_filtering.finish <= task.ignition_control.start;
  task.ignition_control.finish <= next task.knock_control.start;

  //resource filter_buffer is only one
  mutex{
    task.knock_sensing.start -> task.knock_sensing.finish,
    task.knock_filtering.start -> task.knock_filtering.finish,
  };
  //timing requirements of cylinders on tasks
  task_cylinder_rel(task, cylinder0, sensor_sampling);
  task_cylinder_rel(task, cylinder1, sensor_sampling);
  task_cylinder_rel(task, cylinder2, sensor_sampling);
  task_cylinder_rel(task, cylinder3, sensor_sampling);
}
//In particular
engine_control(1166) is live and safe;
//Or in general
find rpm where engine_control(rpm) is live and safe;
```

Listing 4.5: MRTCCSL specification of Spark ignition control system

As the task durations are unknown, the argument of task calls are marked by ?. While this means, that the analysis engine should try to find the values, these should later be least estimated on the real hardware. In the end of the specification, we express the check that a specific value of RPM satisfies the system defined. As well as the ideal search functionality, where the language would find the possible ranges of solutions.

4.7.3 Brake-by-wire

```
1  brake(freq: Hz, e: interval<time>) assume {
2    ms_scale = freq/1kHz;
3    p = basic_platform(freq, e);
4  } {
5    abs_correction = p.task(?ms);
6    braking = p.task(?);
7    speed = p.task(?);
8
9    receive.cmd <= abs_correction.start;
10   abs_correction.end <= braking.start;
11   braking.end = actuation;
12
13   speed.ready = skip ? every 10*ms_scale p.base_clock;
14   speed.ready <= speed.start;
15   speed.finish <= send.speed.ready;
16 } upper interface {
17   actuation = delay receive.cmd by [0,10ms];
18   send.speed.ready = skip ? every 10*ms_scale p.base_clock;
19 }
```

Listing 4.6: Brake specification

```
21 controller(freq: Hz, e: interval<time>) assume {
22   ms_scale = freq/1kHz;
23   p = basic_platform(freq, e);
24 } {
25   torque_comp = p.task(?);
26   pedal = p.task(?);
27
28   pedal.start = skip ? every 10*ms_scale p.base_clock;
29   //change in from pedal can appear when finish checking
30   pedal.change subclocks pedal.finish;
31   pedal.change <= torque_comp.start;
32
33   torque_comp.finish <= send.fl.cmd;
34   torque_comp.finish <= send.fr.cmd;
35   torque_comp.finish <= send.rl.cmd;
36   torque_comp.finish <= send.rr.cmd;
37 }
```

Listing 4.7: Controller specification

Similarly to Spark ignition control system before, here we present the building blocks first, combining into a complete system. Additionally, the specification is more full than the one pre-

```
39 bbw(freq: Hz, e1: interval<time>, e2: interval<time>, e2e_latency: time) {
40   brakes = {
41     fl = brake(freq, e1);
42     fr = brake(freq, e1);
43     rl = brake(freq, e1);
44     rr = brake(freq, e1);
45   }
46   c = controller(2*freq, e2);
47
48   c.send.fl.cmd < brakes.fl.receive.cmd;
49   brakes.fl.send.speed < c.receive.speed;
50   c.send.fr.cmd < brakes.fr.receive.cmd;
51   brakes.fr.send.speed < c.receive.speed;
52   c.send.rl.cmd < brakes.rl.receive.cmd;
53   brakes.rl.send.speed < c.receive.speed;
54   c.send.rr.cmd < brakes.rr.receive.cmd;
55   brakes.rr.send.speed < c.receive.speed;
56
57   //bus mutex
58   send = c.send | brakes.fl.send | brakes.fr.send | brakes.rl.send |
         brakes.rr.send;
59   receive = brakes.fl.receive | brakes.fr.receive | brakes.rl.receive |
         brakes.rr.receive | c.receive
60   send alternates receive;
61   //or what is the size of the packet
62   receive = delay send by [0.75, 1]ms;
63 } assert {
64   //reaction constraints
65   reaction_deadline = delay c.pedal.change by e2e_latency;
66   brakes.fl.actuation < reaction_deadline;
67   brakes.fr.actuation < reaction_deadline;
68   brakes.rl.actuation < reaction_deadline;
69   brakes.rr.actuation < reaction_deadline;
70
71   //brake synchronization
72   s = slowest(brakes.fl.actuation, brakes.fr.actuation, brakes.rl.actuation,
         brakes.rr.actuation);
73   f = fastest(brakes.fl.actuation, brakes.fr.actuation, brakes.rl.actuation,
         brakes.rr.actuation);
74   s < (delay f by 5ms);
75 }
76
77 find f where bbw(f, +-1% of f, +-1% of f, 10ms) is schedulable and safe;
```

Listing 4.8: Coordination specification

sented in Section 3.4. For this we followed the explanation from [Per+12]. Thus, the specification consists of three modules:

- `brake` module describes the brake functionality and timings, its reaction to the actuation of the brake itself and transmission of torque sensor readings to the controller. We additionally verify by using `upper interface` that the reaction to the actuation request is bounded and that the sensor readings occur at regular intervals;
- `controller` module then specifies that the controller monitors the changes in the pedal and sends out the updates to actuate the brakes, once the computation of torque is complete;
- finally, `bbw` module "synchronizes" the two by instantiating four brakes and specifying how the communication between them and the controller works. Finally, we check that the overall system satisfies the synchronization requirement, i.e. that the activations of brakes occur approximately at the same time, and this time is also bounded in relation to the change in pedal position, i.e. the maximum end-to-end latency is not violated.

4.8 Conclusion

In this chapter we have introduced several extensions to the language of CCSL, namely:

- real-time;
- parameters, expressions and their constraints;
- modules;
- some auxiliary constraints;
- new properties and checks.

Each have their semantics informally and formally defined. And thanks to CCSL's nature and our design there is little traction in integration of these extensions. We additionally explain how the language is applied to the motivational examples we saw in Chapter 3, and demonstrate how the features inspired by them actually materialize. Because we have a good coverage of the extensions in these specifications, we consider these additions to be a success.

CHAPTER 5

Analysis

This chapter explores the analysis techniques applied to MRTCCSL. Specifically analysis using mathematical induction on infinite schedules and abstract interpretation. We discuss how we implement and use them, their strong points and flaws, as well as suggestions to improve. To overcome the encountered and anticipated problems specific to our language, we propose several methods that build on top of the abstract interpretation tools. Lastly, application to use cases using these techniques is shown.

There is a little usability to a language without tools. In case of programming languages it is enough to provide a translation to another language. But, depending on the properties and expressiveness of the target language, the properties of interest in source language may or may not be easily verifiable. Thus, this chapter presents an implementation of symbolic analysis for a fragment of real-time CCSL, experiments with existing abstract interpretation tools and propositions of how to extend them to effectively support the language. We finish with a description of the implementation as a software project.

5.1 Analysis with induction

Here we present a method using inductive reasoning about a subset of infinite schedules in MRTCCSL specifications. It is based directly on the notion of mathematical induction and denotational semantics of a MRTCCSL fragment. In the case of induction, base step and inductive step have to hold to proof a desired clause. We obtain this from the constraints itself by manipulation of the formulas of the denotational definitions. As not all the constraints can be transformed, this is only a partial method, but which nonetheless is effective and efficient on the selected subset, especially in comparison to the abstract interpretation analysis that we present later in Section 5.2. To solve numerical conditions inside inductive and base steps, we use Verimag Polyhedra Library (VPL) [BM18a; BM18b].

5.1.1 Motivational example: Brake-by-wire

The core of the Brake-by-wire system (described in Section 3.4, specification in Section 4.7.3) is temporal synchronization of the reactions on the wheels, relative to the change in pedal. The delays on two wheels are denoted d_1, d_2, and are supposed to be bounded in time with the tolerance t. The clocks are w_1, w_2 for wheel reactions, r for the reaction request, f for fastest reaction of the two, s for slowest respectively and α is an auxiliary clock for an expression. Thus the specification, with an assertion to be checked using subspecification relation $\Subset$, is expressed the following way:

$$\left.\begin{aligned} w_1 &= \texttt{delay}\ r\ \texttt{by}\ d_1 \\ w_2 &= \texttt{delay}\ r\ \texttt{by}\ d_2 \\ f &= \texttt{fastest}(w_1, w_2) \\ s &= \texttt{slowest}(w_1, w_2) \end{aligned}\right\} \Subset s < \texttt{delay}\ f\ \texttt{by}\ t$$

First, we could try to solve it by hand. In this case a simple yet mouthful proof can be made:

$$
\begin{aligned}
w_1 = \texttt{delay}\ r\ \texttt{by}\ d_1 \equiv\ & \forall i \in \mathbb{N} : w_1[i] - r[i] = d_1 \\
\equiv\ & \forall i \in \mathbb{N} : w_1[i] = r[i] + d_1 && (5.1)\\
w_2 = \texttt{delay}\ r\ \texttt{by}\ d_2 \equiv\ & \forall i \in \mathbb{N} : w_2[i] - r[i] = d_2 \\
\equiv\ & \forall i \in \mathbb{N} : w_2[i] = r[i] + d_2 && (5.2)\\
f = \texttt{fastest}(w_1, w_2) \equiv\ & \forall i \in \mathbb{N} : f[i] = \min(w_1[i], w_2[i]) && (5.3)\\
s = \texttt{slowest}(w_1, w_2) \equiv\ & \forall i \in \mathbb{N} : s[i] = \max(w_1[i], w_2[i]) && (5.4)\\
(5.1) \wedge (5.2) \wedge (5.3) \implies\ & \forall i \in \mathbb{N} : f[i] = \min(r[i] + d_1, r[i] + d_2) \\
\equiv\ & \forall i \in \mathbb{N} : f[i] = \min(d_1, d_2) + r[i] && (5.5)\\
(5.1) \wedge (5.2) \wedge (5.4) \implies\ & \forall i \in \mathbb{N} : s[i] = \max(r[i] + d_1, r[i] + d_2) \\
\equiv\ & \forall i \in \mathbb{N} : s[i] = \max(d_1, d_2) + r[i] && (5.6)\\
\alpha = \texttt{delay}\ f\ \texttt{by}\ t \equiv\ & \forall i \in \mathbb{N} : \alpha[i] - f[i] = t \\
\equiv\ & \forall i \in \mathbb{N} : \alpha[i] = f[i] + t && (5.7)\\
s \prec \alpha \equiv\ & \forall i \in \mathbb{N} : s[i] < \alpha[i] && (5.8)\\
(5.7) \wedge (5.8) \implies\ & \forall i \in \mathbb{N} : s[i] < f[i] + t && (5.9)\\
(5.5) \wedge (5.6) \wedge (5.9) \implies\ & \forall i \in \mathbb{N} : \begin{array}{l} \max(d_1, d_2) + r[i] < \\ \min(d_1, d_2) + r[i] + t \end{array} \\
\equiv\ & \max(d_1, d_2) - \min(d_1, d_2) < t \\
\equiv\ & |d_1 - d_2| < t && (5.10)
\end{aligned}
$$

If we would rewrite the formulas a bit differently, we can obtain the following:

$$
\begin{aligned}
w_1 &= \texttt{delay}\ r\ \texttt{by}\ d_1 \\
w_2 &= \texttt{delay}\ r\ \texttt{by}\ d_2 \\
f &= \texttt{fastest}(w_1, w_2) \\
s &= \texttt{slowest}(w_1, w_2)
\end{aligned}
\equiv \forall i \in \mathbb{N} : S(i) = \begin{cases} w_1[i] - r[i] = d_1 \\ w_2[i] - r[i] = d_2 \\ f[i] = \min(w_1[i], w_2[i]) \\ s[i] = \min(w_1[i], w_2[i]) \end{cases}
$$

$$
\begin{aligned}
\alpha &= \texttt{delay}\ f\ \texttt{by}\ t \\
s &\prec \alpha
\end{aligned}
\equiv \forall i \in \mathbb{N} : P(i) = \begin{cases} \alpha[i] - f[i] = t \\ s[i] < \alpha[i] \end{cases}
$$

$$
(\forall i \in \mathbb{N} : S(i)) \implies (\forall i \in \mathbb{N} : P(i)) \equiv \forall i \in \mathbb{N} : S(i) \implies P(i)
$$

To prove this, we can use induction, which is what the method we describe next is capable of doing:

$$
\begin{aligned}
& S(0) \wedge \forall i \in N^+ : S(i) \implies S(i+1) \\
(S(0) \implies P(0)) \wedge \forall i \in N^+ : & (S(i) \implies P(i)) \implies (S(i+1) \implies P(i+1))
\end{aligned}
$$

5.1.2 Constraints to induction

The main principle of this method, is to use denotational semantics of the constraints, derive the base step and induction step that would prove it, and try to check that these steps do actually hold.

Method assumptions First, we assume that all clocks are infinite, this way the conditions related to existence of ticks are eliminated. Second, we notice that the constraints of CCSL are defined as first-order formula and a lot of them are only once universally quantified. Third, the body of the universally quantified formula are semi linear, i.e. a finite disjunction of linear conditions on numerical domain variables.

The assumptions on quantifiers and what the formulas under $\forall$ can be, are dictated by the choice of solving backend, the polyhedra (discussed in Section 2.9.4), and the choice of general proof schema, the induction. No second universal and no existential quantifiers are allowed in the formulas, because they do not translate into the base and inductive steps.

We refer to $c[i]$ as variables, as the task is to find out whatever there is an assignment from each tick to rationals. And so, the variables are uniquely identified by the clock and index. The index cannot be any expression on integers though. The reason is the impossibility to match the variables that are involved in multiplication of indices ahead of induction loop. For example, $a[i] = b[i * p], b[i] < c[i]$ fails to generate an inductive step with our scheme, as the number of b between $a[i]$ and $a[i + 1]$ is unknown in general case. Thus we use variables with indices related to i only, because we allow only one universal quantification, and shifted by a constant only. As we use polyhedra as solving backend, this fact means that not every constraint can be expressed exactly or at all in it. Additionally, polyhedra domain has to operate on finite number of variables, adding the same restriction on the indices.

Thus, in a formula, the variables are constrained by semi linear relations with other variables, i.e. other ticks on other clocks. Generally, the variables are constrained exclusively using rational relations with the exception of absolute periodic constraint, where the relation also involves the index of the tick itself. As it is an integer, the relation with it cannot be represented as convex polyhedra, only an overapproximation.

We classify all the constraints of MRTCCSL in the Table 5.1 using this characterization. As not all the constraints satisfy the assumptions, we provide their over and underapproximation variants.

Assuming all of that and a constraint ϕ defined as $\forall i : P(i)$, we can rewrite it as $P(0) \wedge \forall i : P(i) \implies P(i+1)$, i.e. the induction (Section 2.1.2). The translation is a bit more involved than that though, because inductive step is not explicit about the connection between the predicates on i. For example, in case of proving induction on natural numbers the connection is the arithmetics itself. In our case, we need to provide this connection explicitly. And the connection is the condition of total order on variables of logical clocks and that the first ticks has to be bigger or equal to 0. Thus, if $P(i) = a[i] \prec_I b[i]$ as defined in CCSL constraint $a \prec b$, then:

- base step is $P(0) \stackrel{\text{def}}{=} 0 \preccurlyeq_I a[0] \wedge 0 \preccurlyeq_I b[0] \wedge a[0] \prec_I$;
- inductive step is $(P(i) \implies P(i+1)) \stackrel{\text{def}}{=} a[i] \prec_I b[i] \implies a[i] \prec_I a[i+1] \wedge b[i] \prec_I b[i+1] \wedge a[i+1] \prec_I b[i+1]$.

If it is one constraint satisfying the conditions above, then such transformation is trivial.

Next, we need to handle the synchronization of the constraints. Given that in denotational form, the synchronization is defined as a conjunction (Definition 2.6.7), we need to achieve the following transformation, given two constraints ϕ and ψ and the denotational transformation of

Constraint	Definition, $\forall i$	$\forall$	$i \pm c$	$\angle$	$\mathbb{Q}$	Over-approximation, $\forall i$	Underapproximation, $\forall i$
$a \preccurlyeq b$	$a[i] \le b[i]$	✓	✓	✓	✓		
$a \prec b$	$a[i] < b[i]$	✓	✓	✓	✓		
$a \# b$	$\forall j : a[i] \neq b[j]$	×	×	×	✓	$a[i] \neq b[i]$	$b[i-1] < a[i] < b[i]$
$a = b$	$a[i] = b[i]$	✓	✓	✓	✓		
$a \subseteq b$	$\exists j : a[i] = b[j]$	×	✓	✓	✓		$a[i] = b[i]$
$r = a - b$	$\exists j : \begin{array}{l} r[i] = a[j] \wedge \\ \nexists k : a[j] = b[k] \end{array}$	×	✓	✓	✓		$(r[i] = a[i] \vee r[i-1] = a[i-1]) \wedge$ $a[i-1] < b[i] < a[i]$
$b = a \$ N$	$b[i-N] = a[i]$	✓	✓	✓	✓		
$r = \texttt{fastest}(a_i, \ldots, a_n)$	$r[i] = \min(a_i[i], \ldots, a_n[i])$	✓	✓	×	✓	$\bigwedge_{k=1}^{n} r[i] \le a_k[i]$	$\bigwedge_{k=1}^{n} r[i] = a_k[i]$
$r = \texttt{slowest}(a_i, \ldots, a_n)$	$r[i] = \max(a_1[i], \ldots, a_n[i])$	✓	✓	×	✓	$\bigwedge_{k=1}^{n} r[i] \ge a_k[i]$	$\bigwedge_{k=1}^{n} r[i] = a_k[i]$
$r = a_1 * \cdots * a_n$	$\exists j_1 \ldots j_n : r[i] = a_1[j_1] = \cdots = a_n[j_n])$	×	✓	✓	✓		$r[i] = a_1[i] = \cdots = a_n[i]$
$r = a_1 + \cdots + a_n$	$\bigvee_j^n \exists k : r[i] = a_j[k]$	×	✓	✓	✓		simple: $r[i] = a_1[i] = \cdots = a_n[i]$ complex: $\vee_{S \in \mathcal{P}(n)} \wedge_{j \in S} r[i] = a_j[i]$
$b = \texttt{skip}\ \varphi\ \texttt{every}\ p\ a$	$b \left\lfloor \frac{i-\varphi+1}{p} - 1 \right\rfloor = a[i]$	✓	×	✓	✓		
$b = \texttt{sample}\ a\ \texttt{on}\ s$	$\exists j : s[i] = b[j] \iff$ $\exists k : a[k] \in \begin{cases} (s[i-1], s[i]) & \text{if } i > 0 \\ [0, s[0]] & \text{if } i = 0 \end{cases}$	×	✓	✓	✓		$s[i] = b[i] \wedge a[i] \in \begin{cases} (s[i-1], s[i]) & \text{if } i > 0 \\ [0, s[0]] & \text{if } i = 0 \end{cases}$
$b = a \$ N\ \text{on}\ s$	$\exists j : s[i] = b[j] \iff$ $\exists k : a[k] \in \begin{cases} (s[i-1-N], s[i-N]] & \text{if } i > N \\ [0, s[0]] & \text{if } i = N \\ \bot & \text{if } i < N \end{cases}$	×	✓	✓	✓		$s[i+N] = b[i] \wedge$ $a[i] \in (s[i-1+N], s[i+N]]$
$a\ \texttt{alternates}\ b$	$b[i-1] < a[i] < b[i]$	✓	✓	✓	✓		
$b = \texttt{delay}\ a\ \texttt{by}\ [d_1, d_2]$	$b[i] - a[i] \in [d_1, d_2]$	✓	✓	✓	✓		
$a = p \cdot i^{\text{th}} \pm \texttt{rel.}e + \varphi$	$a[i] \in p + e + \begin{cases} a[i-1] & \text{when } i > 1 \\ \varphi - p & \text{otherwise} \end{cases}$	✓	✓	✓	✓		
$a = p \cdot i^{\text{th}} \pm \texttt{abs.}e + \varphi$	$a[i] \in p \cdot (i-1) + e + \varphi$	✓	✓	✓	×	is overapproximation when $\mathbb{N}$ is not supported	
$f = \texttt{first sampled}\ a\ \texttt{on}\ b$	$f[i] = \min_{b[i-1] < a[k] \le b[i]}(b[k])$	×	✓	×	✓		$b[i-1] < f[i] = a[i] \le b[i])$
$l = \texttt{last sampled}\ a\ \texttt{on}\ b$	$l[i] = \max_{b[i-1] < a[k] \le b[i]}(b[k])$	×	✓	×	✓		$b[i-1] < l[i] = a[i] \le b[i])$
$\texttt{forbid}\ c\ \texttt{in}\ [a, b]$	$a[i] \le b[i] \wedge \nexists k : a[i] \le c[k] \le b[i]$	×	✓	✓	✓		$a[i] \le b[i] \wedge \left(b[i-1] < c[i] < a[i] \vee b[i] < c[i] < a[i+1]\right)$
$\texttt{allow}\ c\ \texttt{in}\ [a, b]$	$a[i] \le b[i] \wedge \nexists k : c[k] < a[i] \vee b[i] < c[i]$	×	✓	×	✓		$a[i] \le c[i] \le b[i]$
$a = \varepsilon\ \texttt{strict sporadic}$	$a[i] > a[i-1] + \varepsilon$	✓	✓	✓	✓		

Table 5.1: Comparison of denotational definition of CCSL constraints and their approximations. $\forall$ means only one universal quantifier is present, $i \pm c$ — only constant offset in index position, $\angle$ — convex relation, $\mathbb{Q}$ — only rational variables (i is an integer).

constraints $[\![\_]\!]_{\text{ind}}$ (given by Table 5.1):

$$\begin{aligned}
[\![\phi]\!]_{\text{ind}} &= \forall i : P_\phi(i) \\
[\![\psi]\!]_{\text{ind}} &= \forall i : P_\psi(i) \\
[\![\phi \parallel \psi]\!]_{\text{ind}} &\equiv [\![\phi]\!]_{\text{ind}} \wedge [\![\psi]\!]_{\text{ind}} \\
&\equiv \forall i : P_\phi(i) \wedge \forall i : P_\psi(i) \\
&\equiv \forall i : P_\phi(i) \wedge P_\psi(i) \\
&\overset{?}{\Longleftarrow} P_\phi(0) \wedge P_\psi(0) \wedge \forall i > 0 : P_\phi(i) \wedge P_\psi(i) \implies P_\phi(i+1) \wedge P_\psi(i+1)
\end{aligned}$$

But is the last step correct? Not always as demonstrated on the following example:

$$\begin{aligned}
b &= a \ \$ \ 3 \ \texttt{on} \ r \\
r &= 5\,\text{s} \cdot i^{\text{th}} \pm \texttt{rel.}1\% + 1s \\
b &= \texttt{delay} \ a \ \texttt{by} \ 10s
\end{aligned}$$

It does not have any schedules, as logically delayed clock b, given that reference is periodic of $5\,\text{s}$, will be of distance at least $5\,\text{s} \cdot 3 = 15\,\text{s}$ compared to a which violates third constraint. If we would translate the constraints as described in Table 5.1 and then try to make an inductive proposition, we would have to ask ourself how to match indices on r defined in relative periodic to logical delay. Without duplicating the periodic relation several times to the past in the inductive step, i.e. for indices $i, i-1, i-2, i-3$, it is not possible to derive that the delay ends up being *both* 15 and 10 seconds, which is of course a contradiction.

Thus, what final induction proposition hides is the common clocks in the two propositions. If clocks are shared, so both the variables and the relations from other constraints can propagate to the other variables. Additionally, in the universally quantified form, it is not required for the indices of variables to match, as $\forall$ checks everything anyway, while in inductive form, if we do not impose the dependency, there will be none. And if some variables are not constrained enough, we get misleading results. More precisely, such a method is only an overapproximation, i.e. only a sound proof of contradiction can be obtained. To improve that, the inductive step should be sound *and* complete (on the infinite schedules for the constraints with semi linear conditions).

We achieve the completeness by saturating the conditions across the indices. It is done by the following algorithm, but for it to work we need to assume that the formulas extracted under universal quantification are conjunctions only. However, we do handle disjunctions later. The saturated product of the relations in indices is described in Algorithm 5.1. We do not define how auxiliary functions work exactly because it is too technical, but we provide examples of what they return for some values:

$$\begin{aligned}
\text{partition} \begin{pmatrix} [a[i] < b[i], \\ b[i] < c[i], \\ e[i+1] = g[i-1]] \end{pmatrix} &= \begin{bmatrix} \begin{bmatrix} a[i] < b[i], \\ b[i] < c[i] \end{bmatrix}, \\ [e[i+1] = g[i-1]] \end{bmatrix} \\
\text{shift}(a[i] < b[i+1] = p \cdot i, j) &= a[i+j] < b[i+1+j] = p \cdot (i+j) \\
\text{vars}(a[i] < b[i+1]) &= [a[i], b[i+1]] \\
\text{dedup}\,([a[i] < b[i+1], a[i] < b[i+1]]) &= [a[i] < b[i+1]]
\end{aligned}$$

Additionally, we use unconventional syntax of `for ... in ... where ...` in order to eliminate even more technicalities of searching and variable matching.

