Methodology for the formal verification of temporal properties for real-time safety-critical applications based on logical time

Fabien Siron

UNIVERSITÉ CÔTE D'AZUR

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

THÈSE DE DOCTORAT

Méthodologie de vérification formelle de propriétés temporelles pour les applications temps-réel critiques basées sur le concept de temps logique

Fabien SIRON

Equipe Kairos, Centre Inria d'Université Côte d'Azur

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

Dirigée par : Dumitru POTOP-BUTUCARU, Chargé de recherche, Centre Inria d'Université Côte d'Azur

Co-dirigée par : Robert DE SIMONE, Directeur de recherche, Centre Inria d'Université Côte d'Azur

Soutenue le : 11 Décembre 2023

Devant le jury, composé de :

Reinhard VON HANXLEDEN, Professor, Kiel University

Pierre-Loïc GAROCHE, Professeur, École Nationale de l'Aviation Civile

Timothy BOURKE, Chargé de recherche, Centre Inria de Paris

Damien CHABROL, Responsable Innovation et Stratégie Produit, Krono-Safe

Amira METHNI, Architecte logiciel, Krono-Safe

Inria

KRONO-SAFE

MÉTHODOLOGIE DE VÉRIFICATION FORMELLE DE PROPRIÉTÉS TEMPORELLES POUR LES APPLICATIONS TEMPS-RÉEL CRITIQUES BASÉES SUR LE CONCEPT DE TEMPS LOGIQUE

Methodology for the formal verification of temporal properties for real-time safety-critical applications based on logical time

Fabien SIRON

Jury :

Rapporteurs

Reinhard VON HANXLEDEN, Professor, Kiel University
Pierre-Loïc GAROCHE, Professeur, École Nationale de l'Aviation Civile

Examinateurs

Timothy BOURKE, Chargé de recherche, Centre Inria de Paris

Directeur de thèse

Dumitru POTOP-BUTUCARU, Chargé de recherche, Centre Inria d'Université Côte d'Azur

Co-directeur de thèse

Robert DE SIMONE, Directeur de recherche, Centre Inria d'Université Côte d'Azur

Membres invités

Damien CHABROL, Responsable Innovation et Stratégie Produit, Krono-Safe
Amira METHNI, Architecte logiciel, Krono-Safe

Université Côte d'Azur

À mon grand-père.

Méthodologie de vérification formelle de propriétés temporelles pour les applications temps-réel critiques basées sur le concept de temps logique

Résumé

Les systèmes temps-réel critiques doivent respecter des contraintes temporelles strictes qui doivent être considérées tout au long du cycle logiciel. Cependant, du fait que les temps d'exécution exacts ne sont généralement pas connus lors de la conception, le *temps logique* fournit un moyen d'abstraire les contraintes temporelles et les exécutions du *temps physique*, dépendant de la plateforme. Dans cette thèse, nous nous concentrons sur deux formalismes basés sur le temps logique. L'approche *Synchrone-Réactive* abstrait totalement le temps physique en utilisant des bases de temps discrète sur lesquelles les calculs sont déclenchés. L'approche de *Temps d'Exécution Logique* utilise des bases de temps logique pour représenter non seulement les instants de déclenchement, mais aussi les durées des calculs élémentaires. Dans notre travail, nous commençons par définir une unification des approches *Synchrone-Réactives* et de *Temps d'Exécution Logique*, fournissant un cadre formel naturel pour définir la sémantique de PSYC, un langage temps-réel industriel expressif. Nous définissons deux sémantiques pour celui-ci : une sémantique native *à grands pas*, préservant les durées logiques des intervalles de temps, définie par des règles structurelles opérationnelles ; et une sémantique synchrone *à petits pas* définie par traduction vers un langage Synchrone-Réactif étendant les durées des intervalles de temps à une succession de transitions atomiques. Nous montrons que les deux sémantiques sont équivalentes. Cette formalisation de la sémantique de PSYC nous permet de définir une méthodologie de vérification formelle pour PSYC basée sur du *model-checking symbolique*. Pour réduire l'espace d'état pendant le model checking, nous définissons une technique d'optimisation inspirée du *model-checking temporisé*. Enfin, nous spécifions les exigences temporelles que nous voulons vérifier via un langage de spécification de contrainte d'horloge — CCSL — qui sont ensuite traduites en observateurs synchrones.

Mots-clés : Systèmes Temps-Réel, Temps d'Exécution Logique, Langages Synchrone-Réactifs, Vérification Formelle.

Methodology for the formal verification of temporal properties for real-time safety-critical applications based on logical time

Abstract

Safety-critical real-time systems have to respect strict timing constraints. Thus, timing constraints must be considered throughout the software development cycle. As exact computation execution time are generally not known during design, *logical time* provides a way to abstract time constraints and execution from platform-dependent *physical time*. In this thesis, we focus on two main formalisms based on logical time. The *Synchronous-Reactive* approach totally abstracts physical time by discrete time bases on which computations are triggered. The *Logical Execution Time* approach uses logical time bases to represent not only triggering instants but also the durations of elementary computations. In this thesis, we start by unifying *Synchronous-Reactive* and *Logical Execution Time* approaches. This provides the natural formal framework for defining the semantics of PSYC, an expressive industrial real-time language. We define two formal semantics for PSYC: a native *big-step* semantics preserving the logical durations of time intervals defined by structural operational rules and a synchronous *small-step* semantics defined by translation to a Synchronous-Reactive language expanding time interval durations to a succession of atomic transitions. We show that the two semantics definitions are equivalent. This formalization of the PSYC semantics enables us to define a formal verification methodology for PSYC based on *symbolic model-checking*. To reduce the state space during model-checking, we also define an optimization technique inspired by *timed automata* model-checking. Finally, we show how to encode high-level timing requirements into a clock constraint specification language — CCSL — which are then translated to synchronous observers.

Keywords: Real-Time Systems, Logical Execution Time, Synchronous-Reactive Languages, Formal Verification.

Remerciements

Cette thèse a été menée conjointement avec la société KRONO-SAFE — renommé depuis ASTERIOS TECHNOLOGIES — et l'équipe-projet KAIROS de l'INRIA. Je remercie grandement ces deux entités d'avoir permis cette collaboration et d'avoir montré un intérêt commun pour ce travail.

* * *

Je tiens ainsi à remercier mes directeurs de thèse — Dumitru POTOP-BUTUCARU et Robert DE SIMONE — qui ont fait preuve d'un soutien sans faille tout au long de cette thèse. Ils ont été particulièrement présents, malgré la distance géographique, et malgré les périodes de confinement. C'est d'abord grâce à eux que je dois cette initiation au monde de la recherche académique ainsi que l'aboutissement de ce travail. Je remercie aussi l'équipe KAIROS pour son accueil, ainsi que plus globalement l'INRIA, pour toutes les discussions intéressantes que j'ai pu avoir, avec Marc POUZET, Yves BERTOT, Julien DEANTONI, Frédéric MALLET, Timothy BOURKE, et bien d'autres. Je remercie aussi l'EPITA dont la formation m'a été précieuse, et plus particulièrement Geoffrey L.G. qui m'a permis de m'initier à l'enseignement. Je n'oublierai pas non plus les multiples rencontres que j'ai pu avoir pendant cette thèse au cours des différents évènements : Adrien R., Nan L., João C., Hugo P., et d'autres.

Ensuite, je tiens aussi à remercier l'entreprise KRONO-SAFE pour le support qu'elle a fourni, ainsi que l'encadrement de Damien CHABROL et Amira METHNI dont les conseils ont été cruciaux tout au long de cette thèse. Après avoir été recruté par cette entreprise, mon espoir de faire une thèse de doctorat s'était amenuisé, remplacé néanmoins par un travail prometteur sur des sujets intéressants. Ce n'est toutefois sans compter sur l'intérêt et la confiance de Damien et Amira qui ont transformé ma volonté initiale en un projet solide et défendable. L'entreprise a fait preuve d'une grande confiance en co-finançant cette aventure avec le support de l'ANRT[*]. Je tiens ainsi à remercier l'ensemble des collègues qui ont encouragé ou contribué, directement ou indirectement, au succès de ce travail par de précieuses discussions : Frédéric T., Christophe A., Jean G., Stéphane-Jean M., Florian A., Guillaume P., et bien d'autres. Je remercie aussi chaleureusment l'entreprise PROVER TECHNOLOGY, et plus particulièrement Nicolas A. pour son aide et son encouragement.

* * *

Je souhaite aussi remercier particulièrement Pierre-Loic GAROCHE et Reinhardt VON HANXLEDEN d'avoir accepté de rapporter cette thèse et d'y avoir consacré leur temps, ainsi que Timothy BOURKE d'avoir accepté de faire partie du jury de cette thèse.

∗. Association Nationale de la Recherche et de la technologie

* * *

Sur une touche plus personnelle, je souhaite aussi remercier mes parents, ma famille, mes amis, ainsi que celle qui est devenue ma femme. Cette aventure a été particulièrement marquée par un évènement heureux, mon mariage, ainsi que par un évènement malheureux, la perte de mon grand-père, qui ne pourra malheureusement pas voir l'aboutissement de ce travail. L'accomplissement de cette thèse est grandement lié à eux.

Contents

Language and Semantics

Appendix

CHAPTER 1

Introduction

1.1 General context

Safety-critical real-time systems have to respect strict time constraints. Thus, time constraints must be considered throughout the software development cycle: from system specification to integration on a specific hardware platform. However, the exact execution time of computations is generally not known during the system specification and design phases, as they depend on a specific target. The *Logical Time* concept provides a way to abstract time constraints and execution from hardware *Physical Time* by providing discrete time bases on which computations react. At integration time, analyses are then used to ensure those physical time constraints — mainly execution time estimations — satisfy the logical time constraints necessary to ensure the system's safety.

In this thesis, we focus on two main formalisms based on logical time. The *Synchronous-Reactive* (SR) approach totally abstracts physical time by discrete time bases — called *logical clocks* — on which computations are executed atomically on specific ticks. Although SR is very expressive (partly due to its logical clocks), it suffers from implementation difficulties when using multi-scale time constraints. As an alternative, the *Logical Execution Time* approach uses logical time bases to represent not only triggering instants but also the durations of elementary computations. This defines time windows which contain the actual computation and on which communication can only happen on the boundaries of those logical time intervals. Both SR and LET provide a deterministic and concurrent way to deal with logical time constraints. ESTEREL and LUSTRE are examples of Synchronous-Reactive languages, while GIOTTO and TDL are examples of Logical Execution Time languages. However, LET suffers from expressivity limitations; in particular, timing durations are based on a unique chronometrical time base, and thus, cannot express timing constraints based on multiform logical time. XGIOTTO, LINGUA FRANCA and PSYC are examples of languages that include features from both SR and LET:

— XGIOTTO is an imperative language that extends GIOTTO and LET with events via a mechanism called event scoping [GHKS04] ;

— LINGUA FRANCA is a dataflow coordination language developed in Berkeley university which supports LET while having a semantics closer to the SR approach [LL22];

— PSYC is a real-time integration language developed initially at the CEA *, and now by KRONO-SAFE, which extends LET with synchronous instants [ACA+96].

This thesis is focused on the PSYC language, developed by the company KRONO-SAFE – now renamed ASTERIOS TECHNOLOGIES as a SAFRAN ELECTRONICS & DEFENSE subsidiary — who is funding this work.

Those languages are typically used to design a specific program satisfying a set of high-level timing requirements such as constraints among the timing of various components (e.g., period, synchronizations) or constraints on data propagation among various components (e.g. end-to-end latency). In a safety-critical process, those requirements should be verified very carefully in the program. Traditionally, this activity is done manually by code inspection or by simulation. However, formal verification provides stronger guarantee to ensure that those requirements are satisfied.

1.2 Contributions

We now describe the main contributions of this thesis illustrated in Figure 1.1 which are split in two topics: language semantics and formal verification.

Language Semantics PSYC inherits from both the Synchronous-Reactive (SR) and the Logical Execution Time (LET) approach. By identifying how SR and LET formalisms relate, we define in this thesis a synchronous generalization of LET called *synchronous LET* in which time interval boundaries are specified by synchronous instants. We use sLET as a semantics foundation for PSYC: we define a native *big-step* semantics preserving the logical durations of time intervals — which is close to the one used in the compiler today — defined by structural operational rules and a synchronous *small-step* semantics defined by translation to the synchronous language ESTEREL expanding time interval durations to a succession of atomic transitions. We show that both semantics definitions are equivalent on the boundaries of the LET time intervals.

Formal Verification From the synchronous semantics of PSYC, this thesis also defines a verification methodology for PSYC based on symbolic model-checking and on the modelization of timing requirements as synchronous observers. However, as the synchronous semantics expand all LET intervals in atomic transitions, this approach does not scale well. To solve this problem, we define an optimization technique inspired by *timed automata* model-checking, which preserves durations from the native semantics assuming that only one base clock is used in the program. Finally, we show how to encode our requirements in a clock constraint logic — CCSL — instead

∗. French Alternative Energies and Atomic Energy Commission

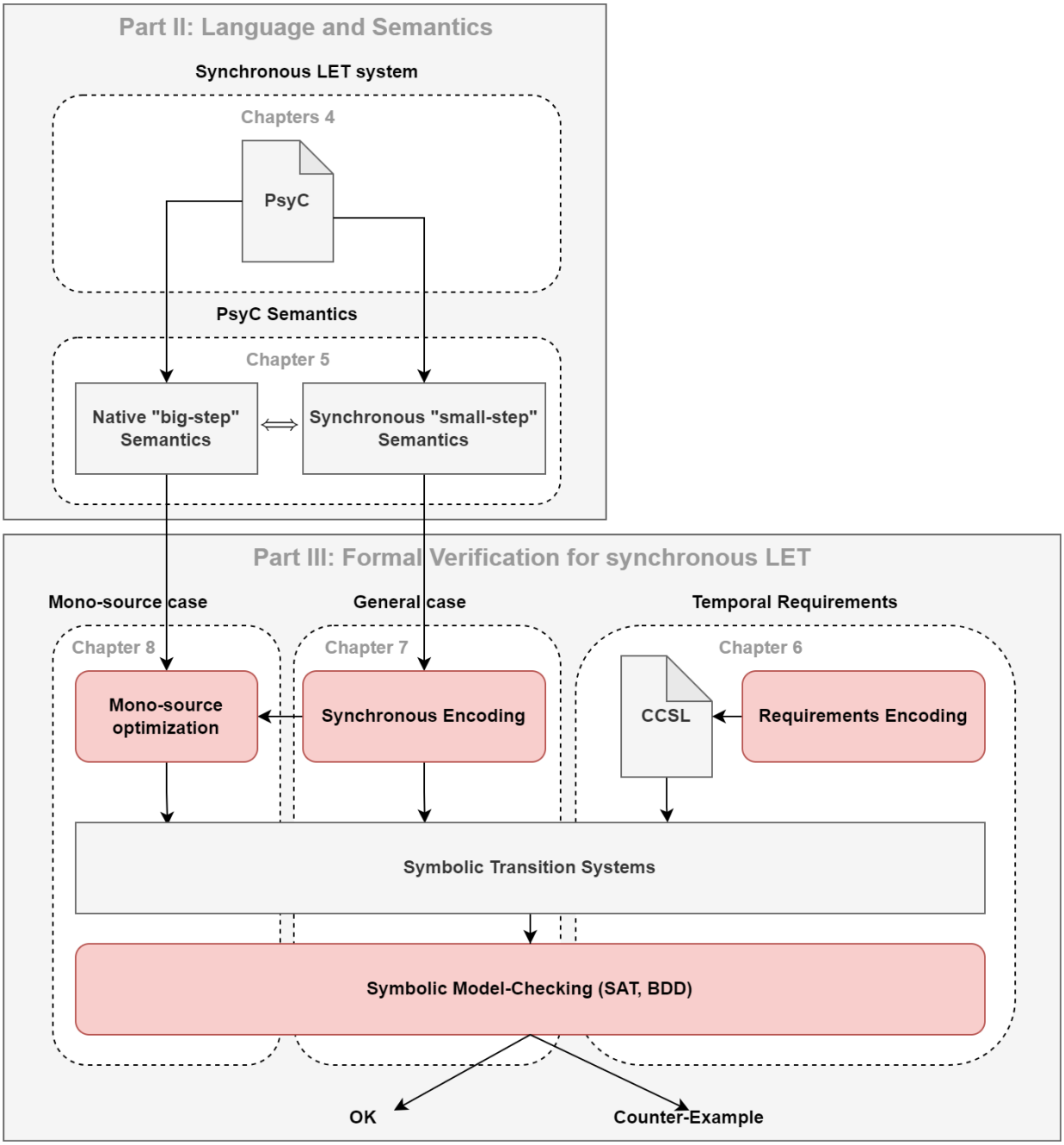


Figure 1.1 – Global methodology proposed in this thesis

of a more traditional temporal logic — such as LTL — as the program typically has multiple time scales among components. In both cases, however, requirements belonging to the *safety* class can be translated to synchronous observers.

1.3 Thesis Outline

The outline of this thesis is composed of three main parts:

— The first part presents the overall context and background of this thesis, both academically and industrially. In Chapter 2, we describe the overall context and industrial background. Then, in Chapter 3, we give an overview of various languages and formalisms used to specify, design and implement real-time systems.

— The second part focuses on semantics foundations for the PSYC language. Chapter 4 gives an overview of the PSYC language and defines synchronous LET as being a synchronous generalization of Logical Execution Time. It is then used in Chapter 5 to define the native and the synchronous semantics of PSYC. We also give an equivalence criterion between those semantics, which proof is sketched in Appendix A.

— The third part focuses on a verification methodology based on the semantics defined on the second part . Chapter 6 formalizes the different temporal requirements that we want to verify on PSYC programs. Then, Chapter 7 describes the overall verification methodology for the general case of PSYC programs. Finally, Chapter 8 focuses on the subcase of PSYC programs using only one base clock to define an optimization method compressing the state space.

— In chapter 9, as a final part, we give a global conclusion of this thesis as well as various perspectives.

1.4 Publications and Communications

In this thesis, we did multiple publications and communications. This section describes them by category.

International Conferences and Workshops

[1] Fabien Siron, Dumitru Potop-Butucaru, Robert De Simone, Damien Chabrol, and Amira Methni. Semantics fondations of PsyC based on synchronous Logical Execution Time. In *TCRS 2023 - Time Centrics Reactive Software*, San Antonio, Texas, U.S.A., May 2023.

[2] Fabien Siron, Dumitru Potop-Butucaru, Robert De Simone, Damien Chabrol, and Amira Methni. The synchronous Logical Execution Time paradigm. In *ERTS 2022 - Embedded Real Time Systems*, Toulouse, France, June 2022.

[3] Fabien Siron, Dumitru Potop-Butucaru, Robert de Simone, Damien Chabrol, and Amira Methni. (wip:)programming and verifying real-time design using logical time. In *FDL 2021-Forum on specification & Design Languages*, 2021. https://inria.hal.science/hal-03537976/.

National Conferences and Workshops

[4] Fabien Siron, Dumitru Potop-Butucaru, Robert de Simone, Damien Chabrol, and Amira Methni. Vérification d'applications temps-réel basées sur le paradigme de logical execution time (let). In *École d'Été Temps Réel 2021*, 2021. https://inria.hal.science/hal-03545758/.

Research Report

[5] Fabien Siron, Dumitru Potop-Butucaru, Robert De Simone, Damien Chabrol, and Amira Methni. Formal Semantics of the PsyC language. Research report, INRIA Sophia Antipolis - Méditerranée (France), 2023.

Communication without proceedings

[6] Fabien Siron, Dumitru Potop-Butucaru, Robert De Simone, Damien Chabrol, and Amira Methni. Synchronous Logical Execution Time: Towards formal verification. In *Synchron 2021, 28th International Open Workshop on Synchronous Programming*, La Rochette, France, November 2021.

Unpublished articles

[7] Fabien Siron, Dumitru Potop-Butucaru, Robert De Simone, Damien Chabrol, and Amira Methni. Formally verifying timing requirements on applications based on synchronous Logical Execution Time. 2023.

First Part

Background

Learning *never exhausts the mind.*

— *Attributed to* Leonardo da Vinci, (1452-1519).

CHAPTER 2

Safety-Critical Real-Time Systems

Today, in fields such as avionics and automotive, software plays an increasingly important role. Among the significant trends in these fields are automation and electrification. Examples include automatic landing systems [Air23] and electrification of aircraft landing gears [Saf23]. This results in increasingly complex systems, and, due to strong economic constraints, also results in harder integration process and bigger function concentration. For all these reasons, software reliability is becoming a major challenge in those fields. This thesis falls within this context. The following sections give global definitions characterizing the overall context.

Critical Systems In this work, we focus on systems that are qualified as *critical* due to the consequences a system failure could have. We give below the definition from the IEEE * glossary [iee90]:

> « *Software whose failure could have an impact on safety, or could cause large financial or social loss.* » (IEEE)

Depending on these consequences, critical systems can be classified into two main classes:

— *Safety-critical*: A system whose failure could endanger people's safety. Examples include control systems of nuclear plants or control systems in civil aircraft;

— *Mission-critical*: A system whose failure could endanger its mission and, thus, business operations. Examples include navigational systems of unmanned spacecrafts or bank management systems.

In this work, we focus mostly on safety-critical systems; the targeted fields of the company KRONO-SAFE include mainly avionics and automotive industries. However, most techniques described in this work could be applied to both classes.

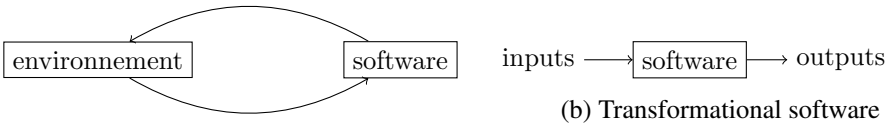
Real-time and reactive software In safety-critical real-time systems, timing is part of the system's functionnality; those systems have to respect a set of strict timing constraints to guarantee the overall safety. [KS22] defines real-time systems as follows:

∗. Institute of Electical and Electronics Engineers

« *A real-time computer system is a computer system where the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when these results are produced.* » (H. Kopetz)

However, real-time systems are not traditional sequential software programs. A typical traditional program computes output value from inputs values, and then stops as shown in 2.1b; they are often called *Transformational* [HP84]. However, real-time systems — and more generally *reactive* systems — continuously react to the environment by a sequence of reactions similar to Figure 2.1b as shown in Figure 2.1a; reactive systems, thus, react at a pace dictated by their environment [HP84]. Reactive systems call for *concurrency*; they are often composed of various communicating parallel components that react with their environment at different rates. Additionally, as reactive systems are often used in a critical context, they should be highly predictable, and thus call for *determinism*. A system is said to be deterministic if the same sequence of inputs always produces the same sequence of outputs [Ber02]. Hence, as advocated in [Lee09], real-time and reactive software should consider time throughout the software cycle as functionality constraints rather than quality of service measurements. To do that, one should consider languages and formalisms with time as a first class citizen ensuring system's predictability.

The control system of an aircraft engine (called FADEC) is an example in which the control accuracy — and thus the whole system's safety — directly depends on temporal requirements dictated by the environment. Not respecting those requirements can lead to an engine failure, which could in turn jeopardize the system's integrity (here, the aircraft) and therefore the passengers safety. A FADEC is a very classical safety-critical real-time system and is one of the use-cases of the ASTERIOS technology, developed by KRONO-SAFE.



(a) Reactive software

(b) Transformational software

Figure 2.1 – Software class

Formal Design for Real-Time Reactive Systems As systems are becoming increasingly complex, traditional techniques face their limit: manual coding of functional source code may insert bugs, and test-based verification could fail to detect bugs due to the incompleteness of testing. Moreover, the later the bugs are detected the more expensive are the consequences. To overcome those issues, *formal methods* provide mathematically rigorous techniques for various activities of the software life cycle. We give below the definition of Clarke [CW96]:

« *Formal methods are mathematically based languages, techniques, and tools for specifying and verifying systems.* » (Clarke)

In real-time systems, the *Multiform Logical Time* concept [AMDS07] (sharing a lot of similarities with the Tagged signal model [LSV98]) mandates to deal with a formal abstraction of time through *logical* timing constraints dictated by high-level requirements and thus, independent from

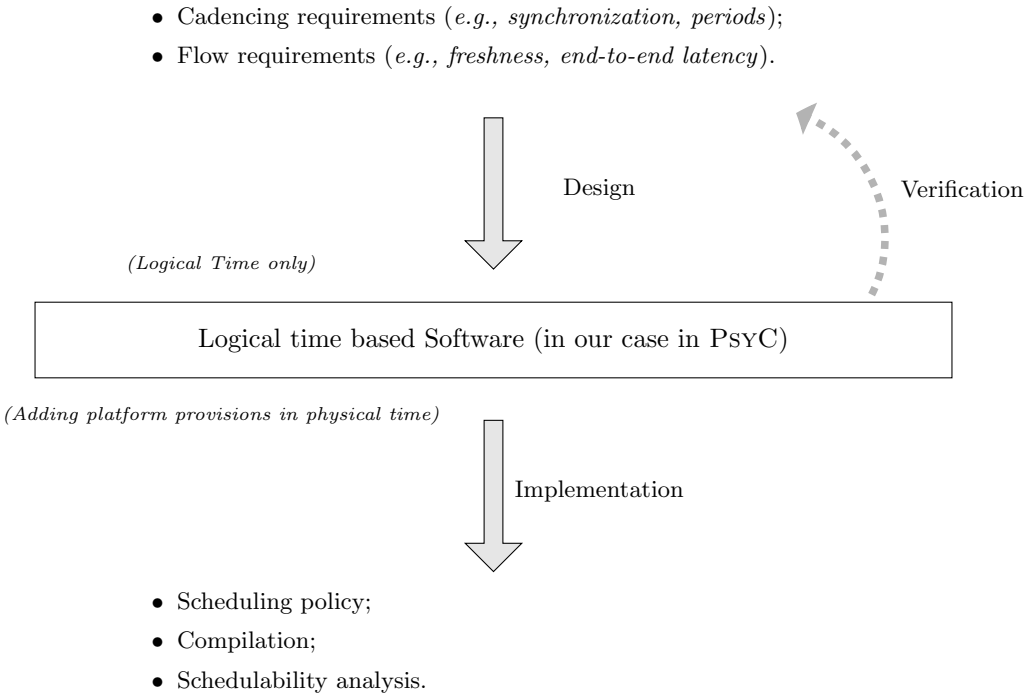


Figure 2.2 – Logical time application design methodology

a specific embedded platform. It's then the role of the implementation to map those logical timing constraints to physical time. Hence, software development is often divided into two main steps as shown in Figure 2.2, similarly to the V cycle or the avionic safety standard software process (DO-178C):

— The *design phase* allows for the design of logical timing constraints providing a deterministic timing behavior to be respected by the implementation.

— Then, the *implementation phase* allows logical timing constraints to be mapped to physical time based on platform timing provisions (e.g., estimation of worst case execution time) and real-time scheduling.

Logical timing constraints rely on *logical clocks* to express time; they denote sequences of instants either based on external events or on specific instants of physical time. Similarly to [Lam78], logical clocks are used to replace physical dates by logical sequencing. Real-time systems often rely on multiple logical clocks to express system components reacting on different paces. They are two cases of multi-clock systems:

— All *logical clocks* are derived (e.g., via periodic refinement) from a unique global *logical clock*, often mapped to a hardware timer during implementation;

— *Logical clocks* rely on multiple sources of logical time which could be independent or partially controlled.

In practice, most industrial systems fall in the first category due to safety considerations. Nonetheless, some examples still need multiple source clocks. For example, an automotive powertrain control is a typical system which relies on two time bases: a source clock based on the crankshaft

position to trigger computations on specific physical state of the engine and a source clock based on a hardware timer to define timing constraints such as initialization time or deadlines [CDO+14].

In this thesis, we focus only on the *design* phase as the *implementation* phase has been heavily covered by previous work from the CEA [LDAVN10]. The *design phase* can be a particularly challenging task due to the number of requirements to satisfy. In this work, we particularly focus on *temporal requirements* specifying either constraints among the rates of multiple components (e.g., synchronizations) or among the propagation of some data across multiple components (e.g., end-to-end latency). Multiple techniques could be used then to ensure that logical timing constraints are compliant with high-level requirements. Among them, formal verification — and more precisely model-checking — is a generic and automatic technique to provide formal guarantees for the verification process. It has been heavily studied and used in formalisms based on logical time in academy [HLR94a] and industry [BVWW09].

Industrial Considerations Formal methods are becoming a trend for safety-critical systems in various industrial domains. In railway, the Meteor project has been designed using the B method, a formal specification language based on proven refinements [AHC96]. In avionics, synchronous reactive formalisms such as SCADE [CPP17] have been used successively in large scale avionics systems and provide formal verification techniques — mainly model-checking — that have been studied for realistic avionics systems [BVWW09].

Safety-critical systems are usually subject to safety standards defining legal and regulatory requirements to specify, design, develop and verify systems at various stages. For example, DO-178C [RDE11] focuses on avionics software, ISO-26262 [Org18] on automotive systems and IEC-62304 [Com06] on medical software. As an illustration, Figure 2.3 shows the typical software process imposed by the DO-178C: the specification process typically produces a set of high-level requirements, then the design process produces both the software architecture and a set of low-level requirements, and finally the implementation and integration process produce respectively the source code and the executable object code. Verification activities are performed at each level, ensuring compliance with upper levels; this is often done with code inspections and test campaigns using strict coverage criteria.

Nonetheless, safety standards are being adapted to the use of formal methods techniques and languages:

1. In the avionics domain, DO-178C has an appendix, named DO-333, defining guidelines to adapt DO-178C objectives to the use of formal methods techniques and tools;
2. In the automotive domain, ISO-26262, advocates for the use of formal methods for the verification process of the most critical systems (ASIL-C and D);
3. In the industrial domain, IEC-61056, highly recommends the use of formal methods for the most critical systems (SIL-4).

In the particular case of the avionics domain – which is our main focus in this thesis due to its industrial context – NASA have worked on a detailed case study providing guidance on how to apply formal methods at various steps of the software process using the DO-333 [CM14]. In this thesis, we focus on model-checking based techniques applied to formalisms based on logical

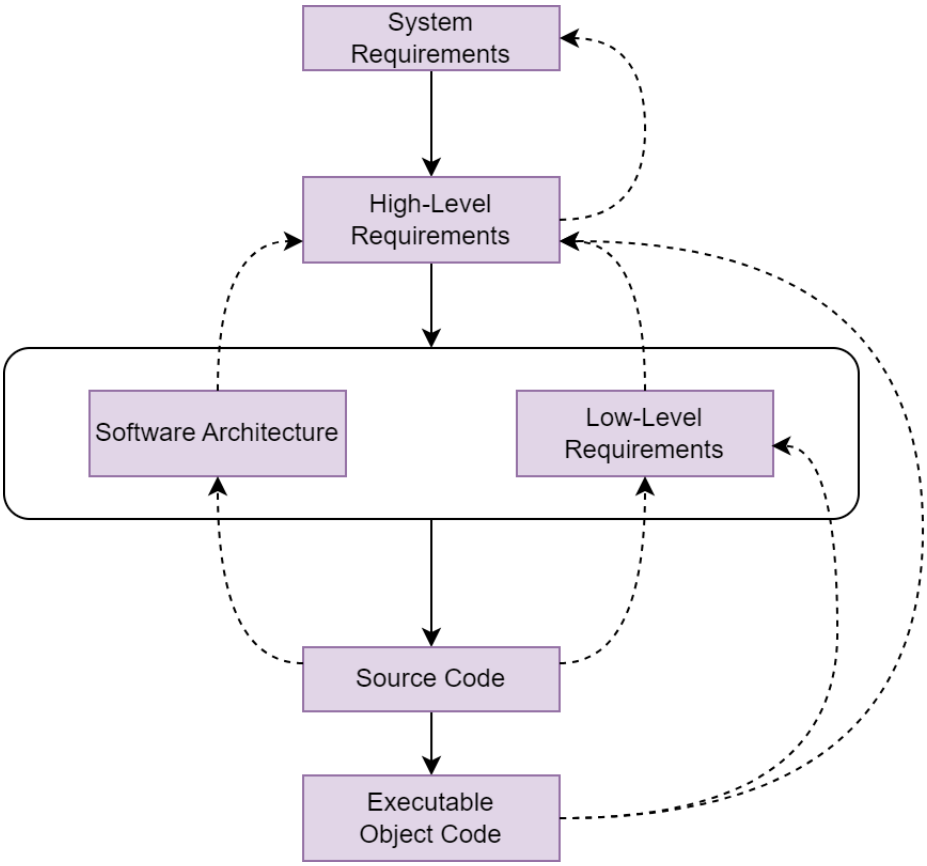


Figure 2.3 – DO-178 Software Process

time to help the design process and its verification. The following chapter details some of these formalisms.

