Applying Formal Methods to Autonomous Vehicle Control

Thèse de doctorat de l'Université Paris-Saclay
préparée à École Normale Supérieure Paris-Saclay

Ecole doctorale n°580 Sciences et Technologiques de l'Information et de la Communication (STIC)
Spécialité de doctorat : Programmation : modèles, algorithmes, langages, architecture

Thèse présentée et soutenue à Cachan, le 26 novembre 2018, par

YANN DUPLOUY

ACKNOWLEDGMENTS

This PhD thesis could not have been achieved without the help of Maria Climent–Pommeret, Nicolas Dandrimont, Jonathan Lajus, Aurore Moisy-Mabille and Allium. They have been here at what was the worst moment in my life (due to family difficulties that led to extreme exhaustion), accompanied me to get psychiatric help, which greatly helped me to get back on track and produce the work presented in this document.

Also, the biggest of thanks to Béatrice Bérard and Serge Haddad for their quite incredible support; obviously for scientific advices, but also for their contribution to the quality of writing in articles and this thesis. They have pushed their best for getting close to perfection, while understanding my lack of energy and my dilemmas caused by familial issues. Thanks to Benoit Barbot for guiding me on my first collaborative software project.

A lot of thanks to Thao Dang and Isabel Demogodin for reviewing this thesis, and Loïc Hélouet, Leïla Kloul and Nihal Pekergin for accepting to be part of the jury.

Thanks to Émilie Baudet, Gaëlle Carrette, Romain Gatelet, Mamadou Gaye and overall the medical team at *CMP Palaiseau* for the quality of their psychiatric help. Thanks to Fahima Hassani for her help in matters relating to my parents. Thanks to Audrey for your help and the moments of brother-sister relationship we could have despite our age difference and the fact we have been quite separated by various matters.

Thanks to Engel Lefaucheux, Clément Beysson, Julien Reichert for being great friends from my years of scholarship at the ENS Cachan.

Thanks to my parents for giving me a computer when I was 5 years old and introducing me to the joys of programming that early. Sadly, my father died a few months before the completion of this thesis. Thanks to Agathe Herrou, Arnaud Martinet, Sophie Pires Doliget, Théo Marchal, Jacqui Samlone, Vincent Lejeune, François Verbeck, Antoine Roche, Louise Xie, Emmanuel Arrighi and Peggy Boulay Luzy for their financial help for his funeral.

Thanks to *monsieursapin* and *Nat'ali* and their communities for the entertainment (during their streams) and cheerful discussions about videogames. Thanks to the *Pokémon France* community on Discord for being a great place to talk[1] about anything.

Thanks a lot to the LSV past or present members. LSV is a very friendly environment; Barbizon is a great way to have up-to-date information on the lab, but also see the whole lab in a different setting. I enjoyed the various discussions at lunch with Jean Goubault, Hugues Moretto-Viry, Francis Hulin-Hubard, Alexis Goudou, and probably others I have forgotten (sorry!).

Late thanks to *La Cinquième Parallèle* for providing support and knowledge on sex-related and gender-related questions, *La LiKa* for their humour and passion of the improvisiational theatre, and *La Médiatek* for being a nice place to stay on the campus.

Thanks to the IRT SystemX for funding this thesis, thanks to Linda Zhao, Emmanuel Arbaretier and Mohamed Tlig for their help in ensuring the industrial partners of the SVA project understand the usefulness of this work. Thanks to Mohamed Tlig for the advices and the help for the relation with the IRT. Thanks to Paul Labrogère for the opportunities to present my work in industry-focused workshops. Thanks to Jean Van Frank, Rémi Regnier, Mohamed Tlig (and others I may have forgotten) for the friendly discussions.

Last but not least, here is a Pikachu, my most beloved character:

Thanks to Junichi Masuda, The Pokemon Company and Game Freak for all your work on the Pokemon series, and thanks Nintendo for your support of *anytime, anywhere* playing with the Nintendo Switch. It was especially helpful in stressful situations (mostly non-thesis related) to have something to relax.

And thank *you*, reader of the thesis for the interest on this work or reader of this page because you were at the defence;

Yann "pikachuyann" Duplouy

[1]or rather, write

CONTENTS

INTRODUCTION

Context

The population growth in most countries, alongside with the increased distance from home to activities, have led to an ever increasing demand in transportation systems. While train and steam buses exist from the industrial age, the invention of personal cars has drastically changed the behaviour of the travelers. More precisely, with the introduction of the Ford Model T in 1908, automobiles became affordable for the middle-class, introducing a better option than public transportation. Unfortunately, the increasing car traffic has triggered some issues: driving induces strain, the increasing number of accidents is a major preoccupation for public safety, and the circulation conditions are degrading. In order to address these issues, a range of advanced driver-assistance systems have been introduced thanks to breakthrough in computation power and electronics.

Today, smartphones and mobile networks enable communication while traveling. However, drivers have to focus on the road circulation. So the next step is to make personal cars capable of performing the entire process of of going to some destination: the so-called *autonomous vehicles* (or self-driving cars). It could even be a part of public transportation, such as the planned night shuttle between Massy and Polytechnique [30]. Waymo and Tesla are known to be pioneers in fully driver-less vehicles ([44], [61]), but it is now a concern of all car manufacturers. Obviously, these companies want to sell these new vehicles to the general public. But this requires ensuring that such vehicles do not increase the risk of accidents, nor pose a threat to the life of their owners. Since the question of safety has an high societal impact and is difficult to solve, most companies want to let their vehicles drive enough kilometers, in order to have sufficient data and design improvements. However, this comes with a huge legislation problem: they must have the right to drive those possibly dangerous cars. In France, the autonomous vehicles are limited to small portions of motorways [62], which can only be a benchmark for *Trafic Jam Controller*s. In the United States, the companies started to test their autonomous vehicles in California and Texas, but this had led to several accidents and the public opinion became uncomfortable with the matter.

Similarly to the crash tests for the safety of drivers and passengers during a crash (which is now entirely done, in Europe, via EURO NCAP test benches [38]), this requires a shift to more abstract verification methods. Such approach has two advantages: first, it does not take as much time nor money to get the target precision on the crash rate (10^{-9} crashes every 100km driven), and most importantly it does not involve taking any vital risk, and formal verification offers better guarantees than testing [22].

Related works. Since an autonomous vehicle is very complex by nature, any direct verification task is extremely difficult. Hence, researchers have defined and tackled many subproblems, like: the effects of weather perturbation on sensors [60], the detection of road signs [65] and other entities [67], inter-vehicle communication [66], human-vehicle interaction [54], design of controllers for road parts [25] and for the single vehicle.