```
1   // Partition into components by relation connections
2   partition(rs: rel list) -> rel list list
3   // Shift relation in index
4   shift(atom: rel, shift: int) -> rel
5   // Return all variables in relation
6   vars(atom: rel) -> var list
7   // Deduplication, leaves only unique elements
8   dedup(atoms: rel list) -> rel list
9   // Saturate
10  saturate(atoms: rel list) -> rel list {
11      first = [atoms[0]]
12      return fixpoint(
13      init=first,
14      f=(prev: rel list) -> {
15          next = prev
16          for c[i + j] in vars(p) where p in prev {
17              for a in atoms where c[i + k] in vars(a) {
18                  next += shift(a, k-j)
19              }
20          }
21          return next
22      },
23      equal=λ(prev: rel list, next: rel list).dedup(prev) == dedup(next)
24      )
25  }
26  saturated_product(atoms: rel list) -> (location set, transition set) {
27      product = []
28      for p in partition(atoms) {
29          product += saturate(p)
30      }
31      return product
32  }
```

Algorithm 5.1: Saturated product of denotational semantics formula on clocks

The intuition of algorithm is the following:

1. we translate each constraint into its denotational formula $[\![\_]\!]_{\text{ind}}$, defined in the Table 5.1 from semantics defined in Section 2.6.2.1, and extract the subformula under universal quantification. These relations have to be purely conjunctive, disjunctions are handled by the next algorithm. The conjunction is treated as a list of relation atoms $e \bowtie e$, where e is a numerical expression involving constants or clock indexing $c[i + k]$ and $\bowtie$ is a numerical relation;

2. we partition the relations by their shared clocks transitively. It is exactly the same idea as with connected components in a graph, but where vertices mean clocks and any relation between clocks with any index is an undirected edge;

3. for each partition we do the saturation procedure:

   (a) select one relation as initial;

   (b) add to that relation the rest of the relations when they have shared clocks;

   (c) when indices of these clock variables do not match, shift the relation in indices to be equal;

   (d) repeat adding until fixpoint (no new relation can be added).

4. the result is then a conjunction of individual saturated products.

While it could remind of Floyd-Warshall algorithm [Flo62], it is not because the final set of variables is discovered in the process.

This listing of the algorithm additionally makes few shortcuts, in order to be compact. It does not cover saturation when variables in relations are not indexed clocks, like index itself in absolute periodic. The fixpoint computation can be greatly speed up by only processing new variables. The saturation also should happen as shifting of range over ranges. It is both more performant, as more relations are added in one operation going though fixpoint comparisons, and required for self referential constraints like relative periodic to not introduce infinite amount of variables. These modifications are available in the actual implementation that we describe in Section 5.3.

Then from such saturated product it is possible to derive everything else:

- the base step: first we substitute i with 0, then this formula is shifted by -1 until the clock indexed with the biggest offset shifts below zero. Among the shifted formula, we remove every clause that contains clocks below zero, as these ticks do not exist and so it should not be possible to place restriction on them. The last addition is that the initial tick has to be bigger than zero. Finally, the resulting list of relation is treated as a conjunction and defines the base step. Such base step checks that there are solutions at the start of a schedule, and because it was generated from the induction step itself, it automatically satisfies it there when the solutions exist;

- inductive hypothesis (or precondition to inductive step) is the saturated product itself;

- and the consequence of the inductive step is the product again but shifted with $+1$. To connect with the hypothesis, we add the total order relations for all logical clocks $\forall c[i] : c[i-1] < c[i]$.

Then by finding that solutions exist for the base step and that inductive step is satisfied under this hypothesis, we prove that the original specification has infinite schedules.

To implement this algorithm for fastest and slowest constraints, we need to handle finite disjunctions. For this, each constraint with disjunction is rewritten into its disjunctive normal form, i.e. the disjunctive terms only appear on the most outer level of the formula. Then, the terms of the disjunction are saturated separately. Because the solver used (polyhedra) does not support disjunctions, we need to generate separate variants of inductive proofs. This means that we have to check the consequences from different disjunction terms satisfy which hypothesises from which disjunctive term. Then, if under the hypothesis of the first disjunction term, the consequence of another disjunction term has solutions and it satisfies the first hypothesis now as a condition, that means that again, there is a relation that can generate an infinite schedule.

For a more visual representation, we propose the following graph. It consists of variants of base step, precondition (hypothesis), consequence and postcondition (hypothesis shifted by $+1$). The graph is then populated with directed edges as the induction specifies: base step should satisfy the hypothesis (it is by construction in our case, thus “auto’), consequence should have solutions under some assumption ($\Longrightarrow$ edge), the solution may satisfy an assumption for the next step

$(P_j(i) \wedge P_k(i+1) \wedge P_l(i+1)$, "auto" when $l = k$). Then a correct system will result in a path starting in one base step resulting in a cycle through such graph. If there is a path that ends not in cycle, then this is one of the counter-examples to infinite-only specification.

For a constraint with two disjunction terms, like slowest, we would have the graph shown in Figure 5.1. This is the most edges it can have. Because we check that all vertices' propositions have solutions, and if they do not, they are removed from the graph, and the same with the edges. Due to this, some vertices become unreachable or only lead to a "dead end". In this case, either there are no infinite schedules at all, or there is a mix of finite and infinite schedules.

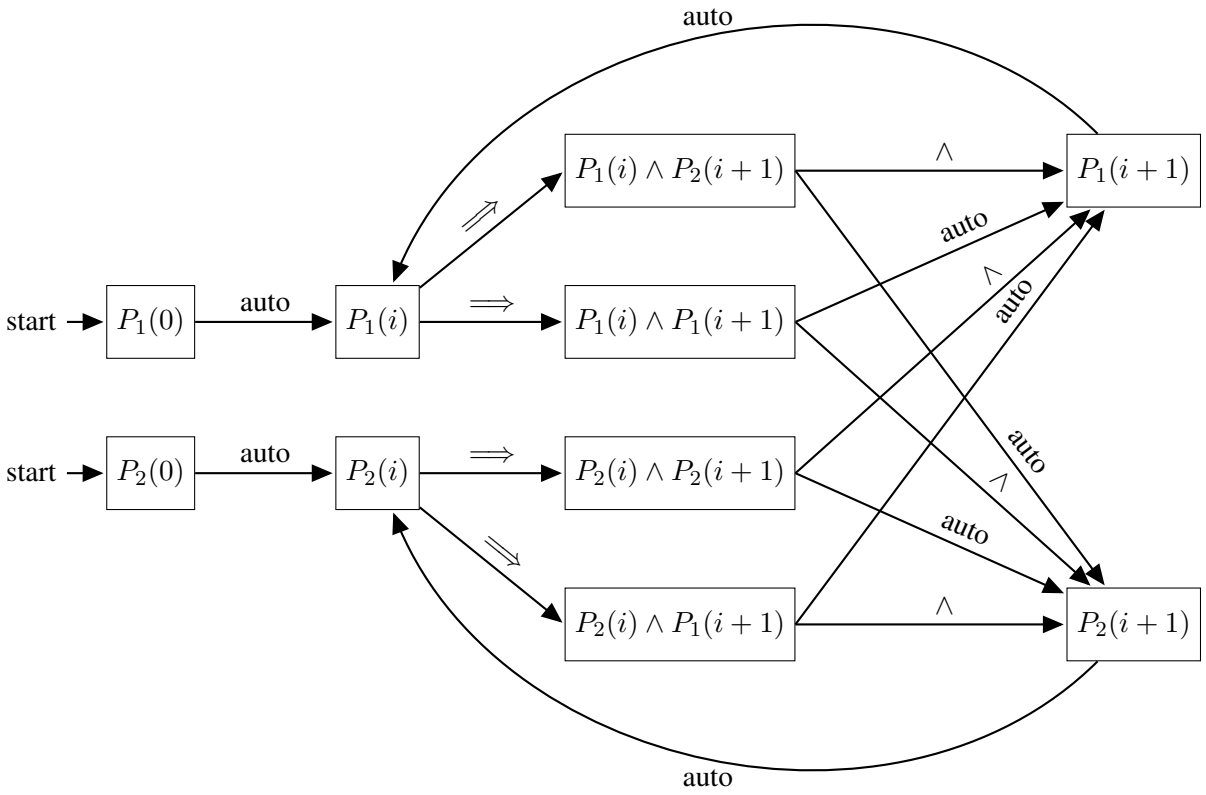


Figure 5.1: Graph of induction parts and proofs for constraints with disjunctions

5.1.2.1 Saturation does not always terminate

Unfortunately, certain combinations of constraints give an infinite sequence of relation additions in saturation step. Of course, then the algorithm does not terminate, which is a problem. An example of such specification is:

$$\begin{aligned} b &= a \ \$ \ d_1 \ \texttt{on} \ r \\ r_d &= r \ \$ \ d_1 \\ c &= b \ \$ \ d_2 \ \texttt{on} \ r_d \end{aligned} \tag{5.11}$$

Thus we implement a solution to detect such infinite loop. Upon detecting the loop, we exit from the procedure with an error.

First of all, the reason for a loop to occur is for a single constraint to relate the same clock several times. In delay, base clock r occurs three times with different indices. To that variable, other constraints attach their own relations. Usually, it does not result in a variable explosion. But when they share two clocks, it is possible for a connection by one variable to introduce another variable with different index from the ones already present, on which in turn the original constraint can attach the same relation. Thus the relations keep adding new terms, leading to an infinite loop.

This explanation is the basis for the algorithm that detects the problem. The algorithm uses a directed graph $G = (V, E)$, where vertices $V \subseteq Rel \times C \times \mathbb{N}$ are some individual relations Rel from formulas obtained from constraints, clocks C and their internal indices $\mathbb{N}$, with edges being $E \subseteq V \times V$. The graph then is populated at each iteration of saturation above (Algorithm 5.1). For this we track which *new* relations were added during computation of fixpoint, in relation to the previous step. More specifically, which previous relations' variable were causing addition of a new relation at this step and any new variables that was added by doing so. Then, if at some point of the computation, there is a cycle in this graph, it means that a variable can infinitely cause addition of itself, with a different index. An example of this graph is provided on Figure 5.2 for Specification 5.11.

5.1.3 Induction to polyhedra

As we have told above, the solving backend is a polyhedra domain. To conduct our induction analysis as described, we need a way to encode it into the domain of polyhedra. For this we employ a simple tactics: every clock indexing $c[i + k]$ is a separate variable `c_i_k`, with k interpreted, i.e. if $k = 1$, the variable is `c_i_1`. It is a valid translation because value of $c[i + k]$ does not depend on the actual value of $i + k$ other than what we can specify with relation, and so it is to differentiate different ticks and to correctly connect them. When index is a variable in an expression, like in $p \cdot i$, it is translated as a separate special variable with name i.

The relations and expressions are translated as they are as we only allow constraints with linear relations to be used, excluding the multiplication between variables due to decidability limitations. The polyhedra library that we use, Verimag Polyhedra Library (VPL), supports both rationals and integers as semantically different types. Of course, the integer support can only help to constraint the resulting set just a bit, as expressions like $c[i] = p \cdot i$ still cannot be exactly represented due to convexity (i should only be $1, 2, 3, \ldots$, but here it can be 3.4). Additional requirement is that the domain used has to implement natively both strict and non-strict comparison. This is caused by the fact that the set of rational numbers is dense and there is no natural successor to its elements. Otherwise, the strict relation in syntax could be translated into a non-strict relation using successors and predecessors. The only library that we know of, that supports this feature, is VPL.

While pretty much everything that we need in this setting, exists natively in polyhedra, exception is the implication $P(i) \implies P(i + 1)$. And equivalent $\neg P(i) \vee P(i + 1)$ is not efficient due to all the disjunctions that occur after the negation of convex $P(i)$, thus we implement another equivalent check. We first construct polyhedra of $P(i) \wedge P(i+1)$ and check if it has solutions, then check that this polyhedra projected to variables of $P(i)$ only, $\Pi_{V(P(i))}(P(i + 1))$, contains whole of $P(i)$ in it, i.e. $P(i) \subseteq \Pi_{V(P(i))}(P(i+1))$. This way, any solution present in the hypothesis, i.e.

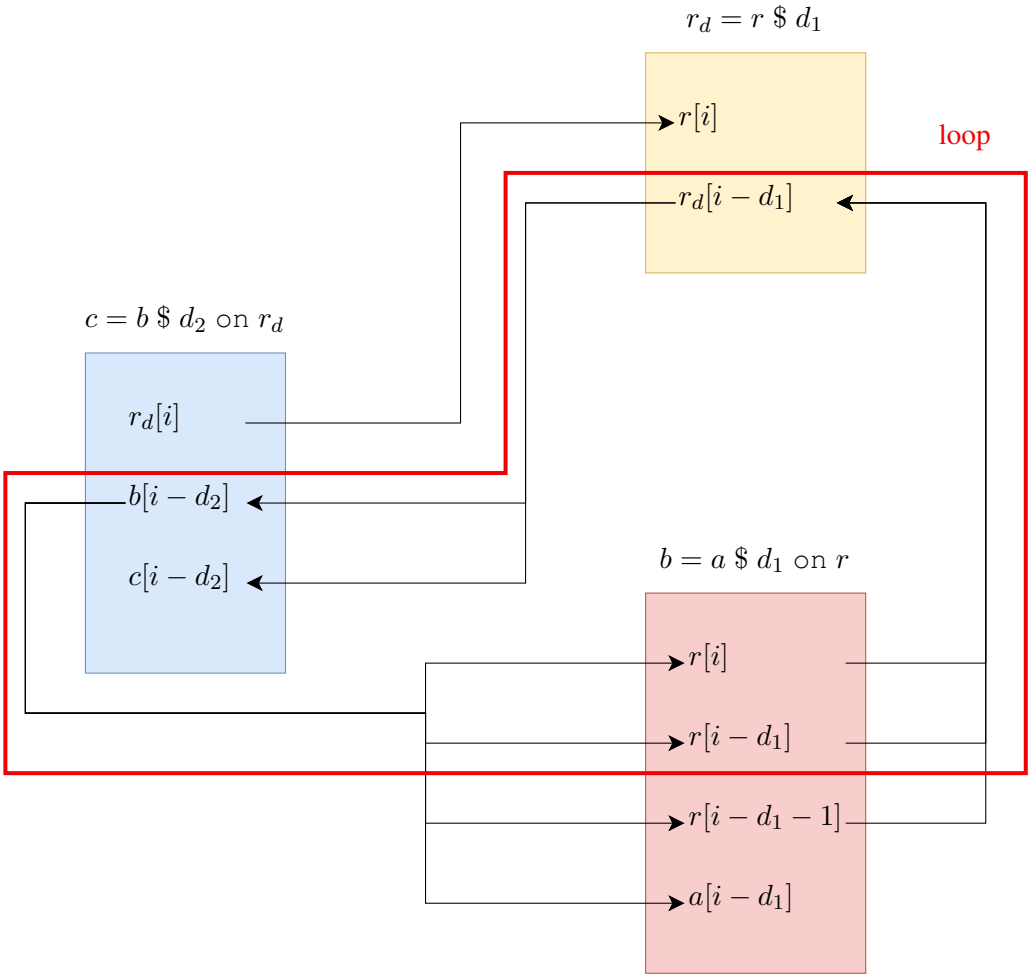


Figure 5.2: Example of graph for infinite saturation with variables grouped by constraint

the past, that has at least one solution in the inductive step, i.e. the future. By applying it infinitely often, we obtain infinite sequences, which is what we were looking for.

5.1.4 Approximations

As shown in Table 5.1, not every constraint is exact given our assumptions (it is when every checkmark is present for a constraint). When a specification is given with such constraints we could just reject it. We have decided to go another path: we automatically decide, given the property we are trying to check, which approximation to use, over or under. We follow the same logic as presented in Section 2.9.1 on approximations: depending on the property we get different conclusion on different approximation we choose.

5.1.5 Existence and emptyness checks

Existence of schedules is checked by proving that at least one disjunctive variant satisfies the induction (if available) or, alternatively, the existence of reachable cycles in the graph (like in Figure 5.1). Then, if an exact representation of the constraints is not possible, two approximations could be built: over and under. As usual, overapproximation can prove that the subset of solutions is not possible, while underapproximation can prove that something does exist. But it is important to keep in mind, that the overapproximation can only prove that the infinite schedule of a really particular form does not exist, not all infinite schedules. The ones where only *some* clocks deadlock, schedules where clocks do not evolve uniformly or finite schedules still can exist. For this reason, the method is only approximate to the full set of solutions, and we can only definitely prove that something exists using it.

5.1.6 Subspecification relation

To check subspecification relation, we define it as a relation between the steps in induction. Thus, given $\forall i \in \mathbb{N} : P_S(i) = [\![S]\!]_{\text{ind}}$ and $\forall i \in \mathbb{N} : P_P(i) = [\![P]\!]_{\text{ind}}$, and if $P_S(0) \wedge \forall i : P_S(i) \implies P_S(i+1)$ and $P_P(0) \wedge \forall i : P_P(i) \implies P_P(i+1)$ are true, subspecification relation $S \Subset P$ is true when the following condition holds:

$$\begin{aligned} P_S(0) &\implies P_P(0) \\ \forall i : P_S(i) &\implies P_P(i) \end{aligned}$$

As with implication above, it is rewritten as projection and then subset check on the polyhedra.

To be more technical, the actual check is implemented as $S \Subset S \parallel P$. It is equivalent as if S violates P, the $S \parallel P$ removes some behaviour and so results in smaller set than S. Next, we do the saturated product of $S \parallel P$ first, and then *remove* atomic relations from $P \setminus S$. It is made so that individual $P_S(i), P_P(i)$ share the same variables with the same indices and with the same relationships between them. Otherwise, the $\implies$ is not sound as due to the semantics of subset relation in polyhedra: if variable is not defined, then it is assumed to be any value. Meaning that if we saturate the products of S and $S \parallel P$ separately, S may result in not covering the same range of indices as $S \parallel P$ because it is a smaller constraint set. Then if we try to prove the relation, it will always fail.

As for the approximations, only underapproximation and exact relations can be used. It is due to the nature of overapproximation: it may include the solutions that do not actually satisfy

the specification. Checking that something is included in a set which itself may contain wrong behaviour, does not prove that the included set is free of the bad behaviour.

5.1.7 Parametric verification

The parameters may be put instead of constants in relations as described in Section 4.3. The parameter variables can be additionally restricted with constraints between them, like $a > 0 \wedge b = c_1 \$ a$. The parameters freely translate into the polyhedra relations as long as they are linear combination of variables. Then the task is to find if there are solutions satisfying the CCSL constraints along with constraints on parameters. Finally, we can extract the relations on parameters itself, to find out for which parameters the specification exists or the subspecification is satisfied.

The parameter placement is limited by the same few factors as it was discussed before. First, due to impossibility of handling anything else than simple $i \pm c$ expressions, where i is the index variable and $c \in \mathbb{Z}$ is an index offset. Second, not all operations are supported by polyhedra. For example, multiplication between variables is to be avoided because of undecidability issues. Thus, it is not possible to make neither logical delay nor real-time periodic constraints accept parameters, using the proposed approach.

Parametric verification is also limited to the case where every part of induction contains the same constraint about the parameter. It is a limitation of the approach, as in the disjunctions, induction and initial conditions exist in different variable spaces. It is possible to extract the parameters in one part and propagate in another, but when different parts provide different constraints on parameters, we would need to consider different combinations, further worsening the complexity. Additionally, the propagation will not necessary terminate.

Thus, in the implementation, we extract the parameters by polyhedra projection from each disjunctive term, without the propagation. And these individual projections are an overapproximation of the actual parameters. So we cannot really rely on the parametrized results, as they may contain not exactly the set of solutions, but they still provide an idea of what parameters' ranges should be, and this information can be plugged into other tools.

5.1.8 Complexity

Polyhedra operations are exponential in number of variables in the worst case. The variables in each base or induction step are also exponential in number of constraints. A specification with n disjunction terms creates at least 2^n variants to check, thus exponential again. Thus, the final algorithm is exponential in both number of clocks and constraints. But since, we use constraint-based implementation of polyhedra (VPL) and our method does not include join, widening or assignment that contribute the most to the complexity, we seem to avoid most of the performance problems. Additionally, in the implementation, we use the lazy evaluation and sharing of polyhedra when we can, meaning that if a base step is UNSAT, the corresponding inductive step is not checked at all.

So far the performance was not a problem. Our test suite, consisting of 15 tests with specifications ranging from two to nine constraints, finishes in 14 seconds on 2020 high-end laptop. We describe the specification and modules that we check in Appendix A.3.

5.2 Using abstract interpretation

Our interest in using abstract interpretation is to provide finite time analysis of the existing and newly defined properties of MRTCCSL. In this section we explain how we use the abstract interpretation, the problems encountered and propositions to the method to improve the analysis, unique to CCSL and its extensions.

As the tools of analysis, we use NBac and ReaVer (Sections 2.9.6.1 and 2.9.6.2). These tools, given a description of a program and an undesired state inside of this program, can derive if it is reachable or not, and so if the system violates whatever is meant by the undesirability of that state. There, the system is defined as a conjunction of transition relations on some numerical or Boolean states and inputs. A bad state is defined as a Boolean formula with relation to the state variables only.

After encoding CCSL using the language of NBac, we explain its modification to support the real-time extension, and the motivation of why we chose this particular encoding over another one. We continue with the description and results of our experiments using the available tools. These experiments cover an implementation of a limited version of finiteness property and sub-specification relation. As the results are not exactly satisfactory and as we envision some other general problems not possible to express right now in the tools, we propose modifications in order to improve these checks and to the approaches to solve some of the modeling patterns we find important from the use cases.

5.2.1 Pure CCSL analysis

In this subsection, we describe a translation of CCSL into the NBac language. As we present two encodings, we start with the common translation logic for both. Please refer to the description of the language in Section 2.9.6.1.

In both, clocks are translated into input Boolean variables. The constraints then impose restrictions on what they can be by relating them to the state via assertion Boolean formula. Semantically, the state variables are always known inside the assertion, thus we need to solve it only for the free variables, i.e. clock ticking or not. The conditions of different constraints are put in conjunction and is semantically equivalent to the synchronization.

Every specification starts at an initial condition, denoted with the special variable `init`. While not strictly required, it is done so that each transition relation could specify its initialization value in a composable way, we do not need to change the initial condition when we add or remove states and transitions. Although, usually integer variables are initialized with zero and Boolean with false, here, we always specify it explicitly for the sake of clarity. Another reason is that introducing another variable lets the engine splits the state space, potentially eliminating starting conditions from the main loop. Additionally, it does mean that in terms of clocks anything can happen in the initialization transition, but we ignore it as it is an setup transition and it does not matter for the program as the state will start to get updated only *after* the initialization.

5.2.1.1 Delta-counter encoding

This encoding translates previously defined in Section 2.6.3.2 symbolic automata into NBac format. Locations of the automata are usually translated into Boolean variables, though we do not always follow it with each individual constraint as there are sometimes better ways to express

```
state
    init: bool;
    delta_ab: int;
input
    a,b: bool;
transition
    delta_ab' = if init then 0 else delta_ab + (if a then 1 else 0) - (if b
        then 1 else 0);
    init' = false;
assertion (delta_ab = 0 => not b) or init;
initial init;
```

Listing 5.1: Precedence $a \prec b$ (δ-counters)

```
state
    init: bool;
    i_a,i_b: int;
input
    a,b: bool;
local
    in_a,in_b: int;
definition
    in_a = if init then 0 else i_a + (if a then 1 else 0);
    in_b = if init then 0 else i_b + (if b then 1 else 0);
transition
    i_a' = in_a;
    i_b' = in_b;
    init' = false;
assertion ((i_a=i_b => not b)) or init;
initial init;
```

Listing 5.2: Precedence $a \prec b$ (clock-counter encoding)

the same behaviour symbolically rather than encoding each location as a number or a sequence of binary variables. The delta counters are expressed as integer variables. The transition is then expressed the following way: for clocks c_1 or c_2, if there is a delta-counter $\delta(c_1, c_2)$, variable `delta_`c_1`_`c_2 increases by 1 when c_1 ticks (input variable c_1 is true), and decreases by 1 if c_2 ticks, respectively.

As an example, we describe how precedence from Figure 2.10 is expressed, see Listing 5.1. We implement this encoding and synchronization of several constraints into the same NBac program in the project called ccsl-rs for all CCSL original constraints.

5.2.1.2 Clock-counter encoding

While delta-counter encoding is exactly what is needed to express the constraints in a minimal state space, they are not exactly convenient in matching patterns for analysis, like congruences. Thus, we see separate clocks variables as an opportunity for the saturation of polyhedra to propagate relations between the variables, which may take more iterations to be discovered if delta-counters are used. Finally, relations between clock that are not specified directly, can still be implicitly captured, which would not always be the case with differences (delta-counters).