CHAPTER 3

Languages and Models for Real-Time Systems

In this chapter, we give an overview of different languages and models used for the modelisation of real-time systems at various stages of the software life cycle. We first describe in real-time scheduling models that are typically used during the integration process and in real-time operating systems. We then give an overview of various languages and programming models dedicated to the modelisation of real-time constraints and typically used during the design process. We finally describe formalisms used to formally define high-level temporal requirements that are often used in a formal verification process.

3.1 Real-time scheduling models

A traditional way to model real-time systems is to consider a finite set of concurrent *tasks* (usually periodic), whose execution is dictated by a *scheduler* which decides which task is executed next at each invocation. In real-time systems, tasks are subject to strict real-time constraints

coming from high-level requirements. However, in embedded real-time systems, tasks also have time provisions dictated by the platform's computing capabilities. The real-time scheduling field is therefore interested in tasks models, and the choice of scheduling policies as well as schedulability (or feasibility) analyses to ensure that platform's provisions could not invalidate real-time task constraints. The reader might want to read [LS16] which gives a good introduction to real-time scheduling models.

3.1.1 Task models

Task constraints We usually consider a finite set of tasks $\mathcal{T}$ in which each task $\tau \in \mathcal{T}$ is executed by a sequence of task instance $\tau_1, \tau_2, \ldots$. Tasks are usually defined to be either *periodic* or *sporadic* depending on the regularity of task instances. Formally, we define r_i as being the *activation instant* of a task instance τ_i.

In general, we define a periodic task by the activation function $r_i = O + (i-1) \times T$ where O and T are respectively the offset and the period of the task. Sporadic tasks do not have a specific activation function for r_i, however, activation should usually satisfy $r_{i+1} \geq r_i + T_{min}$ where T_{min} is the minimal interval between two successive task activations.

The second typical temporal constraint of a task is the *deadline* of a task written δ_i. It corresponds to the instant on which the task instance τ_i has to be terminated. In a real-time system with hard critical constraints, a task instance τ_i should fit between r_i and δ_i, otherwise it is considered as an error. Considering that s_i and f_i are respectively the execution start and termination of τ_i, then τ_i has to satisfy $r_i \leq s_i < f_i < \delta_i$. All those constraints are shown in Figure 3.1.

Task provisions The constraints given above are induced by temporal requirements. However, to compute the instants on which computations start and terminate, s_i and f_i, dictated by a scheduling policy, it is necessary to give an estimate of the task execution time, which we write e_i. In a preemptive model, e_i is not necessarily equal to $f_i - s_i$; execution can be broken down into several sub-windows in which the sum of the execution time should be e_i. In traditional scheduling models, e_i is often considered to be constant. Today, however, it is commonly accepted that this time is variable due to multiple factors: program control-flow, cache management ... It is usually defined by a *Best* and *Worst Case Execution Time* (respectively BCET and WCET). e_i is thus often an estimation of the WCET.

3.1.2 Scheduling policy and schedulability analysis

A scheduling policy elects which task should run at each invocation according to a desired objective. However, a scheduling policy may fail to satisfy some of the time constraints — typically the deadline — due to an incompatibility between real-time constraints and target provisions. In order to avoid this issue, the use of so-called schedulability analysis is needed to ensure that time provisions satisfy task constraints. This analysis depends directly on the scheduling policy used, and is usually defined as a criterion demonstrating that $f_i < \delta_i$ for any execution.

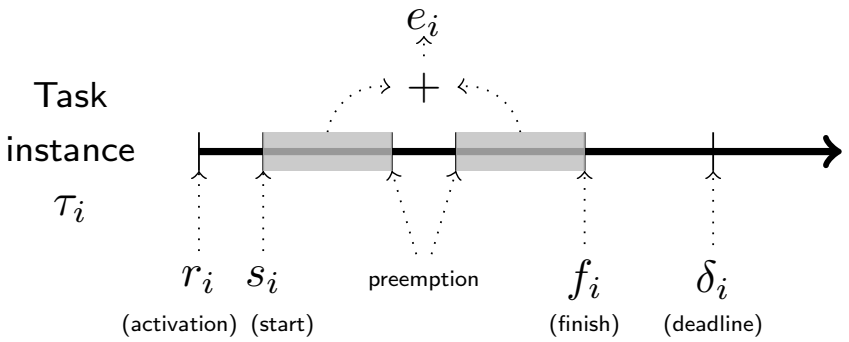


Figure 3.1 – Task model for real-time scheduling

To illustrate the concept of schedulability analysis, let's take the EDF scheduling algorithm (i.e., for *Earliest Deadline First*). At each scheduling decision (e.g., at the end of a task's execution), the next task to be executed is the one with the earliest deadline. In the case of a set of periodic tasks and with deadline $\delta = T$, the schedulability analysis is defined as follows for a set of tasks $\mathcal{T} = \{\tau^1, \tau^2, \dots \tau^n\}$:

$$\sum_{j=1}^{n} \frac{e^j}{T^j} \leq 1 \tag{3.1}$$

This criterion, introduced in [XP90], ensures that the computational load is less than 100%. Hence, if the condition is met, real-time constraints are guaranteed to hold.

In practice, real-time scheduling models are rather low-level, and do not allow reasoning from a programmatic point of view on real-time software. This is because a scheduling model is not a programming model: there is no communication model nor structured control. Additionally, response times are difficult to predict ($f_i - r_i$ is variable at runtime) and can be a cause of non-determinism. Nonetheless, those models are very useful to study the capacity of a hardware platform to satisfy real-time task constraints. In this work, we focus on real-time constraints, that we will call *logical constraints*. We make the assumption that WCET estimation and schedulability analysis exist for the various languages described in this work.

3.2 Real-time languages and programming models

One of the greatest results of the mid-20th century in computer science is the Church-Turing thesis, which shows the equivalence of several models of computation, such as the lambda-calculus or the Turing machine. However, those models are *sequential*. As discussed in the previous chapter, reactive real-time systems need models of computations relying on both *concurrency* and *timing constraints*. This section gives an overview of the main languages and programming models highlighting those aspects that are all based on logical abstraction of time. For each, we also discuss the notion of *determinism*, which is often mandatory in the context of safety-critical systems.

3.2.1 Process algebra

Process algebras are specific languages defined in the context of concurrency theory defined by Milner [Mil82]. These languages enable us to describe formally interactions, synchronizations and communications between process — or agents. Most algebras define the following elements [Mil82] [Hoa78]:

— Inter-process communication mechanisms;

— Sequential composition;

— Parallel composition which is generally asynchronous;

— And finally, replication mechanisms such as loops or recursion.

One of the most famous examples of process algebra is the modelisation of a Coffee Machine (CM). We give here its behavior using a simplified syntax of CCS:

$$\mathbf{CM} = \mathit{coin}.(\mathit{coffee} + \mathit{tea}).\mathit{brewing}.\mathbf{CM}$$

This machine is defined by a recursive sequence of actions. We use two operators "+" which defines choice and "." which defines sequential composition. The behavior is then the following: first the machine waits for a coin, then it proceeds to either brewing coffee or tea depending on the one selected before returning to its initial state. Let's now define the typical behavior of users who either try to get a coffee for free if it's possible or give a coin before talking.

$$\mathbf{USER} = (\mathit{coffee} + \mathit{coin}.\mathit{coffee}).\mathit{talk}.\mathbf{USER}$$

Using the parallel composition "||", we can now define the interactions between the two process, **USER** and **CM**. This illustrates the mechanism of handshaking which synchronizes corresponding actions.

$$(\mathbf{CM} \parallel \mathbf{USER}) = \mathit{coin}.\mathit{coffee}.(\mathit{brewing} \parallel \mathit{talking}).(\ \mathbf{CM} \parallel \ \mathbf{USER})$$

However, generally, process algebras allow defining concurrency but at the price of determinism. Due to the asynchronous composition of process, the algebras cannot guarantee that the resulting process is deterministic. Furthermore, process algebras are not adequate to specify temporally predictable requirements. Nonetheless, multiple approaches extend process algebras to support determinism or predictable temporality: SCCS [Mil83], Meije [AB84] or CoReA [Bon94] to name a few.

3.2.2 Synchronous-Reactive languages

To combine concurrency and determinism, dedicated languages based on the Synchronous-Reactive (SR) programming model have been introduced [BB91]. In a SR language, time is abstracted by a sequence of instants on which computations (often called *reactions*) are triggered. Those sequences of instants are called *logical clocks* and this time abstraction, *logical time*. Multiple logical clocks can exist in SR programs; however, the semantics of SR languages usually impose to expand each behavior on a unique base clock. The SR programming model is based on two main principles:

— *Synchrony*: the output of a computation is available to all other computations triggered on the same instant; hence, communications are *synchronous*. The *synchronous hypothesis* states that all computations triggered on some instant should terminate before the next one.

— *Causality*: the output of a computation must be available *before* any other computations could use it on the same instant; hence, communications are *causal*.

Based on both synchrony and causality properties, SR languages provide support for modeling complex concurrent programs while ensuring determinism. SR programs focuses on temporal constraints — based on logical time — while physical time mapping is left to the implementation. Figure 3.2 shows a typical execution of a synchronous-reactive program

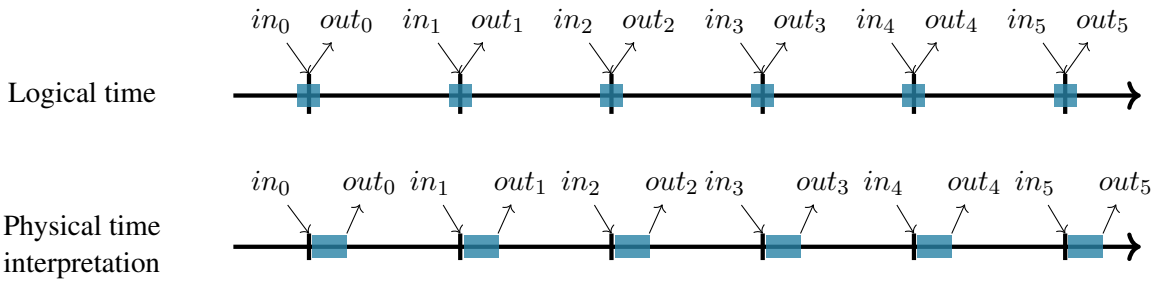


Figure 3.2 – Typical execution of a synchronous-reactive program in logical and physical time

Imperative approach: ESTEREL

ESTEREL [Ber02] is an imperative synchronous-reactive language. Contrary to data-flow languages described in the next section, ESTEREL code is control-flow oriented. Communications in ESTEREL are carried out using broadcasted signals. These signals are logically instantaneous — they are available on the same instant on which they are produced — and are broadcasted to all the program. Signals can be inputs, outputs of a component as well as internal. They are defined by a *status* indicating the presence or absence of the signal on each instant and by an optional *value*. ESTEREL provides two main mechanisms: imperative concurrency allowing multiple instruction pointers in a single program; and synchronous preemption aborting an execution. Statements are divided into two classes:

— Statements that don't take time, which are *logically instantaneous* like `emit s` which sends a signal s or `present s then P else Q` which executes P or Q depending on the presence of signal s (note that even if reacting to s does not take time, P or Q may take time);

— And statements that take time — one or multiple instants — like `pause` which waits for the next instant or `await S` which awaits an instant in which signal s is present.

The example in Listing 3.1 illustrates the main features of ESTEREL. This program awaits signals A and B on its inputs — in any order — and emits O when both signals are received. Signal R allows the reset of the program. A detailed description of the language is given in Chapter 5 in the semantics definitions of PSYC.

```
1  module ABRO:
2    input A, B, R;
3    output O;
4    loop
5      [ await A || await B ];
6      emit O
7    each R
8  end module
9
```

Listing 3.1 – ESTEREL ABRO example

Dataflow approach: LUSTRE, SIGNAL

Imperative SR languages are however not adapted for control systems. In this domain, computations are usually described by block-diagrams in which blocks correspond to operators and wires to dataflow dependencies.

Lustre LUSTRE [CPHP87] is a declarative dataflow SR language used to model such control systems. It defines a set of mutually recursive equations operating on data streams. A stream is a sequence of values based on a (logical) clock, which specifies the presence or absence of values on each instant of a global base clock. LUSTRE defines two types of equation operators:

— Traditional operators which depend on the value's type; arithmetic operators for integers or boolean operators for boolean values.

— Synchronous operators which allow the manipulation of stream history; `pre(X)` delayes values of X by one instant and `X -> Y` is equal to X on the first instant, then Y. LUSTRE also defines sampling/merging operators that we omit here.

```
1  node Edge(X : bool);
2  returns (Y : bool);
3  let
4    Y = False -> X and not (pre X) ;
5  tel
6
```

Listing 3.2 – LUSTRE rising edge example

Listing 3.2 shows the most classical LUSTRE example which defines a rising edge detector. Y is set to true when X is true on the current instant but was not on the previous instant. Listing 3.3 shows how the rising edge detector can be used to define a similar program to the ABRO ESTEREL example. Basically, O is the rising edge of the conjunction of A and B which are held while R is not present. Table 3.3 gives a possible execution of the ABRO program.

```
1   node Hold(X, R : bool);
2   returns (Y : bool);
3   let
4     Y = if R then False else X or (False -> pre Y);
5   tel
6
7   node Abro(A, B, R : bool);
8   returns (O : bool);
9   var seenA, seenB : bool;
10  let
11    O = Edge(seenA and seenB);
12    seenA = Hold(A, R);
13    seenB = Hold(B, R);
14  tel
15
```

Listing 3.3 – LUSTRE ABRO example

A	False	**True**	False	False	False	False	**True**	False
B	False	False	False	False	False	**True**	False	False
R	False	False	False	**True**	False	False	False	False
seenA	False	**True**	**True**	False	False	False	**True**	**True**
seenB	False	False	False	False	False	**True**	**True**	**True**
pre(seenA)	nil	False	**True**	**True**	False	False	False	**True**
pre(seenB)	nil	False	False	False	False	False	**True**	**True**
O	False	False	False	False	False	False	**True**	False

Figure 3.3 – Possible execution of the ABRO example in LUSTRE

Signal SIGNAL [BLGJ91] is also a dataflow SR language, really similar to LUSTRE. However, contrary to LUSTRE, SIGNAL can deal with the clocks of its streams. These clocks are typically only partially defined by relations (instead of deterministic operators). Then, it is the job of the compiler to synthesize these clocks while respecting various relation constraints. Hence, SIGNAL allows the design of programs containing multiple — partially-related — clocks; hence, it is often considered as a *polychronous* language.

However, while SR languages allow mixing both concurrency and determinism, their semantics usually impose to expand reactions on a common global base clock. Consequently, even "long" computations (i.e., whose physical duration lasts multiple instants of the base clock) are

semantically projected on the base clock and have to terminate before the next instant (i.e., synchronous hypothesis). Hence, programs composed of various components with multiple time scale (both logically and physically) are often impossible to compile using traditional SR compilation techniques.

3.2.3 Logical Execution Time languages

The *Logical Execution Time* (LET) [KS12] paradigm takes a different approach to specify temporal constraints; instead of specifying the computations by a reaction triggered on a clock tick, the execution time of those computations is abstracted such that a computation always behaves as if it lasts exactly a constant and fixed logical execution time. This defines an abstract time window which contains the actual computation. However, to guarantee determinism, communication are done only on the boundaries of the LET time interval; inputs are read automatically at the start of the interval and outputs are produced automatically at the end of the interval. Figure 3.4 illustrates how LET abstracts computations: the upper part is the logical view interpreted by the model and the lower part is the physical view of a typical real-time implementation. The yellow symbol shows that in LET, execution can be typically preempted by other concurrent tasks while maintaining the communication behavior. Hence, contrary to SR, the semantics of LET allow more execution time variability for computations as (logical) computation durations are part of the semantics. This allows classical scheduling algorithms — such as EDF — to be used in the implementation.

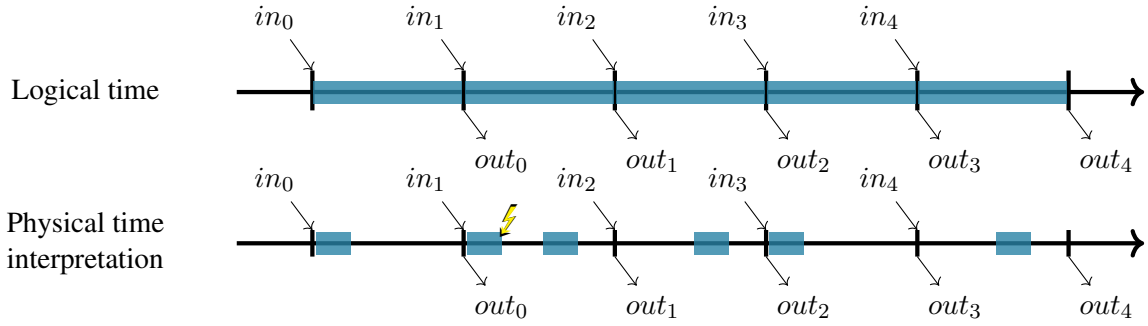


Figure 3.4 – Typical execution of a logical execution time program in logical and physical time

GIOTTO is a time-triggered language implementing the logical execution time paradigm. It allows the definition of multiple modules parametrized by their periods which are, in turn, composed of tasks defined by their frequency. Tasks also have deterministic communication channels implementing LET communication to communicate together. Even if GIOTTO permits to describe mode and mode transitions, tasks are basically strictly periodic which limits considerably its expressivity. Additionally, all constraints are expressed respectively to physical time; there is no clear distinction between logical time and physical time, nor discrete time basis as logical clocks. TIMING DEFINITION LANGUAGE (TDL) is another language implementing LET extending GIOTTO concepts with a more advanced module and mode system. However, TDL suffers from the same issues as GIOTTO.

3.2.4 Other languages based on logical time

Traditional Logical Execution Time languages suffer from expressivity limitations; they are often limited to multi-periodic behaviors with limited mode handling, but more importantly, timing durations are based on a unique chronometrical time base. This section describes languages mixing aspects from both synchronous-reactive and logical execution time languages; they all rely on multiform logical time while also allowing logical durations of computations to be expressed.

The xGIOTTO language

xGIOTTO is a language implementing LET (as GIOTTO) but supporting events to specify synchronizations (i.e., to specify LET boundaries) [GHKS04]. In its principle, this language is closer to SR languages as logical time could express both timing or events. However, inter-arrival time assumptions of events have to be encoded implicitly in the program using a rather complex mechanism called *event scoping* [KS12]. Event scopes define the computation to be taken in a given time span — specified by chronometrical time or events — and can be nested. Hence, the logical duration of a computation has to be refined to a chronometrical time basis for inter-arrival time to be considered. Hence, in this thesis, we consider that xGIOTTO computations can be triggered by events, but their deadlines are still specified by a chronometrical time basis, as in traditional LET.

The example below shows part of an automotive engine controller inspired by [GJSH05] based on two time bases: crankshaft and milliseconds. The controller is started on event $start$ and immediately stopped — due to the ASAP policy — on event $stop$. Two concurrent tasks are shown: `time` (executing function $task1$) which is triggered each millisecond and lasts one millisecond, and `cycle` (executing function $task2$) which is triggered each crankshaft event and lasts two milliseconds.

```
1   react R1() { release task1(); }
2   react R2() { release task2(); }
3
4   react time() {
5     whenever [ms] react R1() until [ms];
6   }
7   react cycle() {
8     whenever [crank] react R2() until [2ms];
9   }
10  {
11    when [start]
12      react time() until asap [stop] ||
13      react cycle() until asap [stop] ||
14      ...
15  }
```

Listing 3.4 – xGIOTTO controller example

The PRELUDE language

PRELUDE is a multi-periodic variation of LUSTRE in which real-time primitives are added to specify the duration of the tasks. Contrary to LET languages, PRELUDE keeps the synchronous communication model from traditional synchronous languages. However, its expressivity is limited to multi-periodic tasks without mode handling. The compilation process then generates real-time tasks that could be executed by a dynamic scheduling algorithm such as EDF.

In the example below, we show a PRELUDE code with two tasks, `F` and `S`, with an input `i` and an output `o` of period 10. The task `S` has a period of 30 because `vf` has a period of 10 — the same rate as `o` — and the operator `/^` makes the `S` input stream 3 times slower. Similarly, task `F` has a period of 10 because `vs` has a period of 30 and the operator `*^` makes the input stream 3 times faster*.

```
1  node sampling(i: rate(10, 0)) returns (o)
2    var vf, vs;
3  let
4    (o, vf) = F(i, (0 fby vs) *^ 3);
5    vs = S(vf /^ 3);
6  tel
```

Listing 3.5 – PRELUDE sampler example

The LINGUA FRANCA language

LINGUA FRANCA (or LF) [Loh20] is a coordination language based on a logical time framework called the Tagged Signal Model [LSV98]. The components of the language are called *reactors* and are essentially composed of *input ports*, *output ports*, and *reactions* that reacts to input events — called *tags* — coming from *input ports*. An application is defined by multiple *reactors* composed concurrently and by their port connections.

We show in Listing 3.6 a very simple pipeline example using LINGUA FRANCA taken from [LL22]. The *reactors* comprise a sensor task, two computing tasks and an actuator task connected in this order. The rate of the reactors are given by a logical clock (called *timer* in LF) triggering the `sensor` reaction each `period`, by default 10 milliseconds. In the semantics, by default, connected tasks are synchronous; when some reaction receives a message on a logical instant t, then it can produce an output message on the same logical instant t.

*. the `fby` operator is equivalent to `-> pre` in traditional LUSTRE, and is used for initialization

```
reactor Sensor(period:time(10 ms)) {
  output out:int;
  timer t(0, period);
  reaction(t) -> out {=
    ...
  =}
}
reactor Task1 {
  input in:int;
  output out:int;
  reaction(in) -> out {=
    ...
  =}
}
...
main reactor(period:time(10 ms)) {
  /* we omit instantiation part */
  sensor.in -> task1.in;
  task1.out -> task2.in;
  task2.out -> actuator.out;
}

```

Listing 3.6 – LINGUA FRANCA pipeline example

As explained in [LL22], LF can model LET behaviors using delayed connections between ports as shown in Listing 3.7 below; reactors `task1` and `task2` are assigned a LET duration of 10 milliseconds using the `after` construction. Note that, unlike the traditional LET of GIOTTO or TDL, the semantics of LF is fully compatible with the synchronous-reactive one. Hence, reactors can be triggered by any event, or more generally, logical clocks. It is not clear however if the `after` construction allows the LET duration to be defined similarly, using a clock or event synchronization.

```
...
main reactor(period:time(10 ms)) {
  ...
  task1.out -> task2.in after period;
  task2.out -> actuator.out after period;
}

```

Listing 3.7 – LINGUA FRANCA LET pipeline example

The PSYC language

PSYC is an industrial language developed by Krono-Safe [KS23]. It is a language inspired by process algebras, synchronous-reactive languages, and with a principle very similar to Logical Execution Time. PSYC actually stands for *Parallel SYnchronous C*, thus highlighting the two main

principles of the language: synchronous and parallelism (both in the model and in the implementation). In this thesis, we define PSYC based on an intermediate formalism that inherits from both synchronous-reactive and logical execution time approaches, called synchronous Logical Execution Time (sLET). In sLET, both the triggering instants and the duration of the computations are expressed using logical clocks. Hence, whereas in traditional LET a timing window is specified by a triggering instant and a chronometrical duration, in sLET, a timing window is specified by two successive synchronous instants that we call *synchronization points* (or *instants*).

Thus, compared to ESTEREL, sLET (and PSYC) can specify the logical duration of computations, but inherit the delayed communication model from LET. While PRELUDE can also specify logical durations and keep the synchronous communication model, it is limited to multi-periodic programs without mode handling, which is not the case of sLET and PSYC. Compared to traditional LET languages such as GIOTTO, sLET uses logical clock instants to specify interval boundaries instead of a unique chronometrical time basis. Both XGIOTTO and LINGUA FRANCA can rely on logical clock instants (respectively *events* and *tags*) to specify triggering instants and can specify the logical durations of the computations. However, logical durations are expressed by a chronometrical time basis in both cases; XGIOTTO relies on event scoping to implicitly encode inter-arrival times [KS12] and LINGUA FRANCA uses chronometrical time to express delays between reactors. The table 3.5 gives an overview of the main differences.

A PSYC application is composed of a set of concurrent *agents* (or processes) communicating through data-flow channels, called temporal variables. Agents are composed of C code and a dedicated temporal synchronization primitive: the `advance` instruction. This instruction is used to set a synchronization instant, acting as a time barrier, between *elementary actions*, which are timing intervals between two successive synchronizations. According to LET communication semantics, communications only take place atomically at these synchronisation instants. On the other hand, unlike LET, these instants are modeled relatively to logical clocks. In the following example, the end of the first *elementary action* containing f, corresponds to a synchronization of 2 ticks of the clock C. At this point, y is made visible to the other agents, and g can begin execution with the x input captured at this same instant.

```
y = f(); advance 2 with C; g(x);
```

We give an illustration of a PSYC task in Listing 3.8 describing an agent performing conditional filtering (i.e., typically according to a system mode). First, the agent starts on the first tick of clock $c1s$, hence, 1000 milliseconds. Then, when the condition is true, the function `filter` has a logical duration of 3 ticks of clock $c100ms$, which is equivalent to 300 milliseconds. It is then followed by an empty *elementary action* terminating on the next $c1s$ tick; this is used to resynchronize on the task period. Indeed, when the condition is false, the task only performs an empty *elementary action* terminating on the next $c1s$ tick.

Like LINGUA FRANCA, PSYC is a language based on logical time which inherits both from LET and Synchronous-Reactive languages. Like in LF, it generalizes LET using a semantics based on the synchronous-reactive approach. However, while LINGUA FRANCA is a general purpose coordination language, PSYC is an industrial language dedicated to safety-critical real-time systems.

```
1     clock c100ms = 100 * ms;
2     clock c1s = 1000 * ms;
3
4     agent Filter(starttime 1 with c1s) {
5       consult raw_data, mode /* inputs */;
6       display filtered_data /* outputs */;
7       body start {
8         if (mode == FILTER) {
9           filtered_data = Filter(raw_data);
10          advance 3 with c100ms;
11        }
12        advance 1 with c1s;
13      }
14    }
15
```

Listing 3.8 – PSYC filter example

Hence, in this thesis, we focus on PSYC for various reasons. First, the PSYC language is developed by the company KRONO-SAFE who is funding this work. But most importantly, PSYC allow us to focus on a language that inherits from multiple logical time approaches while having major industrial projects using it (e.g., from the SAFRAN group). A detailed description of PSYC is given in Chapter 4.

Language	Triggering instants	Logical duration	Communication model
ESTEREL	logical clocks	synchronous	synchronous
PRELUDE	logical clocks	chronometrical	synchronous
GIOTTO	chronometrical	chronometrical	delayed
XGIOTTO	logical clocks (events)	chronometrical (event-scoping)	delayed
LINGUA FRANCA	logical clocks (tags)	chronometrical	both
PSYC	logical clocks	logical clocks	delayed

Figure 3.5 – Comparison of some languages based on logical time

3.3 Temporal Requirements Formalisms

To define temporal requirements — also called *properties* — on the languages defined in the previous sections, we also need specific specification languages — actually closer to logics — allowing for a specific set of temporal behaviors. Ultimately, those specification languages are often used in formal verification to model-check the requirements on the program.

3.3.1 Temporal logic

Formally reasoning about time goes back to Aristotle to interpret sentences such as "Tomorrow, there will be a sea fight" for philosophical purposes [Var08]. In the 19th century, Pierce described the need to extend classical logic with temporal operators:

« *Time has usually been considered by logicians to be what is called "extra-logical" matter. I have never shared this opinion. But I have thought that logic had not yet reached the state of development at which the introduction of temporal modifications of its forms would not result in great confusion; and I am much of that way of thinking yet. Pierce* » (C.S. Pierce)

However, it wasn't until the work of Pnueli in the late 1970s that a logic with temporal operators — called *temporal logic* — had been introduced. In particular, Pnueli proposes a *linear* temporal logic with future operator, called *LTL*, which can be defined by 4 main operators extending the traditional propositional logic [BK08]:

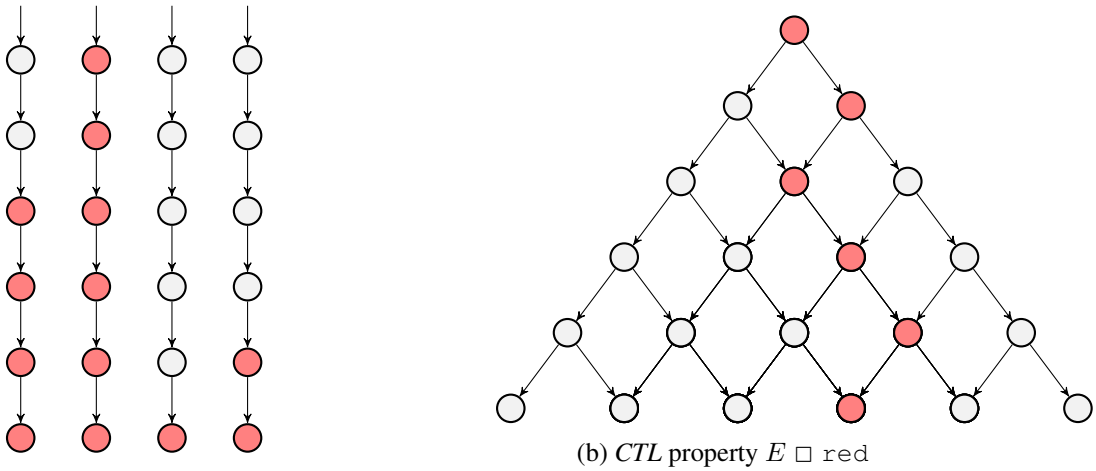
— $\bigcirc\ p$ (or *Next* p), holds if p holds on the next instant;
— $U\ p\ q$ (or *Until* $p\ q$), holds if p holds on all instants until an instant on which q holds;
— $\Box\ p$ (or *Globally* p), holds if p holds on all the following instants;
— $\Diamond\ p$ (or *Finally* p), holds if there exists a future instant on which p holds.

Additionally, properties are often classified into two main classes:

— *Safety* properties state that "something bad should not happen." Typically, we want to prevent a bad condition to happen often modeled as an invariant that should hold on all reachable states of the program.
— *Liveness* properties state that "something good will happen." Contrary to safety properties, liveness properties guarantee that a good condition will hold sometime in the future.

The vision of time described by LTL is, by definition, linear; to satisfy an LTL property, a program has to satisfy the property on each of its execution paths. Another approach is to consider path quantification as in the Computational Tree Logic (CTL) [BK08]. CTL has two operators dedicated to path quantification: $A\ p$ and $E\ p$ to consider respectively universal and existential quantification for property p on execution paths of the program. Examples of LTL and CTL properties are shown in Figure 3.6. Additionally, MTL is another classical variation of LTL dedicated to *quantitative* temporal properties [Koy90]; classical LTL operators are extended with intervals that specify when the operator applies. As an example, $\Diamond_{[0;10]}\ p$ specifies that p should be reachable in the next 10 instants.

In our case, we want to focus on *linear time* properties (in opposition to the *branching* properties of CTL) as we want them to hold for all execution paths and, more specifically, to safety properties to prevent "bad things to happen" due to the safety-critical context. We also consider *bounded liveness* properties for latencies, but this is actually a special case of *safety*. However, in practice, programs based on logical time often rely on different timelines due to the use of multiple external clocks or tasks with different timing constraints. Temporal logic does not capture well properties that rely on multiple time scales — or multiple logical clocks.



(a) *LTL* property $\Diamond\Box$ `red`

(b) *CTL* property $E\ \Box$ `red`

Figure 3.6 – LTL and CTL property examples

3.3.2 Clock Constraint Specification Language

CCSL is a specification language dedicated to the modelisation of temporal requirements defined as a set of constraints between logical clocks [And09]. A logical clock can be any sequence of events, similarly to the one considered in temporal logic. In this work, we choose CCSL to capture requirements as relations between the timing of multiple tasks because in PSYC, agents usually have different rates or time scales.

CCSL defines four main constraints:

— $c_1 \boxed{\subset} c_2$ which denotes that c_1 should be a subclock of c_2; thus, on each instant, if c_1 holds then c_2 should hold.

— $c_1 \boxed{\prec} c_2$ which denotes that c_1 strictly precedes c_2; thus, the n^{th} tick of c_1 strictly precedes the n^{th} tick of c_2.