In this thesis, we focus on the verification of such controllers. Several related works will be presented in Chapter 2. They can be classified according to two criteria:

- Geometrical versus logical. In the former case, the behaviour of the vehicles in a small time interval is modeled by the set of all positions they occupy, using differential equations, and a check is performed on the interserction of these sets; In the latter case, the expected properties on vehicles and roads are formalised as logical constraints, which are then solved by existing tools.
- Synthesis versus verification. In the former case, a controller satisfying the expected properties is automatically generated. In the latter case, the expected properties are checked on an already existing controller.

Objectives

This PhD thesis takes place in the IRT SystemX project *Simulation pour la Sécurité du Véhicule Autonome* [68], whose aim is to provide simulation methods ensuring the safety of autonomous vehicles. This project includes as partners various automobile manufacturers such as PSA Peugeot Citroën and Renault, but also suppliers such as Continental and Valeo. It is also in partnership with software developers, such as ANSYS, Oktal, Apsys-Airbus. In this thesis, we propose a statistical model-checking based approach which could be used by automobile manufacturers who design controllers of autonomous vehicles.

More precisely, our objectives are as follows.

Formalism. As a first step, it is necessary to model the scenarios for autonomous vehicles with an appropriate formalism. This formalism should be expressive enough for quantifying risks and describing various situations. High-level stochastic Petri nets (see for instance [31]) are such a candidate. However, it misses some features of cyber-physical systems such as continuous evolution that occurs in these scenarios. Thus, our first objective is to extend the high-level stochastic nets to include these features.

Simulink® Integration. The industrial partners of the project use Simulink as a modeling tool to design controllers for their autonomous vehicles. This choice is motivated by the sheer number of toolboxes and simulation tools that can be connected to Matlab-Simulink, namely SCANeR (developed by Oktal, partner of the project) that serves as another tool for the simulation in the SVA project. However, the input formalism of Simulink is not compatible with Cosmos, and the documentation given by MathWorks is at best incomplete. Thus, this requires to define a formal operational semantics for Simulink. Such a semantics could be useful outside the scope of the project, emphasizing the interest of this objective. Moreover, we want to combine the high-level Petri nets with Simulink to introduce a new stochastic hybrid system formalism that could entirely describe controlled vehicle case studies.

Cosmos Implementation. These formalisms should then be implemented into Cosmos in order to run simulations and produce relevant performance indices. This requires to also study scalability issues, and undesirable features like the rare event phenomenon, which leads to our final objective.

Empirical validation. The final objective is to design case studies that illustrate the capacities of the newly designed formalism in the context of autonomous vehicle verification. This brings two main questions: Are we able to model realistic scenarios with this formalism? Are the performances of the simulation efficient enough to manage the analysis of the case studies in an iterative process?

Organisation

This thesis is divided in two parts and five chapters.

Part I - Context. The first chapter gives a background of the computer science and mathematical concepts about formal verification. It is mostly theoretical, but also gives a comparison between various tools for statistical model-checking, enforcing the choice of Cosmos in the thesis. The second chapter is a brief state-of-the-art on approaches for the control of autonomous vehicles.

Part II - Contributions. The third chapter is dedicated to the definition of an operational semantics for Simulink models, and proceeds in three steps: first, we define a proper syntax for the block diagrams of Simulink. Secondly, we define an exact semantics which will be used as a target for the accuracy of the operational semantics. Finally, we define an operational semantics in the form of an algorithm. The fourth chapter starts by a brief description of Cosmos workflow, then presents the extensions brought to Cosmos. We then show how, and when, the performances for high-level nets have been largely improved. Finally, we explain how our operational semantics for Simulink were integrated into the tool, and how the model combining high-level stochastic Petri nets and Simulink is handled via a multi-model simulation on a single queue. Finally, the last chapter presents two case studies: the first one is a heavy traffic on a motorway; the second one is an entrance ramp.

Part I

Context

CHAPTER 1 MODELING AND ANALYSIS OF CYBERPHYSICAL SYSTEMS

Abstract. A cyberphysical system consists of a physical part handled by a set of softwares. The analysis of such a system requires three stages. First one has to specify it based on some formal model. Then the expected properties must be expressed in some logic. Once the two stages are achieved, the model and the formulas are provided as inputs of a model-checker that verifies whether the formulas are satisfied by the model and in the negative case produces some counter-example. Figure 1.1 depicts this process. The system designer must select an appropriate formalism depending on the features that occur in the system (discrete and/or continuous time, discrete and/or continuous space, non deterministic and/or probabilistic behavior, etc.). So we present several formalisms for system modelling in Section 1.1. Similarly in Section 1.2 we give an overview of the (temporal) logics that may be relevant to express the properties. Section 1.3 ends the chapter by an overview of several verification methods and tools, some of them being applied in this work.

Contents

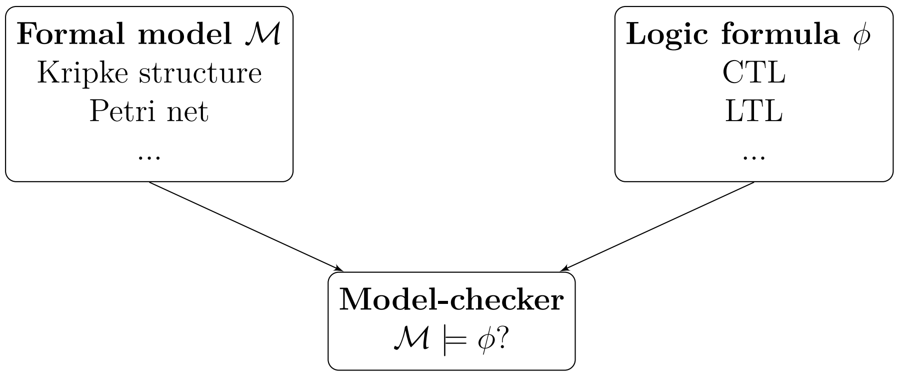


Figure 1.1: Model-checking answers the question: does $\mathcal{M}$ satisfies ϕ?

1.1 Formal Specifications of Systems

The first step of the process shown in Figure 1.1 consists in specifying a formal model of the system expressed with some formalism. So here, we present some of them. *Transition Systems* which is the basic one is not used for design. However it is provided as a semantic for more concise formalisms and for defining the truth value of a formula. Still at the semantical level, *Discrete Event Stochastic Processes* may be viewed as an extension of Transition Systems by integrating probabilities and time. At a syntactical level, we introduce *Petri Nets* and their extensions which are widely used for concurrent and distributed systems. Finally, we present *Hybrid Systems* which allow to describe a continuous dynamics and thus to express cyberphysical systems.