```
state
    init: bool;
    ok: bool;
    i_a,i_b: int;
input
    a,b: bool;
local
    in_a,in_b: int;
definition
    in_a = if init then 0 else i_a + (if a then 1 else 0);
    in_b = if init then 0 else i_b + (if b then 1 else 0);
transition
    i_a' = in_a;
    i_b' = in_b;
    init' = false;
    ok' = if init then true else ok and (in_a >= in_b);
assertion ((i_a=i_b => not b)) or init;
initial init;
final not (init or ok);
```

Listing 5.3: Proving $a < b \implies i_a \geq i_b$

Thus, we use a slightly different encoding, which instead of delta-counters uses counters for each clock. Formally: for each clock c we add a counter i_c, which is increased each time c ticks. Then the relations described on delta-counters are replaced by differences between the counter variables of the respective clocks. We demonstrate this by providing a translation in Listing 5.2. We also list some other constraints in Appendix A.2.

Additionally, we define next values of integer counters too, written in code as `in_c`, where `i` means index and `n` means next value. This way we can check properties of interest on next values directly, which removes the lag in property checking. In both encodings presented we did not describe the properties that we try to prove, but if we would write one, it is usually expressed as an `ok` variable. The property is then expressed in a way to make the variable false, when it is violated. Had we expressed the property in the current state, for a property to be *set* as violated, the state would have had to be already violated in the preceding state. While in a lot of cases, specifications continue to be live and so the violation will be reached, in case when it is not, such schema is not correct. Thus, we define the next values of counters and use them, making property value `ok` synchronous with the violations.

To showcase how it works, we ask the analysis to prove a simple (inductive) property of precedence $a < b$: $i_a \geq i_b$. The program itself is listed in Listing 5.3 and can be proved in NBac when we increase the precision of postcondition computation (option "-dselect"). Otherwise the analysis makes extrapolation of the relation, leading to locations violating the property. We demonstrate that on a compact version of the control flow graph of the problem with abstractions of the state on Figure 5.3.

5.2.2 Real-time CCSL encoding

The encoding of real-time extensions consists of two parts: first, the modification to CCSL-native clocks, and second, NBac definitions of the new constraints itself.

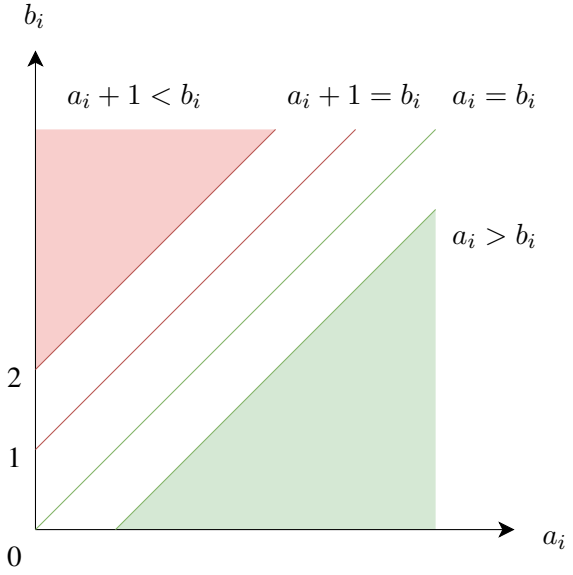


Figure 5.3: State regions of $a < b \implies i_a \geq i_b$, red unreachable when using "-dselect 2"

```
1  state
2      init: bool;
3      i_a,i_b: int;
4      a_lts, b_lts: rational;
5      prev: rational;
6  input
7      a,b: bool;
8      next: rational;
9  local
10     a_ticked, b_ticked: bool;
11     in_a, in_b: int;
12 definition
13     in_a = if init then 0 else i_a + (if a then 1 else 0);
14     in_b = if init then 0 else i_b + (if b then 1 else 0);
15     a_ticked = a_lts >= 0;
16     b_ticked = b_lts >= 0;
17 transition
18     i_a' = in_a;
19     i_b' = in_b;
20     init' = false;
21     a_lts' = if init then -1 else if a then next else a_lts;
22     b_lts' = if init then -1 else if b then next else b_lts;
23     prev' = if init then 0 else next;
24 assertion ((i_a=i_b => not b) and (prev < next) and next >= 0) or init;
25 initial init;
```

Listing 5.4: Precedence $a \prec b$ real-time encoding

```
1  state
2      init: bool;
3      o_lts: rational;
4      prev: rational;
5  local
6      ticked: bool;
7  definition
8      ticked = o_lts >= 0;
9  input
10     o: bool;
11     next: rational;
12 transition
13     init' = false;
14     o_lts' = if init then -1 else if o then next else o_lts;
15     prev' = if init then -1 else next;
16 assertion ((prev < next) and next >= 0 and (if ticked then (o => e1 <= next -
       o_lts - p and next - o_lts - p <= e2) else o = (next = offset))) or init;
17 initial init;
```

Listing 5.5: Relative periodic encoding $a = p \cdot i^{\text{th}} \pm \texttt{rel}\,.[e_1, e_2] + \varphi$

As the basis we use counter encoding above with rational variable `prev`, which indicates the previous real time, and a rational *input* variable `next` for the time of the next step. A necessary condition is that variable `next` is strictly bigger than the current time. With second change, we add an additional variable of "last time seen", c_{lts}, per clock c. This variable is updated when clock ticks with the value of the next transition. The example of precedence from Listing 5.2 is modified to include these changes in Listing 5.4.

Now we define the new constraints from the real-time extension: relative periodic in Listing 5.5 and absolute periodic in Listing 5.6.

As we have told before in Section 4.2.1, by putting question marks "?" in periodic constraints, we indicate that we do not care when the periodic constraint starts, or that we would like to get all the options when it could start. Or by expressing it as a variable, we may want to know the possible values relative to other clocks, which would satisfy desired property. For example, to do correct initialization sequence. To achieve it we omit the else branch on `ticked` variable from the assertion, i.e. it will be just `ticked => (o => e1 <= next - o_lts - p and next - o_lts - p <= e2)`.

As for real-time delay, it cannot be encoded without significant changes to the NBac language and infrastructure. We need a queue type with push, pop, head and empty operations. If these were available, the NBac program for it would look as in Listing 5.7.

For the implementation of a queue we propose to use lattice automata domain [Gal08]. The library does implement all the operations needed and was used before in similar setting in the tool McScM [HLS12]. An important note: McScM cannot be used as it is in this setting because Communicating Finite-State Machines do not have any variables nor their manipulation or control based on them, thus making it not impossible to express RTCCSL in it.

```
state
    init: bool;
    o_lts: rational;
    prev: rational;
input
    o: bool;
    next: rational;
local
    ticked: bool;
definition
    ticked = o_lts >= 0;
transition
    init' = false;
    o_lts' = if init then -1 else if o then o_lts + p else o_lts;
    prev' = if init then 0 else next;
assertion ((not init => prev < next) and (if ticked then (o => e1 <= next -
    o_lts - p and next - o_lts - p <= e2) else o = (next = offset))) or init;
initial init;
```

Listing 5.6: Absolute periodic encoding $a = p \cdot i^{\text{th}} \pm \texttt{abs}\,.\,[e_1, e_2] + \varphi$

```
state
    init: bool;
    ab_q: queue<rational>;
    a_lts, b_lts: rational;
    prev: rational;
input
    a,b: bool;
    next: rational;
local
    ab_q_inter: queue<rational>;
definition
    ab_q_inter = if a then push(q, a_lts) else q;
transition
    init' = false;
    a_lts' = if init then -1 else if a then next else a_lts;
    b_lts' = if init then -1 else if b then next else b_lts;
    ab_q' = if init then [] else if b then pop(ab_q_inter) else q;
assertion ((b => head(ab_q_inter) + d1 <= next and next <= head(ab_q_inter) +
    d2) and (prev < next) and next >= 0) or init;
initial init;
```

Listing 5.7: Real-time delay encoding $b = \texttt{delay}\ a\ \texttt{by}\ [a, b]$

5.2.3 Subspecification relation

We remind that subspecification relation is defined in Definition 4.4.1 and is the basis for module semantics and thus modular proofs. Here and later by S and P we mean specifications from $S \Subset P$.

Given the proof scheme of NBac and ReaVer, i.e. search for non-reachability of some bad state, we define the subspecification relation check as violation of the constraints the property P specification consists of. Thus, the constraints become observers to constraints of S. This implies an important restriction to this particular implementation: the bad state can only be defined in terms of what is already present, i.e. what the S specification defines as behaviour. It may sound that we would be restricted to the state of S implementation only, but it also includes state variables of P solely defined by clocks of S. Meaning that the actual restriction to the method is $C(P) \subseteq C(S)$, i.e. clocks of the property specification P cannot include clocks other than ones in S.

In most cases, it is more an annoyance than a problem, as some of the constraints can be shifted to the S side. But not all the relations can be rewritten this way. For example, $a < b < c \Subset a < d < c$ is true, because the common clocks are a and c. Then $a < b < c \Subset a < c$ and $a < d < c \Subset a < c$, i.e. they are abstracted with the same relation, thus the subspecification relation should actually hold both ways. But in neither case we can move any constraint without making the relation false. Implementation wise, $a < c$ should be deduced in both cases thanks to the choice of counter-based encoding described earlier.

Speaking of the whole module $M = (A, S, G, I)$, while it is possible to check the interface, both under and overapproximations, and the assertion, assumption is never possible. The assumption check $A \Subset A \parallel S$ is always violating the requirement $C(A) \cup C(S) \not\subseteq C(A)$, except when $S = \emptyset$, which is a trivial and error case on its own.

As abstract interpretation tool employs the scheme with inversion of the property, it is sound but not complete (the general idea described in Section 2.9.1). Meaning that we can be sure that subspecification relation holds when we get the positive result, but not sure if it does not. To check it for sure, we would need to do underapproximation. Their non-empty intersection is a sufficient evidence that property is violated. Unfortunately, underapproximation domains are inherently more difficult to construct due to their nature [ABG22]. Thus, other techniques should be employed. There are works [DSi13] that try to provide simulation relation in abstract interpretation, but they contain an assumption we do not have: the simulation is expressed under ACTL [Mai00] properties, i.e. the simulation has to preserve any property expressed in the language, while we need to preserve the whole language. We leave the development of a simulation that supports any $C(S)$ and $C(P)$ for future work.

To summarize, the approach to check subspecification relation consists of:

- *check assumptions*: $C(P) \subseteq C(S)$ should be true;
- *translate*: both S and P should be translated into their NBac form;
- *merge*: we follow the synchronization pattern described before. Statements under all sections should be gathered together, and as we use same uniform schemas for all variables, no two variables can have the same name and a different definition. Thus variables can be safely deduplicated. The only difference is the assertion condition of property program becomes part of `ok` variable definition, not `assert`;

- *run analysis*: we try both NBac and ReaVer due to implementation differences (and bugs), and if any of them concludes that the negation of the property is unreachable and then subspecification relation holds.

5.2.3.1 Experiments

Next, we would like to present some of the examples we tried to prove and their results. The examples are:

- definition of alternation using several other constraints is indeed alternation: $a < b < a \$ 1 \Subset a \texttt{ alternates } b$;
- alternation is a particular version of precedence: $a \texttt{ alternates } b \Subset a < b$;
- precedence is transitive: $a < b < c \Subset a < c$;
- intersection of periodic clocks is periodic too:

$$\begin{aligned} n &= \texttt{every } 7\ r \\ m &= \texttt{every } 5\ r \quad \Subset \quad nm = \texttt{every } 7 \cdot 5\ r \\ nm &= n * m \end{aligned}$$

First, $a < b < a \$ 1 \Subset a \texttt{ alternates } b$. The left specification actually consists of three constraints, and we make it obvious before any translations. These are $a < b, b < \alpha$ and $\alpha = a \$ 1$, where α is any free name for the previously anonymous clock. We translate both specification into NBac program in Listing 5.8, and we highlight the property-only part.

As for the results of the analysis, NBac proves the property, while ReaVer does not. The reason is that ReaVer does not partition by numerical conditions in regular analysis, but NBac does.

Second, $a \texttt{ alternates } b \Subset a < b$ is translated in Listing 5.9. And it can be proved by both NBac and ReaVer, but we had to add a Boolean bisimulation step in the ReaVer analysis strategy, making it "aB;aB:b;pIF;pB;rT;aS".

Third, transitivity $a < b < c \Subset a < c$ is only possible to prove with NBac if we tweak the precision of computing postcondition of a relation (option "-dselect"). ReaVer cannot prove it as it requires partitioning by numerical conditions outside of an *accelerable* loop, and there are no accelerable loops in this encoding.

Last is a specification that defines a clock nm as a hyperperiod of clocks n and m. Because both are defined on a common base clock r, the hyperperiod should be periodic on it too, with period 35. The program is presented in Listing 5.11 and is impossible to prove in neither NBac or ReaVer. In Section 5.2.5.1 we propose a change to the partitioning step as a solution targeting this problem.

5.2.4 Properties of interest

As we have defined first for CCSL itself (Section 4.6) and then later in Chapter on MRTCCSL (Section 4.6), there are several properties that we are usually interested in case of reactive systems and their implementation. In this subsection we introduce how the problems we encountered with implementing such checks and what we would need to implement in full.

```
state
    init: bool;
    ok: bool;
    i_a,i_b,i_alpha: int;
    ab_turn : bool;
input
    a,b,alpha: bool;
local
    in_a,in_b,in_alpha: int;
definition
    in_a = if init then 0 else i_a + (if a then 1 else 0);
    in_b = if init then 0 else i_b + (if b then 1 else 0);
    in_alpha = if init then 0 else i_alpha + (if alpha then 1 else 0);
transition
    i_a' = in_a;
    i_b' = in_b;
    i_alpha' = in_alpha;
    ab_turn' = if init then false else if a then true else if b then false
        else ab_turn;
    init' = false;
    ok' = if init then true else ok and (not ab_turn => not b) and (ab_turn =>
        not a);
assertion ((i_a = i_b => not b) and (i_b=i_alpha => not alpha) and (i_a < 1 =>
    not alpha) and (i_a >= 1 => a = alpha)) or init;
initial init;
final not (init or ok);
```

Listing 5.8: Checking $a < b < a \ \$ \ 1 \Subset a$ `alternates` b

```
state
    init: bool;
    ok: bool;
    i_a,i_b: int;
    ab_turn: bool;
input
    a,b: bool;
local
    in_a,in_b: int;
definition
    in_a = if init then 0 else i_a + (if a then 1 else 0);
    in_b = if init then 0 else i_b + (if b then 1 else 0);
transition
    i_a' = in_a;
    i_b' = in_b;
    ab_turn' = if init then false else if a then true else if b then false
        else ab_turn;
    init' = false;
    ok' = if init then true else (ok and i_a>=i_b and (i_a=i_b => not b) and
        i_a<=i_b+1);
assertion ((ab_turn => not a) and (not ab_turn => not b)) or init;
initial init;
final not (init or ok);
```

Listing 5.9: Checking a `alternates` $b \Subset a < b$

```
state
    init: bool;
    ok: bool;
    i_a,i_b,i_c: int;
input
    a,b,c: bool;
local
    in_a,in_b,in_c: int;
definition
    in_a = if init then 0 else i_a + (if a then 1 else 0);
    in_b = if init then 0 else i_b + (if b then 1 else 0);
    in_c = if init then 0 else i_c + (if c then 1 else 0);
transition
    i_a' = in_a;
    i_b' = in_b;
    i_c' = in_c;
    init' = false;
    ok' = if init then true else (ok and (in_a >= in_c) and (in_a = in_c =>
        not c));
assertion ((i_a=i_b => not b) and (i_b=i_c => not c)) or init;
initial init;
final not (init or ok);
```

Listing 5.10: Checking $a < b < c \Subset a < c$

```
state
    init: bool;
    ok: bool;
    i_r,i_n,i_m,i_nm: int;
input
    r,n,m,nm: bool;
transition
    init' = false;
    i_r' = if init then 0 else i_r + (if r then 1 else 0);
    i_n' = if init then 0 else i_n + (if n then 1 else 0);
    i_m' = if init then 0 else i_m + (if m then 1 else 0);
    i_nm' = if init then 0 else i_nm + (if nm then 1 else 0);
    ok' = if init then true else (ok and (i_r=35*(i_nm+1)-1 => nm=r) and
        (i_r<>35*(i_nm+1)-1 => not nm));
assertion ((i_r=7*(i_n+1)-1 => n=r) and (i_r<>7*(i_n+1)-1 => not n) and
    (i_r=5*(i_m+1)-1 => m=r) and (i_r<>5*(i_m+1)-1 => not m) and ((n and m) =
    nm)) or init;
initial init;
final not (init or ok);
```

Listing 5.11: Hyperperiod of 5 and 7 is period of 35

5.2.4.1 Finiteness

Given the definition of finiteness in Definition 2.6.18, it is not possible to define such a condition in the language of NBac. It has to do with the fact that we need an existential quantification, but only a propositional formula on state can be written as a property. However, this does not mean that we can not implement the check in principle.

First, we start with the simplest approach, which is to guess the bound. By replacing the variable bound by the existence quantifier with some value, like maximum integer value on the solver or target machine. Surely it is better than nothing and can be good enough if there is a hard limit, but in general it does not answer the original question. Additionally, if only a counter violates the bound, it could be acceptable to handle it especially in the implementation, but such a case is not possible with this approach.

We then propose the following sequence of reasoning. As the analysis finds the best overapproximation of the state given the domain it uses, we only have to check if the invariant is present in this abstraction. Essentially, we look for a constraint of form $|i_{c_1} - i_{c_2}| < k$ to be always present in the domain for each pair of c_1, c_2 that we need to check. Another way to think about it is that $|i_{c_1} - i_{c_2}|$ must be evaluated to something other than ∞ on the abstraction. Then if it does, the specification is finite. A way to achieve the best abstraction is to use the most granular partitioning. Then the check of finiteness comes down to checking the existence of bounds on each part of the partition.

Experiments In our experiments, we have first verified already proven results of the previous work [MMS13a; MdS15]. The idea of this specification is essentially the same: once backpressure is added, the whole specification becomes finite. This specification consists in modeling a task with two input dependencies and one output:

$$\begin{aligned} in_1 &\prec proc \\ in_2 &\prec proc \\ proc &\prec out \end{aligned}$$

This specification is not finite though as there could potentially be infinite amount of in_1, in_2 or both, before any out. Thus, we add backpressure $f = \texttt{fastest}(in_1, in_2)\ \texttt{alternates}\ out$ to it. The principle is the following: if in_1 ticks, f clock ticks. This tick triggers the alternation to switch and then allows out only. Then, unless out ticks, f cannot tick again, meaning that the distance in number of ticks between in_1, in_2 and out is at most one. Only NBac is able to derive this fact.

The next experiment was not possible to prove without using abstract interpretation. The specification is about expressing a *logical* jitter. While we define a new constraint, relative periodic, in MRTCCSL (Section 4.2.1), that expresses the intention more precisely, the original implementation of jitter in CCSL is:

$$\begin{aligned} min &= t\ \$\ 2\ \texttt{on}\ r \\ max &= t\ \$\ 3\ \texttt{on}\ r \\ t_d &= t\ \$\ 1 \\ min &\prec t_d \prec max \end{aligned}$$