— $c_1 \boxed{\preccurlyeq} c_2$ which denotes that c_1 precedes c_2; thus, the behavior is similar to the constraint above except that tick coincidence is allowed.

— $c_1 \boxed{\#} c_2$ which denotes exclusion between c_1 and c_2; thus, c_1 cannot tick on the same instants as c_2.

Additionally, each clock can be refined using CCSL expressions. We omit them here as an extensive description of CCSL is given in Chapter 6.

As a tiny illustration, consider the following set of constraints: $b \boxed{\subset} a$, $c \boxed{\subset} a$, $b \boxed{\#} c$, $b \boxed{\prec} c$, $c \boxed{\prec} b\ \$\ 1$. The expression $\$\ n$ is a delay expression; it specifies that a clock is delayed with itself by n ticks. The chronogram in Figure 3.7 shows a possible execution of clocks a, b and c that fulfills all CCSL constraints. Graphically, red dashed lines denote simultaneity and blue arrows denote strict causality. In a more realistic system, this example can typically represent a requirement specifying a communication pattern between a producer (clock b) and a consumer (clock c) based on a common clock (clock a).

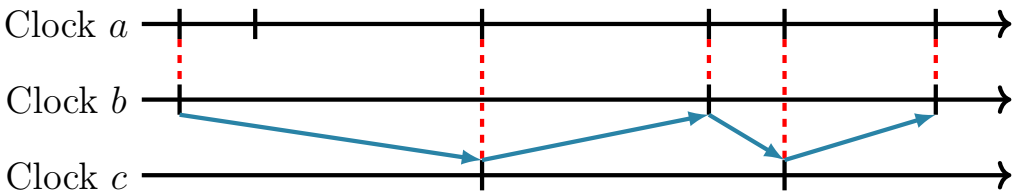


Figure 3.7 – Chronogram of a possible scheduling of the CCSL example

From a theoretical aspect, CCSL and temporal logic are very close. [GMD11] shows that the *safety* subset of PSL — an industrial variant of LTL — is equivalent to a *bounded* subset of CCSL. Unbounded CCSL specification is due to the $\preccurlyeq$ operators generating counters to encode the difference between two clocks; however, most specification can be bounded by adding additional causality constraints as in the tiny example above.

3.3.3 Synchronous observers

Synchronous observers is not a specific language to model properties, but instead, an approach to model properties inside SR languages by continuously observing the program [Rus14]. This approach has been introduced in the context of SR language verification, and, in particular, for the language LUSTRE [HLR94b]. In this approach, a SR program is considered as a model containing both the program to be verified and the requirements. Observers are then specific tasks containing requirements that *observe* the program's state and raise an error if any of the requirements are not satisfied. Typically, the supported properties are linear safety properties.

Both LTL [Hal93] and CCSL [And10] can be translated into synchronous observers for safety properties; they are then used as a means to verify the requirements either dynamically — by runtime monitoring — or statically — by model-checking. In both case, the verification problem is reduced to a reachability analysis on the observer output; if an error is reachable, then the requirement is false. In Chapter 7, we show how synchronous observers can be used to model and verify CCSL properties on PSYC code.

SECOND PART

Language and Semantics

*Every **Kairos** is a **Chronos**, but not every **Chronos** is a **Kairos**.*

— Hippocrates, Praeceptiones (460-377 av.J.-C.).

CHAPTER 4

Synchronous Logical Execution Time in PSYC

This thesis focuses on the PSYC language, developed by the company KRONO-SAFE. This language is based on both Synchronous-Reactive and Logical Execution Time foundations, making it ideal for studying its semantics based on logical time as well as verification techniques, detailed

respectively in Chapters 5 and 7 of this thesis. Hence, this chapter briefly describes the industrial context and tooling of the technology — called ASTERIOS — in section 4.1, then defines in section 4.2 a formalism unifying *Logical Execution Time* and *Synchronous-Reactive* approaches — called synchronous Logical Execution Time — used to give an overview of the PSYC language in section 4.3.

4.1 The ASTERIOS Software Suite

4.1.1 Origins of the technology

KRONO-SAFE is a *spin-off* company from the CEA * created in 2011 and whose objective is the industrialization of two research projects — OASIS and PHAROS — under the name ASTERIOS. The company is now targeting domains such as avionics, railways and the automotive industry; those are all safety-critical domains requiring certified tools to the highest level of criticality. The origins of the ASTERIOS technology goes back to the end of the 1990s with the OASIS project defined as following in [ACA+96]:

« *The* OASIS *proposes rules and methods for safety-critical application engineering allowing to design and implement concurrent programs with fully deterministic behaviours, and thus to guarantee some specified properties of dependability* » (C. Aussagues and al)

At that time, the main targeted domain of OASIS was civil nuclear power. In particular, it was used to build a Qualified Display System (QDS) used in modern nuclear plant by FRAMATOME [DAL+04]. A decade later, OASIS has been adapted to the automotive domain under the name PHAROS before becoming ASTERIOS with KRONO-SAFE in 2011.

4.1.2 Industrial context

In industry, functional modeling and software integration activities are generally separated. The former is responsible for the functional behavior of a subset of the system, while the latter is responsible for the software and hardware integration of multiple functional components. Code generation of functional models has been automated for years in tools such as SCADE or SIMULINK based on synchronous reactive approaches. These tools are not, however, well suited for the integration of functional components using different time scales. Also, they do neither propose seamless tooling for on-target integration with features such as real-time scheduling nor memory protection. For all these reasons, software integration is still largely done manually in industry. ASTERIOS proposes methodology and tooling to automate software integration activity based on a parallel and synchronous language called PSYC; the toolchain is detailed in the next section.

∗. French Alternative Energies and Atomic Energy Commision

4.1.3 Tooling

The ASTERIOS toolchain is built around a language called PSYC. It is composed mainly of a compiler called PSYKO compiling PSYC applications to binaries which can be executed on an embedded target by a dedicated real-time kernel. Figure 4.1 shows how those tools interact.

The compiler, called PSYKO, takes C code corresponding to functional components (e.g., generated by SCADE) as well as PSYC tasks, and generates various binaries. Firstly, it generates a set of *Partitions* each of which containing a set of compiled PSYC tasks corresponding to the same criticality level. Secondly, the compiler generates a *Runtime* containing all the configuration entries necessary to the execution model, such as a static scheduling plan, bounded communication buffers and memory descriptors.

Both *Partitions* and the *Runtime* are executed on an embedded target along with a dedicated real-time kernel certified to the highest level of criticality for the avionics safety standard (DAL-A, DO-178C). The kernel implements the execution model with spatial and temporal isolation constraints specified in the *Runtime*. Note that this kernel is not meant to be run with any other application than the one generated by PSYKO and such applications can only be executed by the kernel.

ASTERIOS also gives the possibily to simulate the behavior of the PSYC execution model. Instead of generating binaries for an embedded target, PSYKO can generate them for the host machine. Then, those binaries can be executed with a specific simulation library which simulates the behavior of the real-time kernel. Note that in this approach, the *Partitions* and the *Runtime* are generated very similarly as if an embedded target was selected to be as faithful as possible.

Finally, one might remark that in this process, only the real-time kernel is certified for the avionics safety standard. However, the compiler could insert errors during compilation. To solve this problem, ASTERIOS also contains a validation tool called CHECKER which verifies that the compiler's outputs are compliant with the corresponding inputs. For example, it verifies that the static scheduling plan contained in the *Runtime* is correct with respect to the temporal constraints described in PSYC files. This tool is qualified using the software tool qualification considerations of the avionics safety standard (DO-330) as described in [MOT20].

4.2 A synchronous generalization of Logical Execution Time

We now describe a generalization of Logical Execution Time (LET) compatible with the Synchronous-Reactive (SR) approach, called synchronous Logical Execution Time (sLET). sLET provides a natural framework for defining the PSYC language and its semantics given respectively in section 4.3 and chapter 5.

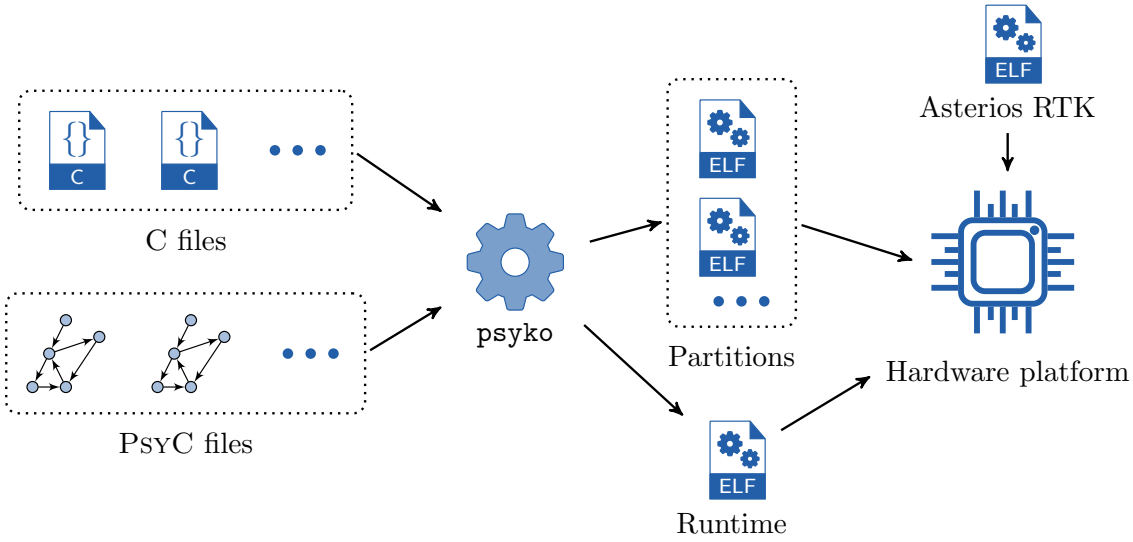


Figure 4.1 – PSYC toolchain overview

4.2.1 Overview of synchronous Logical Execution Time

The objective of synchronous LET (or sLET) is to extend LET with SR aspects. Contrary to LET, in synchronous LET, the boundaries of LET intervals are actually synchronous instants, hence, defined using logical clocks. The computations are specified similarly to the ones from LET as they are defined with a strictly positive *logical duration* forming temporal intervals; although unlike LET, each interval is defined by two synchronous instants called *Temporal Synchronization Points (TSP)* forming its boundaries. As in SR, those *TSP* are defined relatively to some *logical clock*.

As an illustration, Figure 4.2 shows the typical behavior of a synchronous LET task. The synchronization instants are specified respectively to logical clock A and B, in an alternation pattern. Thus, each computation has a duration defined as either 1 tick with clock A or B. This forms intervals — called *elementary actions* — with a delayed communication model similar to the traditional LET approach. More generally, in PSYC, the span of an elementary action is defined by the n^{th} tick of the specified clock after the activation instant.

Compilation, and more precisely, real-time analysis, is also very similar to that of LET and relies on the same implementation hypothesis: a computation has to terminate before its deadline. Thus, a language based on synchronous Logical Execution Time is compliant with classical schedulability policy ensuring that a computation fits in its logical execution time interval.

4.2.2 Definitions

This section covers the main definitions of synchronous LET that will serve as semantics foundations in the next sections.

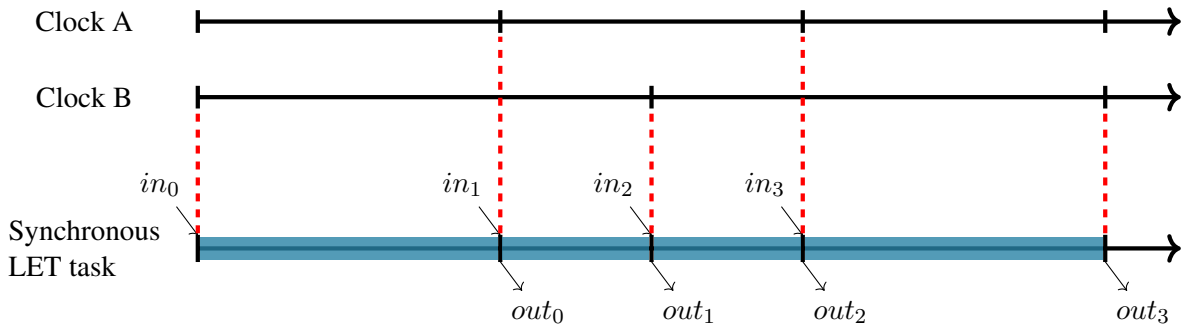


Figure 4.2 – Execution of a synchronous logical execution time program in logical time

We start by defining *logical clocks*, the base construction used to define all the temporal constraints:

Definition 4.2.1 (Logical Clocks)**.** A logical clock c is defined by a totally ordered set of ticks $(t_i)_{i\in\mathbb{N}}$.

Based on the logical clocks defined above, sLET allows the modelisation of time windows for computations — called *elementary actions* — similar to PSYC. These elementary actions are used to abstract the execution time of some computation by defining two synchronous instants: one corresponding to the activation instant and one corresponding to the deadline. Unlike LET, these two instants are synchronous and specified using logical clocks.

Definition 4.2.2 (Elementary Actions)**.** A Logical Execution Time (or *Elementary Action*) is defined by some computation reaction, constrained by two temporal constraints:

— an activation date (or earliest start date) d_{esd} defined on some logical clock c_{start} ;

— a deadline date d_{ddl} defined on some logical clock c_{ddl}, by the number of ticks to await

These two constraints are special synchronous instants called *Temporal Synchronization Points (TSPs)* and are used to form *elementary actions*.

Finally, to guarantee model determinism, communications have to respect a visibility criterion similar to the ones from LET.

Definition 4.2.3 (Visibility Principle)**.** Consider the elementary action of some producer (d^w_{esd}, d^w_{ddl}) and the elementary action of some consumer (d^r_{esd}, d^r_{ddl})

— An output produced by the producer task defines a visibility date such that $d_{visibility} \geq d^w_{ddl}$;

— An input can only be consulted by the consumer if its visibility date satisfies $d_{visibility} \leq d^r_{esd}$

Hence, LET corresponds to the particular case in which $d_{visibility} = d^w_{ddl}$ and should then satisfy $d^w_{ddl} \leq d^r_{esd}$

The definitions above form the basis of the synchronous LET model. Elementary actions are then typically composed sequentially — as in LET. The deadline TSP of some elementary action is also the activation TSP of the next elementary action building sequential agents which can then be composed in parallel.

4.2.3 About functional and temporal determinism

The visibility principle actually gives a key criterion to ensure communication determinism, and thus, to avoid that the instants at which communication happens depend on the *physical* execution time of some computation — its platform provision — which would be unpredictable. Previous work from the CEA on PSYC [LO12] proved this determinism result in the case of one global source clock. The next chapter defines a synchronous semantics for PSYC in the general case of multiple source clocks. Hence, as the semantics of synchronous-reactive languages is deterministic [PBEB07], determinism also holds for the generalized PSYC model — but more generally, synchronous LET — when multiple logical clocks are involved.

Nonetheless, one might argue that, in practice (not in the model per se), we can face two types of non-determinism:

— *Internal non-determinism* can hardly be avoided in practice; while the semantics itself does not face non-determinism issues, the tools (e.g., the compiler) do not typically evaluate branching conditions in the control-flow. Hence, tasks are typically modeled as non-deterministic finite automata simplifying various analysis.

— *External non-determinism* arises typically when multiple logical clocks are used in inputs of the system (corresponding to multiple sources in PSYC). When two tasks depend on different clocks that are not related, the ordering between them can be non predictable, and therefore cause different effects. Various solutions have been proposed in the literature (*input guards* in CSP, *conflict-freeness* in Petri nets, *endochrony* in SR languages . . .). However, the simpler approach is usually to limit the task dependency to unrelated clocks; in PSYC, industrial projects very often rely on a unique source clock — refined multiple times however — for safety reasons.

4.3 The PSYC language

PSYC is a language developed by KRONO-SAFE within the ASTERIOS toolchain. PSYC stands for *Parallel Synchronous C* and takes the form of C code to which synchronous and parallel primitives have been added. As mentioned in the previous section, it inherits some aspects of both Synchronous-Reactive and Logical Execution Time formalisms. The remainder of this chapter gives an informal description of the PSYC language relying on synchronous Logical Execution Time. Its full formalization is described in the next chapter.

4.3.1 Source and clocks

Sources and clocks are the building blocks of the PSYC language modeling a logical time basis which is used to trigger task computations.

4.3.1.1 Source clocks

Source clocks represent logical clocks used as application inputs. They are very similar to ESTEREL input signals or to LUSTRE input streams; source clocks define a time basis on which task computations react. They can be assigned either to a hardware timer or to any input events by the toolchain. Syntactically, source clocks are defined as following:

```
1 source <name>;
```

Listing 4.1 – PSYC source declaration

We give here a proper definition of a source clock which corresponds to the sLET definition given in section 4.2.

Definition 4.3.1 (Source clock)**.** A source clock s is defined by a totally ordered set of ticks $(s_i)_{i\in\mathbb{N}}$

4.3.1.2 Periodic clocks

Similarly to source clocks, periodic clocks in PSYC are also logical clocks defining a time basis on which task computations react. However, contrary to source clocks, they are defined respectively to other source or periodic clocks as periodic refinements. Syntactically, they are defined as follows:

```
1 clock <name> = <period> * <parent> + <offset>;
```

Listing 4.2 – PSYC clock declaration

where <period> $\in \mathbb{N}^*$ and <offset> $\in \mathbb{N}$.

Period and offset can be omitted in the syntax. In that case, they are defined with their default value, respectively 1 and 0.

Definition 4.3.2 (Periodic clock)**.** A periodic clock c is defined by $(c_i)_{i\in\mathbb{N}}$ with respect to its parent cp where $\forall i \in \mathbb{N}, c_i \equiv cp_{P\times i+O}$ in which $\equiv$ denote temporal coincidence (i.e., they are synchronous), P is the period and O is the offset.

4.3.1.3 Illustration: source and clocks

Figure 4.3 illustrates the use of several clocks taken from a classical example using multiple source clocks, the control of an automotive powertrain. Two independent time bases are defined: one representing the angular position of the crankshaft via intermediate cranks, and the other representing a time coming from a hardware timer. These two time bases are defined with two source clocks, respectively, `tooth` and `realtime`:

```
1 source tooth;
2 clock revolution = 6 * tooth;
3 source realtime;
4 clock c2 = 2 * realtime;
5 clock c4_2 = 2 * c2 + 1;
6
```

Listing (4.3) PSYC sources and clocks example

Figure 4.3 – The left part shows the PSYC syntax of the source and clock example, and the right part shows a possible chronogram of their execution

— `tooth` is used to define a periodic clock `revolution` corresponding to the complete revolution of the crankshaft which occurs every 6 ticks of crankshaft `tooth`.

— `realtime` is used to define two periodic clocks `c2` and `c4_2`. Assuming that `realtime` is a hardware timer with a 1 millisecond period, then `c2` has a period of 2 milliseconds as it ticks every two ticks of clock `realtime`. The second periodic clock, `c4_2`, has a period of 4 milliseconds as it ticks every two ticks of clock `c2` which has a 2 millisecond period. Additionally, `c4_2` has an offset of 1 tick of clock `c2` which corresponds to 2 milliseconds.

4.3.2 Tasks

We now describe PSYC tasks which form the basic sequential execution unit of a PSYC application; the latter is composed of a set of concurrent tasks which are run in parallel on the target and can be assigned to a specific CPU core. In this thesis, we only define the main type of PSYC task called `agent`.

4.3.2.1 Elementary Actions

As explained in the section 4.2, *elementary actions* allow the definition of temporal window, similarly to Logical Execution Time, containing computations. An elementary action is defined by two synchronization instants, an *activation* and a *deadline* instant on which communications happen. These are defined relatively to clock ticks by the `advance` statement as follows:

```
1 advance <n> with <c>;
```

Listing 4.4 – PSYC advance statement

Where $n \in \mathbb{N}^*$ and c is a clock. It defines the next synchronization instant as being the n^{th} tick of clock c after the preceding synchronization instant. It acts both as a deadline instant for code preceding it and as an activation instant for code following it.

The following example shows that `f()` has to terminate before $n1$ ticks of clock $c1$. This synchronization instant also acts as the activation instant of the next elementary action composed of `g()` which should terminate before $n2$ ticks of clock $c2$.

```
f(); advance n1 with c1; g(); advance n2 with c2;
```

Listing 4.5 – PsyC advance statement example

4.3.2.2 Body

Body is a dedicated PsyC syntactical construction allowing the definition of structures similar to state machines inside tasks. By default, a `body` is only a named scope containing a C-like compound statement acting as an infinite loop. It is defined as follows:

```
body <name> {
  <PsyC-statements>+
}
```

Listing 4.6 – PsyC body declaration

PsyC statements can be any C statement in addition to dedicated PsyC statements such as `advance`.

To change the default looping behavior, dedicated statements can be used to change the `body` to be executed. `next b` selects the next body to be executed when the current one terminates and `endbody` aborts the current body to execute the next selected one, which is by default itself. Those two statements can be composed to form a new one, `jump b`, that jumps to the specified body directly.

To illustrate these concepts, the example below defines three body scopes corresponding to different states — or modes — of the system's task. The first body is called `start` and is executed only once as body `loop` is selected as the next body to be executed. Then body `loop` loops unless an error is detected; in that case, it jumps directly to the `failure` body.

```
body start {
  next loop;
  /* ... */
  advance 1 with c_init;
}
body loop {
  if (/* errors? */) jump failure; /* or next failure; endbody; */
  advance 1 with c_period;
}
body failure { /* ... */ }
```

Listing 4.7 – PsyC body declaration example

4.3.2.3 Agents

We now describe the definition of PSYC agents. They are defined as following:

```
agent <name>(starttime <n> with <c>) {
  <Body-declarations>+
}
```

Listing 4.8 – PSYC agent declaration

They are defined as a set of body declarations in addition to a `starttime` parameter defining the first task synchronization instant, similarly to an `advance` statement. In this thesis, we consider that the `starttime` parameter can be omitted which is equivalent to immediately starting the first body on startup. Additionally, the first selected body is the `start` one — by convention — which has to be defined.

The example below illustrates the definition of a very simple agent which has a period of 1 tick of clock c and an offset of 1 tick of the same clock.

```
agent MonAgent(starttime 1 with c) {
  body start {
    f_init();
    next nominal;
  }
  body nominal {
    f_step();
    advance 1 with c;
  }
}
```

Listing 4.9 – PSYC agent declaration example

4.3.2.4 Illustration: LED blinking

We illustrate the concepts defined in the sections above with one of the most classical examples of a real-time system: a LED blinking controller. We give two variations of the same application. One defining the blinking of one LED, and the other defining the blinking of two LEDs (red and green) depending on the current mode with different frequencies.

In both cases, the blinking specification is defined by the following requirements:

1. The periodicity of the LED pattern;
2. The causality of the LED pattern, switching on and off should alternate;
3. And the duty cycle of the LED pattern, defining the proportion in which the LED stays on with respect to the period. In other words, this defines the minimum and maximum latency between switching on and off the LED.

The chronogram below shows the typical timing that we expect from the application based on the above requirements. The period is defined by 10 ticks of clock `c_period` whereas the duty cycle is defined as being 40% to 60% of the period, which corresponds to a latency between 4 and 6 milliseconds.

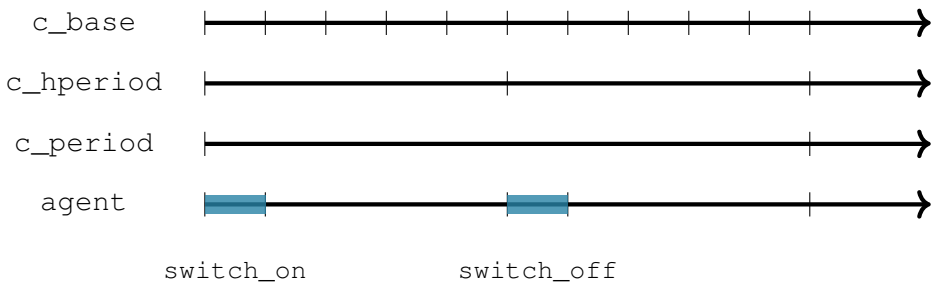


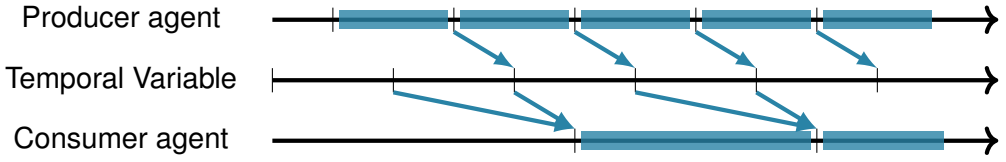
Figure 4.4 shows on the left side the implementation in which only one LED is blinking and on the right side the implementation with two different LEDs, red and green, driven by some mode conditions, with different frequency (and duty cycle) requirements.

4.3.3 Inter-task communication channels

The PSYC language has two deterministic communication means: *temporal variables*, which are dataflow channels that are updated on predefined instants, and *streams*, which are buffers with a *First-In First-Out* policy.

4.3.3.1 Temporal Variables

A *temporal variable* is a shared variable between a producer — which updates it at predefined instants — and its consumers. Additionally, temporal variables are sampled on clocks before being available to consumers; the latter can access the last sampled value, which is the last updated producer value. Note that, in a dataflow channel, values are persistent. If no value is updated by the producer on some synchronization instant, the previous value is used. The chronogram below illustrates the temporal variable mechanism. Each clock tick of the temporal variable corresponds to the last value produced by the producer agent and each value read by the consumer corresponds to the value of the last clock tick of the temporal variable — but previous sampled values can also be accessed using the temporal variable history.



Syntactically, a temporal variable is defined as follows:

```
1  source realtime;
2  clock c_base = realtime;
3  clock c_half_period = 5 * c_base;
4  clock c_period = 2 * c_hperiod;
5
6  agent LED(starttime 1 with c_period)
7  {
8    body start {
9      switch_on();
10     advance 1 with c_base;
11     advance 1 with c_hperiod;
12     switch_off();
13     advance 1 with c_base;
14     advance 1 with c_period;
15   }
16 }
```

Listing (4.10) PSYC example of a LED blinker

```
1  source realtime;
2  clock c_base = realtime;
3  clock c_half_period = 5 * c_base;
4  clock c_period = 2 * c_half_period;
5
6  agent LED(starttime 1 with c_period)
7  {
8    body start {
9      if (...)
10       next blink_red;
11     else
12       next blink_green;
13   }
14   body blink_red {
15     switch_on(RED);
16     advance 1 with c_base;
17     advance 1 with c_half_period;
18     switch_off(RED);
19     advance 1 with c_base;
20     advance 1 with c_period;
21     next start;
22   }
23   body blink_green {
24     switch_on(GREEN);
25     advance 1 with c_base;
26     advance 1 with c_period;
27     switch_off(GREEN);
28     advance 1 with c_base;
29     advance 1 with c_period;
30     next start;
31   }
32 }
```

Listing (4.11) PSYC example of a LED blinker with modes

Figure 4.4 – LED blinking example

```
temporal <type> <variable_name> with <clock>;
```

Listing 4.12 – PSYC temporal variable declaration

Where <clock> corresponds to the sampling clock.

From the producer side, the agent has to declare the temporal variables that it is going to write. It is defined as follows:

```
agent Producer(/* ... */) {
  display <variable_name>;
  /* ... */
}
```

Listing 4.13 – PSYC agent display declaration

Then, writing on a temporal variable is done by a simple assignment as in C:

```
  <variable_name> = <new_value>;
```

Listing 4.14 – PSYC temporal variable assignment statement

From the consumer's side, the agent also has to specify the temporal variables that it is going to read from, as well as the number of past samples to read with respect to the temporal variable's clock. Syntactically, it is defined as following:

```
agent Consumer(/* ... */) {
  consult <N> $ <variable_name>;
  /* ... */
}
```

Listing 4.15 – PSYC agent consult declaration

Where <N> $\in \mathbb{N}^*$ is the number of past samples read; 1 means only the last sampled value. Then, reading from a temporal variable is done by specifying the sample to be read with respect to its clock. Syntactically, it is defined as follows:

```
  /* ... */ = $[<N>]<variable_name>;
```

Listing 4.16 – PSYC temporal variable expression

Where <N> $\in \mathbb{N}$ means the n^{th} previous sampled value; 0 means reading the last sampled value, 1 the previous one ...

```
1  source realtime;
2  clock c_base = realtime;
3  clock c_period = 10 * c_base;
4
5  temporal float fast_to_gnc = 0.0f
6    with c_period;
7
8  agent Fast(starttime 1 with c_period)
9  {
10   consult 1 $ gnc_to_fast;
11   display fast_to_gnc;
12   body start {
13     gnc_to_fast =
14       Fast_sensors(/* ... */);
15     Fast_actuators($[0]gnc_to_fast);
16     advance 1 with c_base;
17   }
18 }
```

Listing (4.17) `Fast` Task

```
1  clock c_gnc = 10 * c_base + 3;
2  temporal float gnc_to_fast = 0.0f
3    with c_gnc;
4
5  agent GNC(starttime 1 with c_period)
6  {
7    consult 1 $ fast_to_gnc;
8    display gnc_to_fast;
9    body start {
10     gnc_to_fast = GNC(fast_to_gnc);
11     advance 3 with c_base;
12     advance 1 with c_period;
13   }
14 }
```

Listing (4.18) `GNC` Task

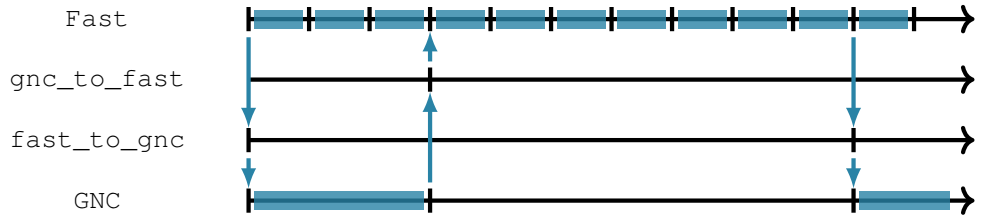
Figure 4.5 – Guidage, Navigation and Control (GNC)

4.3.3.2 Illustration: control system

To illustrate the concept of temporal variables, we show in Figure 5.3 the simplified functional architecture of a guidance, navigation and control system, inspired from the one in [CPBL+15]. In such a system, communication is typically done using dataflow channels which can be implemented in PSYC using temporal variables. The system consists of two main communicating tasks: `Fast` which quickly interacts with hardware and `GNC` which interacts with `Fast` applying guidance commands. Between these two tasks, there are two temporal variables:

— `fast_to_gnc` sending data from `Fast` to `GNC` every 10 milliseconds;

— and `gnc_to_fast` sending data from `GNC` to `Fast`, every 10 milliseconds with an offset of 3 milliseconds to match the end of GNC elementary actions.



4.3.3.3 Streams

A *stream* is a queue belonging to a specific consumer and updated by an agent on predefined instants. A producer can push data into a stream and a consumer can pop them.

In PSYC, a stream is defined as follows:

```
1 stream <type> <stream_name> expiration <N>;
```

Listing 4.19 – PSYC stream declaration

Where <N> $\in \mathbb{N}^*$ corresponds to the maximum duration (with respect to some source) that the data is available. It is used to ensure that the stream is always bounded; however, if some data expires in the stream, the system raises an error.

From the producer's side, the agent has to specify the stream that it is going to push on. Syntactically, it is defined as following:

```
1 agent Producer(/* ... */) {
2   pushto <stream_name>;
3   /* ... */
4 }
```

Listing 4.20 – PSYC agent pushto declaration

Then, sending data to a stream is done by the `push` statement. Contrary to a temporal variable, multiple values can be pushed inside during an elementary action. Syntactically, it is defined as follows:

```
1   push(<stream_name>, <new_value>);
```

Listing 4.21 – PSYC stream push statement

From the consumer's side, the agent also has to specify the stream it is going to pop from. Syntactically, it is defined as follows:

```
1 agent Consumer(/* ... */) {
2   popfrom <stream_name>;
3   /* ... */
4 }
```

Listing 4.22 – PSYC agent popfrom declaration

Then, removing data item from a stream is done using the `pop` statement which returns the oldest unpopped data from the queue.

```
1   /* ... */ = pop(<stream_name>);
```

Listing 4.23 – PSYC stream pop statement

PSYC also has a statement called `count(<stream_name>)` which returns the number of available elements in a stream. It is used to avoid popping from an empty stream.

```
1  source realtime;
2  clock c_base = realtime;
3
4  stream float s with expiration 10 *
       clockperiod(c_base);
5
6  agent A(starttime 1 with c_base)
7  {
8    pushto s;
9    body start {
10     /* ... */;
11     push(s, data);
12     advance 1 with c_base;
13   }
14 }
```

Listing (4.24) A Task

```
1  agent Logger(starttime 1 with c_base)
2  {
3    popfrom s;
4    body start {
5      while [10] (count(s) > 0) {
6        log(pop(s));
7      }
8      advance 10 with c_base;
9    }
10 }
```

Listing (4.25) Logger Task

Figure 4.6 – Logging subsystem

In practice, streams have a few issues:

1. Popping from a stream without checking if the stream is not empty is unsafe;
2. Pushing too much data into a stream is also unsafe as it may raise an expiration error if the consumer does not pop enough data.