1.1.1 Transition Systems

As discussed above, transition systems is the standard low-level formalism. It mainly consists of a set S of *states* (which may be uncountable) and a *transition relation* $\rightarrow \subseteq S \times S$ describing state changes. We first present the particular case of Kripke structures, which have a finite number of states equipped with *labels* representing *atomic properties* that hold in the state.

Definition 1 (*Kripke structure*)

Let AP be a set of atomic propositions.
A Kripke structure *is a 4-uple $\mathcal{K} = (S, I, R, \Sigma, L)$ where:*

- *S is a finite set of states;*
- *$I \subseteq S$ is the subset of initial states;*
- *$R \subseteq S \times S$ is a transition relation satisfying:*

$$\forall s \in S : \exists s' \in S : (s, s') \in R$$

- *$\Sigma = 2^{AP}$ is the set of subsets of atomic propositions;*
- *$L : S \to \Sigma$ is a labelling function*

Remark. *The restriction on the transition relation means that for every state there always exists a transition starting from that state. In fact this restriction can be easily overcomed by adding a special idle state with a loop and such that for all* deadlock *state (i.e. a state without outgoing transitions) one adds a transition to the idle state. Moreover a special property, say deadlock, would only hold in the idle state in order to reason about it in formulas.*

Often, one also includes in the definition of a transition system a subset of initial states S_0 which is sometimes a singleton $\{s_0\}$.

Example 1. A Kripke structure modeling a (very simple) coffee machine is shown in Figure 1.2; the labels on arcs are only provided here for readability of the example, but are not a part of the Kripke structure. The atomic property p means that the machine is busy and the atomic property c means that the coffee is filling the cup. If there is no fairness condition, this machine may prepare coffee indefinitely.

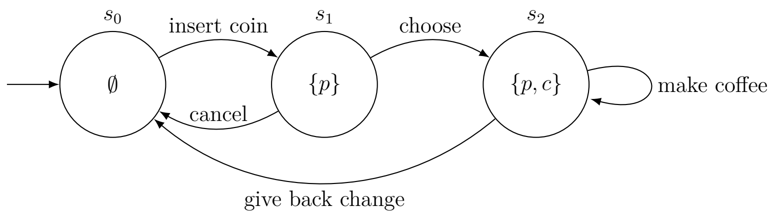


Figure 1.2: An example of Kripke structure : a coffee machine

Definition 2. *An* execution *or* run *of a Kripke structure $\mathcal{K}$ is an* infinite *sequence of states $\sigma = s_0, \ldots, s_n, \ldots$ such that, for every $i \in \mathbb{N} : (s_i, s_{i+1}) \in R$. This run generates an (infinite) word $w = L(s_0) \cdots L(s_n) \cdots \in \Sigma^\omega$, called the* trace *of σ. The* language *of $\mathcal{K}$, denoted by $\mathcal{L}(\mathcal{K})$, is the set of words generated by runs of $\mathcal{K}$.*

Remark. *When S_0 is specified one restricts the runs to those starting in S_0.*

Example 2.

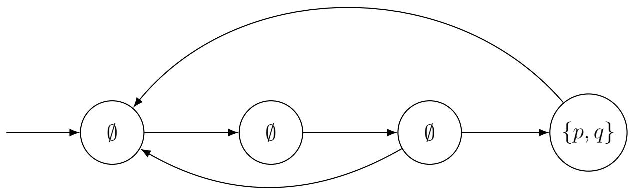


Figure 1.3: Another example of a Kripke structure

The word $(\emptyset\emptyset\emptyset\{p,q\})^\omega$ is generated by the Kripke structure shown in Figure 1.3.

1.1.2 Discrete Event Stochastic Processes

We extend transitions systems by adding probabilities and time to the formalism.

Notation 3. *We denote by $P(e)$ the probability of an event e, and by $P(e|e')$ the conditional probability of e given e' has happened. We denote by $E(X)$ the* expectation *of the (numerical) random variable X. Given a set of* basic events *E equipped with a σ-algebra, we denote by $\mathsf{dist}(E)$ the set of distributions over E.*

Remark. *Observe that the word* event *is overloaded. Within a σ-algebra it denotes a measurable subset while in a transition system it denotes what triggers the state change. In the sequel, the context should clarify which notion is used.*

Behaviors for stochastic discrete event systems can be modeled via two families of random variables:

- $(S_n)_{n\in\mathbb{N}}$ taking values in the system state space. The random variable S_0 represents the initial state of the system, and S_n the state after the n^{th} event.
- $(T_n)_{n\in\mathbb{N}}$ taking values in $\mathbb{R}^+$. The random variable T_0 corresponds to the time elapsed before the first event, and T_n to the time between the n^{th} and the $(n+1)^{\text{th}}$ event. Note that there may be zero delay between two events, like in the case of a sequence of instantaneous instructions.

Note that this definition implies that events are instantaneous, and that there might be a null delay between two events.

Example 3.

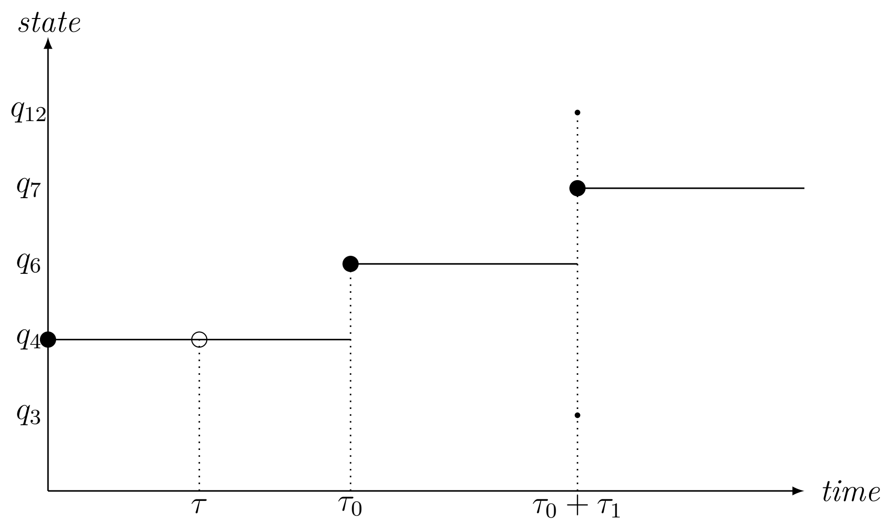


Figure 1.4: A realization of a discrete event stochastic process

The figure above illustrates a discrete event system where the first *realizations* of its random variables are defined as follows:

- $S_0 = q_4$, $S_1 = q_4$, $S_2 = q_6$, $S_3 = q_3$, $S_4 = q_{12}$ et $S_5 = q_7$;
- and $T_0 = \tau_0$, $T_1 = \tau_1$, $T_2 = T_3 = 0$.