```
1  state
2      init: bool;
3      ok: bool;
4      i_t,i_td,i_r,i_min,i_max: int;
5      minr_sampled,tmin_delay0,tmin_delay1: bool;
6      maxr_sampled,tmax_delay0,tmax_delay1,tmax_delay2: bool;
7  input
8      t,td,r,min,max: bool;
9  local
10     i_tn,i_tdn,i_rn,i_minn,i_maxn: int;
11 definition
12     i_tn = if init then 0 else i_t + (if t then 1 else 0);
13     i_tdn = if init then 0 else i_td + (if td then 1 else 0);
14     i_rn = if init then 0 else i_r + (if r then 1 else 0);
15     i_minn = if init then 0 else i_min + (if min then 1 else 0);
16     i_maxn = if init then 0 else i_max + (if max then 1 else 0);
17 transition
18     init' = false;
19     i_t' = i_tn;
20     i_td' = i_tdn;
21     i_r' = i_rn;
22     i_max' = i_maxn;
23     i_min' = i_minn;
24     minr_sampled' = if init then false else if r then false else (minr_sampled
           or min);
25     tmin_delay0' = if init then false else if r then (minr_sampled or min)
           else tmin_delay0;
26     tmin_delay1' = if init then false else if r then tmin_delay0 else
           tmin_delay1;
27     maxr_sampled' = if init then false else if r then false else (maxr_sampled
           or max);
28     tmax_delay0' = if init then false else if r then (maxr_sampled or max)
           else tmax_delay0;
29     tmax_delay1' = if init then false else if r then tmax_delay0 else
           tmax_delay1;
30     tmax_delay2' = if init then false else if r then tmax_delay1 else
           tmax_delay2;
31     ok' = if init then true else (ok and i_minn - i_maxn <= 1 and i_maxn -
           i_minn <= 1);
32 assertion ((min = (r and tmin_delay1)) and (max = (r and tmax_delay2)) and
       (i_t < 1 => not td) and (i_t >= 1 => t = td) and (i_td=i_min => not td)
       and (i_td=i_max => not max)) or init;
33 initial init;
34 final not (init or ok);
```

Listing 5.12: NBac program of logical jitter

We translate the specification in the NBac program in Listing 5.12. In this case, both NBac and ReaVer end up proving the property that the difference between number of min and max is bound, and this bound is exactly 1. ReaVer can prove it with both standard and accelerating analysis.

Heuristic While dividing the state space by every condition would yield the greatest precision we can achieve. Unless we split by conditions not present in the definition of the program, but which may lead to non-termination of the analysis. However, as partition representation is not free and will grow potentially exponentially with each split, it is better to avoid such case and use only a partition that we need to prove the property. Thus, we propose a heuristic for partitioning, specifically for finiteness check, which was inspired by the motivational examples shown before (Chapter 3).

We approach the problem the following way: variables change when clocks tick, while a clock can depend on variables and other clocks. Thus, the difference limit between two counters may only occur when there is some kind of relation between the clocks. More specifically, we need to establish an interdependence, either an alternation using a Boolean variable, a binary relation between counters, a strong correlation among clock inputs themself or via a transitive relation to another interdependent set of clocks and so variables.

Our proposition then is to construct the following graph:

- every variable of NBac program is a vertex;
- changes to state variables and dependencies on state variable are directed edges;
- operations and relations between variables (including clocks) are undirected edges;
- (sub)expressions make anonymous vertices, which in the graph may look like edges connecting to edges.

Then, in the graph we identify strong relations, which we call anchors †. These strong relations hold on Boolean variables (excluding clocks) and equality relations. Then the partitioning is constructed from the relations between variables (numerical or Boolean comparisons) that are met in a tightest loop that spans the counters that need to be bounded and the strong relations (anchor points). With such partitioning, abstract reachability analysis is performed and if it is not proven that there is a bound on counters, other loops can be tried until the worst case is reached: every syntactic condition of the original program has to contribute to the partitioning.

We illustrate the use of such a graph on specification $a < b < \alpha \parallel \alpha = a \$ 1$. As discussed before (Section 5.2.3.1), it is equivalent to alternation, thus the difference between counters of a and b have to be bound between zero and one. This specification generates the graph shown on Figure 5.4, with a green loop passing though the vertices with † symbols. This gives us a partition by conditions $i_a = i_b, b_1 = i_\alpha, a = \alpha$ and their negations. Then because $a = \alpha$ implies $i_\alpha + 1 = i_a$, it propagates to differences between i_a, i_b and i_α. Any other partition would be irrelevant to the proof of finiteness.

5.2.4.2 Liveness and weak-liveness

The definition of liveness Definition 2.6.17 contains universal quantification over all schedules (solutions) to a given specification. To check liveness, we would therefore need to have an exact representation of the language. From abstract interpretation point of view, it would require an

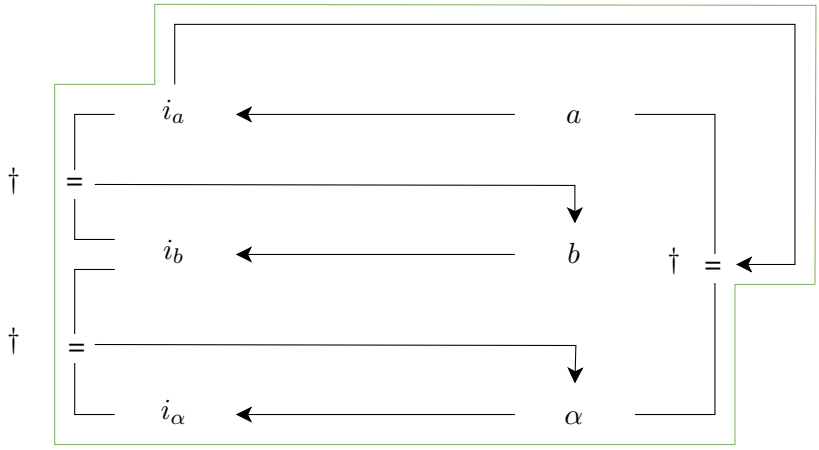


Figure 5.4: Relation graph of $a < b < a \$ 1$

exact representation of the state, which is usually not possible to obtain. If it existed though, we would need to check that every cycle between the locations contains every clock.

One way to relax this condition is to check if there is path in principle, but not required to actually reach it, the weak-liveness Definition 4.6.1. Still, doing it requires effective underapproximation domains for the state, which seems to not exist in general. The main reason is that underapproximation abstract domains must be closed under union, but abstract domains are rarely such [ABG22], with the exception of disjunctive completion (Section 2.9.4.1). Thus, while disjunctive completions can be used to turn any non-closed domain into one, loops in control flow graph may lead to infinite series of disjunctions. And using some construction analogue of widening (for example, intersection) will easily lead to bottom element, from which underapproximation may not recover.

If we assume that this is not the case for our specification (disjunctions are finite and do not quickly lead to bottom), the algorithm to find live schedules is defined as the following sequence of actions:

1. perform analysis until stabilization (because finite it will terminate);
2. during the analysis all disjunctions in underapproximation domains should be split into different locations, as to specialize transitions;
3. go through the control flow graph as a graph and collect all cycles;
4. then if a cycle contains all clocks, existence of a live schedule is proven. To prove weak-liveness, we need to find out if loops form strongly connected components with clocks in the property. Every loop does not have to contain all clocks, but a subset of clocks defining a subsystem. This way we can have only part of the behaviour that becomes infinite, while it is still possible to switch at some points.

5.2.4.3 Time-triggered semantics implementation

To implement time-triggered semantics of real-time CCSL we do not need to change much, as is shown in the operational version of the semantics (Section 4.2.3.2). An additional condition

should be added to the assertion, which we specify below:

$$
\begin{aligned}
c_2 = \texttt{delay}\ c_1\ \texttt{by}\ [d_1, d_2]\texttt{[tt]} \quad &\Rightarrow \quad \mathrm{head}(c_1 c_2 \_ \texttt{q}) + d_2 = \texttt{next} \implies c_2 \\
c = p \cdot i^{\texttt{th}} \pm \texttt{rel.}e + \varphi\texttt{[tt]} \quad &\Rightarrow \quad i_c = 0 \implies (\texttt{next} = \varphi_2 \implies c) \wedge \texttt{next} = c_{lts} + p_2 \implies c \\
c = p \cdot i^{\texttt{th}} \pm \texttt{abs.}e + \varphi\texttt{[tt]} \quad &\Rightarrow \quad \texttt{next} \leq c_{lts} + p + e_2 \wedge \texttt{next} = c_{lts} + p + e_2 \implies c
\end{aligned}
$$

As usual, there is no interpretation in which sporadic constraint can be time-triggered.

5.2.5 Analysis improvement

In this subsection we want to discuss how we can improve the analysis. One way to improve is through acceleration along with some specific optimisations. As such, we do not present modifications to the principle of acceleration, but modify the representation such that the preconditions needed for acceleration are satisfied.

5.2.5.1 Hyperperiod expansion

A common case in reactive system is to mix periodic behaviours with different periods yet on the same source. In case the periodic behaviours define tasks that exchange the data, or there are requirements (like end-to-end latency [FH07]) and properties that depend on them, we need a way to check every unique combination of their occurrences and not occurrences. In such cases, we usually compute their hyperperiod. Though, depending on the period durations and offsets, these clocks may or may not end up ticking at the same instant. In CCSL, this is denoted as follows:

$$
\begin{aligned}
a_1 &= \texttt{skip}\ \varphi_1\ \texttt{every}\ p_1\ r \\
a_2 &= \texttt{skip}\ \varphi_2\ \texttt{every}\ p_2\ r
\end{aligned}
$$

When we try to analyse systems that contain such constraints using abstract interpretation, we have troubles in being precise. It has to do with the representation that we get if we try to analyse it as it is: the present guards in periodic relations do not give enough to determine the whole order of ticks that appear before the two tick synchronously and then loop. To fix this we propose a solution that detects and expands the hyperperiod in minimal amount of locations. An important note, is that the expansion works regardless of an intersection of periodic clocks, as in the case of Section 5.2.3.1.

The principle of the method is enumeration of the composed state needed to express the hyperperiod, with individual a_1 and a_2 still ticking inside the hyperperiod, with the caveat that we compress (make symbolic) the states where the behaviour is to stutter on r until either a_1 or a_2 ticks. In other words, we partition the state space of i_{a_1}, i_{a_2}, i_r such that guarding condition of a period, $i_{a_1} = p_1 \cdot i_r - 1$, required for a_1 to occur, is present several times inside the hyperperiod. For this we introduce an additional variable, i_{hp}, for counting the hyperperiods. It is a reference point from which we then can redefine each of the periodic behaviours to appear several times inside the hyperperiod. In this case it induces a partition P_{hp}, which we define the following way:

$$
\begin{aligned}
P_j \stackrel{\text{def}}{=} \quad & \{[\![i_r = p_j \cdot x + \mathrm{lcm}(p_1, p_2) \cdot i_{\mathrm{hp}} + \varphi_j]\!] \mid 0 \leq x \leq \mathrm{lcm}(p_1, p_2)/p_j\} && (5.12) \\
\bigcup \quad & \{[\![i_r < p_j \cdot x + \mathrm{lcm}(p_1, p_2) \cdot i_{\mathrm{hp}} + \varphi_j]\!] \mid 0 \leq x \leq \mathrm{lcm}(p_1, p_2)/p_j\} && (5.13) \\
\bigcup \quad & \{[\![i_r < \varphi_j]\!]\} && (5.14) \\
P_{\mathrm{hp}} \stackrel{\text{def}}{=} \quad & \{p_1 \cap p_2 \mid (p_1, p_2) \in P_1 \times P_2 : p_1 \cap p_2 \neq \emptyset\} && (5.15)
\end{aligned}
$$

where j is 1 or 2 given c_1, c_2, $[\![v_1 = v_2]\!]$ is interpreted as the set of solutions to the expression $v_1 = v_2$, i.e. the expression is treated as a singular object. If to put informally, the P_j defines a partition to one of the periodic constraints with period p_j and this partition contains the part of the state, where the clock c_j has just ticked (Equation (5.12)), before it can tick (Equation (5.13)) and the initial part (Equation (5.14)). Then the individual partitions are "synchronized" into the common partition P_{hp} (Equation (5.15)) via a Cartesian product. The combinations of partitions that do not have solutions are removed, i.e. when $p_1 \cap p_2 = \emptyset$. Because all conditions relate to the same variable i_r, the control flow graph will be simplified after forward analysis to become a chain of transitions (see Figure 5.5). Additionally, they can then be accelerated, if the periods p_1, p_2 are big enough.

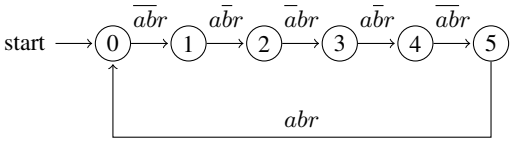
A constructive algorithm that does the same partition while avoiding the whole analysis part (we know that other partitions are unreachable by construction) and can handle not coprime and not zero offsets is described in Algorithm 5.2. Notes about the syntax:

- as in denotational definition, $[\![a < b]\!]$ means a symbolic Boolean expression and not executed in the algorithm itself;
- but when we use underline as in $\underline{\texttt{prev}}$, for example in $[\![\underline{\texttt{prev}} < i_r < \underline{\texttt{next}}]\!]$, the underlined expression will be interpreted and embedded as a *value* into the symbolic expression when the algorithm runs. Thus, if $\texttt{prev} = 2, \texttt{next} = 3$ at the point when the expression is constructed, it will appear as $[\![2 < i_r < 3]\!]$;
- when we write $\texttt{loc}_1 \xrightarrow{\text{label}} \texttt{loc}_2$ we construct a transition in the control flow graph between the previously defined locations $\texttt{loc}_i$, and it is triggered when a specific combination of clock ticks (their associated Boolean variables are set to true in the program). We use the same automata notation as in the rest of the document. We do not specify the loop numerical condition for intermediate locations, as it is implied by the partitioning itself.

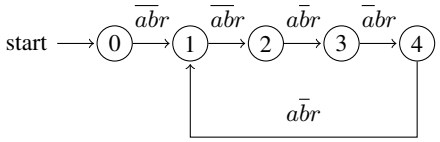
We present some examples on Figure 5.5. For periods like 5 and 7, the saving in location size is already 42%. In the same example, there are 7 loops to accelerate (marked red on the figure), meaning that the number of transitions and locations can drop even further.

This approach is only possible when the parameters are known statically from preprocessing and when the analysis with partitioning by other conditions did not start yet. It is more preferable to run it during the analysis fully, if we can detect the pattern in the control flow graph itself. Fortunately detecting hyperperiods is relatively easy: if conditions of form $i_r = p * (i_a + 1) + \varphi - 1 \implies a = r$ are detected in the *definition* of a transition guard, the algorithm can be performed on p and φ. By using only definitions present we are sure this partitioning terminates. While we do not believe that a program can produce infinite chains of hyperperiod expansions, we have no proof of that. Otherwise, we could try to detect that the reachable state space have the same pattern, for example if some relation has induced an equivalence in between some clocks, and so the hyperperiod would be discovered after the first round of the analysis.

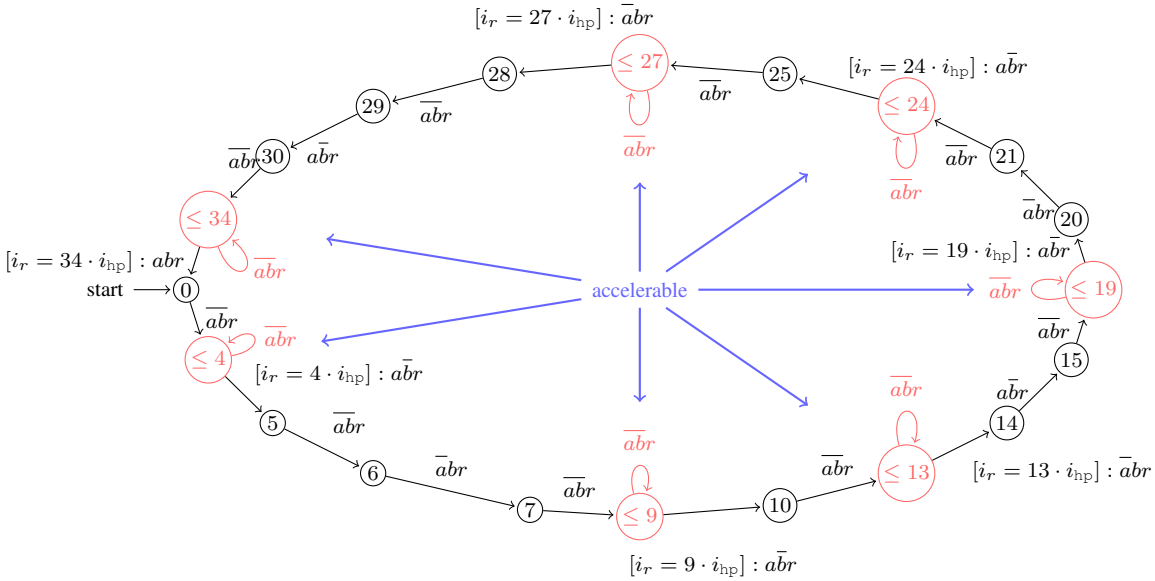
Use in subspecification relation The following specification is a prevalent pattern in embedded systems. But neither NBac or ReaVer can encode it exactly because the hyperperiod is not handled. It consists in checking that intersection of periodic constraints is periodic too, with the period of



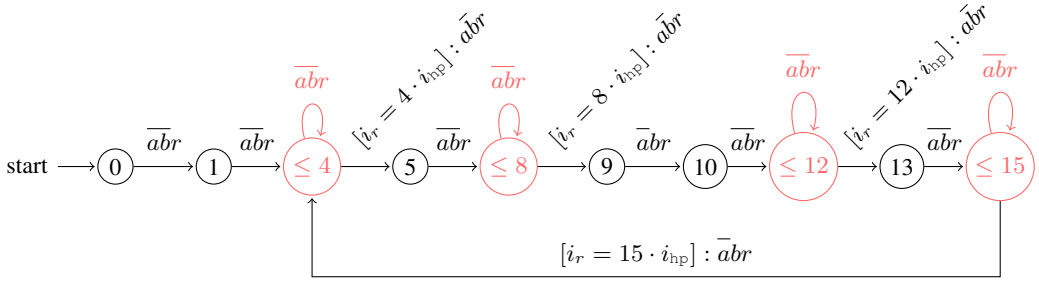
(a) $p_a = 2, \varphi_a = 0, p_b = 3, \varphi_b = 0$



(b) $p_a = 2, \varphi_a = 1, p_b = 4, \varphi_b = 0, \varphi_a \not\equiv \varphi_b \mod \gcd(p_a, p_b)$



(c) $p_a = 5, \varphi_a = 0, p_b = 7, \varphi_b = 0$



(d) $p_a = 4, \varphi_a = 1, p_b = 6, \varphi_b = 4, \varphi_a \not\equiv \varphi_b \mod \gcd(p_a, p_b)$

Figure 5.5: Hyperperiod expansion of $a = \texttt{skip}\ \varphi_a\ \texttt{every}\ p_a\ r \parallel b = \texttt{skip}\ \varphi_b\ \texttt{every}\ p_b\ r$

their multiple. Formally it is:

$$\begin{aligned} &a = \texttt{skip}\ \varphi_a\ \texttt{every}\ p_a\ r \\ &b = \texttt{skip}\ \varphi_b\ \texttt{every}\ p_b\ r \quad \Subset \quad i = \texttt{skip}\ \varphi\ \texttt{every}\ p_a \cdot p_b\ r \\ &i = a * b \end{aligned}$$