In practice, most applications — which are generally control systems — use temporal variables instead of streams. Hence, this thesis focuses mainly on temporal variables; nonetheless, the methodology shown in this thesis also applies to streams.

4.3.3.4 Illustration: logging

Let's now consider a simple logging use-case in which a `Logger` agent pops data from a stream and logs them. The PSYC code is shown in Figure 4.6. The code shows two tasks, an agent called `A` sending data to stream `s`, and an agent called `Logger` with a period 10 times slower. `A` pushes one data item every 1 millisecond whereas `Logger` pops up to 10 data items every 10 milliseconds (the *up to* is due to the loop decorator `[10]` which specifies an upper bound). Every 10 milliseconds, all data are popped from a stream. Because the stream is defined with an expiration of 10 milliseconds, no data expires.

4.4 Summary

This chapter gave an overview of the PSYC language and its toolchain. In PSYC, agents can specify not only the triggering instants of their computations but also their logical duration relative to a number of clock ticks. Consequently, the PSYC language relies on both the synchronous-reactive approach for its synchronization instants, and on the logical execution time approach for

its elementary actions. Both concepts are unified in the synchronous logical execution time model. The following chapter will give two different formal semantics for PSYC based on these concepts.

CHAPTER 5

Formal Semantics of PSYC

This chapter describes the semantics of a subset of the PSYC language using two different approaches:

— A native "big-step" semantics preserving the logical duration of time intervals which is defined by a structural operational semantics. This semantics can be used to build automata — similar to Timed-Constrained Automata [LDAVN10] — used as a formal model for PSYC compilation.

application	::=	*decl+*	*agent*	::=	agent id (starttime n with c) $body+$, $n \in \mathbb{N}$
decl	::=	*clock*	*body*	::=	body id $stmt$
	|	*source*	*stmt*	::=	$id := f(exp*)$
	|	*agent*		|	$stmt_1$; $stmt_2$
	|	*temporal*		|	if (exp) $stmt_1$ else $stmt_2$
source	::=	source c		|	while exp do $stmt$
clock	::=	clock $c_1 = n_1 * c_2 + n_2$		|	advance n with c, $n \in \mathbb{N}^*$
temporal	::=	temporal $id = v$ with c		|	endbody
				|	next b
				|	jump b
				|	nothing

Figure 5.1 – Abstract syntax of a PsyC subset

— A synchronous "small-step" semantics defined by translation to a Synchronous-Reactive language — ESTEREL — expanding time interval durations to a succession of atomic transitions. This semantics directly supports formal models such as Mealy machines which is used later in this work for formal verification.

We first describe a subset of the PSYC grammar considered in the semantics, then we formally define both semantics and we finish by giving an equivalence criteria between both semantics.

5.1 Abstract Syntax of PSYC

Before describing the formal semantics of PSYC in the next section, we first give the abstract syntax of the language. We choose to take a subset of PSYC excluding all advanced expressions and statements that are part of the C language (in fact, most of the C expressions are not interpreted by the PSYC compiler). The abstract syntax is described in Figure 5.1. The left column specifies the PsyC constructions defined at the application level (e.g., communication channels, clocks) while the right column describes the PSYC constructions at the agent level (e.g., statements). Aside from the abstracted expressions, we have the following assumptions on the grammar of PSYC:

— A PSYC application has to declare at least one agent and one clock so that the agent can specify synchronization instants;

— Temporal variables are associated to a unique producer agent to avoid non determinism;

— An agent should contain at least one body called `start` which is the first body to be executed;

— Finally, in a body, we assume that all paths should declare at least one `advance` statement to avoid instantaneous loop.

The above assumptions are syntactical restrictions that must be respected to guarantee the well-foundedness of PSYC programs. They can be easily checked by some static analysis, which are implemented in the current PSYC compiler.

5.2 Native "big-step" Semantics of PSYC

5.2.1 Native semantics of one PsyC agent

We now provide the Structural Operational Semantics (SOS) semantics [Plo04] [Ber02] for single agents; the case of multiple agents — or agent networks — will be considered in next section. The semantics amalgamate structurally all computations inside an elementary action until the next `advance` statement. This results in a single transition for an elementary action, labeled with the specified logical duration specified relatively to a source clock.

As always with SOS rules, operational transitions can be collected in a transition system, which in the case of PSYC is actually a Finite State Machine (FSM). This FSM can be seen as a generalization of Timed-Constrained Automata (TCA) introduced by Lemerre in [LDAVN10]; TCA abstract elementary actions by transitions labeled with absolute durations and with nodes corresponding to a mix of *before* and *after* constraints but cannot specify logical durations specified *relatively* to a source clock. We first provide necessary notations, then provide SOS rules for the non-temporal control-flow features, then for temporal clocks advancements, and finally for the merge at the agent declaration level.

5.2.1.1 Notations

Operational semantics rules are usually given as a relation over a possible rewriting. Let's consider the relation $s \longrightarrow s'$ which basically means that a given state s can be rewritten into s' (also called the residual) after the transition. In a rule, either the relation is given alone, then it's said to be an *axiom*, or it's given with respect to some assumptions. In this article, inference rules are represented as follows:

$$\frac{h_1\ h_2 \cdots h_n}{s \longrightarrow s'} \tag{5.1}$$

This means that if all assumptions are true (e.g., $h_1, h_2 \cdots h_n$), then the transition relation can be used. Additionally, the program state often has a program environment that keeps track of the program variable values. For a given environment E, all declared variables x can have their values accessed using the notation $E(x)$. $[\![\ exp\ ]\!]_E$ denotes a more general evaluation of exp with respect to the environment E and $E[x \leftarrow [\![\ exp\ ]\!]_E]$ denotes the update of the environment E with the evaluation of exp assigned to x. Similarly, we use the same notation $B(b)$ and $B[b \leftarrow stmt]$ to respectively fetch body b and assign $stmt$ to a body b from the set of agent bodies B.

Configuration syntax

In this paper, an agent configuration — or state — is defined by the following tuple:

$$\langle E, T, b \rangle$$

where E is the private environment of the agent, b is the identifier of the next body to be executed and T is the public environment of the agent. This allows to distinguish values that can be accessed by other tasks (through the communication mechanism) and values that shouldn't be accessed.

Transition Syntax

The semantics of PSYC is described in this work using two complementary relations, one for logically instantaneous behavior and one that represents time progress in terms of logical instants. We shall call the former *non-temporal transitions* and the latter *temporal transitions*.

For *non-temporal* transitions, the following relation is used:

$$C \vdash t \longrightarrow C' \vdash t'$$

where C, C' denote the agent configuration respectively before and after the transition, and t, t' denote the agent statements in the same way.

Temporal transitions are due to *advance* statements making the time progress and are defined similarly:

$$C \vdash t \Longrightarrow_{n \times s} C' \vdash t'$$

where $n \in \mathbb{N}^*$ and s is a source clock, denoting a temporal transition that has a duration of n ticks of the source s.

They can be composed with statements which don't take time as long as the sequence of non-temporal transitions terminate with a temporal transition:

$$\frac{C \vdash t \longrightarrow C_1 \vdash t_1 \longrightarrow \cdots C_n \vdash t_n \Longrightarrow C' \vdash t'}{C \vdash t \Longrightarrow C' \vdash t'}$$

Finally, we note $\longrightarrow^*$ the sequential composition of zero, one or more rules $\longrightarrow$,

5.2.1.2 Sources and Clocks

Consider the following clock $c = n_1 \times c_p + n_2$ where $n_1 \in \mathbb{N}^*$ and $n_2 \in \mathbb{N}$. Its semantics is given by the following equations, which give respectively the source, the period and the offset of the clock:

$$Source(c) = \begin{cases} c, & \text{if } c_p \text{ is a source} \\ Source(c_p), & \text{if } c_p \text{ is another clock} \end{cases} \quad \text{(clk-source)}$$

$$\Pi(c) = \begin{cases} 1, & \text{if } c_p \text{ is a source} \\ \Pi(c_p) \times n_1, & \text{if } c_p \text{ is another clock} \end{cases} \quad \text{(clk-period)}$$

$$\Phi(c) = \begin{cases} 0, & \text{if } c_p \text{ is a source} \\ n_2 \times \Pi(C_p) + \Phi(C_p), & \text{if } c_p \text{ is another clock} \end{cases} \quad \text{(clk-offset)}$$

Definition 5.2.1 (Clock State)**.** For a given absolute source date d, a clock state d_c is defined as $d_c = (d - \Phi(c))\ mod\ \Pi(c)$.

Additionally, to avoid keeping an unbounded date for each source, we assume that d_c gives the (current) date of c modulo its period.

Note that, for each of these definitions, we assume that clock dependencies cannot be cyclic and form either a tree or a forest, depending on whether a unique source is used or not.

5.2.1.3 Semantics Rules

Basic statements We first consider basic statements, which corresponds to all non-temporal statements apart from control-flow statements.

— nothing terminates instantaneously and can be rewritten to itself.

$$C \vdash \mathsf{nothing} \longrightarrow C \vdash \mathsf{nothing} \qquad \text{(nothing)}$$

— $x := exp$ terminates instantaneously with an updated environment due to the evaluation of exp.

$$E, T, b \vdash x := exp \longrightarrow E[x \leftarrow [\![exp]\!]_E], T, b \vdash \mathsf{nothing} \qquad \text{(assign)}$$

— next b statement changes the value of the target body instantaneously

$$E, T, b_1 \vdash \mathsf{next}\ b_2 \longrightarrow E, T, b_2 \vdash \mathsf{nothing} \qquad \text{(next)}$$

— jump b statement changes the value of the target body instantaneously and exits the current body

$$E, T, b_1 \vdash \mathsf{jump}\ b_2 \longrightarrow E, T, b_2 \vdash \mathsf{endbody} \qquad \text{(jump)}$$

— endbody statement exists the current body and starts the target body instantaneously

$$E, T, b \vdash \mathsf{endbody} \longrightarrow E, T, b \vdash \textit{AbortBody()} \qquad \text{(endbody)}$$

Where *AbortBody()* terminates instantaneously the current body. The full definition of this rule would require extending transitions with dedicated abort signal, similarly to the `exit` statement of ESTEREL. Note that aborting the current body to start a new one does not take logical time; it is logically instantaneous.

If statements

— `if` statements terminate instantaneously and are rewritten with the branch selected by the condition.

$$\frac{[\![exp]\!]_E \neq 0}{E, T, b \vdash \mathsf{if}\ (exp)\ s_1\ \mathsf{else}\ s_2 \longrightarrow E, T, b \vdash s_1} \qquad \text{(if-1)}$$

$$\frac{[\![exp]\!]_E = 0}{E, T, b \vdash \mathsf{if}\ (exp)\ s_1\ \mathsf{else}\ s_2 \longrightarrow E, T, b \vdash s_2} \qquad \text{(if-2)}$$

While statements

— While statements are instantaneous and are rewritten into the sequence of the body followed by itself if the condition is evaluated to true

$$\frac{[\![\ exp\ ]\!]_E \neq 0}{E, T, b \vdash \mathsf{while}\ exp\ \mathsf{do}\ s \longrightarrow E, T, b \vdash s\ ;\ \mathsf{while}\ exp\ \mathsf{do}\ s} \quad \text{(while-1)}$$

— While statements are instantaneous and are rewritten into nothing if the condition is evaluated to false.

$$\frac{[\![\ exp\ ]\!]_E = 0}{E, T, b \vdash \mathsf{while}\ exp\ \mathsf{do}\ s \longrightarrow E, T, b \vdash \mathsf{nothing}} \quad \text{(while-2)}$$

Sequence statements

— The sequence statement is instantaneous if the first statement terminates instantaneously.

$$\frac{C \vdash s_1 \longrightarrow^* C' \vdash \mathsf{nothing}}{C \vdash s_1\ ;\ s_2 \longrightarrow C' \vdash s_2} \quad \text{(seq)}$$

— The sequence statement takes time if the first statement takes time.

$$\frac{C \vdash s_1 \Longrightarrow_{N \times s} C' \vdash s_1'}{C \vdash s_1\ ;\ s_2 \Longrightarrow_{N \times s} C' \vdash s_1'\ ;\ s_2} \quad \text{(seq)}$$

Advance statement Unlike the previous statements, `advance` is not logically instantaneous. In fact, it is the only PSYC statement that takes time. Thus, it is rewritten to `nothing` and takes exactly the logical duration specified by the advance statement (with respect to the source clock).

$$\frac{N = \mathit{Duration}(n, c, \mathit{date}_s) \quad T' = \mathrm{UpdateOutputs}(E) \qquad s = \mathit{Source}(c)}{E, T, b \vdash \mathsf{advance}\ n\ \mathsf{with}\ c \Longrightarrow_{N \times s} E, T', b \vdash \mathsf{nothing}} \quad \text{(advance-1)}$$

As written in the rule above, the *Duration* function needs a *date* parameter to convert a deadline relative to a clock tick to an absolute deadline represented as a duration in source ticks; this *date* parameter is relative to the source of the clock parameter. Additionally, the *UpdateOutputs* function extracts all the outputs from the internal environment to the externally visible one.

Definition 5.2.2 (Logical Duration)**.** Considering a date d_s assigned to a source s and updated at each s source tick, *Duration* can be defined with respect to a deadline date:

$$\mathit{LastTick}(c, d_s) = \left\lfloor \frac{d_s - \Phi(c)}{\Pi(c)} \right\rfloor$$

$$\mathit{Deadline}(n, c, d_s) = \Pi(c) \times (n + \mathit{LastTick}(c, d_s)) + \Phi(c)$$

$$\mathit{Duration}(n, c, d_s) = \mathit{Deadline}(n, c, d_s) - d_s$$

We consider below an other version of the *Duration* function based on the (finite) clock state d_c defined in section 5.2.1.2 instead of an absolute source date d_s.

Definition 5.2.3 (Finite State Logical Duration)**.**

$$\mathit{Duration}'(n, c, d_c) = n \times \Pi(c) - d_c$$

Proposition 5.2.1. *Let* $d_c = (d_s - \Phi(c))\ mod\ \Pi(c)$*, then* $\mathit{Duration}(n, c, d_s) = \mathit{Duration}'(n, c, d_c)$

Proof.

By definition of mod:

$$d_c = (d_s - \Phi(c))\ mod\ \Pi(c) = (d_s - \Phi(c)) - \Pi(c) \times \left\lfloor \frac{d_s - \Phi(c)}{\Pi(c)} \right\rfloor)$$

$$n \times \Pi(c) - d_c = n \times \Pi(c) - (d_s - \Phi(c)) + \Pi(c) \times \left\lfloor \frac{d_s - \Phi(c)}{\Pi(c)} \right\rfloor)$$

$$\mathit{Duration}_2(n, c, d_c) = \Pi(c) \left(n + \left\lfloor \frac{d_s - \Phi(c)}{\Pi(c)} \right\rfloor \right) + \Phi(c) - d_s$$

$$\mathit{Duration}_2(n, c, d_c) = \mathit{Duration}_1(n, c, d_s)$$

□

Body declaration For body declaration, we assume that a special body, called *current* is used only to keep track of rewriting. That is, it's initially a copy of the *start* body, and it is then updated at each source tick of the agent. When it's empty, its content becomes a copy of the next body.

— A body takes time if its content takes time (i.e., typically due to an advance statement). Then, at the next step, the `current` body will be assigned to the residual statement of the latter.

$$\frac{C \vdash B(\mathit{current}) \Longrightarrow C' \vdash \mathit{stmt}' \quad B' = B[\mathit{current} \leftarrow \mathit{stmt}']}{C \vdash B \Longrightarrow C' \vdash B'} \quad \text{(body-1)}$$

— A body terminates instantaneously if its content is composed of a sequence of instantaneous relations terminating by a nothing statement. Then, at the next step, the `current` body will be assigned to the last target body given by b'.

$$\frac{C \vdash B(\mathit{current}) \longrightarrow^* E', T', b' \vdash \mathsf{nothing} \quad B' = B[\mathit{current} \leftarrow B(b')]}{C \vdash B \longrightarrow E', T', b' \vdash B'} \quad \text{(body-2)}$$

Agent declaration We finally define rules for PsyC agents that compose the rules defined above.

— An agent first transforms its starttime expression into one that depends only on source ticks. Note that, given the specification, the target body b is initially equal to "start".

$$\frac{n > 0 \quad N = \mathrm{Duration}(n, c, 0) - 1 \quad s = \mathit{Source}(c)}{C \vdash \mathsf{agent}\ id\ (\mathsf{starttime}\ n\ \mathsf{with}\ c)\ B \Longrightarrow_{N \times s} C \vdash \mathsf{agent}\ id\ (\mathsf{starttime}\ 0\ \mathsf{with}\ c)\ B} \quad \text{(agent-1)}$$

— An agent with a starttime expression equal to 0 starts the execution of its bodies.

$$\frac{C \vdash B \Longrightarrow_{N\times s} C' \vdash B'}{C \vdash \mathsf{agent}\ id\ (\mathsf{starttime}\ 0\ \mathsf{with}\ c)\ B \Longrightarrow_{N\times s} C' \vdash \mathsf{agent}\ id\ (\mathsf{starttime}\ 0\ \mathsf{with}\ c)\ B'} \quad \text{(agent-2)}$$

— An agent with a starttime expression equal to 0 with multiple instantaneous relations on its bodies can squash them, as long as they are followed by a non-instantaneous relation.

$$\frac{C \vdash B \longrightarrow^* C' \vdash B' \quad C' \vdash B' \Longrightarrow_{N\times s} C'' \vdash B''}{C \vdash \mathsf{agent}\ id\ (\mathsf{starttime}\ 0\ \mathsf{with}\ c)\ B \Longrightarrow_{N\times s} C'' \vdash \mathsf{agent}\ id\ (\mathsf{starttime}\ 0\ \mathsf{with}\ c)\ B''} \quad \text{(agent-3)}$$

The semantics rules of a PSYC agent always yield temporal transitions because all paths in a body have at least one `advance` statement.

5.2.2 Native semantics of a PsyC agent network

5.2.2.1 Global states vs intermediate states

The native sLET semantics of PSYC agents is defined quite classically by a transition expressing rewriting of the agent. However, more interestingly, these transitions also express some (logical) duration with respect to some logical clock (or source clock) which is the foundation of synchronous LET; they allow to express sLET intervals (as well as classical LET intervals).

However, problems arise when building the network of agents. Indeed, in a classical synchronous language, the semantics naturally gives a synchronous composition as each instant has the same duration, that is, the cycle rate. In (synchronous) LET, multiple sLET intervals might have different durations. Consequently, in the native semantics, one cannot express directly the agent composition with only agent states.

To resolve this issue, we define two kinds of agent states in the context of an agent network:

1. *Global states*, which correspond to the actual agent states from the native semantics (i.e., the boundaries of an sLET interval)
2. *Intermediate states*, which correspond to intermediate states in an sLET interval, typically resulting from the overlapping of the global state of another agent.

Futhermore, multiple agent transitions might have different durations based on different sources. There are two main cases:

— PSYC clocks rely on multiple source clocks and thus have a partial ordering due to external non-determinism.

— PSYC clocks rely on a unique source clock and thus have a total ordering allowing direct comparison of the agent intermediate states.

The next section illustrates the different overlapping patterns between agents for the mono-source case.

5.2.2.2 Overlapping patterns

The figure 5.2 illustrates all the different overlapping patterns between sLET intervals. All the cases have two agents, a blue and a red one executing both an sLET interval (but not necessarily synchronized). For each agent, a filled half circle represent a global state while an empty half circle represent an intermediate state. `advance` denotes the logical time of the whole interval while `remaining` denotes the logical time remaining in the current interval, due to overlapping.

— Case 5.2a shows two synchronized agents, they both start and end in a global state;

— Case 5.2b shows one agent (the blue one) longer than the other one, the longer interval shall then be split into sub-intervals, thus introducing an intermediate state and a remaining time;

— In the previous case, both agents still start at the same time. However, in case 5.2d, the blue agent is already in the middle of an interval, thus starting in an intermediate state. Nonetheless, they both end synchronously as the remaining time of the blue agent and the logical time of the red one are equal.

— Case 5.2c extends the previous case in having a longer remaining time for the blue agent than the logical time of the red one. Thus, the blue agent starts and ends on intermediate states.

— Case 5.2e is another variation of the two previous cases in which the red agent has a remaining time smaller than the logical time of the blue one. Hence, the blue agent ends on an intermediate state while the red one starts on an intermediate state but ends in a global state.

The reader might remark that the case where both agents end in an intermediate state is not covered. This should actually not happen for two agents as the one with the smaller logical time should end with a global state.

The next section introduces rules to formalize these patterns and extend them to vector of agents.

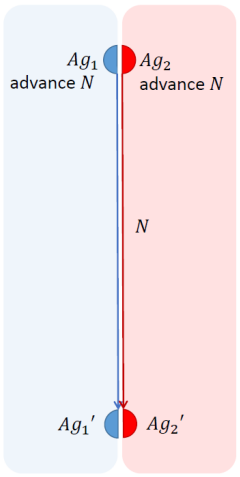
5.2.2.3 Notations

In a parallel network, an agent can be either in a global state, that is on the boundary of one of its interval or in an intermediate state, which is an intermediate instant during its interval. To convey this idea, we extend the notation of the state of an agent. We note:

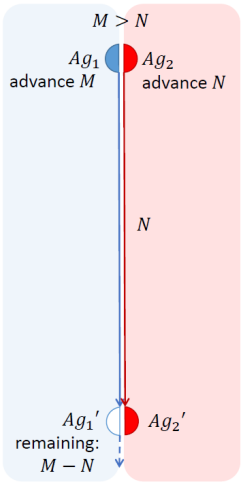
— $C \vdash P@N \times s$ which denotes an intermediate state of an agent with a remainder of N ticks of source s. If only one source is used we may simplify to the following notation $C \vdash P@N$;

— $C \vdash P$ which denotes a global state, as in the previous sections.

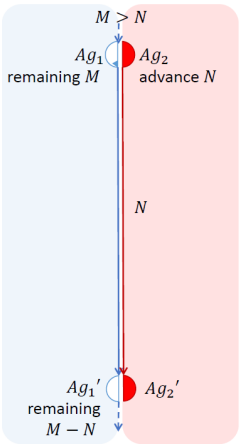
Futhermore, as we consider multiple agents, we denote a vector of agents as follows: $[ag_1, ag_2, \ldots ag_n]$ with ag_i being an agent state as described above.



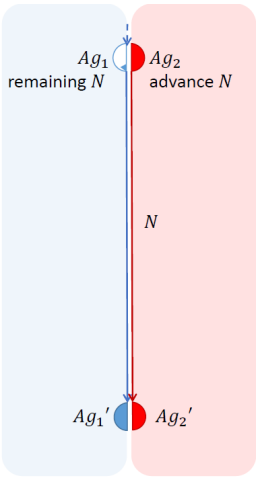
(a) Case 1: synchronized elementary actions



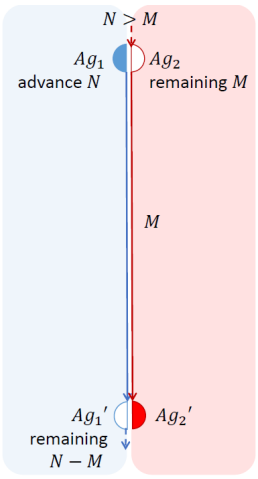
(b) Case 2: both agent starts simultaneously, but the elementary action of the blue agent terminate after the red one



(c) Case 3: the elementary action of the red agent starts and terminates during the elementary action of the blue agent



(d) Case 4: the elementary action of the red agent starts during the elementary action of the blue agent, but both terminate simultaneously



(e) Case 5: the elementary action of the blue agent starts during the elementary action of the red agent, but terminates before

Figure 5.2 – sLET Composition patterns

5.2.2.4 Semantics rules

This section gives the semantics rules that define the native semantics of a network of agents as explained in the two previous sections. First of all, we have to define how to handle an intermediate state for a given agent. For that, we define an additional agent relation that to handle intermediate states as shown below:

$$\frac{T' = UpdateOutputs(T, E)}{E, T \vdash ag@n \times s \Longrightarrow_{n \times s} E, T' \vdash ag} \quad \text{(agent-5)}$$

Furthermore, we define an helper function $\Delta_{k/k_{ag}}$ which denotes an agent partial advance of k instants to a possible intermediate state, while the remaining time of the agent elementary action is of k_{ag} instants. It takes the old and new agent states — both intermediate and global — as parameter, in addition to the global environment composed of all displayed temporal variables. The operator is defined below first for stuttering (no source tick), then for intermediate state and finally for global states.

$$\Delta_{k/0}(ag, \_, \_) = ag \quad \text{(delta-stuttering)}$$

$$\Delta_{k/k_{ag}}(\langle E,\, T\rangle \vdash P \,@\, k_{ag}, \langle E,\, T'\rangle \vdash P,\, T_{app}) = \begin{cases} \langle E \cup T_{app},\, T'\rangle \vdash P, & \text{if } k = k_{ag} \\ \langle E,\, T\rangle \vdash P \,@\, (k_{ag} - k), & \text{otherwise} \end{cases} \quad \text{(delta-intermediate)}$$

$$\Delta_{k/k_{ag}}(\langle E,\, T\rangle \vdash P,\, \langle E',\, T'\rangle \vdash P', T_{app}) = \begin{cases} \langle E' \cup T_{app}\, T'\rangle \vdash P', & \text{if } k_{ag} = k \\ \langle E',\, T\rangle \vdash P' \,@\, (k - k_{ag}) & \text{otherwise} \end{cases} \quad \text{(delta-global)}$$

In the equations above, all the patterns described in figure 5.2 are covered. For *multi-source* application *delta-stuttering* handles the case of agent stuttering (the agent does nothing on this step), *delta-intermediate* covers the case of figures 5.2c and 5.2d in which the blue agent starts in an intermediate state while *delta-global* covers the case of the figures in 5.2a, 5.2b and 5.2e in which the blue agent starts in a global state.

The rules should now handle two cases, *agent-network-mono-source* which is used for an agent network defined on a unique source and *agent-network-multi-source* which is used for an agent network defined on multiple sources. In the former, delta is used to perform intermediate advance as shown in Figure 5.2 while in the latter, the semantics is closer to an asynchronous product as found in process algebra. The transition is labeled by a duration with respect to a set of source that ticks on the current instant written S.

$$
\begin{gathered}
ag_1 \Longrightarrow_{k_1 \times s} ag'_1 \\
ag_2 \Longrightarrow_{k_2 \times s} ag'_2 \\
\dots \\
ag_n \Longrightarrow_{k_n \times s} ag'_n \\
k = min(k_1, k_2, \dots, k_n) \quad s \in S \\
T = \bigcup_{i=1}^{n} \begin{cases} Temporal(ag'_i) & \text{if } k = k_i \\ \emptyset & \text{otherwise} \end{cases}
\end{gathered}
$$

$$
[ag_1,\ ag_2\ \dots\ ag_n] \overset{S}{\Longrightarrow}_{k \times \{s\}} [\Delta_{k/k_1}(ag_1,\ ag'_1,\ T),\ \Delta_{k/k_2}(ag_2,\ ag'_2,\ T),\ \dots\ \Delta_{k/k_n}(ag_n,\ ag'_n,\ T)]
$$

(agent-network-mono-source)

$$
\begin{gathered}
ag_1 \Longrightarrow_{k_1 \times s_1} ag'_1 \text{ if } s_1 \in S \text{ otherwise } k_1 = 0 \\
ag_2 \Longrightarrow_{k_2 \times s_2} ag'_2 \text{ if } s_2 \in S \text{ otherwise } k_2 = 0 \\
\dots \\
ag_n \Longrightarrow_{k_n \times s_n} ag'_n \text{ if } s_n \in S \text{ otherwise } k_n = 0 \\
T = \bigcup_{i=1}^{n} \begin{cases} Temporal(ag'_i) & \text{if } k_i = 1 \text{ and } s_i \in S \\ \emptyset & \text{otherwise} \end{cases}
\end{gathered}
$$

$$
[ag_1,\ ag_2\ \dots\ ag_n] \overset{S}{\Longrightarrow}_{1 \times S} [\Delta_{1/k_1}(ag_1,\ ag'_1,\ T),\ \Delta_{1/k_2}(ag_2,\ ag'_2,\ T),\ \dots\ \Delta_{1/k_n}(ag_n,\ ag'_n,\ T)]
$$

(agent-network-multi-source)

5.2.2.5 Communication channels

The section above described only communications between tasks through a generic temporal variable channel which is available only on producer ticks and transmitted directly to any consumer. From that mechanism, which is similar to ESTEREL valued signals, more practical communication channels can be built using intermediate synchronous components (i.e., fifo, shared variable, etc ...) as long as they only synchronize using PSYC clocks. The PSYC language define two of these, although more can be defined. We discuss now those two channel types and how they can be handled:

— The actual *Temporal Variable* which, in practice, is persistent and also features sampling. This can be defined with a function $tv_{consumer} = Temporal(tv_{producer}, clk)$ where $tv_{consumer}$ is a persistent $tv_{producer}$ sampled on clk.

— The *Stream* which acts as a fifo between the producer and the consumer. Its definition is more complex as we have to deal both with the $push()$ statements of the producer as well as the $pop()$ statements of the consumer. Intuitively, all the $push()$ statements of an sLET interval outputs an array from the producer and all the $pop()$ statements outputs a pop counter from the consumer. Then, the communication function can be defined as $s_{consumer} = Stream(s_{producer}, nb_{consumer})$ where each time an $s_{producer}$ is sent by the

producer, it is concatenated in the global fifo where $nb_{consumer}$ previously read elements have been deleted (in a fifo fashion).

The communication functions defined above are synchronous, and thus their semantics is better defined using the synchronous approach. Basically, they can create a new temporal environment T' such that $T' = \{t' \mid \forall c \in \text{Channels}, f_c(T) = t'\}$ where Channels denotes the set of communication channels of the application and f_c its communication function.

Note that, even if communication channels only synchronize on PSYC clocks, like the agents, the logical instants on which they read or write values from agents is of prime importance. A good rhythm of data exchange is actually the key factor for the correctness of the whole system. It is needed in properties such as end-to-end latencies or freshness. To verify those properties is one of the possible applications of this semantics as well as the synchronous one defined in the next section.

The next section introduces the synchronous semantics and defines one communication function: the *Temporal Variable* with a sampling history which is the main communication channel of PSYC.

5.3 Synchronous "small-step" Semantics of PSYC

The last section presented a native semantics of synchronous LET that preserves the logical durations of the computations. It is used in practice to get efficient compilation and scheduling analyses. In this section, we define a simpler semantics — called *synchronous semantics* — defined as a translation from PSYC to the Synchronous-Reactive language ESTEREL. We chose ESTEREL because its syntax is very close to the PSYC language; they are both imperative and control-flow oriented. Contrary to the native semantics, the synchronous semantics does not preserve durations. Instead, they are all expanded into unit step transitions; it is hence considered as "small-step". Due to its synchronous foundation, this semantics directly supports formal foundations such as Mealy machines and digital circuits which are universally recognized formalisms, and are heavily used for formal verification. Hence, the goal of this semantics is twofold:

1. to study the relation between the two semantics, and thus, to model the synchronous behavior of sLET languages;
2. to have a simpler model to start formal verification activities by reusing state-of-the-art synchronous verification techniques.

The semantics is defined by structural translation from PSYC to ESTEREL. Thus, we start by giving a brief overview of ESTEREL before defining the translation rules.

5.3.1 The ESTEREL language

This section gives a quick overview of the ESTEREL language inspired from [Ber02].

5.3.1.1 Signals and Reactions

ESTEREL is a Synchronous-Reactive language and thus reacts continuously to inputs to give, *logically instantaneous* outputs. In other words, an ESTEREL program maps an input sequence to an output sequence. At each reaction, outputs are given by a function from inputs as well as an internal state computed at the preceding reaction.