According to physical constraints, the only restriction for these families of random variables is to forbid Zeno phenomenon, where an infinite number of actions can occur in a finite duration. This is mathematically equivalent that almost surely $\sum_{n=0}^{\infty} T_n = +\infty$.

Example 4. In this section, we use the Tandem Queue Network (TQN) illustrated in Figure 1.5 as a running example. This system consists of two queues: the event *in* represents the arrival of a client in the first queue; *move* indicates the move of a client from the first to the second queue; finally, *out* represents a client leaving the system. While the first event could happen at any moment, the other events require the presence of at least one client in their corresponding queue.

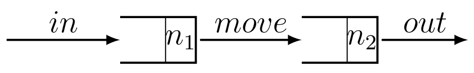


Figure 1.5: Tandem Queue

Since we have not yet specified its probabilistic behavior, this high level model leads to the transition system shown in Figure 1.6 which a countable number of states, each state being specified by the numbers of clients in the two queues.

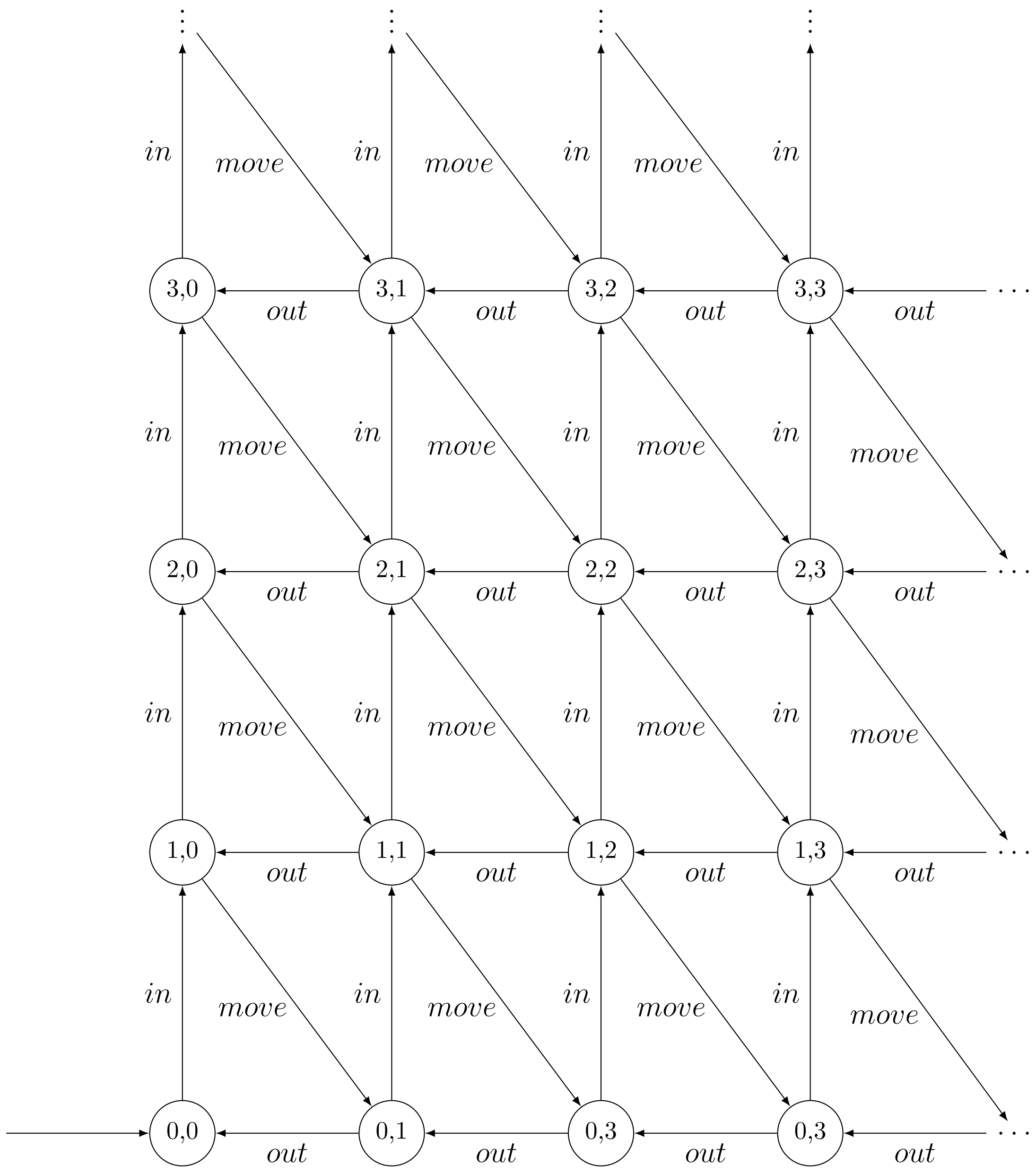


Figure 1.6: Tandem queue, seen as a transition system

For modeling purposes, it is useful to have a syntactical formalism for which the semantics are described by the two random variable families $(S_n)_{n\in\mathbb{N}}$ and $(T_n)_{n\in\mathbb{N}}$ previously defined.

Definition 4 (*Discrete Event Stochastic Process*)

A Discrete Event Stochastic Process (DESP) is a tuple $\mathcal{D} = (S, \pi_0, E, \mathsf{Ind}, \mathsf{enabled}, \mathsf{delay}, \mathsf{choice}, \mathsf{target})$ *where:*

- S *is a (possibly infinite) set of states;*
- $\pi_0 \in \mathsf{dist}(S)$ *is the initial distribution over states;*
- E *is a finite set of events;*
- Ind *is a set of state indicators, i.e. functions* $S \to \mathbb{R}$ *that may be constants;*
- $\mathsf{enabled} : S \to \mathcal{P}(E)$ *indicates the set of enabled events for each state, satisfying: for all* $s \in S, \mathsf{enabled}(s) \neq \emptyset$*;*
- $\mathsf{delay} : S \times E \to \mathsf{dist}(\mathbb{R}^+)$ *is a mapping defined for pairs* (s, e) *such that* $s \in S$ *and* $e \in \mathsf{enabled}(s)$*;*
- $\mathsf{choice} : S \times \mathcal{P}(E) \times \mathbb{R}^+ \to \mathsf{dist}(E)$ *is a mapping defined for triples* (s, E', d) *such that* $E' \subseteq \mathsf{enabled}(s)$ *and the possible outcomes of the distribution are restricted to* $e \in E'$*;*
- $\mathsf{target} : S \times E \times \mathbb{R}^+ \to S$ *is a mapping indicating the state changes triggered by events, defined for triples* (s, e, d) *with* $e \in \mathsf{enabled}(s)$.