```
1  state
2      init: bool;
3      ok: bool;
4      i_r,i_n,i_m,i_nm: int;
5  input
6      r,n,m,nm: bool;
7  transition
8      init' = false;
9      i_r' = if init then 0 else i_r + (if r then 1 else 0);
10     i_n' = if init then 0 else i_n + (if n then 1 else 0);
11     i_m' = if init then 0 else i_m + (if m then 1 else 0);
12     i_nm' = if init then 0 else i_nm + (if nm then 1 else 0);
13     ok' = if init then true else (ok and (i_r=35*(i_nm+1)-1 => nm=r) and
           (i_r<>35*(i_nm+1)-1 => not nm));
14 assertion ((i_r=7*(i_n+1)-1 => n=r) and (i_r<>7*(i_n+1)-1 => not n) and
       (i_r=5*(i_m+1)-1 => m=r) and (i_r<>5*(i_m+1)-1 => not m) and ((n and m) =
       nm)) or init;
15 initial init;
16 final not (init or ok);
```

Listing 5.13: Intersection of periodics is periodic with their periods' LCM

This specification is translated into the NBac program shown in Listing 5.13. Thanks to the generalized Chinese remainder theorem, the relation is satisfied by offsets $\varphi_a \equiv \varphi_b \mod \gcd(p_a, p_b)$, regardless of whether the periods are coprime, and always if they *are* coprime.

5.2.5.2 Concurrent analysis

Another pattern that we would like to handle is that of loosely synchronized components, heavily inspired by the spark engine controller case study (Section 3.3). In this example there are two parts that are related with only four clocks, and so we would like to not have to do the typical partitioning, as it will be exponential. Meaning, we would like to propose a method to split loosely coupled subparts inside a given specification, and allow these parts to be analysed . First, we propose to optimize a particular pattern, then we generalize it with an automatic partitioning.

The pattern is characterized by a clock a and consists of a loop, with no occurrence of a in it, and a transition to another location with a always present. Then our proposition is the following: if we trace the dependencies of the guard of the transition away from the loop into the transition of the loop, and discover that the clocks and state variables are independent modulo a, i_a and current time, then we can split such a pattern into several *concurrent* transition systems synchronized at the end by a. We illustrate this translation on Figure 5.6 using specification $a \prec b \prec c$.

Concurrent transition partitioning More technically, we do an analysis similar to the original NBac's approximation of transition relations [Jea02], but the transition inside the analysis is a control flow graph instead of *one single* transition. We call it *concurrent transition partitioning* and the algorithm to obtain it is the following:

1. build the relation graph, where variables are vertices and any interaction is an edge (as on Figure 5.7), between the guard of the away transition ρ and the loop transition τ (as on Figure 5.6);

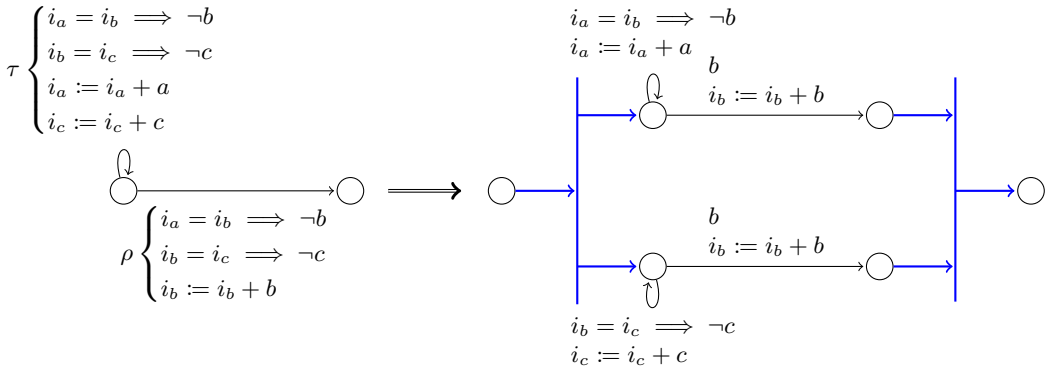


Figure 5.6: Splitting of a transition into concurrent branches

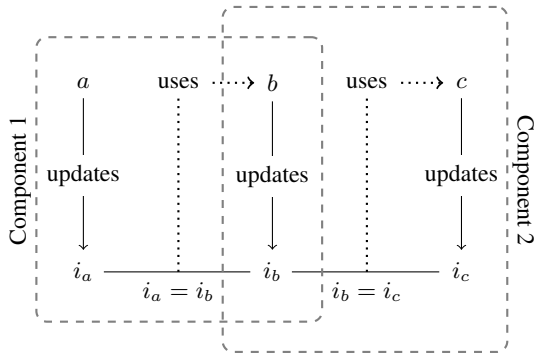


Figure 5.7: Relation graph of $a \prec b \prec c$

2. determine if it is possible to cut it in half by variables a and `prev`;
3. if possible, the parallel transitions are then interpreted the following way:
   (a) first concurrent transition system gets updates of the first component, second others respectively;
   (b) each is rewritten such that every variable has both an old and a new versions, and starting relation is $v_{\text{old}} = v_{\text{new}}$;
   (c) internally, the transition consists of two locations, start and finish, with a loop on the first location, and ending with a ticking; practically the same as the matched pattern itself, but specialized;
   (d) each branch is analysed as previously described but on *new variables only*, possibly including subdividing with the same method;
   (e) the forward analysis then intersects together with branches' transition relation;
   (f) in backward analysis the intersection is performed at the beginning of the branching.
4. the obtained transition relation is then used as any other transition during the analysis.

Concurrent location partitioning However, there is no guarantee for this pattern to naturally occur, and if so, it might be altered by other partitioning techniques. Thus, we propose a new

partitioning technique that forces this pattern to occur more frequently. We call it *concurrent location partitioning*. And it uses the following algorithm:

1. construct an hypergraph for the specification as follows: vertices are constraints, hyperedges are clocks;
2. find a cut through edges separating periodic sources (for example by using techniques from [VBK22]); we consider clocks defined using them chronological and usually these are separate physical processes, suggesting concurrency in modeling; the clocks defined by them (and so neighbouring in the graph) should not be cut;
3. then the clocks, which we will call *significant*, of the cut define partitioning: the parts are characterized not by their state, but by the clock of their previous transition. Additionally, there is a starting location into which the initial one leads and is supposed to be the entry and stuttering point before some significant clock ticks. It is important to understand that the location is changed only when the significant clock ticks. Until then, any other transition can fire;
4. Then each partition we define above, if has incoming transitions, will satisfy a part of the requirements for concurrent transition partitioning.

We do not have experimental data to confirm how big should be the final components, but it seems right for at least one component to have size of at least 4 constraints, otherwise the dimensionality of the state space is not big enough to try to optimize. We illustrate how it can be applied in Section 5.2.6.

The size of the cut defines the partitioning size, which results in reduction of complexity but also in loss of precision. Further research and use case analysis are required to find the optimal cut.

A similar idea is described in [Dan+17; SM18], but the key difference is that in CCSL any constraint can be a “thread”, so it is part of the method to figure out how and when to split them.

5.2.5.3 Flattening of task-mutex pattern

Coordination of environment and reactions (computations) is a common usage of CCSL, part of which is a task-mutex pattern. There a CPU is considered a resource which is reserved (to execute a task) via a mutex. This very frequent, shared resource pattern, must be made explicit in CCSL, as otherwise tasks could execute in whatever order without being sure that the execution resource (CPU) is available.

This pattern is implemented in CCSL using a combination of realtime periodic constraint for ticks of reference clock, mutex for task allocation on CPU and logical delay constraint for execution of the task on the CPU’s reference clock. In this particular combination, using a delay causes problems to conduct an efficient analysis. Even though, each of the constraints are finitely representable, delay requires using $d + 1$ Boolean variables*, or 2^{d+1} state, where d is the delay parameter. These variables are usually the first to be split in the analysis, which in this particular case is *not* beneficial.

*This point makes delay impossible to parametrize with the currently used abstract interpretation, as variables should be statically known before the analysis begins.

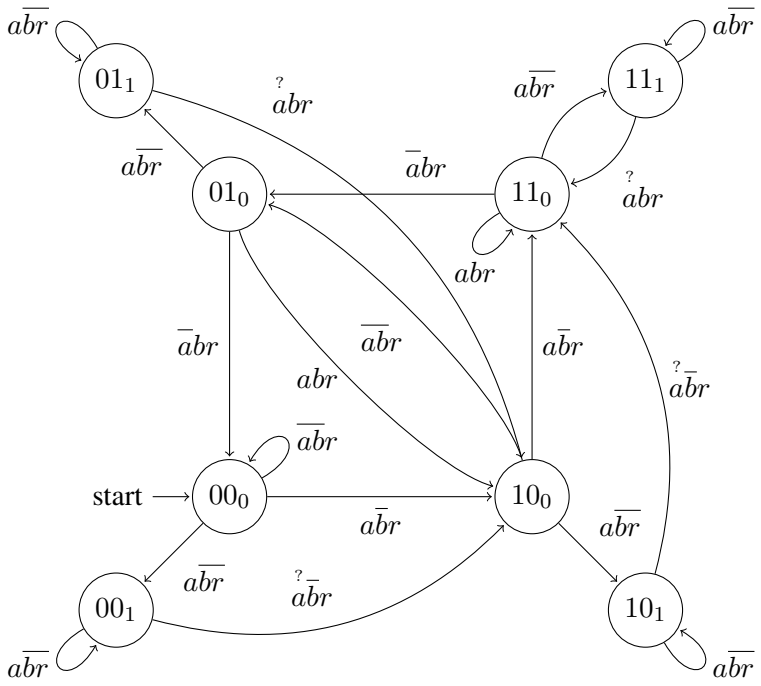


Figure 5.8: Delay $b = a \$ 2 \texttt{ on } r$ ($\emptyset$-loops skipped)

As a minimal example, we show a specification consisting of two constraints, ternary delay and mutex that need to be synchronized, respectively Figure 5.8 and Figure 4.11:

$$\texttt{mutex}\{s \mapsto f\} \tag{5.16}$$

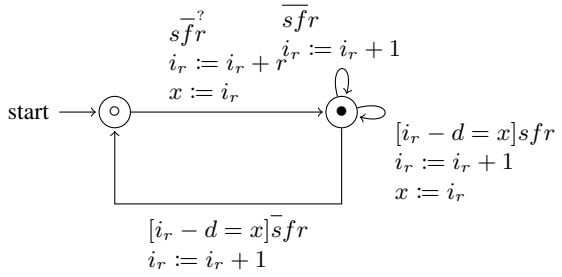
$$f = s \$ d \texttt{ on } r \tag{5.17}$$

Thus, we propose a new domain, based on BDDs, with new operations, and a scheme to accelerate this particular case to the end result shown on Figure 5.9. This domain is further described in the next paragraph.

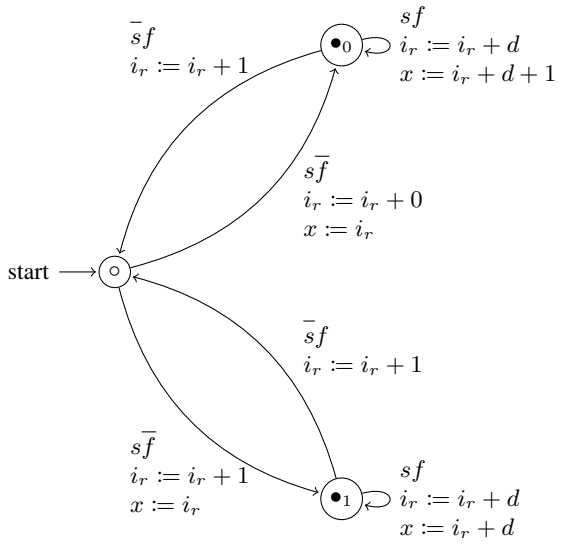
Constant size Boolean array domain This domain $\mathbb{A}_n$ of size n encapsulates n Boolean variables. Internally it is represented by a BDD (Section 2.8.3) plus a bottom element. The two are disconnected from each other, as there is actually no action that could be taken on regular elements of the domain to result in the bottom element. The domain represents an array that is initialized with n false elements and always maintains its size, but its operations are a mix between arrays and queues.

The domain supports the following operations:

- it is important to understand, that all the operations below are and will be defined as they act on a particular value of the array, but as a domain it is applied on all the variants it contains;
- $q[i] : \mathbb{A}_n \to \mathbb{N} \to \mathbb{B}$ indexing of the array to obtain a value at index i; the indexing starts from 1;



(a) Intermediate partitioning



(b) Final result

Figure 5.9: Acceleration of mutex with delays

- values of the domain are expressed as a formula on the $q[i]$ or using notation $?010$, where $?$ means any value, $0, 1$ are Boolean. Alternatively, $0 \dots 0$notation can be used to introduce or match on ranges of same elements, with subscript as in $0 \dots 01_i0 \dots 0$ to match the index too;
- $\mathrm{pushpop}(q, v) : \mathbb{A}_n \to \mathbb{B} \to \mathbb{A}_n$ combines push and pop of a regular queue, so that the queue stays of constant size, but in fact overflows with the last element at every invocation; takes an array and a Boolean variable as arguments, returns the modified array; its definition is $\mathrm{pushpop}(q, v) = q'$ where $q'[1] = v \wedge \forall 1 < i \leq d : q'[i] = q[i-1]$;
- widening is not needed as the domain is finite;
- operations $\sqcup^\sharp$ and $\sqcap^\sharp$ are standard $\wedge\vee$ operators of BDDs;

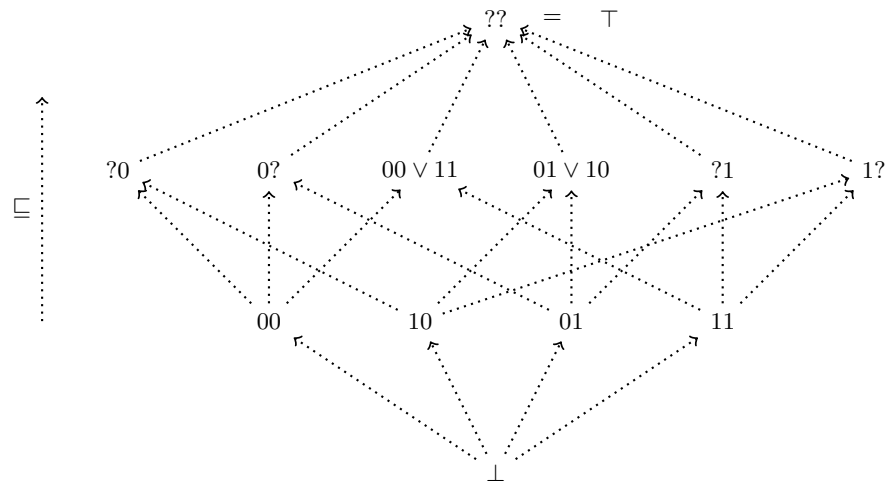


Figure 5.10: Hasse diagram of constant size Boolean array domain of size 2 (? is $0 \vee 1$)

- partial order $\leq^\sharp$ is defined as on Figure 5.10; a solution to formula $a \wedge b$ is less than formula b;
- lub $\top^\sharp$ and $\bot^\sharp$ exist, i.e. it is a complete lattice (Definition 2.9.5); additionally, it is not possible to reach bottom element by applying $\mathrm{pushpop}$; $\top^\sharp$ acts as one and zero for $\sqcap^\sharp$ ($\bot^\sharp$ for $\sqcup^\sharp$ respectively):

$$\top^\sharp \sqcap^\sharp x = x \qquad \bot^\sharp \sqcup^\sharp x = x$$
$$\top^\sharp \sqcup^\sharp x = \top^\sharp \qquad \bot^\sharp \sqcap^\sharp x = \bot^\sharp$$

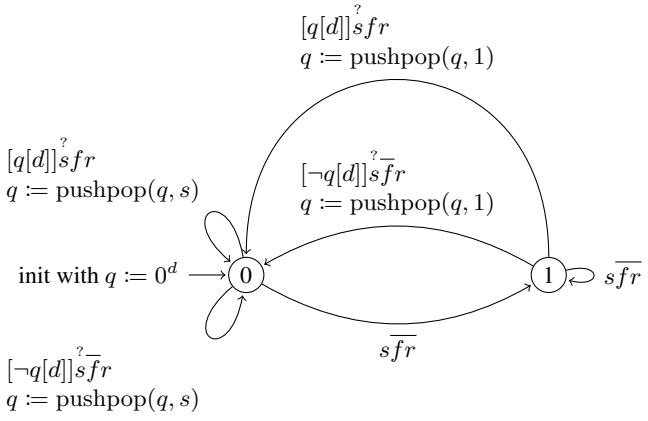


Figure 5.11: Delay $f = s \ \$ \ d \ \mathtt{on} \ r$ with constant size Boolean array domain

Then by using this domain we are able to implement an alternative version of the ternary delay, shown on Figure 5.11.

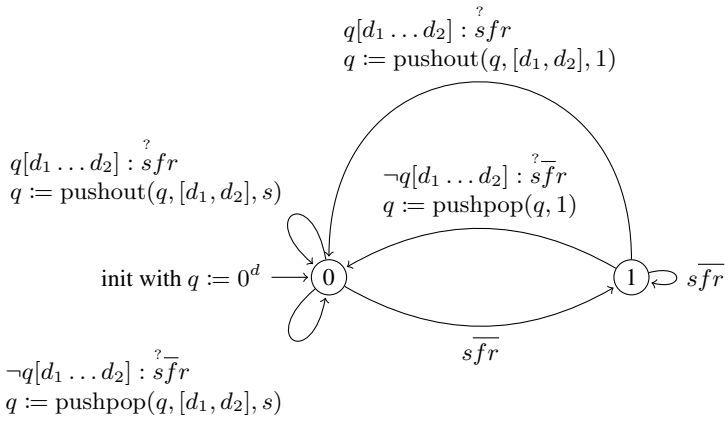


Figure 5.12: Delay $f = s \ \$ \ [d_1, d_2] \ \texttt{on} \ r$ with constant size Boolean array domain

Case of interval delay As we extend delay by replacing a single scalar parameter by an interval, there are two ways to implement this new construct. Either we unwrap the syntax into a list of constraints with two regular delays and then use the technique above, or we modify it. We chose the second option.

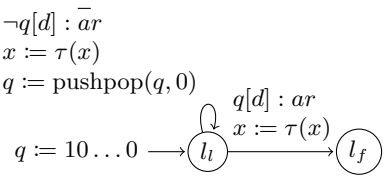
Delay with interval, written as $b = a[n, m] \ \$ \ r$, where a, b, r are clocks, and $0 \leq n \leq m$, uses the same domain, but with a size m, as it is the biggest out of the interval bounds. The modification consists in introducing a new conditional and a new operation, slice access and pop at the distance respectively. The first checks if there is a true value for some index inside a slice, written as $q[x \dots y] : \mathbb{A}_n \to \mathbb{I} \to \mathbb{B}$, where $x \leq y$. It is defined as $q[x \dots y] \overset{\text{def}}{=} \bigwedge_i^{x \leq i \leq y} q[i]$. Second makes false the value with the biggest index from the slice, i.e. $\text{pushout}(q, [x \dots y]) : \mathbb{A}_n \to \mathbb{I} \to \mathbb{A}_n = q' \iff i = \arg\max_{x \leq i \leq y}(q[i])$. The automaton of the interval domain is shown on Figure 5.12.

Acceleration Given the data structure above and the operations, we have identified conditions similar to those of the original acceleration in which the array is entirely or mostly eliminated.

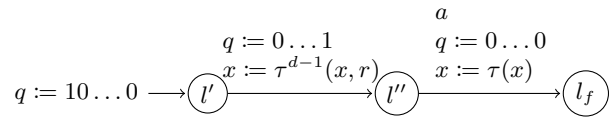
On Figure 5.13 we can see the idea: when we push a true value to the array in the first location, in the second location there is a loop triggered by reference clock r, which lasts until the true value reaches the head. Then the last transition makes the array empty again, and the cycle may continue indefinitely.

If there is a pattern, shown on Figure 5.13, we can accelerate it. The idea is the following: in this case, we have to know that the transition to the loop is the only one, and that it brings a true value (token) at the start of the array. The loop is void of the changes to the array, other than moving it to the end. Then the only exit is when the token reaches the end of the queue. In case of an interval delay we modify the pattern, and then the acceleration is not a function anymore, but a relation, as the token moves within an interval (leaving multiple possible choices).

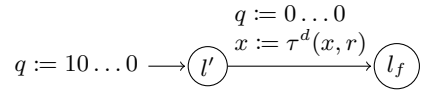
Next, we introduce the case where we have two delays on the same reference clock, yet they reset the mutex when the latest clock ticks ($\texttt{mutex}\{a \mapsto \texttt{slowest}(b, c)\}$, Figure 5.14), which statically depends in this case on their parameters. There the position, and so the accelerating transition depend on the delay before. So the smaller of two delays can be accelerated using



(a) Original control flow graph



(b) Without ignoring a



(c) If ignoring a

Figure 5.13: Acceleration of constant size Boolean array domain

partitioning information, while the second delay has to wait for the results of the first analysis. Then the pattern is the same, except that we start not from the first position in the array but directly at the *accelerated* position.

Before matching on a control flow graph though, we introduce a shifting operation, an accelerated version of pushpop, in Definition 5.2.1.

Definition 5.2.1 (Array shift)**.** Array domain shift $\oplus$ is an operation defined as

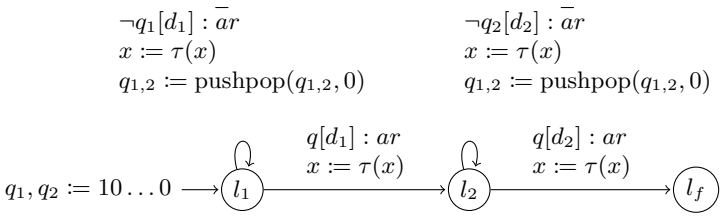
$$q' = q \oplus [a, b] \iff \forall 1 \leq i \leq n : q'[i] = \bigwedge_{j}^{a \leq j \leq b} i - j \geq 0 \implies q[i - j]$$

where $a \leq b < n \in \mathbb{N}$ is the shift interval, $q \in \mathbb{A}_n, i - j \geq 0$ checks that the index is in defined range, otherwise ignores the result. Meaning that if the sum of the present indices and the shift is bigger than n, the domain removes these elements from the array.

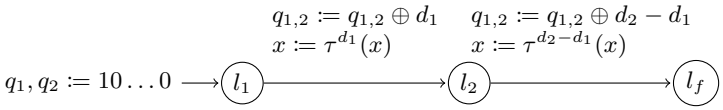
As a reminder, if given a $x \coloneqq \tau(x)$, we denote τ^n a function τ applied n times to itself. Such a function can be accelerate and this can be done in an analytical (fast) way. For $x \coloneqq \tau^{[a,b]}(x)$, interpolation between $x \coloneqq \tau^a(x)$ and $x \coloneqq \tau^b(x)$ can be performed, but only if it is linear. Then, when τ is pushpop, we propose an acceleration based on the shift.

Proposition 5.2.1 (pushpop acceleration)**.** *Given* $q \coloneqq \tau(q) \iff q \coloneqq \text{pushpop}(q, 0)$ *its acceleration* $\tau^{[\varphi_1, \varphi_2]}$ *is defined as*

$$q \coloneqq \overset{[\varphi_1, \varphi_2]}{\text{pushpop}}(q, 0) \iff q \coloneqq q \oplus [\varphi_1, \varphi_2]$$



(a) Original control flow graph



(b) With acceleration

Figure 5.14: Acceleration of two delays with mutex reset on slowest with parameters $d_1 \leq d_2$

With this we propose a method to accelerate both simple and complex delays presented before by using Proposition 5.2.1 and matching on locations, transitions and their guards.

Proposition 5.2.2 (Generalized array acceleration)**.** *The control flow graph of the program can be rewritten using the following rule:*

$$\exists k_1 \leq k_2 < i_1 \leq i_2 \in \mathbb{N}, l_l, l_f \in L :$$

$$\bigwedge \gamma(l_l) = (q = \bigvee_{j}^{k_1 \leq j \leq k_2} 0 \dots 1_j \dots 0) \tag{5.18}$$

$$\bigwedge l_l \xrightarrow[x \coloneqq \tau(x)]{\neg q[i_2] : r \quad q \coloneqq \text{pushpop}(q, 0)} l_l \tag{5.19}$$

$$\bigwedge l_f \xrightarrow[q \coloneqq \text{pushpop}(q, 0)]{q[i_1 \dots i_2] : r \quad x \coloneqq \tau(x)} l_f \tag{5.20}$$

$$\bigwedge l_l \not\to l_l \tag{5.21}$$

$$\bigwedge l_l \xrightarrow[x \coloneqq \tau^{[i_1, i_2] - [k_1, k_2]}(x)]{q \coloneqq q \oplus [i_1, i_2] - [k_1, k_2]} l_f \tag{5.22}$$

where τ is an accelerable function guarded by reference clock r, $q = 0 \dots 1_i \dots 0$ denotes that a value 1 is in position i in the array.

To explain:

- by Equation (5.18) we demand that the state of the array contains at least one token or that the abstract array has tokens arranged between k_1 and k_2, but as disjunctive variants; this is expressed as a condition on *concretization* because it is not a relational domain and a partition of the same shape will not appear;
- then in the loop state l_l, it must exist two transitions Equation (5.19) and Equation (5.20), with the second one leading out of the loop when the array reaches the allowed region $q[i_1 \dots i_2]$;
- then it is rewritten with removal of the loop and replacing exiting transition by the acceleration of difference in number of positions between current $[k_1, k_2]$ and the guard $q[i_1 \dots i_2]$, in both array and τ; because the array is not relational with respect to x, we lose precision and this is an overapproximation of the behaviour we could obtain. To be precise, the actual relation is for an array with 1 in position i to happen only in pair with τ^i;
- it is important to ensure that acceleration does not push tokens beyond $i_2 + 1$ as in non-accelerated version. Thankfully, subtraction of intervals already ensures that. For example $[3, 4] - [1, 2] = [1, 2], [1, 2] + [1, 2] = [3, 4]$. Otherwise, we would need to place additional condition $\sqcap ? \dots ?_{i_2+1} 0 \dots 0$ in the guard of the final transition, exactly like numerical acceleration of τ, showed in Section 2.9.6.2;
- also, as shown before how to accelerate an array, it can also be part of that function τ, thus accelerating other arrays with the same acceleration. This is exactly what allows us to handle the case of Figure 5.14.

It is important that the final scheme allows to accelerate both numerical variables and arrays. For example, when modelling tasks running on a processor, we would use a mutex and delays on its base clock. Their clocks can be specified as relative periodic. This means, that such an analysis then can yield not only exactly the difference between the start and end of a task in logical clocks, but also in the attached real-time scale, which in turn can be compared with a deadline.

On precision In summary, while this domain does not lose precision in its operations, it does introduce some imprecision from its limited partitioning, compared to pure Boolean variables and more involved interaction with variables in other domains.

As the automaton (Figure 5.11) only checks the end of the array, we can only split by this condition. Meaning that the domain is an abstraction of previous explicit variables, and we cannot obtain the same precision as before without additional transformations. But, at any point, we can opt out to unpack the array into variables in the external state space. This makes the approach better in the favorable case, and worse in non-favorable cases as we have to do the analysis twice.

As for other domains, for example, in case of delay intervals with some other accelerable function τ, the array induces a relationship between the actual last index (reminder, there could be several as it is an abstract domain) and the value after acceleration $\tau^{\otimes}$. But because this is inexpressible, the domain is not relational even with other arrays, nothing else is communicated to other domains, it does not happen.

5.2.5.4 Parametric verification

Parametric verification is natively supported with abstract interpretation, but some assumptions need to hold. The variables are introduced as any regular variable and used as any other variable. We must be careful not to use theses variables in operations, not (easily) supported by domains, like multiplication of variables. Otherwise such operations put us beyond Presburger arithmetics, which is undesirable from the concern for decidability and efficiency. By treating the new variables as regular, it is possible to refine locations by the parameters. And considering that these variables are constant during the evolution of the system (this is CCSL assumption), after the analysis ends, we can go though the locations and see what combination of parameters have violated the properties, or for what parameters solutions always exist. An important note is that both forward and backward analysis *have* to be used, in order for the parameter variables to be consistent across the locations, regardless if a cycle exists in the program.

Additionally, some constraint parameters are inherently difficult to parametrize. This is the case of ternary delay constraint: since delay parameter d decides how many variables are needed and as the variables should be statically known before the analysis begins, we have a contradiction. It is not solved with the domain from Section 5.2.5.3 either, as the parameter is a part of the domain type itself and so no constraints can be placed on it.

It is possible to relax CCSL assumption about the constant parameters and it is actually really desirable in some use cases (for example, in the spark engine use case (Section 3.3), the speed of the engine should be variable). But this variation is usually not linear, and so implementing an efficient acceleration is challenging (though interesting). This addition is not part of this work, but it is a promising direction, and similar to the work of Zelus [BP13].

5.2.6 Illustration: Spark ignition control system

To illustrate how individual propositions of Section 5.2.5 are applied on a real system, we use the Spark ignition control system. In particular, we show that by using our technique the analysis becomes simpler and tractable. In this document, the system was first presented in Section 3.3 and then described as a specification in Section 4.7.2.

We remind the important facts about the system: the engine consists of four cylinders and the controller manages the moment when the spark occurs in each. The cylinders operate in a cycle, where their pistons movement and so the phases it undergoes are related to the crankshaft rotation with a periodic relation. The significant events of the cycle are Overlap Top Dead Center (OTDC), Ignition Top Dead Center (ITDC), First and Second Bottom Dead Center (FBDC and SBDC respectively). Computations that the controller does are related to this cycle and so it places additional constraints on timings of tasks it needs to execute. In the specifications, the task constraints appear as either data dependencies between themself, causality with regards to dead center event and the execution time.

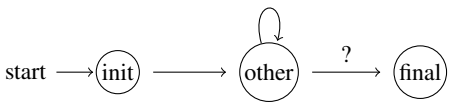


Figure 5.15: Spark ignition control analysis: initial partitioning

Every abstract analysis starts with the initial partitioning into initial, final and everything else locations, as shown on Figure 5.15 and explained in Section 2.9.6.1 as we base our techniques on the approach of this tool.

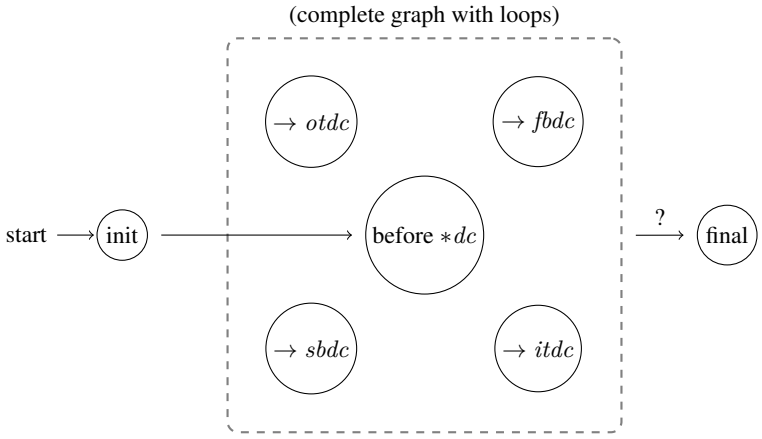


Figure 5.16: Spark ignition control analysis: concurrent location partition

Next, we apply the concurrent location partitioning from Section 5.2.5.2. To remind, the idea is to split the system into several components that only aware of each others state when a significant event happens. We do it by analysing the relationships between the variables of the state and the clocks, and determine the minimal cut we need to make in order to separate them. The partitioning then replaces the location we try to refine, in this case `other`, with a set of completely connected locations, each identified by the clock used to split the system. In this partition, shown on Figure 5.16, the locations are only switches when a significant event happens, like OTDC or SBDC, while allowing other clocks to tick in the same location, which make the system progresses as it normally would.

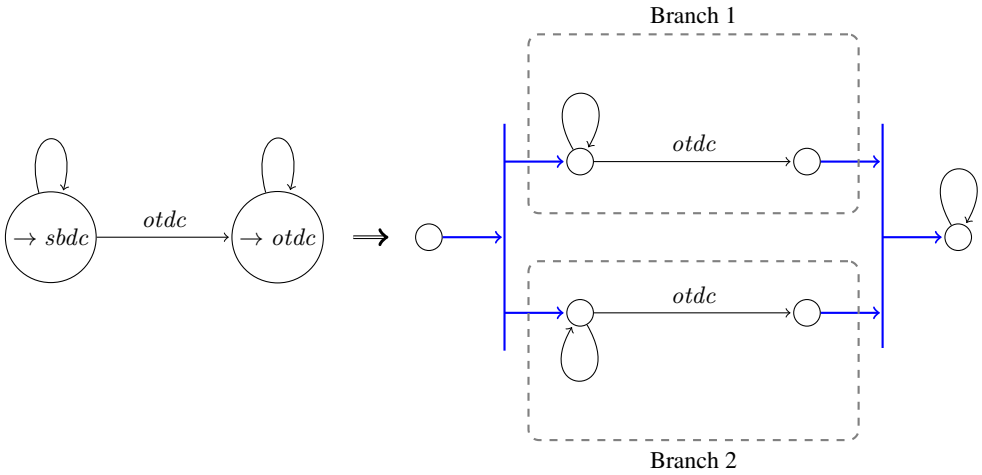


Figure 5.17: Spark ignition control analysis: concurrent transition partition

Each transition that satisfies the conditions described in Section 5.2.5.2 is then split. In each such transition, exist several independent branches, shown on Figure 5.17. In this case, as we

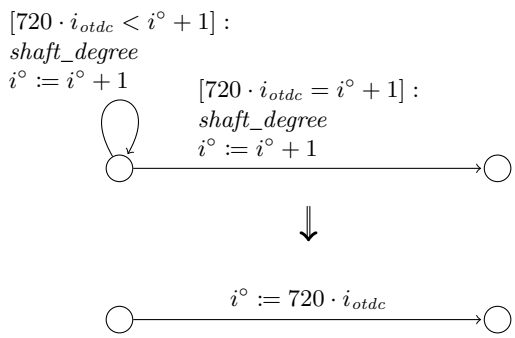


Figure 5.18: Spark ignition control analysis: acceleration of branch 1

split by dead center events and these events only occur when specific conditions are met, it places a strong exit condition in that branch. In the second branch, as we consider the OTDC event after SBDC event, the internal control flow graph is simplified thanks to the timing dependency of oxygen sensing tasks on the exhaust phase which happens precisely between these two events. In both branches, the relations are delays, which are expressed using the constant size array domain from Section 5.2.5.3 and is accelerated as defined in Proposition 5.2.2, as shown on Figure 5.18.

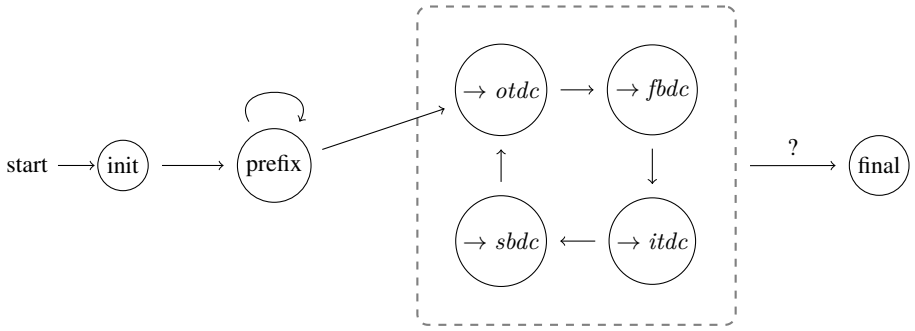


Figure 5.19: Spark ignition control analysis: partition overview after analysis

In the end of the analysis we end up with the control flow graph as shown on Figure 5.19. There, the system follows the global cycle, which is natural, as the controller follows the cylinder cycles. The exception is at the beginning, which we abstractly denote as `prefix`, where the setup for the nominal behaviour happens. Its existence follows from the fact that before all the tasks start one cycle in each task should be executed, and this splits the behaviour of the system. The split in behaviour depends on changes in the state, which manifests in a location by partitioning, and which is what we see here.

In this case, the classical analysis would produce a comparable result with one caveat: it has to use many more locations, each of which would have to be accelerated separately. It is related to the fact that the tasks are often independent from the cylinder unless in rare cases where they do depend on it. In this case the analysis has to make a product of all combinations of the task arrangements and the cylinder-defined state.