Like other SR languages, ESTEREL is inherently concurrent and deterministic. Internal and external communication is done through broadcasted *signals* that carry both:

— A *status* that denotes the *presence* or the *absence* of the signal;

— And an optional *value* which is persistent when assigned. To distinguish both cases, signals are called *valued* when they carry a value and *pure* when they do not.

An ESTEREL program is wrapped-up in *modules* that can instantiate other *modules*:

```
1  module M;
2    -- input
3    input I : T;
4    -- output
5    output O : T;
6
7    signal L : T in
8      -- statements
9    end
10 end module
```

Listing 5.1 – Esterel module

Here, the module `M` takes input `I` to compute output `O` using its internal statements as well as the local signal `L`. A reaction then computes a set of output *statuses*/*values* at each instant.

5.3.1.2 Statements

The core of the ESTEREL language is composed of imperative statements. In this work, we use a variation of those core statements that handle variables and valued signals as defined in [PBEB07].

$$
\begin{array}{rcl}
p, q & ::= & \mathsf{nothing} \\
& | & \mathsf{pause} \\
& | & \mathsf{emit}\ S \\
& | & \mathsf{await}\ S \\
& | & v := exp \\
& | & \mathsf{if}\ c\ \mathsf{then}\ p\ \mathsf{else}\ q\ \mathsf{end} \\
& | & \mathsf{while}\ c\ \mathsf{do}\ p\ \mathsf{end} \\
& | & p\ ;\ q \\
& | & p \parallel q \\
& | & \mathsf{trap}\ T\ \mathsf{in}\ p\ \mathsf{end} \\
& | & \mathsf{exit}\ T
\end{array}
$$

Most of the statements detailed above are classic and defined in books such as [Ber96]. nothing and pause respectively do nothing and pause (i.e., wait for) the current reaction. emit and await respectively send and await a signal. A variable v can be assigned and used in if and while which are generalizations of, respectively, a condition and a conditional loop with condition c based on boolean variable evaluation. Finally, the last statements express the more classic control-flow. We also omit signal and variable declaration for clarity.

5.3.1.3 Overview of the semantics

The behavioral semantics of ESTEREL is classically described in terms of rewriting rules, similarly to our native semantics described in the previous section. Each statement is rewritten to another one and reacts instantaneously to input signals with output signals. As in our native semantics, a statement rule can either take time or not; however, contrary to our native semantics, a rule taking time only takes one instant of some global base clock. Additionally, we consider a variation of the classical semantics taking into account variables and valued signals as shown in [PBEB07].

Definition 5.3.1 (Esterel semantics)**.** Given a constructive ESTEREL program P and a set of $data$, the semantics of its transition is given by the following relation:

$$P, data \xrightarrow[E_i]{E_o \quad k} P', data'$$

where E_i and E_o are, respectively, the input and output signal set active on the current reaction, and k is a boolean denoting the termination status (i.e., if it takes time or not).

Let’s describe more precisely the different statements given in the preceding section:

— nothing rewrites to itself without taking time;

— pause rewrites to nothing taking a single instant;

— emit S emits S synchronously and rewrites to nothing without taking time;

— $v := exp$ assigns a variable and rewrites to nothing without taking time;

— if exp then p else q end rewrites without taking time to p or q depending on the expression evaluation;

— while exp do p end rewrites without taking time to its body p followed by itself or to nothing depending on the expression evaluation;

— p ; q rewrites to q if p rewrites to nothing or to the rewriting of p followed by q;

— $p \parallel q$ rewrites to the parallel execution of both of its statements being rewritten and takes time only if one of them does;

— exit T is a special statement which does not take time but rather breaks the current control flow emitting a special signal;

— trap T in p end works with the previous statement catching special exit signals and rewriting to nothing in this case;

— await S takes an instant and rewrites to itself while the signal S is not present in the inputs. There are two variants of this statement: *immediate* if it can terminate instantaneously or *non immediate* if the first instant always takes time. In this work, we use the second variant with an additional counter n on the number of the awaited signal await n S.

More details on the ESTEREL semantics is given in appendix A.

5.3.2 ESTEREL translation principle

The translation principle is quite straightforward. All PSYC clock ticks are translated using ESTEREL signals. These "synchronization" signals are either application inputs (i.e., source clock), or are generated by periodic clock modules (i.e., periodic clock). Agents are translated to modules which only synchronize on these clock signals using the ESTEREL `await` statement, replacing the PSYC `advance` statement. Finally, the control flow of the agent is very similar in PSYC and ESTEREL considering that body declarations and statements can be simulated by ESTEREL `trap` and `exit` statements.

The main difference lies in the communication handling. In PSYC, an elementary action starts on a clock tick and terminates on another one. Any output value in an agent has to wait for the end of its elementary action to be displayed (i.e., made visible to others). Similarly, any input value in an agent has to be consulted at the start of its elementary action. In ESTEREL, all computations are virtually instantaneous with respect to some global instant. In our ESTEREL translation of PSYC, all computations are done synchronously to the start of their elementary action, thus reading input on the same instant. However, outputs have to be kept internally (i.e., in a variable) until the end of the elementary action, on which it is then emitted by a valued signal.

An application is then composed of multiple concurrent agent modules, clock modules as well as temporal variable modules. There is no causality issue as 1) clock signals are only used by agents and cannot be emitted by them, 2) circular clock definition is forbidden in PSYC and 3) communication signals are always emitted in a delayed fashion, thus breaking all potential causality loops.

5.3.3 Translation Rules

Sources and Clocks Source clocks are just pure input signals whereas periodic clocks are signals generated by a clock module defined with respect to another clock signal. It is composed of three `advance` statements: one to model the periodic delay, one to model the offset delay and an initial one to model the synchronization with the first tick of the input parent clock, only necessary when the latter is not present on the initial instant (i.e., translated by `immediate` in ESTEREL).

```
1  module Clock;
2    -- input clock
3    input clk_in;
4    -- output clock
5    output clk_out;
6    -- constants
7    constant Period, Offset : integer;
8
9    await immediate clk_in;
10   await [Offset mod Period] clk_in;
11   loop
12     emit clk_out;
13   each Period clk_in;
14 end module
```

Listing 5.2 – Esterel translation rule of PSYC clock

Temporal variables A temporal variable is modeled by a sampling component that fetches the corresponding signal displayed by the producer agent on each tick of the temporal clock and emits a sampled signal to consumer agents. The history of the temporal variable is modeled by additional sampling components that propagate previous values on each clock tick.

Let’s consider first the module below that implements a basic sampler.

```
1  module Sampler;
2    type T;
3    input In : T; input Clk;
4    output Out : T;
5
6    loop
7      emit Out(?In);
8      pause;
9      sustain Out(pre(?Out));
10   each Clk
11 end module
```

Listing 5.3 – Esterel sampler module

From this sampler module, we define the temporal module as N concurrent instances of the sampler to model a temporal variable with history of size N as shown below.

```
module Temporal;
  generic constant N: unsigned;
  type T;
  input C;
  input Temporal_In: T;
  output Temporal: T[N];
  [
    run Sampler[signal Temporal_Input/In, Temporal[0]/Out, C/Clk] ||
    run Sampler[signal pre(?Temporal[0])/In, Temporal[1]/Out, C/Clk] ||
    run Sampler[signal pre(?Temporal[1])/In, Temporal[2]/Out, C/Clk] ||
    ...
    run Sampler[signal pre(?Temporal[N - 2]/In, Temporal[N - 1]/Out, C/Clk)]
  ]
end module
```

Listing 5.4 – Esterel translation rule of PSYC temporal variable

Agent The declaration of an agent is more complicated as it is not possible to implement a generic module agent similarly to clocks or temporal variables. Hence, each agent has to be translated to a dedicated ESTEREL module. Let's consider an agent, named ID, that is composed of the body b_{start}, b_1, $...b_K$ and uses the clocks c_1, c_2, ..., c_J. Additionally, the temporal variables consulted (i.e., in input) by the agent are $temporal\_in_1$, $temporal\_in_2$ ..., $temporal\_in_N$ and the displayed ones (i.e., in output) are $temporal\_out_1$, $temporal\_out_2$, ..., $temporal\_out_M$. Their respective types are T_IN_i for inputs and T_OUT_i for output. Furthermore, inputs temporal variables also have a depth parameter $DEPTH_i$, which correspond to the history size. Finally, the agent starttime value is $STARTTIME\_VALUE$ while its clock is $STARTTIME\_CLOCK$. For simplicity, we first give the definition of the agent interface below.

```
interface Agent_ID_Intf:
  -- clocks
  input c1, c2, ..., cJ;
  -- inputs
  input temporal_in1 : T_IN1[DEPTH1];
  input temporal_in2 : T_IN2[DEPTH2];
  ...
  input temporal_inM : T_INM[DEPTHM];
  -- outputs
  output temporal_out1 : T_OUT1;
  output temporal_out2 : T_OUT2;
  ...
  output temporal_outN : T_OUTN;
end interface
```

Listing 5.5 – Esterel interface of PSYC agent

The main ideas of the agent translation are the following:

— Initially, the agent waits for its starttime.

— Then, after initializing its variables (i.e., its next body state, its private temporal variables and potentially its local variables) the agent enters an infinite loop in which the corresponding body is called — i.e., initially its `start` body;

— When a body has terminated, the control loops and the switch statement calls the corresponding next body;

— When a body has aborted — i.e., via a `jump` or an `endbody` statement — the trap catches the body exit and then the behavior is similar to the above.

```
1  type Agt_ID_bodies =
2    enum {BODY_B_START, BODY_B_1, ..., BODY_B_K};
3
4  module Agent_ID:
5    extends Agent_ID_Intf;
6
7    -- starttime
8    await STARTTIME_VALUE STARTTIME_CLOCK;
9
10   -- body handling
11   var body_target : Agt_ID_bodies := BODY_B_START,
12       private_temporal_out_1 : T_OUT_1 := ⟨init⟩,
13       private_temporal_out_2 : T_OUT_2 := ⟨init⟩,
14       ...
15       private_temporal_out_M : T_OUT_N := ⟨init⟩ in
16     loop
17       trap body_exit in
18         switch
19           case body_target = BODY_B_START do T(b_start)
20           case body_target = BODY_B_1 do T(b_1)
21           ...
22           case body_target = BODY_B_K do T(b_k)
23         end switch
24       end trap
25     end loop
26   end var
27 end module
```

Listing 5.6 – Esterel translation rule of PSYC agent

Note that each displayed temporal variable has copies of private variables. They correspond to the update of the temporal variables inside an agent. Body state is also handled by a dedicated variable `body_target`. As the body content b_i is actually a PSYC statement, $T(s)$ actually denotes the translation of a statement described in the next section.

Expressions and Statements PSYC nothing is translated the equivalent ESTEREL statement as shown below.

```
1    nothing
```

Listing 5.7 – Esterel translation rule of PSYC nothing statement

PSYC assignment of temporal variable x, $x := exp$, is translated by the ESTEREL assignment statement to the private variable corresponding to the selected temporal variable as shown below.

```
1  private_temporal_out_x := T(exp)
```

Listing 5.8 – Esterel translation rule of PSYC assignment statement to temporal variable

PSYC assignment of variable x, $x := exp$, is translated by the equivalent ESTEREL assignment statement to the variable as shown below.

```
1  var_x := T(exp)
```

Listing 5.9 – Esterel translation rule of PSYC assignment statement to private variable

PSYC advance statement, of the form advance n with c, and assuming that `var1,var2...varN` are all the output temporal variables of the agent, is translated by an ESTEREL await statement followed (logically instantaneously) by an emit statement for each output temporal variable as shown below.

```
1  await n c;
2  emit temporal_var1(private_temporal_var1);
3  emit temporal_var2(private_temporal_var2);
4  ...
5  emit temporal_varN(private_temporal_varN);
```

Listing 5.10 – Esterel translation rule of PSYC advance statement

PSYC statement next, next b, is translated by an assignment to the target body as shown below.

```
1  body_target := body_b;
```

Listing 5.11 – Esterel translation rule of PSYC next statement

PSYC statement, endbody, is translated by an ESTEREL exit statement as shown below. It jumps to the agent parent scope, which then dispatches the control to the corresponding body.

```
1  exit body_exit;
```

Listing 5.12 – Esterel translation rule of PSYC endbody statement

PSYC jump statement, jump b, is translated by a target body assignment followed by an exit statement as shown below.

```
1  body_target := body_b; exit body_exit;
```

Listing 5.13 – Esterel translation rule of PSYC jump statement

The PSYC sequence statement is translated by the equivalent ESTEREL sequence statement as shown below.

```
1  T(stmt1); T(stmt2)
```

Listing 5.14 – Esterel translation rule of PSYC sequence statement

PSYC condition statement, if (exp) $stmt1$ else $stmt2$ is translated by the equivalent ESTEREL condition statement as shown below.

```
1  if T(exp) then
2    T(stmt1)
3  else
4    T(stmt2)
5  end if
```

Listing 5.15 – Esterel translation rule of PSYC condition statement

The PSYC while statement, while exp do $stmt$, is translated by the equivalent ESTEREL while statement as shown below. It can be recreated with the core ESTEREL statements using the a combination of a trap/exit to model the exit condition and a loop.

```
1  while T(exp) do
2    T(stmt)
3  end
```

Listing 5.16 – Esterel translation rule of PSYC loop statement

PSYC agent expressions can be translated into an equivalent form in *Esterel*. However, we will not dive into the details of expression translation as it is out of the scope of this thesis. Indeed, in PSYC, most complex C expressions are actually defined outside of the language. The only specificity of PSYC expressions is the accesses to a temporal variable. This expression is expressed as $\$[n]$`temporal_in` which denotes the access to `temporal_in` on the n^{th} last sampled value. It is translated by the ESTEREL expression below which access to the input temporal variable array at the corresponding depth.

```
1  temporal_in[n]
```

Listing 5.17 – Esterel translation rule of PSYC temporal variable expression

Application The global PSYC application translation pattern is then the global parallel composition of all the modules defined above. Let's assume that we have an application composed of:

— the sources `s1, s2, ..., sn`

— the clocks `c1, c2, ..., cJ`;

— the temporal variables `tv1, tv2, ..., tvN` with the respective history depth `DPETH1, DEPTH2, ..., DEPTHN`;

— and the agents `ag1, ag2, ..., agM`.

```
1  module Application:
2    -- sources
3    input s1, s2, ..., sn;
4
5    signal tv1 : T1, tv2 : T2, ..., tvN : TN,
6           tv1_sampled : T1[DEPTH1],
7           tv2_sampled : T2[DEPTH2],
8           ...,
9           tvN_sampled : TN[DEPTHN],
10          c1, c2, ... cJ in
11   [
12     run Clock[signal c1/c, ⟨c1 parameters⟩] ||
13     run Clock[signal c2/c, ⟨c2 parameters⟩] ||
14     ...
15     run Clock[signal cJ/c, ⟨cJ parameters⟩] ||
16     run Temporal[signal tv1/Temporal_In,
17                         tv1_sampled/Temporal,
18                  ⟨tv1parameters⟩] ||
19     run Temporal[signal tv2/Temporal_In,
20                         tv2_sampled/Temporal,
21                  ⟨tv2parameters⟩] ||
22     ...
23     run Temporal[signal tvN/Temporal_In,
24                         tvN_sampled/Temporal,
25                  ⟨tvNparameters⟩] ||
26     run Agent_ag1 ||
27     run Agent_ag2 ||
28     ...
29     run Agent_agM
30   ]
31 end module
```

Listing 5.18 – Esterel translation rule of a PSYC application

In this translation pattern, agent parameters are given implicitly. This means that, in the agent modules, the exact parameter name should be used. Also, it may be surprising that there is no input nor output apart from the source clocks. But actually, in practice, inputs or outputs might be dedicated tasks (potentially agents). For semantics purposes, some temporal variables could be, however, made inputs or outputs of the application.

5.4 Equivalence between both semantics

Based on the introduced PSYC semantics and its ESTEREL translation, this section gives an observational equivalence between them for individual agents. Informally, both semantics define equivalent states on the boundaries of the elementary actions since the synchronous one only expand time interval into a succession of atomic transitions. By equivalent state, we mean that the syntaxical states from each semantics are equivalent with respect to the synchronous translation pattern and that the local environments from each semantics are also equivalent. However, the intermediate states of the synchronous semantics *inside* an elementary action cannot be compared to any states of the native semantics for individual agents; informally, they represent evolutions of the clock constraints but not the computations. This result is based on the PSYC operational semantics that has been given in chapter 5.2 and the ESTEREL semantics that is sketched in section 5.3.1.3.

We first consider equivalence of data between the PSYC native semantics and the ESTEREL translation. Similarly to the PSYC native semantics environment, the ESTEREL data environment corresponds to agent private variables, temporal variable copies, next body values ... Thus, we write $E \approx data$ which denotes that PSYC agent environment E is equivalent to ESTEREL environment $data$.

The equivalence theorem yields naturally from both semantics (PSYC and ESTEREL) and the data equivalence relation. Considering an sLET interval, the theorem states that both the PSYC and the ESTEREL representation have an equivalent behavior on the boundaries of the interval. The PSYC semantics yields only one transition while the ESTEREL semantics yields a sequence of unitary transitions (i.e., with respect to the source). If both data representation are equivalent at the start of the interval, then, they are also equivalent at the end.

Theorem 5.4.1. *For a* PSYC *agent* p_{ag} *and the* ESTEREL *translation* $T(p_{ag})$ *and for any agent transition of the form:*

$$C \vdash p_{ag} \Longrightarrow_{n \times s} C' \vdash p'_{ag}$$

Assuming, $C \approx data$ *and* $s \in E$*, we have an equivalent sequence of* ESTEREL *transitions:*

$$T(p_{ag}), data \xrightarrow{E}\!\!\!\!\!\rightarrow P^1, data^1 \ldots \xrightarrow{E}\!\!\!\!\!\rightarrow P^n, data^n$$

such that $T(p'_{ag}) = P^n$ *and* $C' \approx data^n$.

Proof.

By structural induction on the structure of the native rules. The full proof is given in appendix A.

□

Note that theorem D.1 does not consider stuttering, that is, transitions without source tick. However, this result extends easily to allow stuttering, ESTEREL transitions do not change the program when no source tick is present. Additionnally, this equivalence result only adresses a single agent while an application is actually a system of multiple agents, as described in both

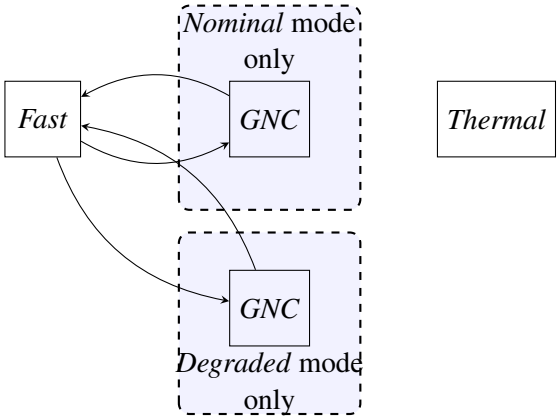
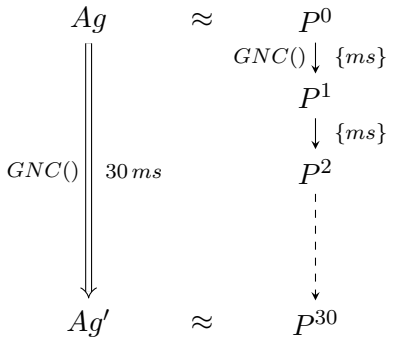


Figure 5.3 – The GNC functional architecture

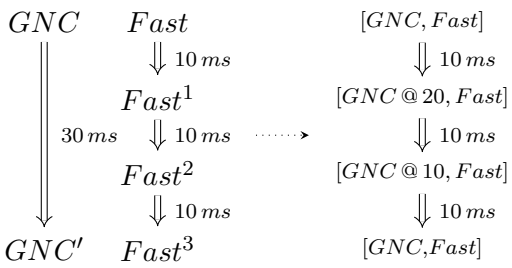
semantics. However, as this equivalence holds for each agent, it naturally generalizes multiple agents.

5.5 Illustration

As an illustration of both semantics, we shall consider a very simplified version of the control system of a spaceship introduced in [CPBL+15], called a Guidance, Navigation and Control system (GNC). This use-case is composed of three tasks: *Fast* which interacts with the sensors and actuators, *GNC* which is responsible for the control and guidance functions and *Thermal* which is responsible for the thermal regulation. *GNC* has two different modes, one nominal mode and one degraded mode that has different period constraints. The functional architecture is described in Figure 5.3.



(a) GNC semantics simulation diagram



(b) GNC agent network semantics

```
clock c10ms = 10 * source_ms;
clock c100ms = 10 * c10ms;
...

agent GNC(starttime 1 with
 c100ms) {
  display updateCommands : all;
  consult 1 $ filteredSensors;
  consult 1 $ mode;
  body start {
    if ($[0]mode == NOMINAL) {
      /* compute GNC */
      updateCommands =
        GNC($[0]filteredSensors);
      advance 3 with c10ms;
    }
    advance 1 with c100ms;
  }
}
```

(a) PsyC implementation of task GNC

```
run Clock[signal source_ms, c10ms,
          constant 10, 0] ||
run Clock[signal c10ms, c100ms,
          constant 10, 0] ||
...
module Agent_GNC:
  ...
  await 1 c100ms;
  var private_updateCommands :
   t_cmd in
    loop
      if mode[0] = NOMINAL then
        -- compute GNC
        run GNC[signal
          filteredSensors[0],
          private_updateCommands];
        await 3 c10ms;
        emit update_commands(
          private_updateCommands);
      end if;
      await 1 c100ms;
    end loop
  end var
end module
```

(b) Esterel translation of task GNC

Figure 5.5 – Task GNC

Based on the functional and temporal requirements, the *GNC* task could be implemented in PSYC as detailed in Figure 5.5a and based on the ESTEREL translation, it can be translated to ESTEREL as shown in Figure 5.5b.

In nominal mode, the native semantics yields the following infinite trace for agent *GNC*:

$$C \vdash ag \Longrightarrow_{100} C^1 \vdash ag^1 \xRightarrow{GNC()}_{30} C^2 \vdash ag^2 \Longrightarrow_{70} C^3 \vdash ag^3 \xRightarrow{GNC()}_{30} C^4 \vdash ag^4 \ldots$$

The source `source_ms` is omitted for clarity and the function *GNC()* has been annotated on its corresponding transition. If we take this transition, the ESTEREL translation yields the following unit transitions:

$$P, data \xrightarrow{GNC() \quad E^1} P^1, data^1 \xrightarrow{E^2} P^2, data^2 \ldots \xrightarrow{E^{30}} P^{30}, data^{30}$$

where $data$ contains $filteredSensors$ used for the computation in the first transition which outputs the updated $updateCommands$ value only in $data^{30}$, thus having an equivalent behavior as the native semantics described above. The simulation diagram in figure 5.4a illustrates this equivalence at the agent level while figure 5.4b illustrates how the parallel operator from section 5.2.2 allows building a unique automaton from a network of agent automata.

THIRD PART

Formal Verification for synchronous LET

For it would suffice for them to take their pencils in their hands and to sit down at the abacus, and say to each other : ***calculemus****.*

— G.W.Leibniz, Calculus Ratiocinator (1670).

CHAPTER 6

Temporal Requirements

In the first part, we detailed the need for an approach based on logical time to specify precise and deterministic temporal constraints, mandatory for safety-critical domains. In the second part, we gave the formal semantics of PSYC based on synchronous LET, a formalism inherited from both Synchronous-Reactive and Logical Execution Time. PSYC allows the temporal behavior of such programs to be specified precisely.

More specifically, the PSYC language focuses on the specification of explicit temporal constraints *inside* sequential agents (i.e., using the `advance` statement). These agents then communicate together through dedicated dataflow channels. Hence, more global temporal requirements such as the relative rate of multiple agents or the end-to-end latency of communication chains are largely implicit. The goal of this chapter is to formalize a set of common temporal requirements that will be used in the following chapters in PSYC formal verification problems.

6.1 Temporal requirements for PSYC

Formally, the synchronization instants of agents are also logical clocks. We can also consider a subset of them corresponding to the activation instants of some specific function, as well as some conditional activation corresponding to a PSYC conditional branch. In this chapter, we assume that we already have these clocks modeling some agent instants. The next chapter clarifies the link between these clocks and the PSYC code. Considering that the execution of a PSYC application can be represented as a set of logical clocks, a PSYC application is actually multi-clock even if all agents are based on a unique source clock. Thus, the requirements that we want to specify are mainly temporal constraints among multiple agents. They specify constraints among the rates of multiple agents or constraints among the data propagation among multiple agents.

In this thesis, we focus on four kinds of requirements inspired by specification languages such as TADL2 [PFGDN12] or AUTOSAR Timing extensions [SAG12]:

— *Repetition requirements* specify a constraint on the repetition of successive instants of a single logical clock. They are typically used to show the periodicity of a clock.

— *Synchronization requirements* specify the synchronizations among multiple logical clocks. They are typically used to show that multiple concurrent computations are triggered on the same instant. More generally, we consider that synchronization requirements correspond to invariants among multiple agents. This includes synchronization, but also exclusion or subclocking constraints.

— *Causality requirements* specify that the instants of an agent *precede* the instants of another agent. More generally, causality requirements can specify alternation patterns between two agents. They are often used to specify communication patterns between agents such as *no data is lost* or *the consumer is twice as fast as the producer*.

— *Delay (or latency) requirements* specify the delay between a *stimulus* instant and a *response* instant. Delay requirements can be applied either on two instants of a single agent or on instants of multiple agents to model the propagation of data. If the delay is applied for each *response instant*, this requirement is often called *Freshness*, and if the delay is applied for each *stimulus* instant, this requirement is often called *Reaction*. The former is usually used in systems such as *automatic cruise control* in which the age of data is really important, and the latter is usually used in systems such as *airbag control* in which the presence of a response is critical.

In addition to these requirements, agent instants can be causally related to model agent communication. This is often called *functional chain* or *event chain* [FRNJ09]. Informally, if a data item is *tagged* and used as a *stimulus* to the model, assuming that the tag is propagated in all computations, then its corresponding *response* also contains the tag. However, defining *functional chain* usually faces two issues:

1. When a consumer has a higher frequency than a producer, then some values are read multiple time. This is called *oversampling*. In this case, the functional chain is split into multiple branches, going to different consumer instants.
2. When a producer has a higher frequency than a consumer, then some values are overwritten. This is called *undersampling*. In this case, the functional chain faces a "*dead-end*". The data is not propagated.

Usually, in the literature [FRNJ09], a *Reaction* requirement considers only the first branch derivation (issue 1) and a *dead-end* is considered to be an error, due to infinite latency (issue 2). On the contrary, a *Freshness* requirement considers all the branch derivations (issue 1) that yield a *response*; so *dead-ends* are ignored (issue 2).

As mentioned in the introduction, the most common logics used to formalize requirements are temporal logics. However, in our case, the requirements are expresses as constraints among multiple logical clocks — the agent synchronization instants. Thus, we use CCSL as it naturally express constraints among different logical clocks whereas the operators of temporal logics focus on a single timeline. The next section gives an overview of the CCSL specification language.

6.2 The CCSL language

This section gives an overview of the Clock Constraint Specification Language (CCSL) language inspired by the presentation given in [MDS15].

6.2.1 Time model

A CCSL specification is defined as a set of logical clocks, as defined in Chapter 4, constrained by a set of constraints defining when the clocks can tick. The definition of these logical clocks is the same as the one used for synchronous LET; it is a sequence of instants, totally ordered, that can represent any action of the system (e.g., task activation, communications . . .). The CCSL constraints specify causal and simultaneity constraints between clocks. They are used to constrain the clocks either for synthesis (finding a schedule of the clocks) or verification (verify the constraint on the system events).

Definition 6.2.1 (CCSL clock)**.** A CCSL clock is a logical clock defined by an infinite sequence of instants $(c_i)_{i\in\mathbb{N}^*}$ (i.e. the clock *ticks*).

Definition 6.2.2 (CCSL specification)**.** A CCSL specification is defined as a tuple $\langle C, Cons\rangle$, where C is a finite set of logical clocks and $Cons$ a finite set of constraints.

A *schedule* of a CCSL specification states when clocks tick in the system. It assigns a date to each tick of a clock set using a common time basis. It can be seen as a possible execution of the model, and is considered *valid* only if it satisfies the constraints of the specification defined in the next section.

Definition 6.2.3 (CCSL Schedule)**.** The schedule of a set of clocks C is a function $\sigma \in \mathbb{N} \to 2^C$ in which $\sigma(s)$ denotes the set of clocks that tick at step s. It is *valid* if for a given specification $\langle C, Cons\rangle$ it satisfies all the constraints $\forall cons \in Cons, \sigma \models cons$.

We also define the state of a clock that we call *history*. Informally, the history of a clock at an instant i with respect to some schedule is the number of ticks the clock has made before, and including, this instant.

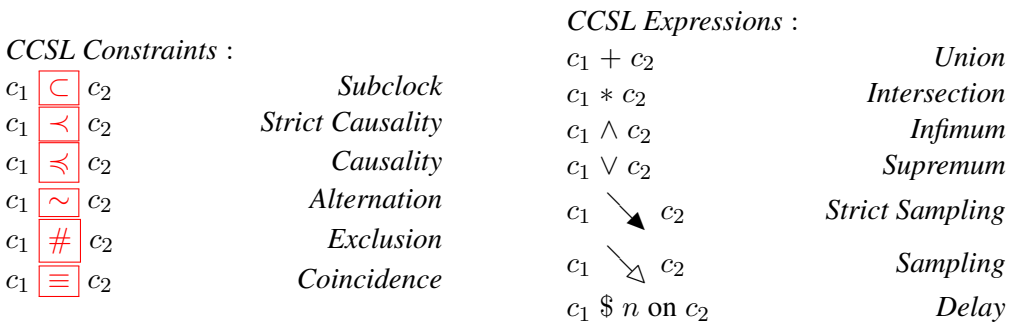


Figure 6.1 – Abstract syntax of CCSL

Definition 6.2.4 (CCSL history)**.** For a schedule σ, a history of a clock is a function $\chi_\sigma : C \times \mathbb{N} \to \mathbb{N}$ defined on a set of clocks C such that:

$$\begin{aligned} \chi_\sigma(c,0) &= 0 \\ \forall n \in \mathbb{N}^*, \chi_\sigma(c,n) &= \begin{cases} \chi_\sigma(c,n-1)+1 & \text{if } c \in \sigma(n) \\ \chi_\sigma(c,n-1) & \text{otherwise} \end{cases} \end{aligned}$$

In the literature, CCSL is often used in *synthesis* problems: one wants to find an execution, or a *schedule* which satisfies the specification. But CCSL can also be used in *verification* problem. In that case, the clocks are mainly mapped to events of the system to be verified. Thus, they already have schedules corresponding to system executions. In our case, the events of the system are the *agent clocks*.

Definition 6.2.5 (CCSL verification)**.** A system *satisfies* a CCSL specification $\langle C, Cons \rangle$ if and only if ever execution path of the system σ_{system} is a *valid* schedule (thus, satisfying $Cons$).

Additionally, CCSL clocks can also be refined using expressions. The next section describes both CCSL constraints and expressions.

6.2.2 Constraints and Expressions

There are two kinds of CCSL constraints: stateless and stateful. The former only relies on the presence or absence of clock ticks while the latter also relies on the state of the clock. Figure 6.1 shows the syntax for the main constraints and expressions of CCSL that we use in this work.

A first basic constraint is the subclock constraint, which states that all the ticks of a clock coincide with those of another clock.

Definition 6.2.6 (subclock)**.** Let a, b be two logical clocks, then a is a subclock of b (noted $a \subseteq b$) if: $a \subseteq b \triangleq \forall i \in \mathbb{N}, a \in \sigma(i) \implies b \in \sigma(i)$

We will also use the exclusion constraint which states that two clocks never have ticks that coincide.

Definition 6.2.7 (exclusion)**.** Let a, b be two logical clocks, a is in exclusion with b (noted $a\#b$) if: $a\#b \triangleq \forall i \in \mathbb{N}, \neg(a \in \sigma(i) \wedge b \in \sigma(i))$

Some other constraints are stateful. As an example, the causality constraint states that for a clock on a given index, a clock instant a_i always happen before another clock instant b_i.

Definition 6.2.8 (causality)**.** Let a, b be two logical clocks, a causes b (noted $a \preccurlyeq b$) if: $a \preccurlyeq b \triangleq \forall i \in \mathbb{N}, \chi(a_i) \leq \chi(b_i)$.