The set $\mathsf{enabled}(s)$ is the set of events enabled from state s. The delay function, when applied to a pair (s, e) gives the distribution of the delay between the activation of e in state s and its possible occurrence. The function choice gives an additional probabilistic distribution in the case of several events having the same due date, to randomly solve conflicts. The function target denotes the (random) destination state given the source state, the occurrence of an event and the sojourn time in the source state. The set of functions Ind are related to the observed behavior of the model and will be used for specifying formulas.

Example 5. In the case of the tandem queues (see Figure 1.5), the set of states is $\mathbb{N} \times \mathbb{N}$, $\pi_0 = \text{Dirac}\,((0,0))$ meaning that the initial distribution is concentrated on state $(0,0)$, $E = \{in, move, out\}$. As discussed above,

$$\mathsf{enabled}(s) = \begin{cases} \{in\} & \text{when } s = (0,0), \\ \{in, out\} & \text{when } \exists n \in \mathbb{N}^+ \text{ such that } s = (0,n), \\ \{in, move\} & \text{when } \exists n \in \mathbb{N}^+ \text{ such that } s = (n,0), \\ \{in, move, out\} & \text{otherwise.} \end{cases}$$

Let event in follow an exponential law of parameter λ denoted by $\exp(\lambda)$ with density $x \mapsto \lambda e^{-\lambda x}$ and events $move$ and out follow exponential laws of respective parameters μ_0 and μ_1, we have, for $s \in S$ and $e \in \mathsf{enabled}(S)$:

$$\mathsf{delay}(s, e) = \begin{cases} \exp(\lambda) & \text{if } e = in, \\ \exp(\mu_0) & \text{if } e = move, \\ \exp(\mu_1) & \text{if } e = out. \end{cases}$$

The function choice is irrelevant as we use continuous laws: the simultaneous occurrence of two events has null probability. We might for example use $\mathsf{choice}(s, E', t) = \mathrm{U}(E')$ (a uniform distribution). Finally, as can be seen in Figure 1.6, $\mathsf{target}((n, m), e, d)$ is defined as follows for $\mathsf{enabled}(s)$:

$$\mathsf{target}((n,m),e,d) = \begin{cases} (n+1, m) & \text{if } e = in, \\ (n-1, m+1) & \text{if } e = move, \\ (n, m-1) & \text{if } e = out. \end{cases}$$

Observe that here the target function does not depend on the sojourn time.

Definition 5. *A* configuration *of a DESP is a triple* $(s, \tau, \mathsf{sched})$ *where s is the current state, $\tau \in \mathbb{R}^+$ the current time and* $\mathsf{sched} : E \to R^+ \cup \{\infty\}$ *the function describing the times at which each enabled event will occur ($+\infty$ if the event cannot be activated).*

Informal semantics of a DESP. Starting from a given configuration $(s, \tau, \mathsf{sched})$ of a DESP, its dynamics corresponds to an infinite loop, where each iteration consists of the following steps:

1. Using sched, one get the set $E' = \{e \in \mathsf{enabled}(s) \mid \forall e' \in \mathsf{enabled}(s) : \mathsf{sched}(e) \leq \mathsf{sched}(e')\}$ the set of enabled events with minimum delay. The corresponding delay d is then $\mathsf{sched}(e) - \tau$ for all $e \in E'$. If E' contains several elements, we then use a random variable of distribution $\mathsf{choice}(s, E', d)$. The corresponding sampling provides an event e, which will be activated.
2. The next state s' is then defined by $s' = \mathsf{target}(s, e, d)$ and sched is updated: for all event $e' \neq e$ still enabled, the value of $\mathsf{sched}(e')$ is kept. For any other event e' that is enabled in s', we sample a new delay d' using $\mathsf{delay}(s', e')$ and then $\mathsf{sched}(e') = \tau + d + d'$. For all disabled event e' in s', $\mathsf{sched}(e') = +\infty$.

The initial configuration is obtained by sampling s using the distribution π_0, τ is the initial time ($\tau = 0$) and sched is obtained by sampling every event delay of $\mathsf{enabled}(s)$.

A particular case of DESP corresponds to the classical formalism of Discrete Time Markov Chain (DTMC).

Definition 6. *A square matrix* $\mathbf{P} \in \mathbb{R}^{S \times S}$ *is* stochastic *(or a* transition matrix*) if all its coefficients are non-negative and the sum of coefficients over each line is equal to 1, i.e.:*

$$\forall i, j \in S, \mathbf{P}(i,j) \geq 0 \text{ and } \sum_{j \in S} \mathbf{P}(i,j) = 1$$

Definition 7 (*Discrete-Time Markov Chain*)

A sequence $(X_k)_{k \geq 0}$ of random variables with values in S is a Markov chain *of*

transition matrix $\mathbf{P}$ *and initial law* π_0 *if:*
- $P(X_0 = i) = \pi_0[i]$ *for every* $i \in S$;
- *for all* $k \in \mathbb{N}$ *and* $i \in S$:

$$P(X_{k+1} = i \mid X_k = i_k, \dots X_0 = i_0) = P(X_{k+1} = i \mid X_k = i_k) = \mathbf{P}(i_n, i).$$

When the initial distribution is a Dirac distribution, the initial state is usually denoted s_0. The sequence $(X_k)_{k \in \mathbb{N}}$ is a stochastic process. This definition highlights the characteristic property of Markov chains: the future evolution of the process is independent from its past evolution; it only depends on its current state.

Definition 8 (*Graph of a DTMC*)

A DTMC is represented by a weighted directed graph:
- *The set of vertices is* S.
- *Ther is an edge* $i \to j$ *with weight* $\mathbf{P}(i,j)$ *if* $\mathbf{P}(i,j) > 0$.

In addition to the interest of a graphical representation (see Figure 1.7) which may be more readable for small DTMC, the analysis of such graphs leads to characterizations of properties of the states like transience, recurrence, periodicity, etc.

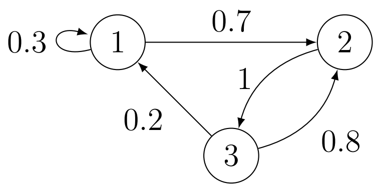


Figure 1.7: The graph of a finite DTMC

Implicitly in DTMC, an event occurs every time unit which implies that w.r.t. time elapsing DTMC are not memoryless. A contrario Continuous Time Markow Chains (CTMC) are "fully" memoryless since the time between two events follows an exponential distribution whose rate only depends on the current state.