```
1   hyper_partition(p1: int, p2: int, phi1: int, phi2: int) -> (location
        set, transition set) {
2       assert p1 > 0 and p2 > 0 and phi1 >= 0 and phi2 >= 0;
3       hyperperiod = lcm(p1, p2);
4       offset1, offset2 = 0, 0;
5       hyperoffset = if phi1 = 0 or phi2 = 0 {
6           max(phi1, phi2)
7       } else {
8           lcm(phi1, phi2)
9       };
10      locations = {};
11      transitions = {};
12      prev = 0;
13      step(i: int, is_one: bool, prev_loc: location, next_loc: location)
            {
14          locations += prev_loc;
15          locations += next_loc;
16          if is_one { // difference between prev and next is one, not
                need for intermediate location
17          match i with
18          | 1 -> transitions += prev_loc --a1 ā2 r--> next_loc;
19          | 2 -> transitions += prev_loc --ā1 a2 r--> next_loc;
20          } else {
21              inter_loc = ⟦prev < i_r < next⟧;
22              locations += inter_loc;
23              transitions += prev_loc --ā1 ā2 r--> inter_loc;
24              transitions += inter_loc --ā1 ā2 r--> inter_loc;
25              match i with
26              | 1 -> transitions += inter_loc --[i_r=(next−1)·i_hp]:a1 ā2 r--> next_loc;
27              | 2 -> transitions += inter_loc --[i_r=(next−1)·i_hp]:ā1 a2 r--> next_loc;
28          }
29          match i with
30          | 1 -> offset1 = offset1 + p1;
31          | 2 -> offset2 = offset2 + p2;
32          prev = next;
33      };
34      while offset1 < hyperoffset or offset2 < hyperoffset {
35          next, i = min_argmin(hyperoffset, offset1 + p1, offset2 + p2);
36          prev_loc = ⟦i_r = prev⟧;
37          next_loc = ⟦i_r = next⟧;
38          step(i, (prev - next = 1), prev_loc, next_loc);
39      } // prev = hyperoffset;
40      while prev < hyperperiod + hyperoffset {
41          next, i = min_argmin(hyperperiod + hyperoffset, offset1 + p1,
                offset2 + p2); // i is for which argument of min was chosen
42          prev_loc = ⟦i_r = hyperperiod · i_hp + prev⟧;
43          next_loc = ⟦i_r = hyperperiod · i_hp + next⟧;
44          step(i, (prev - next = 1), prev_loc, next_loc);
45      }
46      return locations, transitions
47  }
```

Algorithm 5.2: Constructive partitioning and transition specialization of hyperperiods

5.3 Implementations

There two project implementing all these propositions:`ccsl-rs` [Tok23a] and `mrtccsl` [Tok24].

The first project has started as an implementation in Rust based on ideas developed early during the thesis [Tok21] and became the first place to make experiments with abstract interpretation. Thus it features the first translations of CCSL symbolic automata to NBac format, which ended up being difficult to write and not entirely correct. As Rust ecosystem did not have any abstract interpretation domains available and reimplementing it would be waste of time, the development shifted to OCaml and `mrtccsl` project.

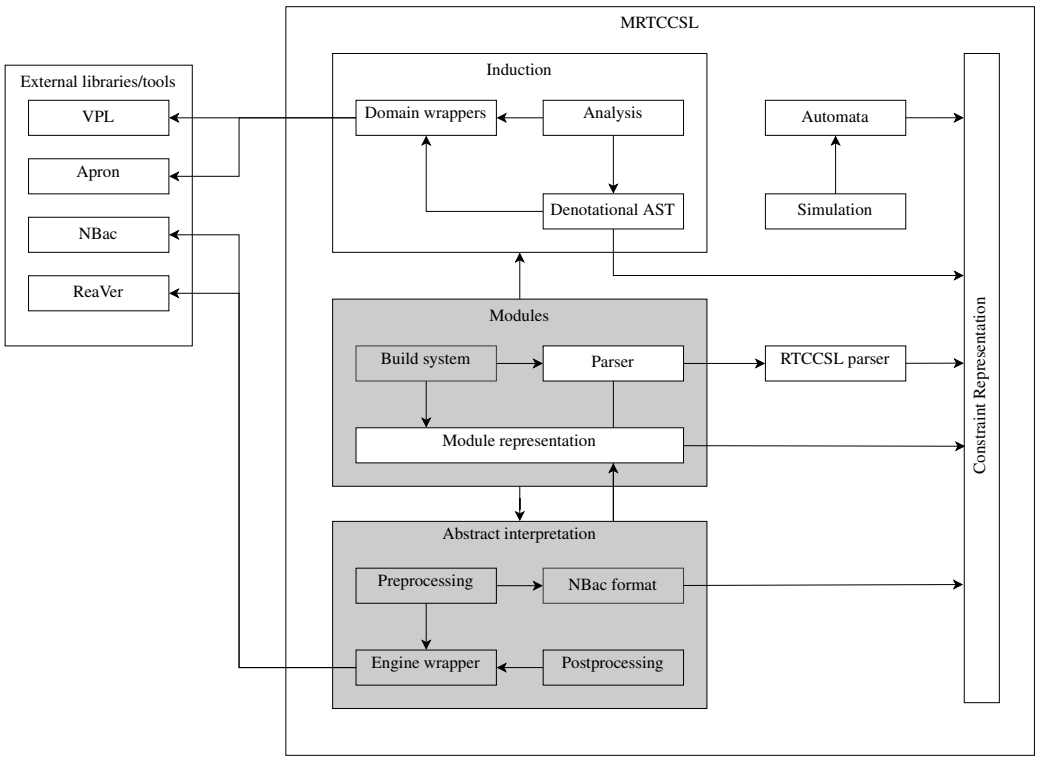


Figure 5.20: Modules and parts of MRTCCSL implementation

We present an overview of the `mrtccsl` project on Figure 5.20, with grayed out planned, but not yet implemented parts. It consists of implementations of intermediate constraint representation, a data structure that defines the constraints and on which everything else depends and transforms further, simulation based on semi symbolic automata and analysis using induction and polyhedra. The implementation of modules is currently limited to the parsing, type checking and subsequent translation into module representation is under development.

Simulation We have implemented a simulator for the RTCCSL part of the language. It is able to produce traces, an example being Figure 5.21. As RTCCSL is not a deterministic *program*, to simulate it we need to define two strategies: first to select clocks, second to select how much to advance time. For this we have implemented a set of standard strategies, like choose first, last or random transition, and slow, fast or bounded random time step. To confirm that the implementation

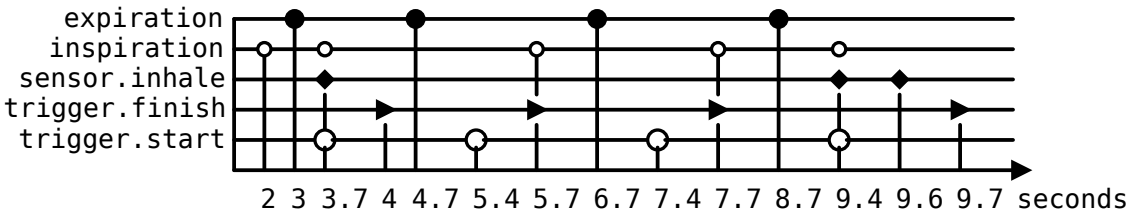


Figure 5.21: Trace of PCV mode from Mechanical Lung Ventilator use case

conforms to the semantics, we have wrote an extensive test suite spanning 108 typical and corner cases for both regular CCSL as well as for the new features.

Inductive reasoning Inductive reasoning starts with constraints passed as modules defined in Section 4.4. The modules are then translated into denotational formula, which are prepositional and numerical expressions. There are three possible cases: exact, over and underapproximations, all encoded inside. They are chosen depending on which part of the modules it is trying to check, overapproximation for emptiness check and underapproximation for satisfaction of assumption, interface and assertion. With the priority for a constraint to be translated exactly if possible. These formulas are then saturated and checked for solutions and the results are printed. As with the simulation, we have a testing setup, currently containing 15 tests (Appendix A.3). These tests check that the implementation is correct by checking properties for the specifications that we know have them, including the relation of subspecification that did not exist prior to this work. For the polyhedra library we can use either the Verimag Polyhedra Library (VPL) or any that supports Apron interface. Unfortunately, all the domains we found do not support simultaneous strict and non-strict conditions, except VPL, which is critical to express real-time constraints and why we focused on using this library.

5.4 Conclusion

In this chapter, we have shown what analysis is possible for our new language. It consists of translating the language to already existing tools, like NBac and ReaVer, tuning their parameters to check properties we would like to check and making changes to their internal representation to enable more efficient analysis and the specific to CCSL properties. These improvements include an algorithm to detect a hyperperiod and then construct a partition for it, concurrent analysis and accompanying partition scheme to reduce complexity of the analysis and the new constant size Boolean array domain, which allows to optimize the common pattern of mutually exclusive sharing of resources by using acceleration technique. We also provide an ad-hoc solver that uses the idea of mathematical induction applied to logical clocks and polyhedra library to solve numerical inequalities between tags of the clocks. It implements the full language analysis engine, including the real-time constraints, parameters and their relations, and subspecification check, used in modules.

CHAPTER 6

Conclusion and Perspectives

6.1 Summary

This thesis contributes to the field of reactive system modeling with the definition of a new language, which we call the Modular Real-Time Clock Constraint Specification Language (MRTCCSL). Its purpose is to make the descriptions more accessible to the non-specialist public by introducing modularity and additional constructs. At the same time we provide a strong foundation by defining it formally in various semantic styles. We also provide a framework for solvers to take advantage of the new features, like real-time constraints and modules. We finish the contribution with analysis capabilities exploring several formal techniques and symbolic methods beyond those classically used for synchronous languages. Our ad-hoc solver brings together several analysis techniques to apply to the specific nature of our language. We also propose a new abstract domain, named constant size Boolean array domain, that is better suited for CCSL specifications than available combination of Boolean and numerical analysis. We have developed an acceleration technique for this domain and a concurrent analysis method that consists of location and transition partitioning. In conclusion, we provide an almost complete package for a language: the definition with syntax and formal semantics, based on core concepts of subspecification relation $\Subset$ and synchronization $\|$, several examples defined in the language's syntax, already available tools and a description of how to make them better.

6.2 Perspectives

While we have managed to complete a vast scope of tasks in this work, there were some that we did not have time or expertise to tackle. Additionally, we see several opportunities to improve the current work, but which go much beyond the current topic.

Formal semantics of the real-time subset While we have started on defining formal semantics of real-time extension, we have encountered difficulties to complete it. It is based on previous work that has defined a formal denotational CCSL semantics in Agda [MP18b]. This work did not cover index-related relations that are essential to encode real-time constraints. That would be a complete work in itself that we did not have time to complete in the given timeframe.

General abstract interpretation framework for reactive systems As of now, there are no actively developed tools of abstract interpretation for reactive systems and so languages like CCSL.

While the last one, ReaVer, is free and publicly available, its modification is not easy. For example, to implement the features we require for our language, we would need to:

- to wrap Verimag Polyhedra Library to use Apron interface as we need both strict and non-strict inequalities;
- reimplement numerical condition partitioning;
- integrate lattice automata;
- implement the constant size Boolean array domain and its acceleration, which itself involves detection of patterns;
- modify the language and the internal representation to support types of queues, constant-size queues, and the operations on them;
- same for the concurrent analysis.

While we have built the first and necessary step to take it, this would be also another entire work.

Modules as guide for partitioning Modules in our language are really analogous to functions in general programming languages. Thus, our first idea was to allow modular analysis by using modules. For that we had to define the additional construct of interface. The reason is that as the constraints define sets of behaviour (languages) and approximating it automatically is certainly not possible for such complex language as CCSL. Otherwise, that would mean that we can generate a more general or specific program from the description of an existing one.

Improving precision of constant size Boolean array domain Lastly, the precision of the domain is greatly reduced in our examples, as it is not able to exchange the information with other domains about what range of indices is present in an element of the domain (the explanation is in Section 5.2.5.3). If this was available to other domains symbolically, it would make the analysis much better in cases that have uncertainty.

Bibliography

[ABG22] Flavio Ascari, Roberto Bruni, and Roberta Gori. “Limits and Difficulties in the Design of Under-Approximation Abstract Domains”. In: *Foundations of Software Science and Computation Structures*. Ed. by Patricia Bouyer and Lutz Schröder. Vol. 13242. Cham: Springer International Publishing, 2022, pp. 21–39. ISBN: 978-3-030-99252-1 978-3-030-99253-8. DOI: 10.1007/978-3-030-99253-8_2.

[Abr+91] Jean-Raymond Abrial et al. “The B-Method”. In: *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume 2: Tutorials*. VDM ’91. Berlin, Heidelberg: Springer-Verlag, Oct. 21, 1991, pp. 398–405. ISBN: 978-3-540-54868-3.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge ; New York : Cambridge University Press, 2010. 626 pp. ISBN: 978-0-521-89556-9.

[AD94] 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.

[AFH96] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. “The Benefits of Relaxing Punctuality”. In: *J. ACM* 43.1 (Jan. 1, 1996), pp. 116–146. ISSN: 0004-5411. DOI: 10.1145/227595.227602.

[AH94] Rajeev Alur and Thomas A. Henzinger. “A Really Temporal Logic”. In: *Journal of the ACM* 41.1 (Jan. 2, 1994), pp. 181–203. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/174644.174651.

[AMD10] Charles André, Frédéric Mallet, and Julien Deantoni. “VHDL Observers for Clock Constraint Checking”. en. In: IEEE computer society, July 2010. DOI: 10/bf3jng. URL: https://hal.inria.fr/inria-00587107.

[AMP07] Charles André, Frédéric Mallet, and Marie-Agnès Peraldi-Frati. “A Multiform Time Approach to Real-Time System Modeling: Application to an Automotive System”. In: IEEE Int. Symp. on Industrial Embedded Systems (SIES). IEEE, 2007, p. 234. DOI: 10.1109/SIES.2007.4297340.

[AMS07] Charles André, Frédéric Mallet, and Robert de Simone. “Modeling Time(s)”. In: ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS/UML). Vol. LNCS 4735. Springer, 2007, pp. 559. DOI: 10.1007/978-3-540-75209-7_38.

[And09] Charles André. “Syntax and Semantics of the Clock Constraint Specification Language (CCSL)”. report. INRIA, 2009, p. 37.

[And21] Étienne André. "IMITATOR 3: Synthesis of Timing Parameters Beyond Decidability". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 552–565. ISBN: 978-3-030-81685-8. DOI: 10.1007/978-3-030-81685-8_26.

[And95] C. André. "Syncharts: A Visual Representation of Reactive Behaviors". In: 1995.

[Arn94] A. Arnold. *Finite transition systems - semantics of communicating systems*. International Series in Computer Science. Prentice Hall, 1994.

[AVR19] Adina Aniculaesei, Andreas Vorwald, and Andreas Rausch. "Using the SCADE Toolchain to Generate Requirements-Based Test Cases for an Adaptive Cruise Control System". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). Sept. 2019, pp. 503–513. DOI: 10.1109/MODELS-C.2019.00079.

[Bar+08] Sébastien Bardin et al. "FAST: Acceleration from Theory to Practice". In: *International Journal on Software Tools for Technology Transfer* 10.5 (Oct. 1, 2008), pp. 401–424. ISSN: 1433-2787. DOI: 10.1007/s10009-008-0064-3.

[BBP17] Guillaume Baudart, Timothy Bourke, and Marc Pouzet. "Symbolic Simulation of Dataflow Synchronous Programs with Timers". In: 12th Forum on Specification and Design Languages (FDL 2017). Sept. 18, 2017.

[BBP23] Timothy Bourke, Vincent Bregeon, and Marc Pouzet. "Scheduling and Compiling Rate-Synchronous Programs with End-To-End Latency Constraints". In: 35th Euromicro Conference on Real-Time Systems (ECRTS 2023). Vol. 262. July 11, 2023, 1:1. DOI: 10.4230/LIPIcs.ECRTS.2023.1.

[Ben+18] Albert Benveniste et al. "Building a Hybrid Systems Modeler on Synchronous Languages Principles". In: *Proceedings of the IEEE* 106.9 (Sept. 2018), pp. 1568–1592. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2018.2858016.

[Ber+10] Julien Bertrane et al. *Static Analysis and Verification of Aerospace Software by Abstract Interpretation*. AIAA Infotech at Aerospace 2010. American Institute of Aeronautics and Astronautics Inc., Apr. 2010. ISBN: 978-1-60086-743-9. DOI: 10.1561/9781601988577.

[Ber02] Gérard Berry. "The Esterel v5 Language Primer". In: (Dec. 11, 2002).

[BG24] 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.

[BM18a] Sylvain Boulmé and Alexandre Maréchal. "Refinement to Certify Abstract Interpretations, Illustrated on Linearization for Polyhedra". In: *Journal of Automated Reasoning* (Nov. 2018). DOI: 10.1007/s10817-018-9492-2.

[BM18b] Sylvain Boulmé and Alexandre Maréchal. *Verimal Polyhedra Library*. https://github.com/VERIMAG-Polyhedra/VPL/. 2018.

[BM83] Bernard Berthomieu and Miguel Menasche. “An Enumerative Approach for Analyzing Time Petri Nets”. In: IFIP 9th World Computer Congress. Sept. 19, 1983.

[Bou+14] Frédéric Boulanger et al. “TESL: A Language for Reconciling Heterogeneous Execution Traces”. In: *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference On*. Lausanne, Switzerland, Oct. 2014, pp. 114–123. ISBN: 978-1-4799-5336-3. DOI: 10.1109/MEMCOD.2014.6961849.

[Bou+22] Patricia Bouyer et al. *Zone-Based Verification of Timed Automata: Extrapolations, Simulations and What Next?* Version 1. July 15, 2022. DOI: 10.48550/arXiv.2207.07479. arXiv: 2207.07479 [cs]. Pre-published.

[BP13] Timothy Bourke and Marc Pouzet. “Zélus: A Synchronous Language with ODEs”. In: HSCC - 16th International Conference on Hybrid Systems: Computation and Control. ACM, Apr. 8, 2013, p. 113. DOI: 10.1145/2461328.2461348.

[BR83] Gerard Berry and Jean-Paul Rigault. “Esterel: Towards a Synchronous and Semantically Sound High-Level Language for Real-Time Applications”. In: 1983.

[Bry86] Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* C-35.8 (Aug. 1986), pp. 677–691. ISSN: 1557-9956. DOI: 10/bnrh63.

[Cas+87] 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.

[CE81] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic”. In: *Logic of Programs, Workshop*. Berlin, Heidelberg: Springer-Verlag, May 1, 1981, pp. 52–71. ISBN: 978-3-540-11212-9.

[CH78] Patrick Cousot and Nicolas Halbwachs. “Automatic Discovery of Linear Restraints among Variables of a Program”. In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’78. New York, NY, USA: Association for Computing Machinery, Jan. 1, 1978, pp. 84–96. ISBN: 978-1-4503-7348-7. DOI: 10.1145/512760.512770.

[Che68] N. V. Chernikoba. “Algorithm for Discovering the Set of All the Solutions of a Linear Programming Problem”. In: *USSR Computational Mathematics and Mathematical Physics* 8.6 (Jan. 1, 1968), pp. 282–293. ISSN: 0041-5553. DOI: 10.1016/0041-5553(68)90115-8.

[CHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. “A Calculus of Durations”. In: *Information Processing Letters* 40.5 (Dec. 13, 1991), pp. 269–276. ISSN: 0020-0190. DOI: 10.1016/0020-0190(91)90122-X.

[Cio13] tefan Ciobâc. “From Small-Step Semantics to Big-Step Semantics, Automatically”. In: *Integrated Formal Methods*. Ed. by Einar Broch Johnsen and Luigia Petre. Berlin, Heidelberg: Springer, 2013, pp. 347–361. ISBN: 978-3-642-38613-8. DOI: 10.1007/978-3-642-38613-8_24.

[Com+14] Benoit Combemale et al. “Globalizing Modeling Languages”. In: *Computer* 47.6 (June 2014), pp. 68–71. ISSN: 1558-0814. DOI: 10.1109/MC.2014.147.

[CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. “SCADE 6: A formal language for embedded critical software development (invited paper)”. In: *11th International Symposium on Theoretical Aspects of Software Engineering, TASE*. Sophia Antipolis, France: IEEE Computer Society, 2017, pp. 1–11. DOI: 10.1109/TASE.2017.8285623.

[DAG14] Julien Deantoni, Charles André, and Régis Gascon. “CCSL denotational semantics”. en. Pages: 29. report. Inria, Nov. 2014. URL: https://hal.inria.fr/hal-01082274.

[Dan+17] Andrei Dan et al. “Effective Abstractions for Verification under Relaxed Memory Models”. In: *Computer Languages, Systems & Structures*. Special Issue on the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2015) 47 (Jan. 1, 2017), pp. 62–76. ISSN: 1477-8424. DOI: 10.1016/j.cl.2016.02.003.

[DB03] ROBERT DE SIMONE and ALBERT BENVENISTE. “The Synchronous Languages 12 Years Later | IEEE Journals & Magazine | IEEE Xplore”. In: (Jan. 2003). DOI: 10.1109/JPROC.2002.805826.

[DM12a] Julien DeAntoni and Frédéric Mallet. “TimeSquare: Treat Your Models with Logical Time”. In: *Objects, Models, Components, Patterns*. Ed. by Carlo A. Furia and Sebastian Nanz. Red. by David Hutchison et al. Vol. 7304. 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.

[DM12b] 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.

[DSi13] Vijay D’Silva. “Generalizing Simulation to Abstract Domains”. In: *CONCUR 2013 Concurrency Theory*. Ed. by Pedro R. D’Argenio and Hernán Melgratti. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 485–499. ISBN: 978-3-642-40184-8. DOI: 10.1007/978-3-642-40184-8_34.

[EH83] E. Allen Emerson and Joseph Y. Halpern. “"Sometimes" and "Not Never" Revisited: On Branching versus Linear Time (Preliminary Report)”. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’83. New York, NY, USA: Association for Computing Machinery, Jan. 24, 1983, pp. 127–140. ISBN: 978-0-89791-090-3. DOI: 10.1145/567067.567081.

[FH07] Peter Feiler and Jrgen Hansson. “Flow Latency Analysis with the Architecture Analysis and Design Language (AADL)”. In: (Jan. 1, 2007).

[Flo62] Robert W. Floyd. “Algorithm 97: Shortest Path”. In: *Commun. ACM* 5.6 (June 1, 1962), p. 345. ISSN: 0001-0782. DOI: 10.1145/367766.368168.

[FM97] M Fujita and P C Mcgeer. “Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation”. In: *Formal Methods in System Design* (1997).

[Gal08] Tristan Le Gall. “Abstract Lattices for the Verification of Systèmes with Stacks and Queues”. PhD thesis. Université Rennes 1, July 2, 2008.

[Gam+07] Abdoulaye Gamatié et al. “Polychronous Design of Embedded Real-Time Applications”. In: *ACM Trans. Softw. Eng. Methodol.* 16.2 (Apr. 1, 2007), 9–es. ISSN: 1049-331X. DOI: 10.1145/1217295.1217298.

[GH06] Laure Gonnord and Nicolas Halbwachs. “Combining Widening and Acceleration in Linear Relation Analysis”. In: *Static Analysis*. Ed. by Kwangkeun Yi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 144–160. ISBN: 978-3-540-37758-0. DOI: 10.1007/11823230_10.

[Gok+13] Arda Goknil et al. “Tool Support for the Analysis of TADL2 Timing Constraints Using TimeSquare”. en. In: *2013 18th International Conference on Engineering of Complex Computer Systems*. Singapore, Singapore: IEEE, July 2013, pp. 145–154. ISBN: 978-0-7695-5007-7. DOI: 10.1109/ICECCS.2013.28. URL: http://ieeexplore.ieee.org/document/6601815/.

[Gro08] Object Management Group. *UML Profile for MARTE, beta 2*. OMG document number: ptc/08-06-09. 2008. URL: https://www.omg.org/omgmarte/.

[GS14] Laure Gonnord and Peter Schrammel. “Abstract Acceleration in Linear Relation Analysis”. In: *Science of Computer Programming* 93, part B (125 - 153 2014), pp. 125–153. DOI: 10.1016/j.scico.2013.09.016.

[Gue+91] Paul Le Guernic et al. “Programming Real-Time Applications with Signal”. In: *Proceedings of the IEEE* 79.9 (1991), p. 1321. DOI: 10.1109/5.97301.

[HLS12] Alexander HeuSSner, Tristan Le Gall, and Grégoire Sutre. “McScM: A General Framework for the Verification of Communicating Machines”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Cormac Flanagan and Barbara König. Red. by David Hutchison et al. Vol. 7214. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 478–484. ISBN: 978-3-642-28755-8 978-3-642-28756-5. DOI: 10.1007/978-3-642-28756-5_34.

[Hoa13] Thai Son Hoang. “An Introduction to Event-B”. In: *Industrial Deployment of System Engineering Methods*. Berlin, Heidelberg: Springer, 2013.

[HPS] Frédéric Herbreteau, Gérald Point, and Ocan Sankur. *The TChecker tool and librairies*. https://github.com/ticktac-project/tchecker.

[Jea02] B. Jeannet. “Representing and Approximating Transfer Functions in Abstract Interpretation of Hetereogeneous Datatypes”. In: *Static Analysis*. Ed. by Manuel V. Hermenegildo and Germán Puebla. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 52–68. ISBN: 978-3-540-45789-3. DOI: 10.1007/3-540-45789-5_7.

[Jea03] B. Jeannet. “Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Reactive Systems”. en. In: *Formal Methods in System Design* 23.1 (July 2003), pp. 5–37. ISSN: 1572-8102. DOI: 10.1023/A:1024480913162. URL: https://doi.org/10.1023/A:1024480913162.

[Jea13] Bertrand Jeannet. “Relational Interprocedural Verification of Concurrent Programs”. In: *Software & Systems Modeling* 12.2 (May 1, 2013), pp. 285–306. ISSN: 1619-1374. DOI: 10.1007/s10270-012-0230-7.

[JHR99] Bertrand Jeannet, Nicolas Halbwachs, and Pascal Raymond. “Dynamic Partitioning in Analyses of Numerical Properties”. In: *Static Analysis*. Ed. by Agostino Cortesi and Gilberto Filé. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 39–50. ISBN: 978-3-540-48294-9. DOI: 10.1007/3-540-48294-6_3.

[JM09] Bertrand Jeannet and Antoine Miné. “Apron: A Library of Numerical Abstract Domains for Static Analysis”. In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Berlin, Heidelberg: Springer, 2009, pp. 661–667. ISBN: 978-3-642-02658-4. DOI: 10.1007/978-3-642-02658-4_52.

[Jou+20] Matthieu Journault et al. “Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer”. In: *Verified Software. Theories, Tools, and Experiments*. Ed. by Supratik Chakraborty and Jorge A. Navas. Vol. 12031. Cham: Springer International Publishing, 2020, pp. 1–18. ISBN: 978-3-030-41599-0 978-3-030-41600-3. DOI: 10.1007/978-3-030-41600-3_1.

[Käs+10] Daniel Kästner et al. “Astree: Proving the Absence of Runtime Errors”. In: *Embedded Real Time Software and Systems - ERTS2 2010*. Ed. by J.C. Laprie. Toulouse, France: AAAF, SEE, SIA, May 2010.

[Kir+15] Florent Kirchner et al. “Frama-C: A software analysis perspective”. In: *Formal Aspects of Computing* 27.3 (May 2015), pp. 573–609. ISSN: 0934-5043. DOI: 10.1007/s00165-014-0326-7.

[KM02] Antonín Kuera and Richard Mayr. “Why Is Simulation Harder than Bisimulation?” In: *CONCUR 2002 Concurrency Theory*. Ed. by Lubo Brim et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 594–609. ISBN: 978-3-540-45694-0. DOI: 10.1007/3-540-45694-5_39.

[Koy90] Ron Koymans. “Specifying Real-Time Properties with Metric Temporal Logic”. In: *Real-Time Systems* 2.4 (Nov. 1, 1990), pp. 255–299. ISSN: 1573-1383. DOI: 10.1007/BF01995674.

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1, 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563.

[Lar+98] Kim G. Larsen et al. “Clock Difference Diagrams”. In: *BRICS Report Series* 5.46 (June 16, 1998). [TLDR] Clock Difference Diagrams is presented, a new BDD-like data-structure for effective representation and manipulation of certain non-convex subsets of the Euclidean space, notably those encountered in verification of timed automata. ISSN: 1601-5355, 0909-0878. DOI: 10.7146/brics.v5i46.19491.

[LPW95] K.G. Larsen, P. Pettersson, and Wang Yi. “Compositional and Symbolic Model-Checking of Real-Time Systems”. In: *Proceedings 16th IEEE Real-Time Systems Symposium* (1995). [TLDR] This paper develops and combines compositional and symbolic model-checking techniques that provide the foundation for a new automatic verification tool UPPAAL and indicates that UPPAal performs time- and

space-wise favorably compared with other real-time verification tools., pp. 76–87. DOI: 10.1109/REAL.1995.495198.

[LTL02] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. "Polychrony for System Design". In: *Journal of Systems Architecture* 12 (2002), pp. 261–304. DOI: 10.1142/S0218126603000751.

[MA08] Frédéric Mallet and Charles André. "UML/MARTE CCSL, Signal and Petri nets". en. report. INRIA, 2008. URL: https://hal.inria.fr/inria-00283077.

[Mai00] M. Maidi. "The Common Fragment of CTL and LTL". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. Proceedings 41st Annual Symposium on Foundations of Computer Science. Nov. 2000, pp. 643–652. DOI: 10.1109/SFCS.2000.892332.

[Mal08] Frédéric Mallet. "Clock Constraint Specification Language: Specifying Clock Constraints with UML/MARTE". In: *Innovations in Systems and Software Engineering* 4 (Oct. 1, 2008), pp. 309–314. DOI: 10/dn4ptd.

[MdS15] Frédéric Mallet and Robert de Simone. "Correctness Issues on MARTE/CCSL Constraints". In: *Science of Computer Programming*. Special Issue: Architecture-Driven Semantic Analysis of Embedded Systems 106 (Aug. 1, 2015), pp. 78–92. ISSN: 0167-6423. DOI: 10/f7qbxg.

[Min01] Antoine Miné. "A New Numerical Abstract Domain Based on Difference-Bound Matrices". In: LNCS 2053. Springer, May 2001, pp. 155–172.

[Min06] Antoine Miné. "The Octagon Abstract Domain". In: *Higher-Order and Symbolic Computation* 19.1 (Mar. 1, 2006), pp. 31–100. ISSN: 1573-0557. DOI: 10/bh4k5k.

[Min17] Antoine Miné. "Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation". In: *Foundations and Trends in Programming Languages* 4.3-4 (2017), p. 120. DOI: 10/gnskfg.

[MLM21] Joao Marques-Silva, Ines Lynce, and Sharad Malik. "Chapter 4. Conflict-Driven Clause Learning SAT Solvers". In: *Frontiers in Artificial Intelligence and Applications*. Ed. by Armin Biere et al. IOS Press, Feb. 2, 2021. ISBN: 978-1-64368-160-3 978-1-64368-161-0. DOI: 10.3233/FAIA200987.

[MMR13] Frédéric Mallet, Jean-Vivien Millo, and Yuliia Romenska. *State-Based Representation of CCSL Operators*. Tech. rep. INRIA, July 19, 2013.

[MMS13a] Frédéric Mallet, Jean-Vivien Millo, and Robert de Simone. "Safe CCSL Specifications and Marked Graphs". In: MEMOCODE - 11th IEEE/ACM International Conference on Formal Methods and Models for Codesign. IEEE CS, Oct. 18, 2013, p. 157.

[MMS13b] Frédéric Mallet, Jean-Vivien Millo, and Robert de Simone. "Safe CCSL Specifications and Marked Graphs". en. In: IEEE CS, Oct. 2013, p. 157. URL: https://hal.inria.fr/hal-00913962.

[MN04] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Ed. by Yassine Lakhnech and Sergio Yovine. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 152–166. ISBN: 978-3-540-30206-3. DOI: 10.1007/978-3-540-30206-3_12.

[Mon20] Mathieu Montin. "A Formal Framework for Heterogeneous Systems Semantics". PhD thesis. Institut National Polytechnique de Toulouse - INPT, Sept. 14, 2020.

[MP18a] Mathieu Montin and M. Pantel. "Ordering Strict Partial Orders to Model Behavioral Refinement". In: *Refine@FM* (2018). DOI: 10/gnth9n.

[MP18b] Mathieu Montin and Marc Pantel. "Mechanizing the Denotational Semantics of the Clock Constraint Specification Language". In: *Model and Data Engineering*. Ed. by El Hassan Abdelwahed et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 385–400. ISBN: 978-3-030-00856-7. DOI: 10/gnth9m.

[MP21] Mathieu Montin and Marc Pantel. "Towards Multi-layered Temporal Models: A Proposal to Integrate Instant Refinement in CCSL". In: 41th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2021). Vol. 12719. Springer International Publishing, June 14, 2021, p. 120. DOI: 10.1007/978-3-030-78089-0_7.

[MPA09] Frédéric Mallet, Marie-Agnès Peraldi-Frati, and Charles Andre. "Marte CCSL to Execute East-ADL Timing Requirements". In: *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC). Tokyo, Japan: IEEE, Mar. 2009, pp. 249–253. ISBN: 978-0-7695-3573-9. DOI: 10.1109/ISORC.2009.18.

[MR05] Laurent Mauborgne and Xavier Rival. "Trace Partitioning in Abstract Interpretation Based Static Analyzers". In: *Programming Languages and Systems*. Ed. by Mooly Sagiv. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 5–20. ISBN: 978-3-540-31987-0. DOI: 10.1007/978-3-540-31987-0_2.

[Ngu18] Hai Nguyen Van. "Formalizing Time and Causality in Polychronous Polytimed Models". These de doctorat. Université Paris-Saclay (ComUE), Sept. 27, 2018.

[Oue+19] Amin Oueslati et al. "System Based Interference Analysis in Capella". en. In: *The Journal of Object Technology* 18.2 (2019), 14:1. DOI: 10.5381/jot.2019.18.2.a14. URL: https://hal.inria.fr/hal-02182902.

[OW05] J. Ouaknine and J. Worrell. "On the Decidability of Metric Temporal Logic". In: *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05). Chicago, IL, USA: IEEE, 2005, pp. 188–197. ISBN: 978-0-7695-2266-1. DOI: 10.1109/LICS.2005.33.

[PD11a] Marie-Agnes Peraldi-Frati and Julien DeAntoni. “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*. ISSN: 2375-5261. Mar. 2011, pp. 50–57. DOI: 10.1109/ISORC.2011.16.

[PD11b] Marie-Agnes Peraldi-Frati and Julien DeAntoni. “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*. 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. Mar. 2011, pp. 50–57. DOI: 10.1109/ISORC.2011.16.

[Per+12] Marie-Agnès Peraldi-Frati et al. “A Timing Model for Specifying Multi Clock Automotive Systems: The Timing Augmented Description Language V2”. In: *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems. July 2012, pp. 230–239. DOI: 10.1109/ICECCS20050.2012.6299218.

[Pnu77] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*. 18th Annual Symposium on Foundations of Computer Science (Sfcs 1977). Providence, RI, USA: IEEE, Sept. 1977, pp. 46–57. DOI: 10/dn8cpn.

[PP18] Nir Piterman and Amir Pnueli. “Temporal Logic and Fair Discrete Systems”. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 27–73. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_2.

[Sch98] David A. Schmidt. “Trace-Based Abstract Interpretation of Operational Semantics”. In: *LISP and Symbolic Computation* 10.3 (May 1, 1998), pp. 237–271. ISSN: 1573-0557. DOI: 10.1023/A:1007734417713.

[SJ11a] Peter Schrammel and Bertrand Jeannet. “Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs”. In: *Static Analysis*. Ed. by Eran Yahav. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 233–248. ISBN: 978-3-642-23702-7. DOI: 10.1007/978-3-642-23702-7_19.

[SJ11b] Peter Schrammel and Bertrand Jeannet. “Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs”. en. In: *Static Analysis*. Ed. by Eran Yahav. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 233–248. ISBN: 978-3-642-23702-7. DOI: 10.1007/978-3-642-23702-7_19.

[SM18] Thibault Suzanne and Antoine Miné. “Relational Thread-Modular Abstract Interpretation Under Relaxed Memory Models”. In: *Programming Languages and Systems*. Ed. by Sukyoung Ryu. Vol. 11275. Cham: Springer International Publishing, 2018, pp. 109–128. ISBN: 978-3-030-02767-4 978-3-030-02768-1. DOI: 10.1007/978-3-030-02768-1_6.

[SPV17] Gagandeep Singh, Markus Püschel, and Martin Vechev. "Fast Polyhedra Abstract Domain". In: *ACM SIGPLAN Notices* 52.1 (Jan. 1, 2017), pp. 46–59. ISSN: 0362-1340. DOI: 10.1145/3093333.3009885.

[Sur+13a] Jagadish Suryadevara et al. "Verifying MARTE/CCSL Mode Behaviors Using UPPAAL". en. In: vol. 8137. Springer, Sept. 2013, p. 1. DOI: 10/f3rnq6. URL: https://hal.inria.fr/hal-00866477.

[Sur+13b] Jagadish Suryadevara et al. "Verifying MARTE/CCSL Mode Behaviors Using UPPAAL". In: SEFM 2013 - 11th International Conference on Software Engineering and Formal Methods. Vol. 8137. Springer, Sept. 25, 2013, p. 1. DOI: 10/f3rnq6.

[TM24] Pavlo Tokariev and Frédéric Mallet. "Real-Time CCSL: Application to the Mechanical Lung Ventilator". In: ABZ 2024 10th International Conference on Rigorous State Based Methods. Vol. LNCS-14759. Springer, June 25, 2024, p. 289. DOI: 10.1007/978-3-031-63790-2_24.

[Tok21] Pavlo Tokariev. "Efficient compilation of CCSL for embedded targets". MA thesis. Université Côte d'Azur, 2021.

[Tok23a] Pavlo Tokariev. *Implementation of CCSL and NBac translation*. https://archive.softwareheritage.org/browse/origin/https://github.com/PaulRaUnite/ccsl-rs. 2023.

[Tok23b] Pavlo Tokariev. "Real-time extension to clock constraint specification language". Presentated at The 30th International Open Workshop on Synchronous Programming (SYNCHRON23), slides available online. Kiel, Germany, 2023. URL: https://www.rtsys.informatik.uni-kiel.de/en/synchron-2023/.

[Tok24] Pavlo Tokariev. *Implementation of MRTCCSL*. https://archive.softwareheritage.org/browse/origin/https://github.com/PaulRaUnite/mrtccsl. 2024.

[VBK22] Nate Veldt, Austin R. Benson, and Jon Kleinberg. "Hypergraph Cuts with General Splitting Functions". In: *SIAM Review* 64.3 (Aug. 2022), pp. 650–685. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/20M1321048.

[ZDM18] Min Zhang, Feng Dai, and Frédéric Mallet. "Periodic scheduling for MARTE/CCSL: Theory and practice". en. In: *Science of Computer Programming* 154 (Mar. 2018), p. 42. DOI: 10/gc3zst. URL: https://hal.inria.fr/hal-01670450.

[Zha+19] Min Zhang et al. "SMT-Based Bounded Schedulability Analysis of the Clock Constraint Specification Language". en. In: Apr. 2019. URL: https://hal.inria.fr/hal-02080763.

List of figures

List of tables

List of definitions

Appendix

Appendix A

Additional listings

A.1 MRTCCSL specification listings

Here we list the whole specifications of the examples from Sections 3.2 to 3.4.