The causality constraint can be made strict (noted $a \prec b$) when a strict order is used between histories of both clocks.

Basic logical operators can be used between clocks as expressions. They are applied to each instant of a schedule.

Definition 6.2.9 (union/intersection)**.** Let a, b and c be three logical clocks, c is the intersection (resp., union) of a and b (noted $c \triangleq a \wedge b$, resp., $\vee$) if: $\forall i \in \mathbb{N}, c \in \sigma(i) \iff a \in \sigma(i) \wedge b \in \sigma(i)\ (resp., \vee)$.

Another expression, more specific to CCSL is the Infimum (resp., supremum) expression which defines the slowest (resp., fastest) clock of two other clocks.

Definition 6.2.10 (infinum/supremum)**.** Let a, b and c be three logical clocks, c is the infinum (resp., supremum) of a and b (noted $c \triangleq a \wedge b$, resp., $\vee$) if: $\forall i \in \mathbb{N}, \chi(c,i) = min(\chi(a,i), \chi(b,i))\ (resp., \max)$

CCSL also defines a sampling operator which defines a clock corresponding to a sampling of one clock to another. This can be used to model sampling communication.

Definition 6.2.11 (sampling)**.** Let a, b and c be three logical clocks, c is the (strict) sampling of a on b (noted $c \triangleq a \searrow b$) if: $\forall i \in \mathbb{N}, c \in \sigma(i) \iff (\exists j < i, b \in \sigma(j) \wedge b \in \sigma(i) \wedge (\chi(b,i) - \chi(b,j)) = 1 \wedge (\exists k, j \leq k < i, a \in \sigma(k)))$. If sampling is non strict, the ordering is $j < k \leq i$.

Finally, we also define the CCSL delay operator which is used for latency and delay constraints.

Definition 6.2.12 (delay)**.** Let a, b, and c be three logical clocks, c is a delayed by $N \in \mathbb{N}^*$ based on clock b (noted $c \triangleq a\ \$\ N$ on b) if: $\forall i \in \mathbb{N}, c \in \sigma(i) \iff b \in \sigma(i) \wedge \exists j \leq i, a \in \sigma(j) \wedge (\chi(b,i) - \chi(b,j) = N)$.

We also often use a clock which is delayed on itself. We write it $a\ \$\ N$ which is equivalent to $a\ \$\ N$ on a.

Various extended constraints and expressions operators can be derived using the basic one defined above. Coincidence is derived by a double subclocking - $a \boxed{\equiv} b$ is true when $a \boxed{\subset} b$ and $b \boxed{\subset} a$ - and (strict) alternation is derived by a double causality with delay - $a \boxed{\sim} b$ is true when $a \boxed{\prec} b$ and $b \boxed{\prec} (a \$ 1)$. For an in-depth introduction to CCSL, the reader might consult [And09].

6.2.3 Synchronous observers

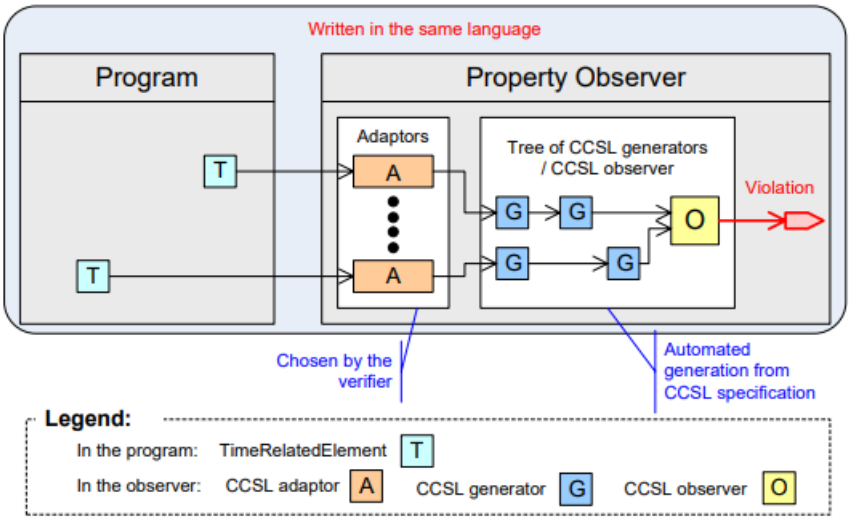


Figure 6.2 – *Synchronous Observer* approach using CCSL [And10]

As explained in [And10], CCSL can also be used along with a system model to perform verification. Traditionally, in the Synchronous-Reactive community, properties to be verified are expressed as *synchronous observers*. They are synchronous components that *observe* the system's behaviors and raise an *error* signal if the property is invalid. This approach is actually quite similar to more traditional model-checking in which a language acceptor — usually a Buchi automata — validate the system's traces. In [And10], CCSL constraints are translated to *synchronous observers* encoded in the ESTEREL language using the CCSL semantics. Figure 6.2 is extracted from [And10] and gives an overview of the methodology: CCSL constraints are translated to observers and CCSL expressions to generators; both are encoded in ESTEREL and connected with the main ESTEREL program. Full verification methodology is detailed in next chapter and uses this encoding along with the ESTEREL translation of PSYC given in chapter 5.

6.3 Encoding requirements in CCSL

6.3.1 Repetition requirement

The repetition constraint is basically a delay constraint between successive instants of a clock. In its most simple form, the repetition constraint can be defined as a coincidence with some periodic clock.

$$\text{Repeat}(c, P, O, b) \triangleq c \boxed{\equiv} c_{P \times b + O} \tag{6.1}$$

where $c_{P \times b + O}$ is a periodic clock defined with respect to b with a period P and an offset O.

However, if we want to omit the offset parameter, we have to specify a constraint in which two successive instants shall *exactly* have a delay of a *period* respectively to some base clock. It is defined as following:

$$\text{Repeat}(c, P, b) \triangleq c \$ P \text{ on } b \boxed{\equiv} c \$ 1 \tag{6.2}$$

where P is the period and b is the base clock.

The constraint can be generalized with a delay interval between successive instants defined by P_{min} and P_{max}.

$$\text{Repeat}(c, P_{min}, P_{max}, b) \triangleq c \$ P_{min} \text{ on } b \boxed{\preccurlyeq} c \$ 1 \boxed{\preccurlyeq} c \$ P_{max} \text{ on } b \tag{6.3}$$

There exists multiple variations of the definitions above such as defining repetition by a jitter on some reference clock. In this work, we only use the basic definitions above, but variations can also be defined using CCSL.

6.3.2 Synchronization requirements

The synchronization constraint, in its strict form, can simply be defined as a coincidence between clock.

$$\text{Synchronization}(c_1, c_2, \ldots c_n) \triangleq (c_1 \boxed{\equiv} c_2) \wedge (c_2 \boxed{\equiv} c_3) \wedge \ldots (c_{n-1} \boxed{\equiv} c_n) \tag{6.4}$$

Additionally, the constraint can be extended with a tolerance parameter that defines the delay between the first and the last synchronized clock. This is defined using the *inf* and *sup* parameters to define the first and last clock a synchronization:

$$c_{min} \triangleq \bigwedge_{i=1}^{n} c_i \tag{6.5}$$

$$c_{max} \triangleq \bigvee_{i=1}^{n} c_i \tag{6.6}$$

$$\text{Synchronization}(c_1,\, c_2,\, \ldots\, c_n,\, T,\, b) \triangleq c_{max} \preccurlyeq c_{min} \$\ T \text{ on } b \tag{6.7}$$

Where T is the tolerence parameter defined on b, a base clock.

6.3.3 Causality requirements

We also consider variations of causality requirements. In its most basic form, causality is directly represented with the causality operator in CCSL:

$$\text{Causes}(c_1, c_2) \triangleq c_1 \preccurlyeq c_2 \tag{6.8}$$

However, implementing this operator would require an unbounded counter. Hence, we consider only bounded variations of this operator mostly based on the alternation constraint.

6.3.4 Delay and Latency requirements

The delay constraint is probably the most complex due to the multiple semantic variations. We consider first a strict delay constraint in which the delay between two instants with index i of two clocks $c_{stimulus}$ and $c_{response}$ is bounded by some interval $[D_{min}, D_{max}]$:

$$\begin{aligned} &StrictDelay(c_{stimulus}, c_{response}, D_{min}, D_{max}) \triangleq \\ &(c_{stimulus} \$\ D_{min} \text{ on } b) \preccurlyeq c_{response} \preccurlyeq (c_{stimulus} \$\ D_{max} \text{ on } b) \end{aligned} \tag{6.9}$$

However, in some situations such as reaction requirements, one only wants to specify the delay between the instants of $c_{stimulus}$ and the *upcoming* instants of $c_{response}$. In other words, we also want to specify for a given instant of $c_{stimulus}$, the minimal and maximal delay up to the next instant of $c_{response}$. We call it forward delay and define it as follows:

$$\begin{aligned} &ForwardDelay(c_{stimulus}, c_{response}, D_{min}, D_{max}) \triangleq \\ &(c_{stimulus} \$\ D_{min} \text{ on } b) \preccurlyeq (c_{stimulus} \searrow c_{response}) \preccurlyeq (c_{stimulus} \$\ D_{max} \text{ on } b) \end{aligned} \tag{6.10}$$

Alternately, in freshness requirements, one may also want to specify the delay between the instants of $c_{response}$ and the *preceding* instants of $c_{stimulus}$. We call it *backward* delay because we use the $c_{response}$ clock as a reference. In this semantics, unlike *Forward* or *Strict* delay, a stimulus event might not be mapped to a response event; however, all response events should have a preceding stimulus event. We specify here the maximal delay between a given instant of $c_{response}$ and the previous instant of $c_{stimulus}$ as following:

$$\begin{aligned} &BackwardDelay(c_{stimulus}, c_{response}, D_{max}) \triangleq \\ &(c_{response} \searrow (c_{stimulus} \$\ D_{max} \text{ on } b)) \preccurlyeq (c_{response} \$\ D_{max} \text{ on } b) \end{aligned} \tag{6.11}$$

Additionnally, as the two constraints *ForwardDelay* and *BackwardDelay* are used for end-to-end latency, they need to focus on a specific data propagation in the system. The next section details how to extract these paths using functional chain definitions.

6.3.5 Functional chains

We now define functional chains that can be used with the requirements defined in the previous sections. Informally, a functional chain represents a possible data propagation in a system modeled by a sequence of task writings and readings (hence, by TSPs as defined in Chapter 4). The modeling of such requirements is rather complex due to the existence of multiple data propagation paths, due to communication over/under-samplings, and it depends on the communication semantics of the system. In this thesis, we consider a semantics based on a buffer-like communication using PSYC temporal variables and sLET synchronization.

We give the semantics of a functional chain composed of tasks t^1, t^2, ... t^n. These tasks are composed of two events t^i_{start} and t^i_{end} corresponding to the activation and deadline instants of the elementary actions of the agents.

A functional chain is composed of specific instances of these tasks corresponding to a specific data propagation path. We note them p^1, p^2, ... p^n for tasks t^1, t^2, ... t^n and we define their start and termination to be a subset of the corresponding task instants:

$$\forall i \in [1, n],\ p^i_{start} \boxed{\subset} t^i_{start} \wedge p^i_{end} \triangleq p^i_{start} \searrow t^i_{end}$$

We then constrain the instants of p by introducing propagation rules that relate producer and consumer instants. The producer termination instant is related to any of the upcoming consumer starting instants before the next producer termination instant (in which a new data item is propagated). This is defined by the two following constraints:

$$\forall i \in [1, n-1],\ p^i_{end} \boxed{\preccurlyeq} p^{i+1}_{start} \qquad \text{(Causality)}$$

$$\forall i \in [1, n-1], p^{i+1}_{start} \boxed{\prec} (p^i_{end} \searrow t^i_{end}) \qquad \text{(Consistency)}$$

The *causality* constraint states that the producer instant should happen before the consumer instant. The *consistency* constraint states that the consumer instant should happen before the next production instant.

Then, for all unique input data $x = p^1_{start}$; the constraints above define all the possible data propagation paths in which p^n_{end} is the functional chain output. By default, the above functional functional chain definition defines all the possible propagation paths; however, in some end-to-end latency semantics, one might want to take only the first propagation. To do that, one might add constraints to retain only a specific propagation path depending on the desired end-to-end latency semantics.

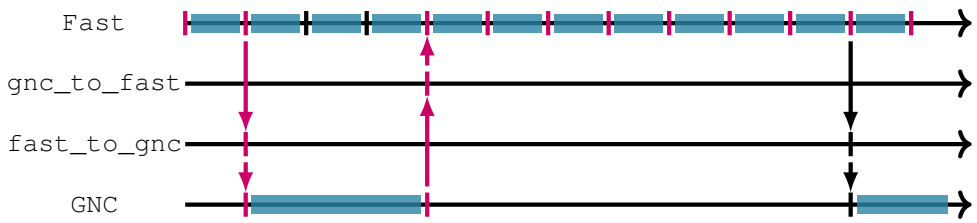


Figure 6.3 – Chronogram of GNC illustrating the end-to-end latency requirement

Additionally, the rule above assumes that the producer and consumer communicate directly. However, in practice, PSYC uses communication means such as temporal variables that can increase latency. The rules are rather generic and can be adapted to introducer temporal variables tv^i, as they can be modeled as synchronous tasks (i.e., $tv^i = tv^i_{start} = tv^i_{end}$).

6.4 Example

Let's consider again the GNC example shown in Chapter 4 and 5. On such use cases, the typical requirements are

— a repetition requirement on the `Fast` agent as a period of 1 millisecond;

— a causality requirement between `Fast` and `GNC` to specify that `Fast` should be 10 times faster than `GNC`. Thus, one value is actually used by `GNC` every 10 occurrences of `Fast`;

— a reaction end-to-end latency requirement between `Fast`, `GNC` and back to the `Fast` task which should be smaller than 130 milliseconds.

Figure 6.3 illustrates the end-to-end latency requirement on the GNC example. The red instants and arrows represent the possible propagation of the functional chain. Such functional chain can be modeled in CCSL using the definitions shown in section 6.3.5, on which the `Delay` definitions from section 6.3.4 can be used.

6.5 Summary

This chapter gave formal definitions of the most common requirements that interest us, in the context of synchronous LET models and the PSYC language. Most of these properties are highly inspired from AUTOSAR related work, such as TADL2. Regarding CCSL, all these definitions are also pretty classical except the end-to-end latency requirement.

CHAPTER 7

Verification: General Case

Two formal semantics were given for PSYC in the second part of this thesis: a native semantics preserving the logical durations of elementary actions and a synchronous semantics in which elementary actions are expanded into a succession of atomic transitions. In this chapter, we consider the synchronous semantics for formal verification as it naturally generalizes to PSYC programs with multiple source clocks and directly supports formal models such as Mealy machines. We detail how CCSL temporal requirements — described in chapter 6 — modeled as synchronous observers can be verified on a PSYC program. To do that, we encode both the observers and the program as Symbolic Transition Systems (STS) — which correspond to symbolic representation of Mealy machines — and then give them to symbolic model-checkers.

However, this approach has an obvious drawback; synchronous Logical Execution Time uses logical time to specify both the triggering instants and the durations of computations. PSYC programs often have long durations, which are split in a succession of atomic reactions by the synchronous semantics. This yields a lot of unnecessary states that makes model-checking blow-up. The next chapter discusses how the native semantics can be used to compress the state space using techniques inspired from timed automata model-checking.

7.1 Background

7.1.1 Digital circuits

In this section, we consider digital circuits with data similarly to [PBEB07]. They are composed of *combinatorial gates* (AND, OR ...) and *registers* that store wire values between reactions. Gates and registers are connected using wires that carry values between a source gate (or a register) and a destination one using data dependencies. Wires also contain the inputs, in which the value is defined by the environment, not by a data dependency. In addition to the classical boolean digital circuits, we also consider an extension based on valued variables with specific gates corresponding to variable tests and assignments.

More formally, a digital circuit $\mathcal{C}$ is defined by a set of wires $\mathcal{W}$, a subset of input wires $\mathcal{I} \subseteq \mathcal{W}$, a set of typed variables $\mathcal{V}$, and a set of causal dependencies. Each causal dependency is defined by a relation between two wires (w_1, w_2), where $w_1 \in \mathcal{W}$ should be available before $w_2 \in \mathcal{W}$. Gates are defined by boolean functions, giving an evaluation for output wires with respect to input wires. Additionally, registers can be considered as specific output wires used as input wires at the next reaction. Finally, we extend the classical definition of boolean circuits by considering specific valued gates — represented by buffers in the graphical diagrams — defining either the status of the output wire depending on some variable evaluation — value test — or assigning data depending on the status of the input wires — value assignment.

7.1.2 Circuit semantics of Esterel

Historically, the ESTEREL Inria compiler had two main compilation techniques: automata-based and circuit-based [PBEB07]. The former generates a global automata based on the ESTEREL behavioral semantics while the latter generates a digital circuit represented as a netlist based on the ESTEREL circuit semantics. In practice, automata-based compilation techniques do not scale to reasonable applications because the generated automata can grow exponentially due to concurrency. Circuit-based compilation, while being usually slower at runtime, scales better and is thus, more adapted to formal verification. Formally, a digital circuit is equivalent to a symbolic representation of Mealy machine that we call a symbolic transition system. It is detailed in the next section. The ESTEREL circuit semantics is used both for PSYC programs using the synchronous semantics and for CCSL models as synchronous observers. We now give a brief overview of the ESTEREL circuit semantics.

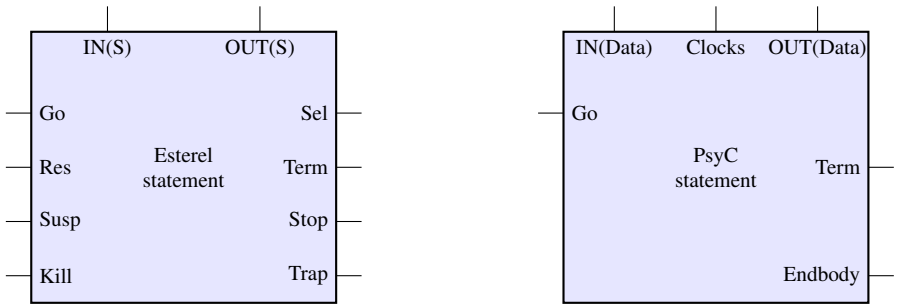


Figure 7.1 – The leftmost circuit models the interface of a ESTEREL statement while the rightmost circuit models the interface of an equivalent PSYC statement.

The interface of an ESTEREL statement is shown in the leftmost circuit of Figure 7.1. The left pins are basically inputs while the right pins are outputs. Additionnally, The top part correspond to input and outputs signals/environment interactions. The meaning of the pins is the following:

— the `Go` pin starts a statement;

— the `Res` pin resumes the execution of a statement;

— the `Susp` pin suspends the execution of a statement;

— the `Kill` pin unset the registers of the statement due to the exit of a trap statement;

— the `Sel` pin indicates if an internal register is set;

— the `Term`, `Stop`, and `Trap(s)` pins are used as completion codes (originally `K0-2`) of the statement. It means that the statement has respectively *terminated*, *stopped* due to a pause statement or *exited* from a trap.

7.1.3 Symbolic Transition System

Most modern model-checking tools use a symbolic representation of the state space to avoid dealing with the expanded state space in input. Some model-checkers, such as CADP [GLMS11], then expand the state space to verify properties — they are called *enumerative* — while others, such as NUXMV [CCD+14] or PROVER PSL [pro23], use symbolic verification techniques to avoid expanding the state space — they are called *symbolic*. In practice, *symbolic* model-checkers scale much better for Synchronous-Reactive languages [Ray08]. We define in this section the concept of Symbolic Transition Systems (STS) as a common input formalism to the two model-checkers that we use — NUXMV and PROVER PSL — as they take a syntactical variation of STS as input, namely SMV and HLL.

Symbolic Transition Systems are, informally, just a symbolic representation of state machines such as Mealy machines. The idea is that a state is modeled by a valuation of a finite set of variables. Initialisation and transitions are then both modeled by predicates defining respectively possible initialisation valuations and possible valuation transitions. The ESTEREL circuit semantics can directly be modeled as STS considering that registers and circuit variables are encoded in STS state variables, and wires — as well as well as gate definitions — are encoded in equations.

Definition 7.1.1 (Symbolic Transition System (STS))**.** A symbolic transition system is a tuple $\langle \mathcal{V}, Init, Trans \rangle$ where:

1. $\mathcal{V}$ is a finite set of (finite) typed variables for which a valuation represents a state of the system;
2. $Init(\mathcal{V})$ is a predicate representing the initial states;
3. $Trans(\mathcal{V}, \mathcal{V}')$ is a predicate representing the state transitions.

As an example, consider a clock of period P. The following STS models the dynamic behavior of this clock:

$$\begin{aligned} \mathcal{V} &= \{c, t\} \\ Init &\triangleq c = 0 \\ &\wedge\, t = true \\ Trans &\triangleq c' = (c+1) \bmod P \\ &\wedge\, t' = (c' = 0) \end{aligned}$$

where c is an integer defined on $[0; N[$ and t is a boolean which denotes the clock ticks.

7.1.4 Symbolic Model-checking

In our approach, symbolic model-checking uses the synchronous composition of an observer and a PSYC program modeled as STS. Basically, these tools only perform a reachability analysis on the observer output to ensure that violation of the observer never happens. Additionally, observers can also be used to model assumption of the environment. In that case, the output of the requirement observer is checked only if the output of the "assumption" observer is correct. This approach is often called "assume-guarantee" [BCE+03].

Definition 7.1.2 (STS Reachability)**.** Consider a program $\langle \mathcal{M}, MInit, MTrans \rangle$ and a property $\langle \mathcal{P}, PInit, PTrans \rangle$. A bad state is never reachable if and only if $\forall n \in \mathbb{N}^*$

$$Init_0 \wedge Trans_1 \wedge Trans_2 \wedge ... \wedge Trans_n \models OK_n$$

where:

— $Init_0 = MInit(M_0) \wedge PInit(P_0)$

— $Trans_i = MTrans(M_{i-1}, M_i) \wedge PTrans(P_{i-1}, P_i)$

— OK_i is the *ok* signal in P_i (at instant i)

In the literature, there are a lot of different techniques to perform reachability analysis on STS-like models. Let's describe the main ones:

— *Enumerative* technique: the traditional approach is to compute the set of reachable states $\mathcal{R}$ by successively applying the transition formula until a fixpoint is reached [Cla97]. Then, one has to ensure that *bad* states $\mathcal{B}$ are not contained in the reachable set $\mathcal{R} \cap \mathcal{B} = \emptyset$. This approach — called *forward reachability* — tends not to scale as it explicitly computes the reachable set.

— *Binary Decision Diagram* techniques (BDD): BDD model-checkers implement a variation of *forward reachability* in which the set of reachable state is represented symbolically using Binary Decision Diagrams [BCM+92]. These data structures provide a more efficient representation and manipulation on the state space. NUXMV is a model-checker that implements *BDD forward reachability*.

— *Bounded Model-Checking* technique (BMC): Bounded model-checking encodes a variation of reachability definition to check successively if the property is true until a bound $k \in \mathbb{N}^*$, using SAT or SMT solvers [BCCZ99]. Both PROVER PSL [pro23] and KIND2 [CMST16] are model-checkers that implement BMC, the former uses SAT while the latter uses SMT.

— *Induction* techniques (IND): induction techniques try to solve the incompleteness of BMC [Bra11]. Instead of verifying the property until a bound k, the property is proven using an induction principle. Basically, the idea is to show that:

1. $Init_0 \models OK_0$
2. $\forall i \in \mathbb{N}^*, \quad Trans_i \wedge OK_{i-1} \models OK_i$

Model-checkers like PROVER PSL [pro23] and KIND2 [CMST16] use a variation of this induction principle called *K-induction* to prove properties using SAT and SMT. As a complement to induction, they also both support invariant generation and interpolation techniques.

7.2 Translating PSYC to Symbolic Transition Systems

This section gives the translation rules to encode PSYC programs as STS. To do that, we combine the *synchronous semantics* of PSYC and the ESTEREL circuit semantics sketched in the preceding section. Encoding of CCSL observers in STS is beyond the scope of this thesis, but is described in [And10].

7.2.1 Extending PSYC agents with verification signals

For verification, agents should be extended with dedicated signals used for verification. They are logical clocks modeling the rate of agents or one of its functions. We extend PSYC using three different annotations:

1. First, we can name `advance` statements using the syntax

   ```
   @name, advance ... with ...;
   ```

   When the `advance` statement terminates, a signal is emitted named after the label.

2. Second, we add `probe` statements in PSYC code using the syntax

```
probe @name;
```

When the preceding statement terminates, a signal is emitted named after the label.

3. Finally, as data is not evaluated in our STS encoding, we can also annotate the conditions using the syntax

```
@name if ... then ... else ...
```

If the name label is present, then the condition of the `if` statement is driven by an input signal named after the label, otherwise, it's a non-deterministic choice as expressions and values are not interpreted.

7.2.2 Translation principle

Instead of taking the circuit generated from the Inria compiler, we chose to give a dedicated circuit translation optimized for the PSYC language as the *synchronous semantics* only deals with a small part of the ESTEREL language.

The ESTEREL circuit interface can be highly simplified due to the structure of PSYC programs. This simplified interface is shown in the rightmost circuit of Figure 7.1:

— First, in the inputs, only the `Go` pin is necessary. `Res` and `Susp` can be removed as the ESTEREL `Abort` and `Suspend` statements are never used. Furthermore, `Kill` is also not necessary because `Exit` and `Trap` correspond to a change of `Body`. In that case, the registers are unset because of the sequential behavior of a PSYC agent.

— Second, in the outputs, `Sel` can be removed for the same reasons as `Res`. Moreover, `Stop` can be deduced from `Term` and `Trap`, which is renamed `endbody` to fit the PSYC vocabulary.

— Finally, we split the input signals in the top part to have a separate input set called `Clock`, which is a set of input signals that trigger PSYC `advance` statements. `Data` correspond to input and output variables of a statement.

The PSYC circuit interface described above is only used for PSYC statements. We describe the STS treatment of source and clocks in the next section. As explained in the last chapter, we do not translate expressions as values are not interpreted. Also, *temporal variables* components are not translated either as they are modeled into the functional chain definitions of the properties.

7.2.3 Sources and clocks

Similarly to the synchronous semantics, *source* clocks are just inputs of the model. *Periodic* clocks are encoded in STS as follows:

$$
\begin{aligned}
\mathcal{V} &= \{parent,\ c,\ t\} \\
Init &\triangleq c = (O \text{ if } parent \text{ else } O + 1) \\
&\quad \wedge\ t = ((c = 0) \wedge parent) \\
Trans &\triangleq c' = ((c - 1 \text{ if } c > 0 \text{ else } P - 1) \text{ if } parent' \text{ else } c) \\
&\quad \wedge\ t' = ((c' = 0) \wedge parent')
\end{aligned}
$$

which corresponds to the PSYC clock definition `clock t = P * parent + O;`.

The initialization of the counter c might seem surprising as depending on the initial value of the $parent$ clock, we set it to O or $O + 1$. It is because the offset is counted starting on the first tick of the parent clock, hence:

— If the parent clock is present on the first instant, we start immediately to count the offset;

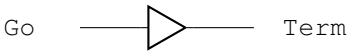
— Otherwise, we wait for the first tick of the parent clock before counting the offset. Hence, we wait for $O + 1$ ticks.

In the *synchronous semantics*, the clock is encoded starting by the statement `await immediate parent;` it can be proved equivalent to this STS definition.

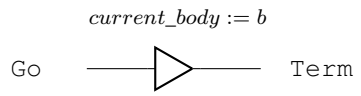
7.2.4 Statements

Basic statements

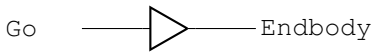
All the basic statements of PSYC are translated straightforwardly using the ESTEREL translation. As shown in Figure 7.2, `nothing` terminates instantaneously, `next` sets the next body variable and `endbody` terminates instantaneously and exits the current control flow. One might remark that we do not consider computations and variable assignments. As we cannot interpret them, they are abstracted in the circuit translation.



(a) Circuit modeling of the `nothing` (and computation) statement of PSYC



(b) Circuit modeling of the `next` b statement of PSYC



(c) Circuit modeling of the `endbody` statement of PSYC

Figure 7.2 – Circuit modeling of basic statements

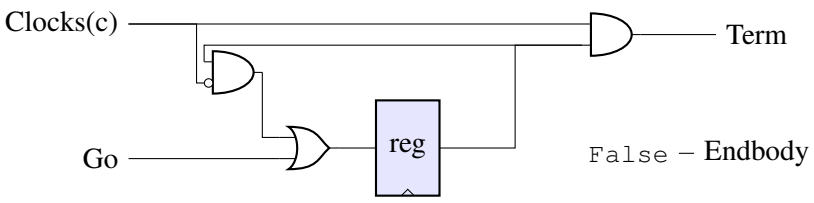


Figure 7.3 – Circuit modeling of the `advance 1 with c` statement of PSYC

Advance statement

The only PSYC statement that takes time is the advance statement. Other statements like communication or control can be directly adapted from the ESTEREL circuit semantics [PBEB07]. Figure 7.3 shows the translation of an `advance 1 with c` statement. The behavior is the one from the synchronous semantics:

1. When `Go` is selected, then `reg` is set and the agent stops on this `advance` statement;
2. While `Clock(c)` is not present, `reg` is reloaded at each instant;
3. When `Clock(c)` is present (and `Go` is not set again), `reg` is unset and `Term` is set; the instruction has terminated.

Remark that `endbody` is always absent as the `advance` instruction never breaks the control flow. Once started, it can only stop or terminate.

This statement can be easily generalized using a counter as shown in Figure 7.4. The `advance1` circuit is reloaded until the counter is bigger than zero. The buffers denote local update or conditions on the counter state variable. One may remark that there is a potential conflict on the `or` gate when merging different assignments of the counter variable. In fact, both are mutually exclusive: as an agent is sequential, if the `go` signal is set, it means that the current statement is either not active or has terminated in the same instant. In the latter case, the counter is equal to zero, so the upper branch of the or gate is not set.

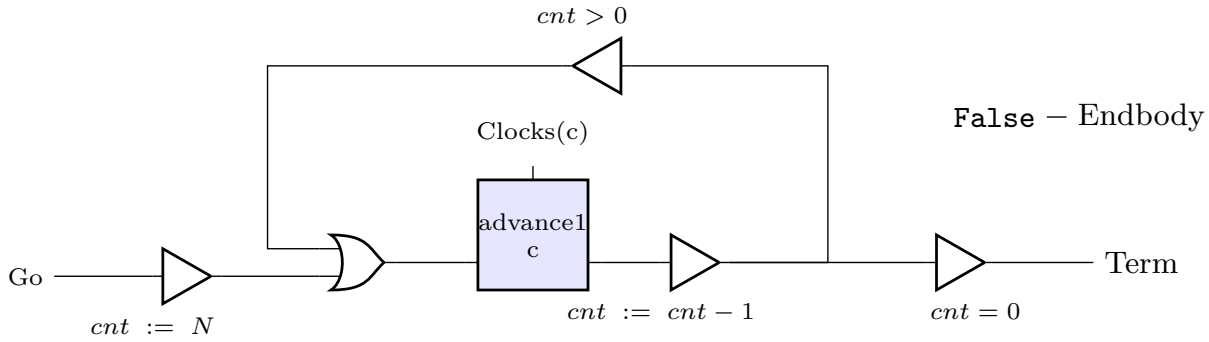


Figure 7.4 – Circuit of the `advance n with c` statement of PSYC

Control statements

The PSYC sequence statement s_1, s_2, ... s_n is translated into the circuit shown in Figure 7.5. Basically, when a statement terminates, the next one starts unless the `endbody` is set (both outputs are mutually exclusive).

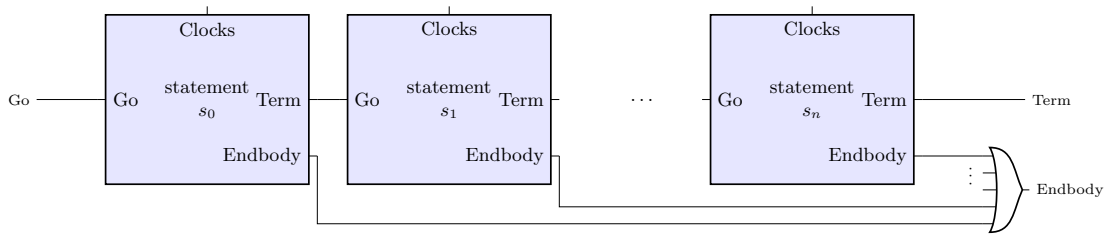


Figure 7.5 – Circuit of the sequence statement of PSYC

The PSYC if statement if (exp) s_1 else s_2 is translated in the circuit shown in Figure 7.6. First, the expression `exp` is replaced by a fresh signal called `condition` as we abstract expressions. This new signal is an input of the model and could also be used in the verification. The next sections will show how it could be done. This `condition` signal is used in the circuit to start the correct statement.