Definition 9 (*Continuous Time Markov Chain*)

Two sequences $(X_k)_{k \geq 0}$ *and* $(T_k)_{k \geq 0}$ *represent a* CTMC *of transition matrix* $\mathbf{P}$, *initial distribution* π_0 *and rate* $\{\lambda_i\}_{i \in S}$ *if:*
- *The sequence* $(X_k)_{k \in \mathbb{N}}$ *is a DTMC with transition matrix* $\mathbf{P}$ *and initial distribution* π_0;
- *for all* $k \in \mathbb{N}$ *and* $\tau \in \mathbb{R}^+$:

$$\begin{aligned} &P(T_k \leq \tau | X_k = i_k, \dots, X_0 = i_0, T_0 \leq \tau_0 \dots T_{k-1} \leq \tau_{k-1}) \\ &= P(T_k \leq \tau | X_k = i_k) = 1 - e^{-\lambda_{i_k} \tau} \end{aligned}$$

The DTMC associated with the CTMC is called the *embedded* DTMC. The graph of a CTMC is the one of embedded DTMC where the states have for labels their rate (see Figure 1.8).

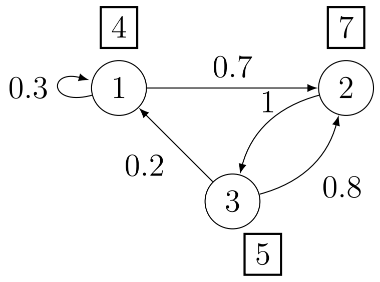


Figure 1.8: The graph of a finite CTMC

1.1.3 Petri Nets

Carl Adam Petri, a German mathematician, has defined during his PhD [57] a formalism for describing relations between conditions and events, thus modeling the behavior of distributed discrete event systems. In this section, we give a short presentation of this model, which is now called Petri net, and of some of its extensions.

Definition 10 (*Petri net*)

A Petri net is a 5-tuple $\mathcal{N} = (P, T, W^-, W^+, m_0)$ where $P \cap T = \emptyset$ such that:
- *P is a non-empty finite set of places;*
- *T is a non-empty finite set of transitions;*
- *$W^- : P \times T \to \mathbb{N}$ is the forward incidence function;*
- *$W^+ : P \times T \to \mathbb{N}$ is the backward incidence function;*
- *$m_0 \in \mathbb{N}^P$ is the initial marking of the net.*

A Petri net is represented by a bipartite graph where:
- vertices are places depicted as circles and transitions depicted as rectangles;
- there is an edge from place t to transition t labelled by $W^-[p, t]$ if $W^-[p, t] > 0$;
- there is an edge from transition t to place p labelled by $W^+[p, t]$ if $W^+[p, t] > 0$;
- Inside place p, one writes either the number $m_0(p)$ of tokens or represent this number (when enough small) by bullets.

When the label of an edge is missing, it means that this label is 1.

Example 6.

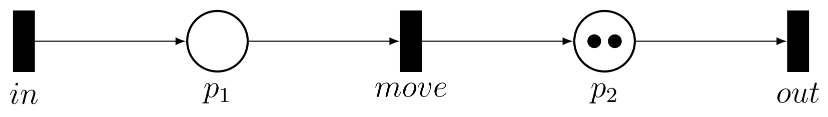


Figure 1.9: A Petri net

The net shown in Figure 1.9 models the tandem queue where the marking of p_1 (respectively p_2) indicates the number of clients in the first (respectively second) queue of the tandem.

A *marking* of a Petri net $\mathcal{N}$ is a mapping $m \in \mathbb{N}^P$ that associates an integer with each place of the net.

A transition $t \in T$ is *firable* in a marking m if $\forall p \in P : m(p) \geq W^-(p,t)$. The firing of such a transition from m leads to a new marking m' defined by: $\forall p \in P : m'(p) = m(p) - W^-(p,t) + W^+(p,t)$. This firing is denoted by $m \xrightarrow{t} m'$.

Let $\sigma = t_1 \dots t_n \in T^*$ be a sequence of transitions. Then σ is firable and and leads to marking m' (written $m \xrightarrow{\sigma} m'$) if there exists a sequence of markings $m_0 \dots m_n$ such that $m = m_0$ and $m' = m_n$ and $\forall 0 \leq k < n, m_k \xrightarrow{t_{k+1}} m_{k+1}$.

We denote by $\mathrm{Reach}(\mathcal{N}, m_0) = \{m | \exists \sigma \in T^* : m_0 \xrightarrow{\sigma} m\}$ the set of *reachable markings.*

Example 7. In Figure 1.9, the initial marking is $m_0 = (0,2)$ and a possible sequence of transitions would be:

$$(0,2) \xrightarrow{in} (1,2) \xrightarrow{out} (1,1) \xrightarrow{move} (0,2)$$

Transitions *in* and *out* are firable from initial marking m_0 while *move* is not.

It is possible to add (quantitative) time in Petri nets, with two main approaches:
- either Time Petri nets [23] in which transitions are equipped with intervals constraining their firing;
- or Timed Petri nets [1] where clocks are added to tokens, and edges are equipped with guards.

The first approach can be extended to obtain stochastic Petri nets, by defining a probabilistic distribution over the time interval of the transition delay. In addition, one has to solve conflicts between simultaneous occurrences of transitions and this is done with the help of priorities and weights.

Definition 11 (*Stochastic Petri net*)

A stochastic Petri net *is a 8-tuple* $\mathcal{N} = (P, T, W^-, W^+, m_0, \Lambda, \Pi, w)$ *where:*
- (P, T, W^-, W^+, m_0) *is a Petri net;*
- $\Lambda : T \to \mathsf{dist}(\mathbb{R})$ *is a function that associates a distribution with each transition;*
- $w : T \to \mathbb{R}$ *a function that associates a weight with each transition;*
- $\Pi : T \to \mathbb{N}$ *a function that associates a priority with each transition.*

Remark. *This definition could be extended by considering Λ as a mapping from markings to rates.*

Stochastic nets can be seen as a particular case of DESP with states $\mathbb{N}^P$, the initial distribution over states $\pi_0 = Dirac(m_0)$, no indicator, and:

- $\mathsf{enabled}(m) = \{t \in T \mid \forall p \in P : W^-(p,t) \leq m(p)\}$;
- $\mathsf{delay}(m,t) = \Lambda(t)$;
- $\mathsf{choice}(m, T', d)$ is a distribution defined by the following procedure: select the subset $T_{max} = \arg\max\{\Pi(t) \mid t \in T'\}$ and then sample some t in T_{max} with distribution $\frac{w(t)}{\sum_{t' \in T_{max}} w(t')}$;
- and $\mathsf{target}(m,t,d) = m'$ with $m'(p) = m(p) - W^-(p,t) + W^+(t,p)$ for all p in P.