```
1  //FUN.19
2  pcv_mode(mode: struct, sensor: struct) assume {
3    //ratio of expiratory time to inspiratory time PER.5, includes PER.13
4    IE in [1, 4];
5    //respiratory rate, breath per minute PER.4, includes PER.12
6    RR in [4,50]/1 min;
7
8    trigger_window_delay = 0.7s; //CONT.45
9
10   //Check that nothing obstructs inspiration to start
11   //(if window is too small,the "faster" will not reset difference in time)
12   trigger_window.start <= fastest(sensor.inhale, trigger_window.finish) <=
         next inspiration.start; //FUN.21
13   //Rationale:  we should not allow inhale sensing outside of trigger window,
14   //otherwise it messes up the logic
15   allow sensor.inhale in [trigger_window.start, trigger_window.finish];
16 } {
17   expiration = delay inspiration by 1/RR/(1+IE); //FUN.20
18
19   trigger_window = {
20     start < finish;
21     start = delay $.expiration by trigger_window_delay; //CONT.45
22     finish = delay $.inspiration by 1/RR; //FUN.20
23   };
24   inspiration_condition = sensor.inhale + trigger_window.finish - (sample
         (sensor.inhale + mode.pcv.finish) on trigger_window.finish) - (sample
         mode.pcv.finish on sensor.inhale); //CONT.25
25   next inspiration = first sampled inspiration_condition on
         trigger_window.finish;
26 } assert {
27   //FUN.20, double check really
28   trigger_window.finish <= delay expiration by IE/RR/(1+IE);
29   inspiration alternates expiration; //same
30 }
31
32 //FUN.24
33 psv_mode(mode: struct, sensor: struct, alarm: struct) assume {
34   min_exp_time_psv in [0.4s, 2s]; //CONT.36.3
35   max_apnea_lag_time in [10s, 60s]; //PER.11, PER.21
36   max_expiration_lag = 7s; //CONT.32
37
```