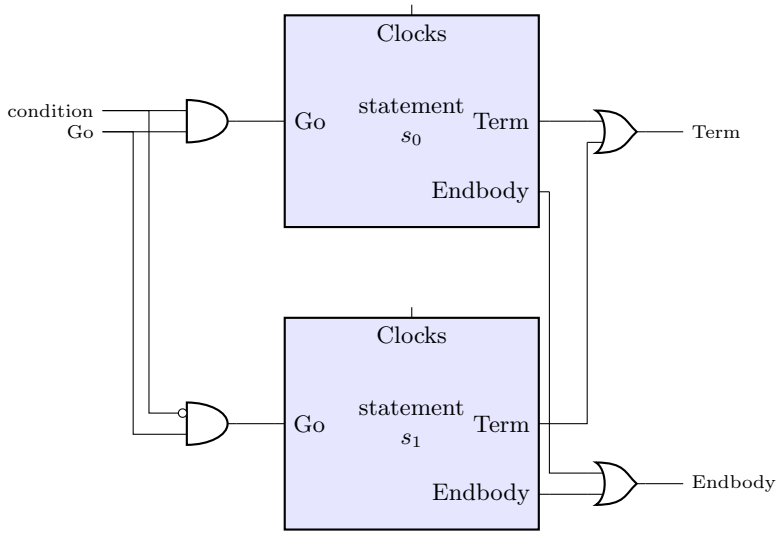


Figure 7.6 – Circuit of the `if` statement of PSYC

The PSYC `while` statement while exp do s is translated into the circuit shown in Figure 7.7. Similarly to the `if` statement, `exp` is replaced by a fresh signal called `condition`. The internal statement circuit is started when `condition` is set and either the whole `while` statement has been started or the internal statement has terminated. Conversely, the `while` statement terminates when the `condition` is unset and either the whole `while` statement has been started or the internal statement has terminated.

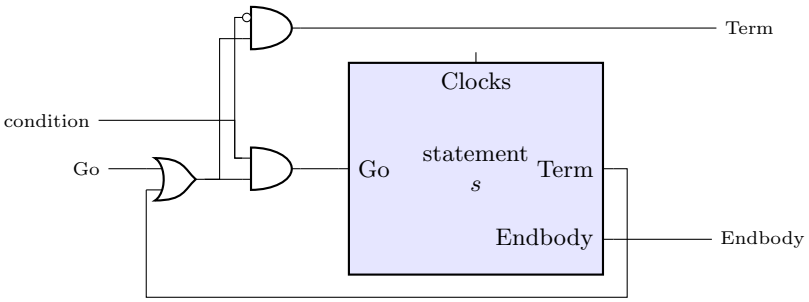


Figure 7.7 – Circuit of the `while` statement of PSYC

7.2.5 Agents

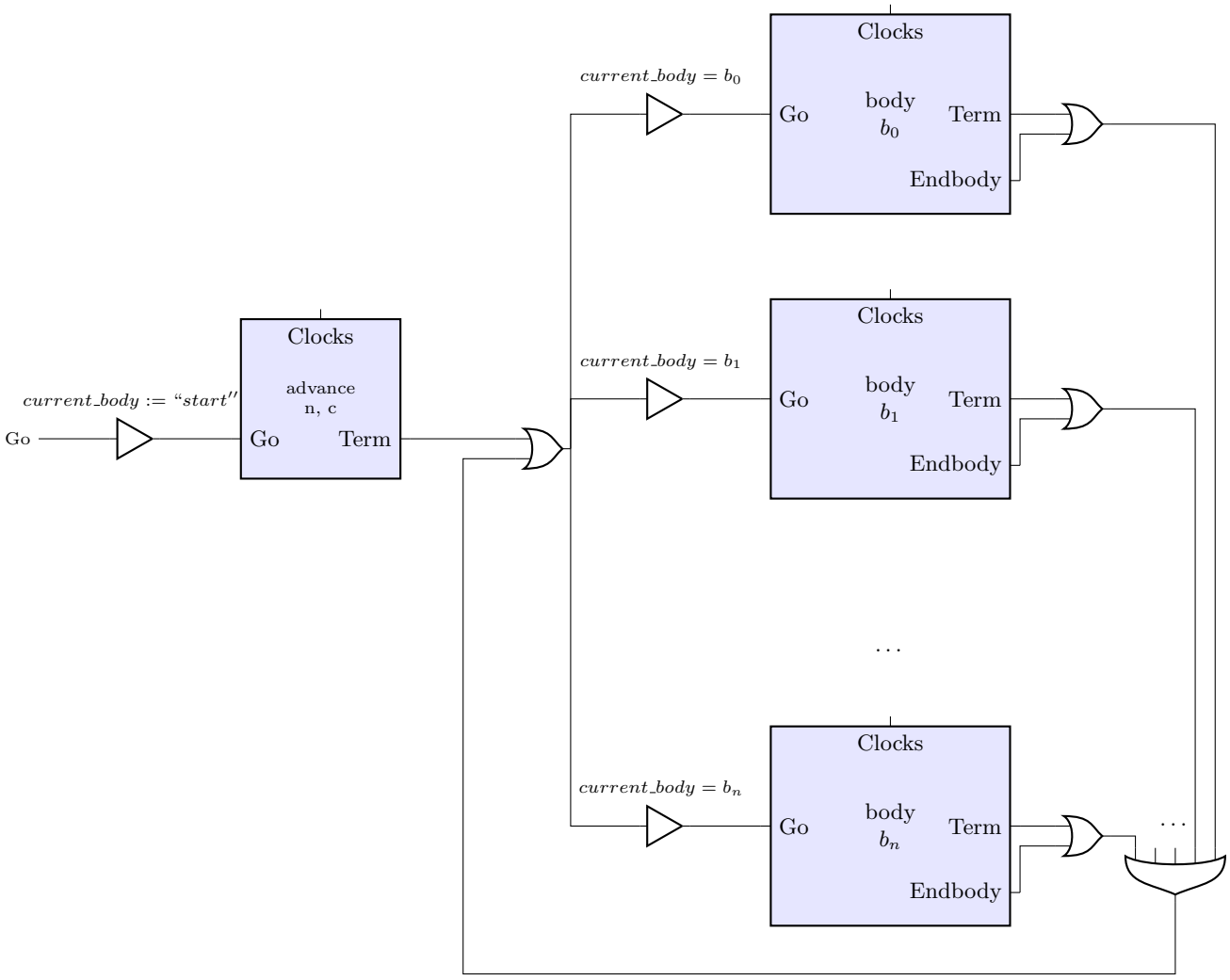


Figure 7.8 – Circuit of the `agent` declaration of PSYC

Consider an agent which defines a list of body declarations b_1, b_2, ... b_n and a `starttime` of parameters n with clock c as following:

$$\text{agent } id \text{ (starttime } n \text{ with } c) \; b_1, \; b_2 \; \ldots b_n$$

The translated circuit is shown in Figure 7.10. For simplicity, we use b_i for both the body name and the body inner statement. The circuit is split into two parts:

1. The `starttime` is translated by the `advance` circuit. This statement is started by a `Go` pin which is set on the initial instant and unset on subsequent ones, and the `current_body` is initially set to *"start"*.
2. When the `starttime` has terminated, the selected body is started. When this body has terminated or has reached an `endbody` statement (early exit), the next selected body is started, and so on.

7.2.6 Example

```
source ms;
clock c20ms = 20 * ms;
clock c50ms = 50 * ms;

agent GNC (starttime 2 with c50ms)
    {
  /* inputs */
  consult sensors, mode;
  /* outputs */
  display cmd;
  /* body infinite loop */
  body start {
    @mode if (mode == NOMINAL) {
      probe @GNC_consult;
      cmd = GNC(sensors);
      @GNC_display, advance 2 with
    c20ms;
    }
    advance 1 with c50ms;
  }
}

```

```
module Agent_GNC:
 input ms, c20ms, c50ms;
 input mode;
 output GNC_consult, GNC_display;

 await 2 c50ms;
 loop
  present mode then
   emit GNC_consult;
   -- private_cmd := GNC(?sensors);
   await 2 c20ms; emit GNC_display;
  end if;
  await 1 c50ms;
 end loop
end module

```

Figure 7.9 – Translation of a PSYC agent example. The leftmost listing shows the PSYC code of the agent while the rightmost figure shows its abstracted synchronous translation using the ESTEREL language

Let's consider again the `GNC` PSYC agent and its ESTEREL translation, used in previous chapters. Figure 7.9 shows on the left side a modified version of the `GNC` agent in which verification signals have been added while the ESTEREL translation is given on the right side. This updated ESTEREL version have additional `emit` statements to broadcast specific synchronization instants

used for verification by the observers. Using the rules described in this chapter, we generate the circuit represented in Figure 7.10.

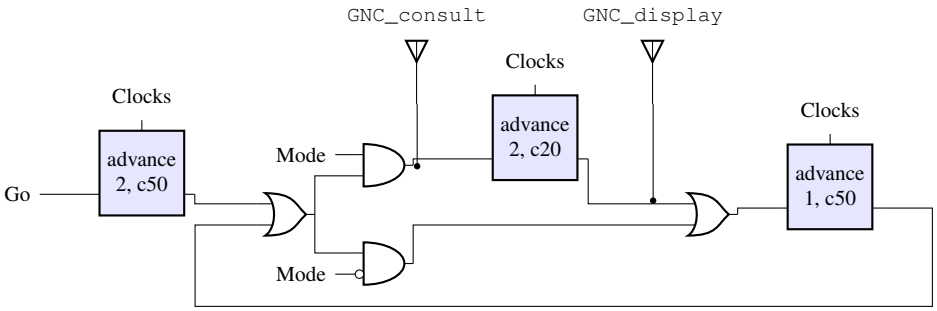


Figure 7.10 – Circuit translation of the agent from Figure 7.9. `Go` is used to launch the agent. It is true on the first instant then always false. `Mode` can be used as a nondeterministic input to model the branch condition. `GNC_consult` and `GNC_display` are the signals used for verification.

7.3 Experiments

7.3.1 Implementation

The encoding has been implemented in a tool called *PsykAnalyst* which takes a PSYC program, a CCSL specification and outputs HLL or SMV models used by external model-checkers — respectively NUXMV and PROVER SAT — for the verification process. Only basic optimisations are performed in *PsykAnalyst*, such as cone of influence analysis or structural simplifications. This tool has been implemented in the OCAML language in approximatively 5000 lines of code. Generation time is basically negligible compared to verification time. The next section shows experiments on various use-cases. They have all been performed on an Intel Core i9-2.3GHz machine, with 32Gb of RAM.

7.3.2 Benchmarks

The benchmark is composed of seven main use-cases. Each with several variations. Unfortunately, we were unable to access real industrial use-cases. However, we think that these benchmark applications cover different complexity aspects of the PSYC language. The following paragraphs quickly detail the different use-cases and various metrics are shown in Figure 7.11.

LED blinking The LED blinking use-case is described in chapter 3. Two tasks are responsible to respectively switch on and off a LED. The blinking period differ depending on a chosen LED color in input. While not really realistic, this tiny example is sufficient to evaluate the verification of basic properties: a blinking period, an alternate causality constraint and a minimal and maximal delay between actions to model the duty cycle. Each of them should be verified in both modes.

Use-cases	#agents	#clocks	#sources	#decisions	#advance	R	C	L	S
Led	1	3	1	1	8	1	1	1	0
ABS	8	4	1	9	30	2	2	2	1
Rosace	9	3	1	0	9	0	0	2	0
LGS	5	3	1	1	16	0	2	2	1
Power	3	8	2	1	9	0	0	1	0

Figure 7.11 – Metrics of the considered use-cases. R denotes the number of repetition requirements, C the number of causality requirements, L the number of latency requirements and S the number of synchronization requirements.

Anti-lock Braking System The Anti-lock Braking System is a simple modal system modeling an automotive braking controller [SSMP13]. When a pedal sends a brake command, braking is computed by three sets of tasks: a sensing task for each wheel, a common controller and a braking task for each wheel. The specification is composed of a repetition constraint on the sensing task, a reaction latency constraint on the whole functional chain and a synchronization constraint on the sensing task (braking should be synchronized on all the wheels).

Rosace Rosace is a simplified longitudinal flight controller commonly used in control system papers [PSG+14]. In practice, this use-cases is very similar to the industrial use-case we have encountered. The latter usually has similar characteristics but are bigger. The particularity of Rosace is its simplicity. All tasks have a very simple periodic pattern. However, the difficulty lies in the verification of end-to-end constraints, namely, reaction and freshness latencies. Of course, in such systems, various approaches perform better than model-checking [FBP17] but our approach tackles the general verification problem in PSYC allowing both advanced control-flow and multiple source clocks. Nonetheless, it is necessary to evaluate the performance of our tool using simpler applications.

Landing Gear System (LGS) The Landing Gear System is a simplified landing gear controller of an aircraft which is detailed more precisely in the next chapter [BW14]. Unlike the Rosace use-case, this system has complex control-flow due to the handling of the extension and retraction gears sequences. The system is composed of a main controller task embedding the system logic as an automaton and four other simpler auxiliary tasks to handle various pre and post processing of the data. As in Rosace, the two main constraints to be verified are a reaction and a freshness constraint. The former applies on the full extension/retraction sequence while the latter applies on the cockpit feedback.

Powertrain The final use-case is a simplified controller of an automotive powertrain [CDO+14]. The architecture is composed of three main sets of tasks: a task responsible to take the user pedal value, a set of tasks to compute the injections and a set of tasks which emit the injection pulses. In our case, we consider the freshness of the emitted pulses according to the pedal command. This property is rather challenging because we have to relate both source clocks: one based on the crankshaft position and one based on a timer. In our case, we assume that the crankshaft clock has

a repetition constraint between 200 microseconds and 1 millisecond (this corresponds roughly to 1000-5000 RPM). All the task activations are based on the crankshaft except the pedal one which has a period of 10 milliseconds.

7.3.3 Evaluation

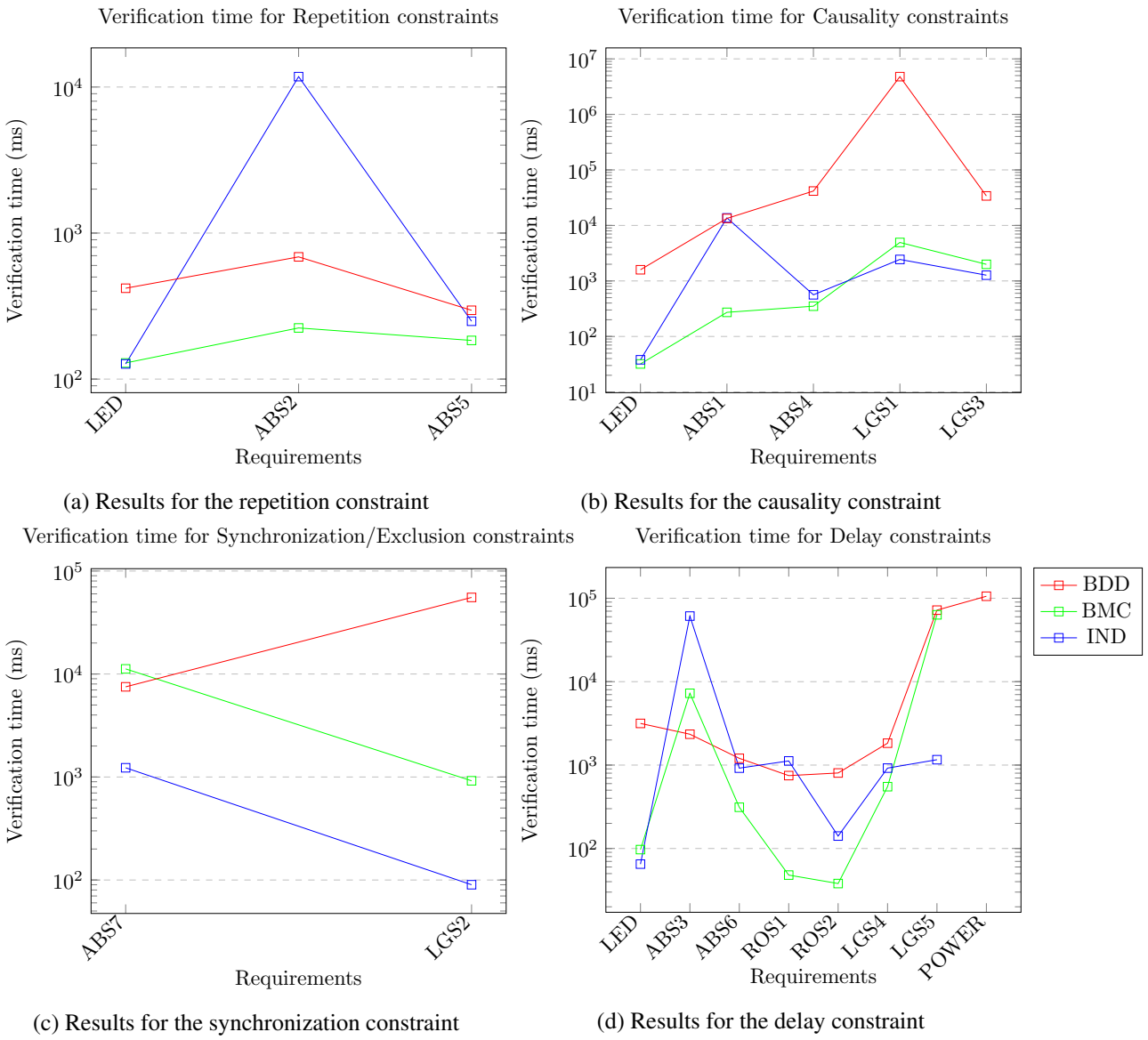
In this section, we give quantitative results for each system model grouped by each constraint type: repetition, causality, synchronization and delay requirements. All requirement verification is evaluated with multiple solvers: *Bounded Model-Checking* (BMC) with PROVER PSL (up to an estimated model's diameter), *Inductive* (IND) and *Interpolation* with PROVER PSL, and *Binary Decision Diagrams* (BDD) with NUXMV. Additionnally, the timeout has been set to 2 minutes.

As shown in Figure 7.12, SAT techniques (both IND and BMC) tend to be quicker than BDD except on two requirements ABS2 and ABS3 which are actually requirements of the "long" mode of the ABS system. BMC tends to be quicker than SAT, but finding a correct model's diameter can be hard. In our case, the diameter corresponds roughly to the hyper-period of the tasks in the system. All results are satisfying except for POWER which has exceeded the timeout value using SAT techniques.

As discussed at the beginning of this chapter, all the verification techniques and tools do not give equivalent guarantees. We now discuss the soundness of our methodology divided into two axes:

— First, the generation of the output model by *PsykAnalyst* is quite direct, and the implemented optimization steps have already been detailed a lot in the litterature. The soundness of the translation procedure is a result of the equivalence criterion between the native and the synchronous semantics.

— Then, the second part is the verification tool itself. BDD techniques, in our case using NUXMV, give sound and complete results. This approach has been used in most classical synchronous verification problems. However, now, SAT and SMT techniques are gradually replacing BDD techniques. In our case using PROVER PSL, SAT techniques often give better results than BDDs. Among them, *Bounded Model-Checking* gives interesting results but, while being also a sound procedure, suffers from completeness problems. It can be hard to find a bound to safely stop the verification process. *Induction* is another SAT technique which solves the completeness problem. The verification process can be quite fast when the model is inductive, but might also not converge otherwise. In our experiments, we heavily used the invariant generation and the interpolation engines of PROVER PSL to avoid non-convergence.



(a) Results for the repetition constraint

(b) Results for the causality constraint

(c) Results for the synchronization constraint

(d) Results for the delay constraint

Figure 7.12 – Verification results

7.4 Summary

In this chapter, we showed how a *circuit semantics* as defined for ESTEREL can be used in the PSYC language, based on its *synchronous semantics* to give Symbolic Transition Systems. Then, we adapted the traditional verification methodology used in Synchronous-Reactive formalisms to PSYC: we showed how to extend PSYC with *verification signals* which are used by *synchronous observers* generated from CCSL constraints. The final model can then be verified using various verification strategies (BDD, BMC, IND ...).

CHAPTER 8

Verification: Mono-Source Case

The encoding detailed in the last chapter gives a direct solution to tackle the PSYC verification problem. However, as it is based on the synchronous semantics, durations are not preserved; they are split into sequences of atomic reactions. Hence, this approach suffers from scaling issues. However, to avoid external non-determinism — and thus ease certification — industrial applications are very often based on a unique source clock, leading to a simpler model. This chapter proposes different solutions to handle the "mono-source" verification problem. The common idea is to re-use, to some extent, the native semantics to keep only the states corresponding to synchronization instants (i.e., interval boundaries).

8.1 Problem statement

Consider again the simple task GNC considered in the previous chapters. This task has a period of 1 second and a duration of 300 milliseconds based on a source clock corresponding to a physical period of 100 milliseconds. Depending on the chosen periods, the number of states might change considerably, although the overall complexity of the system is exactly the same. The table below shows state space variations of the GNC task — without mode handling — for different choices of task period.

GNC period	1 s	10 s	100 s	1000 s
#State vars	37	37	37	37
#States	42	402	4002	40002
Diameter	20	200	2000	20000

We can see that, even if the number of state variables is quite large due to the structural translation, it does not change even if the task period changes. However, the number of states grows linearly when the task period increases as well as the system's diameter. Nonetheless, in the four variations above, the example is functionally equivalent; we might want the state space — and more particularly the verification time — not to change significantly.

An obvious solution to this problem is to use the "best" clocks to limit the size of the state space. However, changing the clock definitions is not always simple or even possible. For example, a system might use very small durations in one mode while having a bigger one in another mode. The system could also have some small durations that happen sparsely, for example once in a system's hyper-period. In such cases, it is not possible to change the clock definitions to obtain a smaller state-space.

Existing approaches

Abstract Interpretation Synchronous-reactive languages usually have a semantics that relies on atomic reactions even for long duration computations; this approach is not only an issue for real-time implementations, but also for formal verification. In this context, [Hal93] introduced an approach to extend BDD model-checking with abstract interpretation in the model-checker LESAR. This work was then extended in the NBAC tool [Jea03]. In both cases, the approach relies on abstract domains — such as polyhedra — for the numerical part of the model while relying on BDDs for the boolean part.

We first tried the tool LESAR using the polyhedra option without any success. However, we got encouraging results with the NBAC tool. Nonetheless, this approach tends not to be highly stable. It was unclear when the verification procedure could converge or not, due to the incompleteness of abstract interpretation.

Timed Transition Systems Another approach is to consider that the system only has one external clock, which is often the case in industrial systems. In this case, timed verification techniques can be used. The most popular one is probably model-checking of timed automata using Difference Bound Matrices (DBM) as in the UPPAAL tool [LPY97]. Other approaches based on Timed Transition Systems and SAT also exists such as the one from NUXMV. However, very few details are given on how the verification is done. This approach could not work directly on STS coming from the *synchronous semantics* as we need the durations of the model. Nonetheless, we could use durations coming from the *native semantics*. We can keep the structure of the STS while having transitions based on those durations.

We tried various examples with UPPAAL. When the examples were very simple, UPPAAL did perform well. However, in more realistic examples, UPPAAL was either significantly slower without optimization or performed similarly with optimizations such as convex hull. Additionnally, it seems that UPPAAL does not scale well when the model contains a lot of control. Also, this approach has been done manually with a lot of "manual" optimization. As the encoding is really different from STS, it has not been automated.

Integer Linear Programming In the particular context of multi-periodic systems, we might also consider techniques independent from the control flow. In this case, the verification instants are reachable using linear expressions. Hence, *Integer Linear Programming (ILP)* could be used for the verification process. This approach also requires a significantly different encoding than STS and would only be usable only on a tiny subpart of PSYC.

The next section presents a more general approach relying on STS with durations that can be directly used with all the model checkers mentioned in the last chapter.

8.2 A temporal optimization approach

We now describe a novel approach to handle durations symbolically. It is motivated by the fact that timed automata or timed transition systems can be simulated using classical transition systems. Some work shows that internal structures of timed model-checkers — namely, Difference Bounded Matrices — can be used to symbolically simulate the durations of a timed automata [BBP19]. Additionaly, intermediate instants in a synchronous program can be abstracted by dynamically specifying the wake-up time of the next reaction [VHBG17]. In our case, our approach is strongly inspired by these two concepts. However, our problem is simpler as we only have to deal with constant durations (due to the *mono-source* constraint), but we focus on verification instead of simulation or execution.

8.2.1 Symbolic Transition System with durations

The full `advance` statement circuit adds a counter to repeat n times the `advance 1` circuit as shown in chapter 9. However, another approach is to keep the logical duration of the `advance` statement from the native semantics. Figure 8.1 shows the encoding of an advance in which a trigger of some clock c triggers the transition to the next advance with duration `N` defined as in the rule: $C \vdash \mathsf{advance}\ n\ \mathsf{with}\ c \Longrightarrow_{N\times s} C' \vdash \mathsf{nothing}$.

However, one might notice that in the native semantics, multiple logical durations can be assigned to the same `advance` statement depending on the state of the PSYC periodic clocks. The native semantics shows that a PSYC code can be unfolded so that each `advance` statement has a single duration. In practice, all our examples respect this assumption, so we don't consider this case. Nonetheless, if needed, the circuit can be generalized to multiple durations.

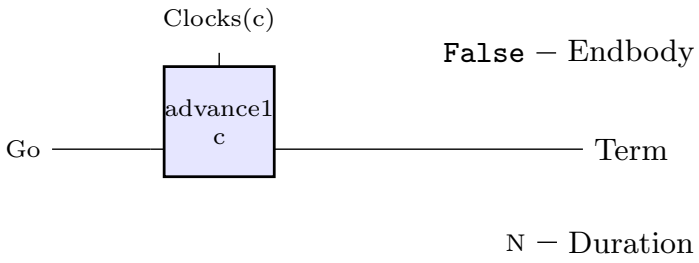


Figure 8.1 – Optimized circuit of the `advance n with c` statement of PSYC

8.2.2 Symbolic abstraction of durations

As explained in previous sections, in the case of long durations, a lot of intermediate states representing temporal progression are generated. We sketch here an approach to aggregate these states. Based on the `advance` circuit shown in the previous section, we can add a *scheduler component* that activates each agent depending on their constraints and *jumps* multiple cycles at once. We call this methodology *temporal optimization*. At the end of this chapter, experiments show that in some cases, this temporal optimization might give significant verification speed-up.

Basically, in an agent, each `advance` statement is triggered by a unique clock and a new $duration$ output is added to them giving the duration of the current elementary action based on the active `advance` statement. The scheduler outputs a $Delta$ signal which denotes the number of cycles jumped at each step depending on the state and the duration of each agents as shown in Figure 8.2. Δ can then be used by the properties to adapt the synchronous observers — especially the delay constraint.

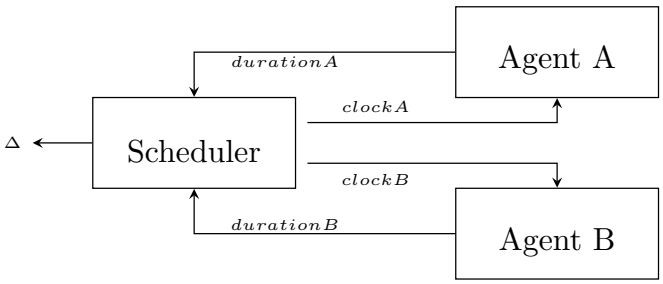


Figure 8.2 – Temporal optimization architecture

We show now the STS of the scheduler for a list of agents $ag_1, ag_2, \ldots, ag_n$:

$$\mathcal{V} = \{\Delta, duration_1, duration_2, \ldots duration_n, state_1, state_2, \ldots state_n, clock_1, clock_2, \ldots clock_n\}$$

$$\begin{aligned}
Init \triangleq\ & clock_1 \wedge clock_2 \wedge \ldots \wedge clock_n \\
& \wedge state_1 = 0 \wedge state_2 = 0 \wedge \ldots \wedge state_n = 0 \\
& \wedge \Delta = min(duration_1, duration_2, \ldots duration_n) \\
Trans \triangleq\ & clock'_1 = (\Delta = (duration_1 - state_1)) \\
& \wedge clock'_2 = (\Delta = (duration_2 - state_2)) \\
& \ldots \\
& \wedge clock'_n = (\Delta = (duration_n - state_n)) \\
& \wedge state'_1 = 0 \text{ if } clock'_1 \text{ else } (state_1 + \Delta) \\
& \wedge state'_2 = 0 \text{ if } clock'_2 \text{ else } (state_2 + \Delta) \\
& \ldots \\
& \wedge state'_n = 0 \text{ if } clock'_n \text{ else } (state_n + \Delta) \\
& \wedge \Delta' = min(duration'_1 - state'_1, duration'_2 - state'_2, \ldots duration'_n - state'_n)
\end{aligned}$$

The detailed behavior is the following:

1. First, depending on the previous agent states and the previous Δ, the scheduler computes which agent is active in the current state. That is, when the previous Δ corresponds to the agent's remaining time after the previous step, i.e., $duration - states$.
2. Then, the scheduler computes the new agent states which model the offset in their current elementary actions. If the agent is active on the current step, it means that it has reached a synchronization point, so the new state is 0. Otherwise, the new state corresponds to the previous one plus the new delta.
3. Finally, depending on the new agent states and the agent durations, the scheduler computes the new Δ which jumps as the minimum of the agent's remaining time.

This behavior is, unsurprisingly, the one shown in the composition rule of agent network in the native semantics.

8.3 Experiments

8.3.1 Back to GNC task

Let's take the GNC task and its variations from section 8.1. Using the optimization described in the previous sections we get, as expected, the same compressed state space in each variation, composed of 4 states, as shown in the table below. Hence, this methodology works in terms of state space size.

Source period	100 ms	10 ms	1 ms	100 us
#State vars	43	43	43	43
#States	4	4	4	4
Diameter	4	4	4	4

Now, consider adding a new task called HM (for *Health Monitoring*) with a 30 millisecond period as shown in Figure 8.3. The chronogram shows the timing behavior of GNC and HM, while below, the figure shows the state evolution of the system using the temporal optimization. The bracket values correspond to the states of the system composed of the state of both agents; these states correspond to a counter representing the agent offset inside its timing interval. Additionnaly, each arrow represents a timing transition with a duration represented by Δ. Hence, initially, both agents have a state of 0. Then, the first transition takes 30 milliseconds up to the next agent synchronization, which yields a state in which HM has reached a synchronization instant, but GNC does not. Hence, GNC has a state of 30 milliseconds already elapsed. The second transition takes 10 milliseconds up to the next agent synchronization, which yields a state in which GNC has reached a synchronization instant, but HM does not. Hence, HM has a state of 10 milliseconds already elapsed. On the next state, as HM has still not reached its synchronization, its state is now 20 milliseconds, which have already elapsed from the beginning of its timing interval. The result of the optimization process yields a new notion of instant in which less than 10 states are visible in the chronogram, while there are roughly 100 states without the optimization.

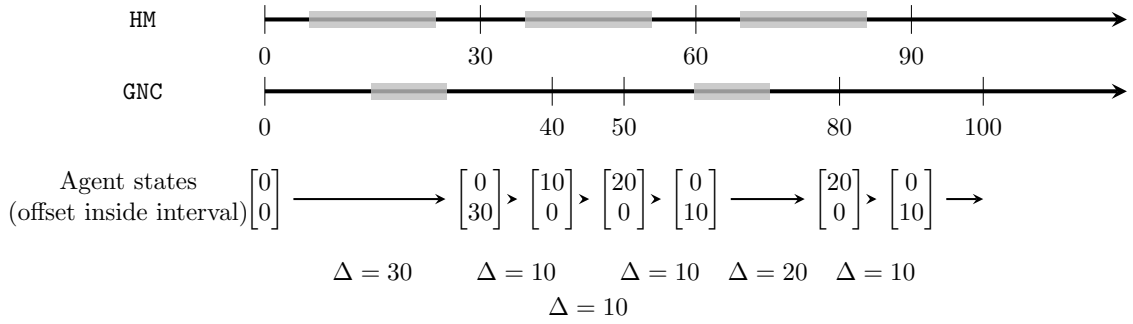


Figure 8.3 – Illustration of the optimization process with the GNC example

However, there is a limitation with this result. The measured state space is actually the *reachable* state space in which non-reachable states are not accounted. When trying to build the GNC models with the model-checker NUXMV, we get, as expected, almost instantaneous property checking even with the larger variation of GNC. However, building the model in itself takes a lot more time than without optimization as we introduce multiple integer variables — up to an hour in the last variation of the GNC task when the original model took a few seconds. It seems that, in NUXMV, building the BDD of the model doesn't scale when large integer variables are used, even if few states are actually reachable.