We now present some high level Petri nets, in which tokens are labelled with information and transition can be fired in multiple ways Let us first recall the a natural extension of sets: *multisets*. A multiset may contain multiple occurrences of the same element.

Definition 12. *A* multiset *is a pair (S,μ) where S is a set and $\mu : S \to \mathbb{N}$ is a mapping from the elements of the set to natural numbers.*

Notation 13. *We denote by $\mathsf{Bag}(S)$ set of multisets using the support set S. A multiset μ is usually written as a symbolic sum $\sum_{s \in S} \mu(s) \cdot s$ with term omitted when $\mu(s) = 0$ and factor omitted when $\mu(s) = 1$.*

Example 8. Using this notation, $2 \cdot b + c$ and $2 \cdot a + b + 3 \cdot c$ are multisets over the set $\{a, b, c\}$.

Definition 14 (*High-level Petri net*)

A high-level Petri net is a 8-tuple $\mathcal{N} = (P, T, W^-, W^+, m_0, C, \mathsf{Domain}, G)$ with:

- *P and T the respective non-empty sets of places and transitions, with $P \cap T = \emptyset$;*
- *$C = \{C_1, \ldots, C_k\}$ a set of finite color classes;*
- *$\mathsf{Domain} : (P \cup T) \to C^*$ a function that associates with each place and transition a sequence of color classes, representing a cartesian product $C_{i_1} \times \cdots \times C_{i_l}$;*
- *W^- and W^+ are respectively forward and backward incidence matrices. For each $p \in P$ and $t \in T$: $W^-(p,t)$ and $W^+(p,t)$ are mapping from $\mathsf{Domain}(t)$ to $\mathsf{Bag}(\mathsf{Domain}(p))$;*
- *G the* guard *is a mapping from T such that $G(t)$ is a subset of $\mathsf{Domain}(t)$;*
- *m_0 the initial marking with $m_0(p) \in \mathsf{Bag}(\mathsf{Domain}(p))$.*

In order to define the incidence functions W^- and W^+, it is necessary to introduce a typed variable per color class occurences of the cartesian product of $\mathsf{Domain}(t)$. They are used to build expressions with values in $\mathsf{Bag}(\mathsf{Domain}(P))$, For instance, with $\mathsf{Domain}(T) = C_1 \times C_1 \times C_2$ and $\mathsf{Domain}(P) = C_1 \times C_2$ such a mapping could be:

$$\begin{array}{rcl} C_1 \times C_1 \times C_2 & \rightarrow & \mathsf{Bag}(C_1 \times C_2) \\ < x_{1,1}, x_{1,2}, x_{2,1} > & \mapsto & 2\cdot < x_{1,1}, x_{2,1} > + < x_{1,2}, x_{2,1} > \end{array}$$

whose type of variable $x_{i,j}$ is C_i.

Example 9.

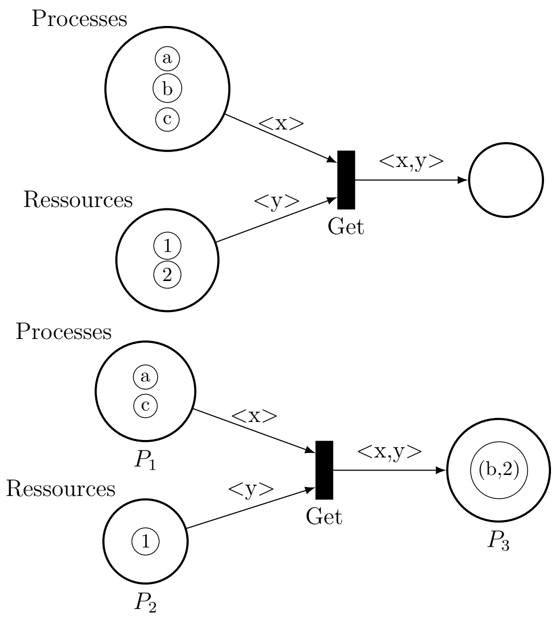


Figure 1.10: A simple high-level Petri net

A simple high-level Petri net is shown in Figure 1.10, with two color classes: Ressources and Processes. The domain of place P_3 is the cartesian product Processes $\times$ Resources. Firing transition Get corresponds to a process acquiring a resource (like a file). In the low part of the figure, a new marking is shown that is obtained after firing Get with x bound to process b and y bound to resource 2.

The incidence functions of this example are:

$$
\begin{array}{rrcl}
W^{-}(Get, P_1) = & \text{Processes} \times \text{Ressources} & \to & \mathsf{Bag}(\text{Processes}) \\
 & <x,y> & \mapsto & <x> \\
W^{-}(Get, P_2) = & \text{Processes} \times \text{Ressources} & \to & \mathsf{Bag}(\text{Processes}) \\
 & <x,y> & \mapsto & <y> \\
W^{+}(Get, P_3) = & \text{Processes} \times \text{Ressources} & \to & \mathsf{Bag}(\text{Processes} \times \text{Ressources}) \\
 & <x,y> & \mapsto & <x,y>
\end{array}
$$

Stochastic Petri nets and high-level Petri nets can be combined to define stochastic high-level Petri nets which is indeed the input formalism of the tool Cosmos used in our case studies.

1.1.4 Hybrid systems

Hybrid systems [43] combine continuous dynamics, i.e. evolution of variables according to flow functions (possibly described by differential equations) in control locations, and discrete jumps between these locations, equipped with guards and variable updates.

Example 10. A classical example is the thermostat depicted in Figure 1.11, controlling a heater by turning it on or off to adjust the room temperature represented by variable x, maintaining it between x_{min} and x_{max}. When the heater is off, the temperature decreases according to differential equation $\dot{x} = -Kx$ and when the heater is on, it increases according to $\dot{x} = K(x_h - x)$, where x_h and K are parameters for the heater and the room.