```
38    //expiration <= next sensor.inhale <= next inspiration; // limit inhale
          sensing to the apropriate window
39    //but also check that nothing obstructs causality
40    sensor.inhale +<= inspiration; //FUN.25
41  } {
42    inspiration alternates expiration;
43    expiration_deadline = delay inspiration by max_expiration_lag; //CONT.32
44    fastest(sensor.expire, expiration_deadline) <= expiration; //CONT.32
45    inhale_deadline = delay expiration by max_apnea_lag_time; //FUN.27, CONT.36.2
46    //Rationale:  if [expiration, deadline] window overlaps, inhale can be lost,
          thus window is shortned
47    shortened_inhale_deadline = fastest(inhale_deadline, sensor.inhale);
48    apnea = shortened_inhale_deadline - sample sensor.inhale on
          shortened_inhale_deadline;
49    apnea <= alarm.apnea; //FUN.27.1
50    apnea +<= mode.pcv.start; //FUN.27.2
51
52    allow sensor.inhale in [delay expiration by min_exp_time_psv, delay
          expiration by max_apnea_lag_time]; //so that controller will trigger
          inhale inside the proper window
53  }
54
55  phase() {start alternates finish};
56
57  cyber(physical: struct, user: struct) {
58    machine = phase();
59
60    selftest = phase();
61    //CONT.3:  only one selftest can happen in power cycle
62    allow selftest.start, selftest.finish in [machine.start, machine.finish];
63    //CONT.4:  if selftest happened, machine was started
64    (sample machine.start on selftest.start) = selftest.start;
65
66    ventilation = phase();
67    //ventilation can be done only while machine works
68    //ventilation has to stop inside the window before machine is off
69    allow ventilation.start, ventilation.finish in [machine.start,
          machine.finish];
70
71    mode = {
72      pcv = phase();
73      psv = phase();
74      //Modes are exclusive
75      mutex{pcv.start -> pcv.finish, psv.start -> psv.finish};
76      start = pcv.start | psv.start;
77      finish = pcv.finish | psv.finish;
78    }
79    //MLV should be considered ventilating in case if it is just a mode change
80    //Modes will finish before ventilation finishes
81    allow mode.start, mode.finish [ventilation.start, ventilation.finish];
82    //if ventilation happened then selftest should have happened or ventilation
          stopped
83    (sample selftest.start+ventilation.finish on ventilation.start) =
          ventilation.start;
84
85    //User can change the modes or to power up and down the machine.
86    //Extensive causality constraint means that other subsystems can also
87    //declare causality in other parts, like in case of apnea.
```

```
88    user.press.on_button +<= machine.start; //CONT.2
89    user.press.off_button +<= machine.finish; //CONT.10
90    user.press.psv_mode +<= mode.psv.start; //CONT.5
91    user.press.pcv_mode +<= mode.pcv.start; //CONT.6
92    user.press.ventilation_finish +<= ventilation.finish; //CONT.4.2
93
94    //Inspiration and expiration commands can only be produced
95    //by the corresponding mode command.
96    inspiration = pcv.inspiration | psv.inspiration;
97    expiration = pcv.expiration | psv.expiration;
98
99    //CONT.39
100   inspiration +<= physical.valve.out.close;
101   inspiration +<= physical.valve.intake.open;
102   physical.valve.out.close < physical.valve.intake.open;
103
104   //CONT.46 power cycle is the only choice after failure
105   //Should add reaction on how fast the failure should be dealt with,
106   //but is not present in the requirements.
107   fail +<= machine.finish;
108   //CONT.19
109   forbid (mode.start, selftest.start, ventilation.start) in [fail,
          machine.finish];
110 }
111
112 valve() {
113   //We assume that valve state is unknown at the beginning,
114   //thus first we need to close it
115   close alternates open;
116 }
117
118 physical() {
119   valve = {
120     intake = valve();
121     out = valve();
122   }
123   //CONT.1.6, safety, FUN.31, only one valve can be open at the time
124   //Prevents creating circuit in the air paths for high pressure oxygen
125   mutex{in.open -> next in.close, out.open -> next out.close};
126 }
127
128 alarm() {
129   high += apnea; //extensive sum constraint
130 }
131
132 spec() {
133   cyber(physical(), {});
134 } upper interface { //Checking higher-level properties
135   //FUN.32:  breathing is not obstructed, valve to exhale opens
136   //at most 5 seconds after closing
137   //5s is just an example
138   spec.physical.valve.out.open <= (delay spec.physical.valve.out.close by 5s);
139 }
140
141 //Checking properties:
142 //- not empty => schedules exist;
```

```
//- weakly live => there is potential infinite behaviour which involves
//all clocks, not necessary in the same schedule
//- safe => representation is finite, can be used safely as monitor
//- property upper interface is checked automatically
spec() is schedulable and weakly live and safe;
```

Listing A.1: Full MRTCCSL specification of Mechanical Lung Ventilator

```
//cylinder timings
cylinder(shaft_degree: clock, offset: int) {
  otdc = skip offset+0   every 720 shaft_degree;
  fbdc = skip offset+180 every 720 shaft_degree;
  itdc = skip offset+360 every 720 shaft_degree;
  sbdc = skip offset+540 every 720 shaft_degree;

  exhaust.start = sbdc $ -[45, 60] on shaft_degree;
  exhaust.finish = (next otdc) $ [5, 20] on shaft_degree;
  ignition_point = (next itdc) $ [-30, 10] on shaft_degree;
  knock_window.start = (next itdc) $ [0, 55] on shaft_degree;
  knock_window.finish = knock_window.start $ [0, 55] on shaft_degree;
}

//timing constraints of cylinders on tasks
task_cylinder_rel(t: struct, c: struct, sensor_sampling: clock) {
  c.exhaust.start |<= t.oxygen_sensing.start;
  t.oxygen_sensing.finish <= c.exhaust.finish;
  t.ignition_control.finish <= c.ignition_point;
  t.knock_sensing.start <= sample c.knock_window.start on sensor_sampling;
  t.knock_sensing.finish <= sample c.knock_window.finish on sensor_sampling;
}

engine_control(rpm: int) {
    rpm in [600, 4500];
  //constants
  shaft_period = 1s/(360*rpm/60);
  frequency = 20MHz;
  ms_scale = frequency/1kHz;
  sensor_scale = frequency/100kHz;

  //controller hardware
  platform = basic_platform(frequency,20ppm of frequency);

  internal_ms = every ms_scale base_clock;
  sensor_sampling = every sensor_scale base_clock;

  //engine generic timing
  shaft_degree = periodic shaft_period rel.error +-1% + ?;
  cylinder0 = cylinder(shaft_degree, 0);
  cylinder1 = cylinder(shaft_degree, 180);
  cylinder2 = cylinder(shaft_degree, 360);
  cylinder3 = cylinder(shaft_degree, 540);

  //processing
  task = {
        knock_sensing = task(?);
        oxygen_sensing = task(?);
        ignition_control = task(?);
```

```
        knock_filtering = task(?);
        knock_control = task(?);
    }
  //data dependencies
  task.oxygen_sensing.finish <= task.ignition_control.start;
  task.knock_filtering.finish <= task.ignition_control.start;
  task.ignition_control.finish <= next task.knock_control.start;

  //resource filter_buffer is only one
  mutex{
    task.knock_sensing.start -> task.knock_sensing.finish,
    task.knock_filtering.start -> task.knock_filtering.finish,
  };
  //timing requirements of cylinders on tasks
  task_cylinder_rel(task, cylinder0, sensor_sampling);
  task_cylinder_rel(task, cylinder1, sensor_sampling);
  task_cylinder_rel(task, cylinder2, sensor_sampling);
  task_cylinder_rel(task, cylinder3, sensor_sampling);
}
//In particular
engine_control(1166) is live and safe;
//Or in general
find rpm where engine_control(rpm) is live and safe;
```

Listing A.2: Full MRTCCSL specification of Spark engine control system

```
brake(freq: Hz, e: interval<time>) assume {
  ms_scale = freq/1kHz;
  p = basic_platform(freq, e);
} {
  abs_correction = p.task(?ms);
  braking = p.task(?);
  speed = p.task(?);

  receive.cmd <= abs_correction.start;
  abs_correction.end <= braking.start;
  braking.end = actuation;

  speed.ready = skip ? every 10*ms_scale p.base_clock;
  speed.ready <= speed.start;
  speed.finish <= send.speed.ready;
} upper interface {
  actuation = delay receive.cmd by [0,10ms];
  send.speed.ready = skip ? every 10*ms_scale p.base_clock;
}

controller(freq: Hz, e: interval<time>) assume {
  ms_scale = freq/1kHz;
  p = basic_platform(freq, e);
} {
  torque_comp = p.task(?);
  pedal = p.task(?);

  pedal.start = skip ? every 10*ms_scale p.base_clock;
  //change in from pedal can appear when finish checking
  pedal.change subclocks pedal.finish;
  pedal.change <= torque_comp.start;
```

```
32
33    torque_comp.finish <= send.fl.cmd;
34    torque_comp.finish <= send.fr.cmd;
35    torque_comp.finish <= send.rl.cmd;
36    torque_comp.finish <= send.rr.cmd;
37  }
38
39  bbw(freq: Hz, e1: interval<time>, e2: interval<time>, e2e_latency: time) {
40    brakes = {
41      fl = brake(freq, e1);
42      fr = brake(freq, e1);
43      rl = brake(freq, e1);
44      rr = brake(freq, e1);
45    }
46    c = controller(2*freq, e2);
47
48    c.send.fl.cmd < brakes.fl.receive.cmd;
49    brakes.fl.send.speed < c.receive.speed;
50    c.send.fr.cmd < brakes.fr.receive.cmd;
51    brakes.fr.send.speed < c.receive.speed;
52    c.send.rl.cmd < brakes.rl.receive.cmd;
53    brakes.rl.send.speed < c.receive.speed;
54    c.send.rr.cmd < brakes.rr.receive.cmd;
55    brakes.rr.send.speed < c.receive.speed;
56
57    //bus mutex
58    send = c.send | brakes.fl.send | brakes.fr.send | brakes.rl.send |
          brakes.rr.send;
59    receive = brakes.fl.receive | brakes.fr.receive | brakes.rl.receive |
          brakes.rr.receive | c.receive
60    send alternates receive;
61    //or what is the size of the packet
62    receive = delay send by [0.75, 1]ms;
63  } assert {
64    //reaction constraints
65    reaction_deadline = delay c.pedal.change by e2e_latency;
66    brakes.fl.actuation < reaction_deadline;
67    brakes.fr.actuation < reaction_deadline;
68    brakes.rl.actuation < reaction_deadline;
69    brakes.rr.actuation < reaction_deadline;
70
71    //brake synchronization
72    s = slowest(brakes.fl.actuation, brakes.fr.actuation, brakes.rl.actuation,
          brakes.rr.actuation);
73    f = fastest(brakes.fl.actuation, brakes.fr.actuation, brakes.rl.actuation,
          brakes.rr.actuation);
74    s < (delay f by 5ms);
75  }
76
77  find f where bbw(f, +-1% of f, +-1% of f, 10ms) is schedulable and safe;
```

Listing A.3: Full MRTCCSL specification of Brake-by-wire system

A.2 Translation of CCSL constraints into NBac

Here we list all the constraints that we defined in the language of NBac, but are not critical to understand the main content of the work.

```
state
    init: bool;
    ok: bool;
    fi,li: int;
    fl_turn : bool;
input
    f,l: bool;
local
    fin,lin: int;
definition
    fin = if init then 0 else fi + (if f then 1 else 0);
    lin = if init then 0 else li + (if l then 1 else 0);
transition
    fi' = fin;
    li' = lin;
    fl_turn' = if init then false else if f then true else if l then false
        else fl_turn;
    init' = false;
    ok' = if init then true else ok and (fin - lin <= 1) and (fin - lin >= 0);
assertion ((not fl_turn => not l) and (fl_turn => not f)) or init;
initial init;
final not (init or ok);
```

Listing A.4: Alternation a `alternates` b using Boolean state

```
state
    init: bool;
    ok: bool;
    fi,li: int;
input
    f,l: bool;
local
    fin,lin: int;
definition
    fin = if init then 0 else fi + (if f then 1 else 0);
    lin = if init then 0 else li + (if l then 1 else 0);
transition
    fi' = fin;
    li' = lin;
    init' = false;
    ok' = if init then true else ok and (fin - lin <= 1) and (fin - lin >= 0);
assertion ((fi = li => not l) and (fi > li => not f)) or init;
initial init;
final not (init or ok);
```

Listing A.5: Alternation a `alternates` b using integer state only

```
state
    init: bool;
    ok: bool;
    ai,bi: int;

input
```

```
    a,b: bool;

local
    ain,bin: int;

definition
    ain = if init then 0 else ai + (if a then 1 else 0);
    bin = if init then 0 else bi + (if b then 1 else 0);


transition
    ai' = ain;
    bi' = bin;
    init' = false;
    ok' = if init then true else ok and (ain-bin >= 0);

assertion ((ai=bi => not (b and not a))) or init;

initial init;
final not (init or ok);
```

Listing A.6: Causality $a \preccurlyeq b$

```
state
    init: bool;
    ok: bool;
    ai,bi,ri: int;
    ar_sampled,ab_delay0,ab_delay1: bool;

input
    a,b,r: bool;

local
    ain,bin,rin: int;

definition
    ain = if init then 0 else ai + (if a then 1 else 0);
    bin = if init then 0 else bi + (if b then 1 else 0);
    rin = if init then 0 else ri + (if r then 1 else 0);

transition
    init' = false;
    ai' = ain;
    bi' = bin;
    ri' = rin;
    ar_sampled' = if init then false else if r then false else (ar_sampled or
        a);
    ab_delay0' = if init then false else if r then (ar_sampled or a) else
        ab_delay0;
    ab_delay1' = if init then false else if r then ab_delay0 else ab_delay1;
    ok' = if init then true else (ok and (bin <= ain and bin <= rin));
assertion (b = (r and ab_delay1)) or init;
initial init;
final not (init or ok);
```

Listing A.7: Ternary delay $b = a \ \$ \ 2 \ \text{on} \ r$

```
1
2  state
3      init: bool;
4      ok: bool;
5      ai,bi: int;
6
7  input
8      a,b: bool;
9
10 local
11     ain,bin: int;
12
13 definition
14     ain = if init then 0 else ai + (if a then 1 else 0);
15     bin = if init then 0 else bi + (if b then 1 else 0);
16
17 transition
18     init' = false;
19     ai' = ain;
20     bi' = bin;
21     ok' = if init then true else (ok and (ai - bi <= 5));
22 assertion ((ai < 5 => not b) and (ai >= 5 => a = b)) or init;
23 initial init;
24 final not (init or ok);
```

Listing A.8: Delay $b = a \$ 5$

```
1  state
2      init: bool;
3      ok: bool;
4      oi,ai,bi: int;
5
6  input
7      o,a,b: bool;
8
9  local
10     oin,ain,bin: int;
11
12 definition
13     oin = if init then 0 else oi + (if o then 1 else 0);
14     ain = if init then 0 else ai + (if a then 1 else 0);
15     bin = if init then 0 else bi + (if b then 1 else 0);
16
17
18 transition
19     oi' = oin;
20     ai' = ain;
21     bi' = bin;
22     init' = false;
23
24 assertion (((ai >= bi and b) or (bi >= ai and a)) = o) or init;
25
26 initial init;
27 final not (init or ok);
```

Listing A.9: Slowest $o = \texttt{slowest}(a, b)$

```
state
    init: bool;
    ok: bool;
    oi,ai,bi: int;

input
    o,a,b: bool;

local
    oin,ain,bin: int;

definition
    oin = if init then 0 else oi + (if o then 1 else 0);
    ain = if init then 0 else ai + (if a then 1 else 0);
    bin = if init then 0 else bi + (if b then 1 else 0);


transition
    oi' = oin;
    ai' = ain;
    bi' = bin;
    init' = false;

assertion (((ai >= bi and a) or (bi >= ai and b)) = o) or init;

initial init;
final not (init or ok);
```

Listing A.10: Fastest $o = \texttt{fastest}(a, b)$

A.3 Inductive reasoning test suite

In this section we give an overview of the tests we did on the implementation [Tok24] of the method, described in Section 5.1. We describe every of them as a MRTCCSL module, which parts are checked for schedulability and subspecification relation, as defined in Section 4.4.2:

- first of all, we check that an empty module is a correct module;
- check that two delays with constant delays do result in precedence, expressed as guarantee im module $(\emptyset, S, G, \emptyset)$:

$$S = \begin{cases} l & = \texttt{delay } in \texttt{ by } 1\,\mathrm{s} \\ r & = \texttt{delay } in \texttt{ by } 2\,\mathrm{s} \end{cases}$$
$$G = \{l \prec r$$

and its slight variation, checking that there is a solution when the delays are uncertain and the second is the slower out of two:

$$S = \begin{cases} l & = \texttt{delay } in \texttt{ by } [1\,\mathrm{s}, 2\,\mathrm{s}] \\ r & = \texttt{delay } in \texttt{ by } [3\,\mathrm{s}, 4\,\mathrm{s}] \end{cases}$$

$$G = \begin{cases} f & = \texttt{fastest}(l, r) \\ f & \prec r \end{cases}$$

- check that module $(A, S, \emptyset, \emptyset)$, implementing sampling with delay on an assumed periodic clock, is valid:

$$A = \begin{cases} b & = 50\,\mathrm{s} \cdot i^{\mathrm{th}} \pm \texttt{rel}\,.\,[-2\,\mathrm{s}, 2\,\mathrm{s}] + 5\,\mathrm{s} \end{cases}$$

$$S = \begin{cases} d & = \texttt{delay } e \texttt{ by } 1\,\mathrm{s} \\ s & = \texttt{sample } d \texttt{ on } b \end{cases}$$

- this is the same specification we have given in Section 5.1.1 as introduction to the method, with the delays d_1, d_2 and tolerance t being parameters of this module:

$$\begin{aligned} w_1 &= \texttt{delay } r \texttt{ by } d_1 \\ w_2 &= \texttt{delay } r \texttt{ by } d_2 \\ f &= \texttt{fastest}(w_1, w_2) \\ s &= \texttt{slowest}(w_1, w_2) \end{aligned} \quad \Subset s < \texttt{delay } f \texttt{ by } t$$

- end-to-end latency check for module $(A, S, G, \emptyset)$ with parameter d_t set either to 600 (satisfies) or 200 (does not) and other parameters set to $d_1, d_s = 3\,\mathrm{s}, n = 4$:

$$A = \begin{cases} r & = 50\,\mathrm{s} \cdot i^{\mathrm{th}} \pm \texttt{rel}\,.\,[-2\,\mathrm{s}, 2\,\mathrm{s}] + 6\,\mathrm{s} \end{cases}$$

$$S = \begin{cases} b & = \texttt{delay } a \texttt{ by } d_1 \\ d & = b \,\$\, n \texttt{ on } r \\ e & = \texttt{delay } d \texttt{ by } d_2 \\ f_b & = \texttt{first sampled } b \texttt{ on } r \\ f_b & = \texttt{delay } f_a \texttt{ by } d_1 \\ f_a & \subseteq a \\ g & = \texttt{delay } f_a \texttt{ by } t \end{cases}$$

$$G = \begin{cases} e \prec g \end{cases}$$

In another test we relax the requirements on parameter t by setting it either to $[262\,\mathrm{s}, 1000\,\mathrm{s}]$ resulting in correct module and $[100\,\mathrm{s}, 261\,\mathrm{s}]$, which does not satisfy the guarantee;

- verification that delaying sampled clock can be substituted by delay $((\emptyset, S, G, \emptyset))$, where n can be any number, but not a parameter:

$$S = \begin{cases} b & = \texttt{sample } a \texttt{ on } r \\ c & = b \,\$\, n \texttt{ on } r \end{cases}$$

$$G = \begin{cases} c & = a \,\$\, n \texttt{ on } r \end{cases}$$

- this module $(A, S, \emptyset, \emptyset)$ with parameters fixed to $n_1 = 2, n_2 = 3, d = 5\,\mathrm{s}$ predictably results in an exception because of a loop in saturation, as we described in Section 5.1.2.1:

$$A = \left\{ r \quad = 10\,\mathrm{s} \cdot i^{\mathtt{th}} \pm \mathtt{rel}\,.\,[-2\,\mathrm{s}, 2\,\mathrm{s}] + 10\,\mathrm{s} \right.$$

$$S = \begin{cases} b & = a \mathbin{\$} n_1 \mathbin{\mathtt{on}} r \\ c & = \mathtt{delay}\ b\ \mathtt{by}\ d \\ r_d & = r \mathbin{\$} n \\ d & = c \mathbin{\$} n_2 \mathbin{\mathtt{on}} r_d \end{cases}$$

- lastly we also test that the parameters behave as expected and satisfy (or not) some basic interval inclusion:

$$x = [100, 200] \Subset x = [100, 400]$$
$$x = [100, 100] \Subset x = [100, 100]$$
$$x = [100, 200] \not\Subset x = [200, 400]$$

Langage Modulaire pour la Spécification de Contraintes d'Horloges Logiques et Temps-Réel

Pavlo TOKARIEV

Résumé

Les systèmes en temps réel critiques (réactifs) sont des systèmes qui contrôlent des processus complexes et dont la faute n'est pas acceptable en raison des graves conséquences pour le système, l'infrastructure et les humains. Dans ces systèmes, le moment de la réaction est aussi critique que l'exécution de la bonne action. Dans ce travail, nous nous concentrons sur le premier. Pour ce faire, nous utilisons une abstraction du temps, connue sous le nom de temps logique. Il abstrait totalement les instants auxquels les événements se produisent par leur position relative. Le langage sur lequel nous basons notre travail, Clock Constraint Specification Language (CCSL), est purement basé sur la notion d'horloge logique et conçu pour décrire les exigences temporelles des systèmes. L'application du langage nous a permis de constater que l'approche purement logique n'est pas toujours adéquate. Le langage de spécification doit permettre de décrire des relations temps réel. Leur simulation purement avec des horloges logiques échoue en général en raison de la différence de complexité. Ceci nous incite à trouver des moyens d'abstraire ou de résoudre en utilisant des méthodes moins exactes.Ainsi, dans ce travail, nous proposons une série d'extensions du langage original, orthogonales mais se complétant. Celles-ci couvrent les contraintes en temps réel et les contraintes auxiliaires pour augmenter l'expressivité, la paramétrisation des contraintes et le cadre modulaire avec une fonction similaire au raffinement.Nous les définissons formellement et motivons leur conception à l'aide de plusieurs cas d'utilisation. Nous rapportons nos expériences avec l'interprétation abstraite dans l'analyse des spécifications et proposons plusieurs modifications pour la rendre plus précise. Enfin, nous introduisons notre propre solveur ad hoc utilisant une représentation polyédrique sur un fragment du langage.

Mots-clés : Systèmes Temps-Réel, Exigences Temporelles, Temps Logique, Temps Réel, Vérification Formelle, Interprétation Abstraite.

Abstract

Safety-critical real-time (reactive) systems are systems in control of complex processes and which failure is not acceptable due to severe consequences for the system, infrastructure and people. In such systems, the timing of the reaction is as critical as doing the right action. In this work, we focus on the former. For this we use an abstraction of time, known as logical time. It completely abstracts away the instants at which events occur by their relation to each other. The language we base our work on, the Clock Constraint Specification Language (CCSL), is purely based on logical clocks and is designed to describe temporal requirements of systems. From the application of the language, we notice that pure logical approach is not always adequate or efficient, as specification languages for such systems do need to express real-time relations. And attempts to simulate using pure logical clocks fail in general for large systems due to the combinatorial complexity. Which in turn prompts us to find ways to abstract or solve the specifications using approximate methods and renounce exact solutions. Thus, in this work, we propose a series of extensions to the original language, orthogonal but complementing each other. These cover real-time and auxiliary constraints to increase expressiveness, parametrization of constraints and modular framework with a mechanism akin to refinement. We define them formally and motivate their design by using several use cases. We report our experiments with abstract interpretation in specification analyses, propose several modifications to make it more precise and demonstrate them on the mentioned use cases. Finally, we introduce our own polyhedra-based ad-hoc solver for a fragment of the language.

Keywords: Real-Time Systems, Temporal Requirements, Logical Time, Real-Time, Formal Verification, Abstract Interpretation.