8.3.2 The Landing Gear System use-case

This section shows how the methodology could be applied to a realistic safety-critical use-case from the avionics domain introduced in [BW14]. Different tasks are added to the original use-case to reflect the complexity of realistic industrial software.

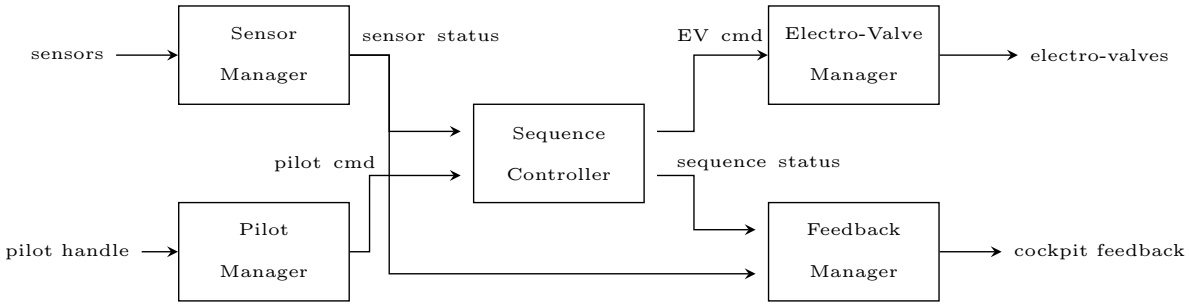


Figure 8.4 – The Landing Gear System (LGS) functional architecture

The use-case describes a controller to handle the landing gear operations of an aircraft: landing gears extension and retraction using a mechanical electro-valve system. It is composed of 5 different tasks. The main task is the `SequenceController` which handles the extension or retraction sequence of the landing gear similarly to a state machine. The `SequenceController` sends its commands to the `EVManager`, which interacts with the electro-valves part of the mechanical system. The task `SensorManager` handles (and filters) sensor data which are used by the `SequenceController` to ensure that there is no discrepancy between the controller state and the real physical state of the system. The last two tasks interact with the pilot. `PilotManager` takes the pilot handle value to send it to the controller, and the `PilotFeedback` task monitors the system to give feedback to the pilot: green light (gears extended), no light (gears retracted), (orange light) gears moving or red light (failure). All these tasks have been implemented in the PSYC language. We cannot show them here due to space limitations.

Evaluation For such systems, both techniques give satisfying results. In the case of PROVER PSL, all requirements are checked in about a second. Temporal optimization gives up to 95% speed-up, although we see that in r1 and r5, there is no significant difference. This is because the model can never jump multiple cycles at once. In the case of NUXMV, all requirements are checked in tens of seconds except r1, and temporal optimization does not give any speed-up except for r4; temporal optimization even makes the results worst. We think that this can be linked to the NUXMV model building and optimizations which are harder in the case of temporal optimization.

8.4 Summary

Based on the assumption that only one clock is used in the model, various approaches may perform better than the classical synchronous model-checking shown in the previous chapter. This

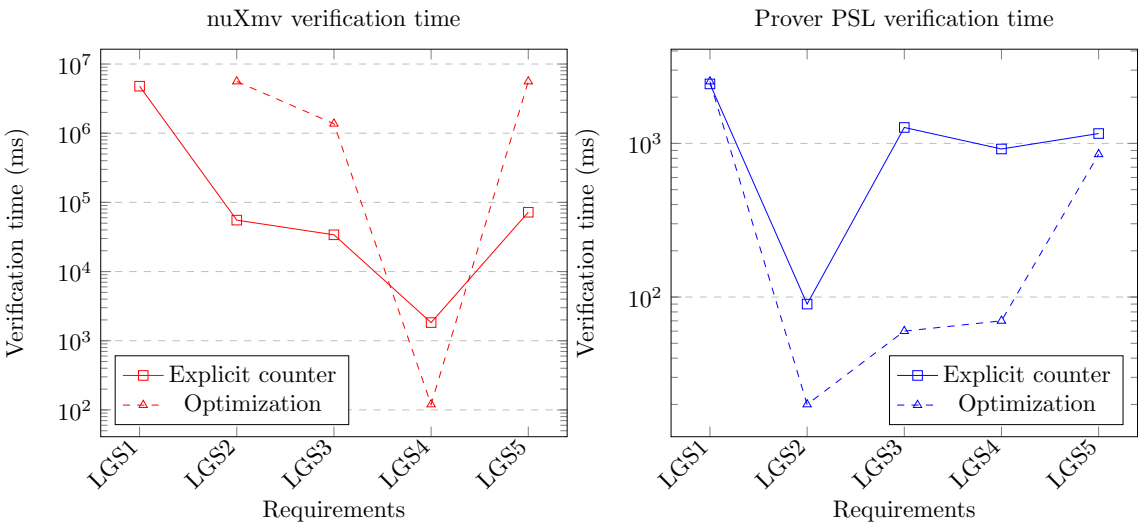


Figure 8.5 – The leftmost plot and the rightmost plot show the verification time for, respectively, nuXmv and Prover PSL. Absent values denote timeout (2h).

chapter presented a new technique — fully compatible with the one shown in the previous chapter — based on simulating durations in the model. This approach is strongly inspired from DBM model-checking in timed automata.

FOURTH PART

Conclusion and Perspectives

CHAPTER 9

Conclusion

9.1 Summary

The main objective of this thesis was to give a formal verification methodology for real-time applications based on logical time. We focused on the PSYC language — a parallel synchronous language based on a variation of Logical Execution Time — mainly due to the industrial context of this thesis. Two main contributions emerge from this thesis. Firstly, we give semantics foundations for the PSYC language based on a new formalism that we call synchronous Logical Execution Time (sLET), and secondly, we give a verification methodology applied to PSYC based on its semantics definitions. The two following sections sum-up these two contributions.

Synchronous LET and operational semantics

The PSYC language, in its current form, has never been formalized totally nor positioned with other existing formalisms such as the Synchronous-Reactive (SR) and the Logical Execution Time (LET) approaches. The first contribution of this work comes from a simple observation: the PSYC language actually inherits from both SR and LET. Hence, we defined a new model called synchronous Logical Execution Time (sLET) as a synchronous generalization of LET with logical clock synchronizations as in SR languages. This model is then used, in turn, to define two operational semantics for PSYC: a classical structural operational semantics called native (or "big-step") that preserves durations along its transitions whereas the second one is defined by a structural translation to a SR language — in our case ESTEREL — which gives a "small-step" behavior in which durations are expanded on atomic transitions. While the native semantics is very close to the one used in the current industrial compiler, the synchronous semantics gives an abstraction, allowing synchronous analysis and verification tools to be reused. Finally, we have also given an equivalence criterion between both semantics; the proof is shown in the appendix.

Formal verification of temporal properties

The main properties that we wanted to verify on the PSYC language come from higher-level temporal requirements and focus mainly on the timing of system functions, and in particular, the relative timing of different functions and how they interact, similarly to requirements of automotive specification [PFGDN12]. Examples of such requirements are synchronizations, or latencies on chains of task communications (called *functional chains*). Hence, the second contribution of this work was to formalize these properties in the CCSL specification language, a language allowing for the formal specification of clock constraints.

Based on the defined synchronous semantics of PSYC, and its equivalence criterion with the native semantics, we have defined a methodology to formally verify properties on PSYC — which can be actually generalized to other (s)LET languages — using a structural translation to symbolic transition systems and synchronous circuits. In this approach, we chose to encode our CCSL requirements as synchronous observers in the same format. To verify the properties, we mainly used symbolic model-checking based on Binary Decision Diagram (BDD) and Satisfiability (SAT). We have proceeded to a series of experiments showing that with most of our applications from our benchmark, requirements were verified in a few seconds.

However, the verification becomes more complicated when applications use particularly long durations. Verification time increases exponentially for programs when increasing logical durations. This is mainly due to the synchronous semantics expanding all durations to atomic transitions. To face this issue, we defined a temporal optimization procedure to reduce the state-space assuming that only one source clock is used — which is often the case in industrial applications. This approach permits to jump multiple cycles at once. Instead of using only the synchronous semantics, we add task durations coming from the native semantics and a scheduler to keep track of the state of each task. This approach has been evaluated on a Landing Gears System case study showing up to 95% speed-up using SAT model-checking.

Finally,the whole methodology described above has been prototyped in a tool called PSYK-ANALYST. All the experiments have been done using this tool and various third-party solvers: NUXMV, and PROVER PSL. The tool takes a representative sub-part of the PSYC language and verifies CCSL requirements using one of the solver strategies. If the property is false, a counter-example is shown based on the one from the solver.

9.2 Perspectives

Semantics of PSYC null-latency communications

Null-latency communication is a special feature of PSYC that allows for the synchronization and communication between two tasks during their elementary actions. This is used, in turn, to simulate causal communication using so called *fractional clocks*. Without going into details, synchronous LET natively supports null-latency, as *fractional clocks* are just another logical clock used to express activation and deadline constraints. There are however multiple syntactical diffi-

culties that makes structural definitions more complex, in addition to dealing with causality issues. As such, both semantics could be extended with null-latency constructions. Nonetheless, verification of properties based on null-latency can already be partially done using CCSL definitions. One could then use those definitions to check the causality of null-latency constraints as well as verifying classical latency properties.

Synthesis of temporal properties Synthesis is another interesting topic to study in PSYC. First, synthesis of temporal constraints from high-level temporal requirements would allow the automatic generation of PSYC code. It is, however, a very difficult subject as it can be quite difficult to automatically choose from multiple possible PSYC designs satisfying the same set of requirements. Nonetheless, this is an ongoing subject at KRONO-SAFE, and is currently prototyped in a tool called DESIGNER. Second, synthesis could also allow PSYC programs containing multiple source clocks to be refined to a unique source clock or to extract sufficient conditions on the relative triggering of clocks, satisfying a set of high-level requirements. This could be used to refine a complex multi-source design to a simpler single-source design simplifying its compilation process.

Compositional model-checking Finally, PSYC verification could be extended by considering *assume/guarantee* contract on C code *inside* elementary actions. Such contracts could be verified using deductive analysis techniques with tools such as FRAMA-C [CKK+12]. Then, model-checking could use those contracts to constrain which branch could be taken by each task on each instant.

Bibliography

[AB84] Didier Austry and Gérard Boudol. Algebre de processus et synchronisation. *Theoretical Computer Science*, 30(1):91–131, 1984.

[ACA+96] Christophe Aussagues, C. Cordonnier, M. Aji, V. David, and J. Delcoigne. OASIS: A New Way to Design Safety Critical Applications. *IFAC Proceedings Volumes*, 29(5):21–26, November 1996.

[AHC96] Jean-Raymond Abrial, A Hoare, and Pierre Chapron. *The B-book*. Cambridge university press, 1996.

[Air23] Airbus. Could the humble dragonfly help pilots during flight? https://www.airbus.com/en/newsroom/stories/2023-01-could-the-humble-dragonfly-help-pilots-during-flight, September 2023.

[AMDS07] Charles André, Frédéric Mallet, and Robert De Simone. Modeling time (s). In *Model Driven Engineering Languages and Systems: 10th International Conference, MoDELS 2007, Nashville, USA, September 30-October 5, 2007. Proceedings 10*, pages 559–573. Springer, 2007.

[And09] Charles André. Syntax and semantics of the clock constraint specification language (CCSL). Technical report, 2009.

[And10] Charles André. Verification of clock constraints: CCSL observers in Esterel. Technical report, 2010.

[BB91] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79:1270 – 1282, 10 1991.

[BBP19] Guillaume Baudart, Timothy Bourke, and Marc Pouzet. Symbolic simulation of dataflow synchronous programs with timers. *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2017*, pages 45–70, 2019.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*, pages 193–207. Springer, 1999.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[BCM+92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[Ber96] Gerard Berry. The constructive semantics of pure Esterel. 06 1996.

[Ber02] Gérard Berry. The Esterel v5 Language Primer. December 2002.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BLGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[Bon94] Frédéric Boniol. CoReA: A synchronous calculus of parallel communicating reactive automata. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, pages 611–622, Berlin, Heidelberg, 1994. Springer.

[Bra11] Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[BVWW09] Thomas Bochot, Pierre Virelizier, Hélene Waeselynck, and Virginie Wiels. Model checking flight control systems: The Airbus experience. In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 18–27, 2009.

[BW14] Frédéric Boniol and Virginie Wiels. The landing gear system case study. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 1–18. Springer, 2014.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 334–342. Springer, 2014.

[CDO+14] Damien Chabrol, Vincent David, Patrice Oudin, Gilles Zeppa, and Mathieu Jan. Freedom from interference among time-triggered and angle-triggered tasks: a powertrain case study. In *Embedded Real Time Software and Systems (ERTS2014)*, 2014.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, Lecture Notes in Computer Science, pages 233–247, Berlin, Heidelberg, 2012. Springer.

[Cla97] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997.

[CM14] Darren Cofer and Steven Miller. DO-333 certification case studies. In *NASA Formal Methods*, 2014.

[CMST16] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The Kind 2 model checker. In *International Conference on Computer Aided Verification*, pages 510–517. Springer, 2016.

[Com06] International Electrotechnical Commission. Medical device software – Software life cycle processes, 2006.

[CPBL+15] Thomas Carle, Dumitru Potop-Butucaru, David Lesens, et al. From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. *Leibniz Transactions on Embedded Systems*, 2015.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, New York, NY, USA, October 1987. Association for Computing Machinery.

[CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–11. IEEE, 2017.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996.

[DAL+04] Vincent David, Christophe Aussaguès, Stéphane Louise, Philippe Hilsenkopf, Bertrand Ortolo, and Christophe Hessler. The OASIS based qualified display system. In *Fourth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT 2004), Columbus, Ohio, USA*, page 11, 2004.

[FBP17] Julien Forget, Frédéric Boniol, and Claire Pagetti. Verifying end-to-end real-time constraints on multi-periodic models. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2017.

[FRNJ09] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*. IEEE Communications Society, 2009.

[GHKS04] Arkadeb Ghosal, Thomas Henzinger, Christoph Kirsch, and Marco Sanvido. Event-driven programming with logical execution times. volume 2993, pages 357–371, 03 2004.

[GJSH05] Arkadeb Ghosal, JCZ Jurado, Marco AA Sanvido, and J Karl Hedrick. Implementation of AFR controller in an event-driven real-time language. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 4428–4433. IEEE, 2005.

[GLMS11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 372–387. Springer, 2011.

[GMD11] Régis Gascon, Frédéric Mallet, and Julien Deantoni. Logical time and temporal logics: comparing UML MARTE/CCSL and PSL. In *2011 Eighteenth International Symposium on Temporal Representation and Reasoning*, pages 141–148. IEEE, 2011.

[Hal93] Nicolas Halbwachs. Delay analysis in synchronous programs. In *Computer Aided Verification: 5th International Conference, CAV'93 Elounda, Greece, June 28–July 1, 1993 Proceedings 5*, pages 333–346. Springer, 1993.

[HLR94a] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous Observers and the Verification of Reactive Systems. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'93)*, Workshops in Computing, pages 83–96, London, 1994. Springer.

[HLR94b] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST'93) Proceedings of the Third International Conference on Algebraic Methodology and Software Technology, University of Twente, Enschede, The Netherlands 21–25 June 1993*, pages 83–96. Springer, 1994.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[HP84] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, 1984.

[iee90] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[Jea03] Bertrand Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, 23:5–37, 2003.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[KS12] Christoph M Kirsch and Ana Sokolova. The logical execution time paradigm. *Advances in Real-Time Systems*, pages 103–120, 2012.

[KS22] Hermann Kopetz and Wilfried Steiner. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer International Publishing, 2022.

[KS23] Krono-Safe. `https://www.krono-safe.com/`, 2023.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[LDAVN10] Matthieu Lemerre, Vincent David, Christophe Aussaguès, and Guy Vidal Naquet. An Introduction to Time-Constrained Automata. In *ICE 2010 3rd Interaction and Concurrency Experience*, pages 83–98, Amsterdam, Netherlands, June 2010.

[Lee09] Edward A Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, 2009.

[LL22] Edward A Lee and Marten Lohstroh. Generalizing logical execution time. In *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, pages 160–181. Springer, 2022.

[LO12] Matthieu Lemerre and Emmanuel Ohayon. A model of parallel deterministic real-time computation. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 273–282. IEEE, 2012.

[Loh20] Hendrik Marten Frank Lohstroh. *Reactors: A deterministic model of concurrent computation for reactive systems*. University of California, Berkeley, 2020.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1:134–152, 1997.

[LS16] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. The MIT Press, 2nd edition, 2016.

[LSV98] Edward A Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 17(12):1217–1229, 1998.

[MDS15] Frédéric Mallet and Robert De Simone. Correctness issues on MARTE/CCSL constraints. *Science of Computer Programming*, 106:78–92, 2015.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, 1982.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical computer science*, 25(3):267–310, 1983.

[MOT20] Amira Methni, Emmanuel Ohayon, and François Thurieau. Asterios checker: A verification tool for certifying airborne software. In *10th European Congress on Embedded Real Time Systems (ERTS 2020)*, 2020.

[Org18] International Standardization Organization. Road vehicles – Functional safety, 2018.

[PBEB07] Dumitru Potop-Butucaru, Stephen A Edwards, and Gérard Berry. *Compiling Esterel*, volume 86. Springer Science & Business Media, 2007.

[PFGDN12] Marie-Agnès Peraldi-Frati, Arda Goknil, Julien DeAntoni, and Johan Nordlander. 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*, pages 230–239, 2012.

[Plo04] Gordon Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 07 2004.

[pro23] Prover Technology. `https://www.prover.com/`, 2023.

[PSG+14] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The rosace case study: From simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 309–318. IEEE, 2014.

[Ray08] Pascal Raymond. Synchronous program verification with Lustre/Lesar. *Modeling and Verification of Real-Time Systems*, page 7, 2008.

[RDE11] RTCA RTCA DO-178 and EUROCAE. Software Considerations in Airborne Systems and Equipment Certification, 2011.

[Rus14] John Rushby. The versatile synchronous observer. In *Specification, Algebra, and Software: Essays Dedicated to Kokichi Futatsugi*, pages 110–128. Springer, 2014.

[Saf23] Safran. Safran green taxiing. `https://www.safran-group.com/fr/videos/electric-taxiing`, September 2023.

[SAG12] Oliver Scheickl, Christoph Ainhauser, and Peter Gliwa. Tool support for seamless system development based on AUTOSAR timing extensions. In *Embedded Real Time Software and Systems (ERTS2012)*, 2012.

[SSMP13] Jagadish Suryadevara, Cristina Seceleanu, Frédéric Mallet, and Paul Pettersson. Verifying marte/ccsl mode behaviors using uppaal. In *Software Engineering and Formal Methods: 11th International Conference, SEFM 2013, Madrid, Spain, September 25-27, 2013. Proceedings 11*, pages 1–15. Springer, 2013.

[Var08] Moshe Y. Vardi. From Monadic Logic to PSL. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science*, volume 4800, pages 656–681. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.

[VHBG17] Reinhard Von Hanxleden, Timothy Bourke, and Alain Girault. Real-time ticks for synchronous programming. In *2017 Forum on Specification and Design Languages (FDL)*, pages 1–8. IEEE, 2017.

[XP90] Jia Xu and David Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, mar 1990.

List of Figures

Listings

Appendix

APPENDIX A

Equivalence Criterion Proof

In this appendix, we demonstrate the equivalence between both semantics, namely native (i.e. big-step) and synchronous (i.e. small-step). We show the proof only for a subpart of the syntax for various reasons: firstly, it makes the proof simpler with less technical details; secondly, it allows to focus on the important part which is actually what should be retained from this criteria and could be applied to other languages.

A Syntax

We show in this section the simplified syntax for both PSYC and ESTEREL.

$$
\begin{array}{rcl}
p, q & ::= & \mathsf{nothing} \\
& | & v := exp \\
& | & \mathsf{pause} \\
& | & \mathsf{await}\ n\ S \\
& | & \mathsf{await\ immediate}\ nS \\
& | & \mathsf{emit}\ S \\
& | & \mathsf{if}\ c\ \mathsf{then}\ p\ \mathsf{else}\ q\ \mathsf{end} \\
& | & \mathsf{while}\ c\ \mathsf{do}\ p\ \mathsf{end} \\
& | & p\ ;\ q \\
& | & p \parallel q
\end{array}
\qquad
\begin{array}{rcl}
\mathit{stmt} & ::= & \mathsf{nothing} \\
& | & id := f(exp*) \\
& | & \mathsf{advance}\ n\ \mathsf{with}\ c, \quad n \in \mathbb{N}^* \\
& | & stmt_1\ ;\ stmt_2 \\
& | & \mathsf{if}\ (exp)\ stmt_1\ \mathsf{else}\ stmt_2 \\
& | & \mathsf{while}\ exp\ \mathsf{do}\ stmt
\end{array}
$$

Figure A.1 – Simplified syntax of PSYC on the left column and ESTEREL on the right column

As shown in Figure A.1, the grammars are highly simplified; only the part used in the translation scheme is kept. In this PSYC syntax, we don't show the `body`/`endbody` mechanism to

avoid the `trap`/`exit` ESTEREL statements in the proof. However, `body` without `endbody` can be rewritten using an `next_body` variable and a global `if` statement to start the correct `body` accordingly as shown below.

```
1 next_body := START;
2 while true {
3   if (next_body = START) {
4     /* start body statements */
5   }
6   else if (next_body = OTHER_BODY) {
7     /* other body statements */
8   }
9 }
```

B ESTEREL Semantics

In this section, we give the ESTEREL semantics rules of the subset that we consider. Let's first recall the definition of the semantics transition rule.

Definition B.1 (Esterel semantics)**.** Given a constructive ESTEREL program P and a set of $data$, the semantics of its transition is given by the following relation:

$$P, data \xrightarrow[E_i]{E_o \quad k} P', data'$$

where E_i and E_o are respectively the input and output signal set active on the current reaction, and k, a boolean denoting the termination status (i.e., if it takes time or not)

Then, we give the main semantics rules of ESTEREL. The rules are given with a structure very similar to the ones from PSYC to ease the proof procedure.

$$\text{nothing}, data \xrightarrow[I]{O\ \ 0} \text{nothing}, data$$

$$\text{emit } s, data \xrightarrow[I]{O\cup\{s\}\ \ 0} \text{nothing}, data$$

$$\text{pause}, data \xrightarrow[I]{O\ \ 1} \text{nothing}, data$$

$$v := exp, data \xrightarrow[I]{O\ \ 0} \text{nothing}, data[v \leftarrow [\![exp]\!]_{data}]$$

$$\text{await } n\ s, data \xrightarrow[I]{O\ \ 1} \text{await immediate } ns, data$$

$$\text{await immediate } ns, data \xrightarrow[I]{O\ \ 1} \text{await immediate } n-1s, data \text{ if } s \in I \wedge n > 1$$

$$\text{await immediate } ns, data \xrightarrow[I]{O\ \ 1} \text{await immediate } ns, data \text{ if } s \notin I$$

$$\text{await immediate } 1s, data \xrightarrow[I]{O\ \ 0} \text{nothing}, data \text{ if } s \in I$$

$$\text{if } exp \text{ then } p \text{ else } q \text{ end}, data \xrightarrow[I]{O\ \ 0} p, data \text{ if } [\![exp]\!]_{data}$$

$$\text{if } exp \text{ then } p \text{ else } q \text{ end}, data \xrightarrow[I]{O\ \ 0} q, data \text{ if } \neg[\![exp]\!]_{data}$$

$$\text{while } exp \text{ do } p \text{ end}, data \xrightarrow[I]{O\ \ 0} s\ ;\ \text{while } exp \text{ do } s \text{ end}, data \text{ if } [\![exp]\!]_{data}$$

$$\text{while } exp \text{ do } p \text{ end}, data \xrightarrow[I]{O\ \ 0} \text{nothing}, data \text{ if } \neg[\![exp]\!]_{data}$$

$$p\ ;\ q, data \xrightarrow[I]{O\ \ 0} q, data' \text{ where } p, data \xrightarrow[I]{O\ \ 0}{}^{*} \text{nothing}, data'$$

$$p\ ;\ q, data \xrightarrow[I]{O\ \ 1} p'\ ;\ q, data' \text{ where } p, data \xrightarrow[I]{O\ \ 1} p', data'$$

$$\frac{p_0, data_0 \xrightarrow[I]{O\ \ 0}{}^{*} p_1, data_1 \quad p_1, data_1 \xrightarrow[I]{O\ \ 1} p_2, data_2}{p_0, data_0 \xrightarrow[I]{O\ \ 1} p_2, data_2}$$

C Sources and clocks

In the proofs, we will assume that source clock always tick (as stuttering doesn't change the code). However, PSYC periodic clocks can refine successively source clocks to give richer patterns. Let's remind the ESTEREL translation of a clock `clk_out` with parent `clk_in` and with `Period` and `Offset`:

```
await immediate clk_in;
await [Offset mod Period] clk_in;
loop
  emit clk_out;
each Period clk_in;
```

Without going in the full detail, it is rather obvious that starting on the first tick of `clk_in`, this code generates `clk_out` with `Offset` and `Period` respectively to `clk_in`. So, considering that those are the *relative* period and offset with its parent clock, we can also define the *absolute* period and offset as following:

$$\texttt{Period}^{clk\_out}_{absolute} = \texttt{Period}^{clk\_out}_{relative} \times \texttt{Period}^{clk\_in}_{absolute}$$

$$\texttt{Offset}^{clk\_out}_{absolute} = \texttt{Offset}^{clk\_in}_{absolute} + (\texttt{Offset}^{clk\_out}_{relative} \times \texttt{Period}^{clk\_in}_{absolute})$$

So, similarly, we can rewrite a relative duration (the `advance` parameter) to an absolute duration (always relative to a source). So, assuming d_c is the state of the clock defined as in chapter 5, we have:

$$N_{absolute} = (\texttt{Period}^{clk}_{absolute} - d_c) + (N^{clk}_{relative} - 1) \times \texttt{Period}^{clk}_{absolute}$$

Thus, in the next section, we assume that clocks are always present since 1) durations relative to periodic clocks can be refined to absolute durations with respect to source clocks and 2) when source ticks are absent (stuttering), agent state does not change.

D Equivalence proof

We sketch here the proof of the equivalence between the native semantics and the behavioral semantics of ESTEREL.

Theorem D.1. *For each agent* PSYC p_{ag} *and its* ESTEREL *translation* $T(p_{ag})$*, and for any transition of the form:*

$$C \vdash p_{ag} \Longrightarrow_{n \times s} C' \vdash p'_{ag}$$

Assuming, $C \approx data$ *and* $s \in E$*, we have an equivalent sequence of* ESTEREL *transitions:*

$$T(p_{ag}), data \xrightarrow{E}\!\!\!\!\!\rightarrow P^1, data^1 \ldots \xrightarrow{E}\!\!\!\!\!\rightarrow P^n, data^n$$

with $T(p'_{ag}) = P^n$ *et* $C' \approx data^n$

Proof.

By induction on the structure of the native semantics rule $\Longrightarrow$:

— Case `skip`: Trivially equivalent to the ESTEREL rule (considering $n = 0$) ESTEREL transition.

— Case `compute`: idem.

— Case `while`: idem.

— Case `if`: idem.

— Case `advance N`: First, the initial ESTEREL transition transforms `await N` in `await immediate N` taking one instant. Then, if $N > 1$, the remaining transitions can be proven equivalent by induction on N resulting in $N - 1$ transitions taking $n - 1$ instants. The remaining ESTEREL statements are await immediate 1 ; emit S. The first statement is instantaneous (and thus can be optimized) and the second one is also instaneous but makes communication visible, which is equivalent to the update of the visible environment T in the PSYC rules.

— Case *sequence* s_1 ; s_2: The ESTEREL translation $T(s_1 \; ; \; s_2)$ is equivalent to $T(s_1) \; ; \; T(s_2)$. Thus, if s_1 terminates without taking time in the PSYC rule, its ESTEREL translation $T(s_1)$ also terminates without taking time, and with an equivalent environment, according to the basic cases described above (induction hypothesis). Thus, the remaining statement of the sequence is s_2 in PSYC and equivalently $T(s_2)$ in ESTEREL. Otherwise, if s_1 takes N instants to s_1', it is equivalent to the N ESTEREL transitions to $T(s_1')$ according to the cases described above (induction hypothesis). Thus, the remaining statement is the sequence $s_1' \; ; \; s_2$ in PSYC and equivalently $T(s_1') \; ; \; T(s_2)$ in ESTEREL.

— Case composition rule: This last case directly holds due to the equivalence of the `advance` case. Instantaneous transition are equivalent, thus, they can be composed with the transitions taking time.

□

Méthodologie de vérification formelle de propriétés temporelles pour les applications temps-réel critiques basées sur le concept de temps logique

Résumé

Les systèmes temps-réel critiques doivent respecter des contraintes temporelles strictes qui doivent être considérées tout au long du cycle logiciel. Cependant, du fait que les temps d'exécution exacts ne sont généralement pas connus lors de la conception, le *temps logique* fournit un moyen d'abstraire les contraintes temporelles et les exécutions du *temps physique*, dépendant de la plateforme. Dans cette thèse, nous nous concentrons sur deux formalismes basés sur le temps logique. L'approche *Synchrone-Réactive* abstrait totalement le temps physique en utilisant des bases de temps discrète sur lesquelles les calculs sont déclenchés. L'approche de *Temps d'Exécution Logique* utilise des bases de temps logique pour représenter non seulement les instants de déclenchement, mais aussi les durées des calculs élémentaires. Dans notre travail, nous commençons par définir une unification des approches *Synchrone-Réactives* et de *Temps d'Exécution Logique*, fournissant un cadre formel naturel pour définir la sémantique de PSYC, un langage temps-réel industriel expressif. Nous définissons deux sémantiques pour celui-ci : une sémantique native *à grands pas*, préservant les durées logiques des intervalles de temps, définie par des règles structurelles opérationnelles ; et une sémantique synchrone *à petits pas* définie par traduction vers un langage Synchrone-Réactif étendant les durées des intervalles de temps à une succession de transitions atomiques. Nous montrons que les deux sémantiques sont équivalentes. Cette formalisation de la sémantique de PSYC nous permet de définir une méthodologie de vérification formelle pour PSYC basée sur du *model-checking symbolique*. Pour réduire l'espace d'état pendant le model checking, nous définissons une technique d'optimisation inspirée du *model-checking temporisé*. Enfin, nous spécifions les exigences temporelles que nous voulons vérifier via un langage de spécification de contrainte d'horloge — CCSL — qui sont ensuite traduites en observateurs synchrones.

Mots-clés : Systèmes Temps-Réel, Temps d'Exécution Logique, Langages Synchrone-Réactifs, Vérification Formelle.

Abstract

Safety-critical real-time systems have to respect strict timing constraints. Thus, timing constraints must be considered throughout the software development cycle. As exact computation execution time are generally not known during design, *logical time* provides a way to abstract time constraints and execution from platform-dependent *physical time*. In this thesis, we focus on two main formalisms based on logical time. The *Synchronous-Reactive* approach totally abstracts physical time by discrete time bases on which computations are triggered. The *Logical Execution Time* approach uses logical time bases to represent not only triggering instants but also the durations of elementary computations. In this thesis, we start by unifying *Synchronous-Reactive* and *Logical Execution Time* approaches. This provides the natural formal framework for defining the semantics of PSYC, an expressive industrial real-time language. We define two formal semantics for PSYC: a native *big-step* semantics preserving the logical durations of time intervals defined by structural operational rules and a synchronous *small-step* semantics defined by translation to a Synchronous-Reactive language expanding time interval durations to a succession of atomic transitions. We show that the two semantics definitions are equivalent. This formalization of the PSYC semantics enables us to define a formal verification methodology for PSYC based on *symbolic model-checking*. To reduce the state space during model-checking, we also define an optimization technique inspired by *timed automata* model-checking. Finally, we show how to encode high-level timing requirements into a clock constraint specification language — CCSL — which are then translated to synchronous observers.

Keywords: Real-Time Systems, Logical Execution Time, Synchronous-Reactive Languages, Formal Verification.