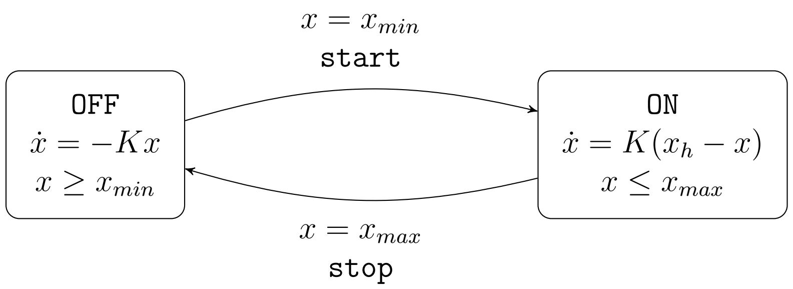


Figure 1.11: Room temperature controller

Formally, hybrid automata over $\mathbb{R}$ can be defined as follows:

Definition 15 (*Hybrid automaton*)

A hybrid automaton of dimension n is a tuple $\mathcal{H} = (S, \Sigma, E, Dyn, Inv, \mathcal{G}, \mathcal{R})$,

where:
- *S is a finite set of states;*
- *Σ is a finite alphabet of events;*
- *$E \subseteq S \times \Sigma \times S$ is a finite set of edges;*
- *Dyn is the* dynamics *assigning to each state a set of differential equations over $\mathbb{R}^n$;*
- *Inv assigns to each state a subset of $\mathbb{R}^n$ called* invariant*;*
- *$\mathcal{G}$ assigns to each edge a subset of $\mathbb{R}^n$ called* guard*;*
- *$\mathcal{R}$ assigns to each edge a subset of $\{1, \ldots, n\}$ called* reset.

In the example above, $n = 1$, $S = \{\texttt{OFF}, \texttt{ON}\}$, $\Sigma = \{\text{stop}, \text{start}\}$, $E = \{(\texttt{OFF}, \text{start}, \texttt{ON}), (\texttt{ON}, \text{stop}, \texttt{OFF})\}$, $Dyn(\texttt{OFF}) = \{\dot{x} = -Kx\}$, $Dyn(\texttt{ON}) = \{\dot{x} = K(x_h - x)\}$, $Inv(\texttt{OFF}) = \{x \in \mathbb{R} \mid x \geq x_{min}\}$, $Inv(\texttt{ON}) = \{x \in \mathbb{R} \mid x \leq x_{max}\}$, $\mathcal{G}(\texttt{OFF}, \text{start}, \texttt{ON}) = \{x_{min}\}$, $\mathcal{G}(\texttt{ON}, \text{stop}, \texttt{OFF}) = \{x_{max}\}$ and $\mathcal{R}(\texttt{OFF}) = \mathcal{R}(\texttt{ON}) = \emptyset$.

Definition 16

The semantics of hybrid automaton $\mathcal{H}$ is defined as the transition system $\mathcal{T}_{\mathcal{H}} = (Q, \to)$ where:
- *the set of configurations is $Q = S \times \mathbb{R}^n$;*
- *the transition relation contains two types of steps:*
  - *discrete step: $(s_1, y_1) \to (s_2, y_2)$ iff there exists $e = (s_1, a, s_2)$ in E with $y_1 \in \mathcal{G}(e)$ and $y_2[i] = 0$ for $i \in \mathcal{R}(e)$ and $y_2[i] = y_1[i]$ otherwise.*
  - *continuous step: $(s_1, y_1) \to (s_2, y_2)$ iff $s_1 = s_2$ and there exist $t \leq t'$ and a continuous function $f : [t, t'] \to \mathbb{R}^n$ solution of the differential equations in $Dyn(s_1)$ such that $y_1 = f(t)$ and $y_2 = f(t')$, with $f(t'') \in Inv(s_1)$ for all $t'' \in [t, t']$.*

For this expressive formalism, where the associated transition system has an uncountable state space, most verification questions are undecidable and in particular the reachability problem. Analysis results were obtained for subclasses, usually by building a finite abstraction based on some *bisimulation* equivalence, preserving a specific class of properties, like reachability or those expressed by temporal logic formulas.

Observations. While this definition seems rather general, it suffers from some limitations. For instance, the specification of the differential equations only depends on the current state and not on the whole history. In Chapter 3, a more general formalism based on Simulink will be detailled.

There are several ways to introduce probabilities in such systems: either by associating weights with discrete transitions or associating random delays with states. In Chapter 4, we propose another way to do so.

1.2 Formal Specifications of Properties

Generally properties are specified using some logic. In the case of dynamical systems temporal logics that express properties related to runs of sytems are the most appropriate ones. Following our presentation of formalisms, we begin by temporal logics for discrete event systems and then we introduce some of their extensions introduced to capture features of stochastic and/or hybrid systems.

1.2.1 Linear Temporal Logic

The aim of temporal logics is to express properties over system executions. Two approaches are possible: linear logics like LTL (Linear Temporal Logic) which consider sequences independently from each other, and tree logics such as CTL (Computation Tree Logic) which consider the execution tree and thus can be used to reason over branchings.

We first focus on LTL, for which a formula is interpreted over a system execution trace.

Definition 17 (Syntax of LTL)**.** *Given a set AP of atomic propositions, LTL formulas are defined by the following grammar:*

$$\varphi ::= a \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \mathsf{X}\,\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \ , \ a \in AP$$

This logic is an extension of propositional logic; we first remark that by using De Morgan rules, it is possible to redefine the $\wedge$ operator. $\top$ (true) can be defined as $a \vee \neg a$ and $\bot$ (false) as $a \wedge \neg a$. The X and U modalities have the following interpretations:

- $\mathsf{X}\,\varphi$ means that at the next moment, φ holds;
- $\varphi_1 \,\mathsf{U}\, \varphi_2$ means that φ_2 will be true at some point in the future, and φ_1 holds until then.

More formally:

Definition 18 (*Semantics of LTL*)

The satisfaction semantics of an LTL formula φ over an infinite word $w = w_0 \dots w_n \dots \in (2^{AP})^\omega$ and a position i of this word, denoted $w, i \models \varphi$, is inductively defined by:

$$\begin{array}{rll}
w,i \models a \in AP & \text{if} & a \in w_i \\
w,i \models \varphi_1 \vee \varphi_2 & \text{if} & (w,i \models \varphi_1) \textit{ or } (w,i \models \varphi_2) \\
w,i \models \neg\varphi & \text{if} & w,i \not\models \varphi \\
w,i \models \mathsf{X}\,\varphi & \text{if} & w,i+1 \models \varphi \\
w,i \models \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{if} & \exists j \geq i : (w,j \models \varphi_2) \wedge (\forall i \leq k < j : w,k \models \varphi_1)
\end{array}$$

The classical operators $\Diamond$ and $\Box$ are defined by:

- $\Diamond \varphi \equiv \top \mathsf{U} \varphi$: the formula φ will be satisfied eventually;
- $\Box \varphi \equiv \neg \Diamond \neg\varphi$: at every moment the formula is satisfied (there is no time in the future where φ is not satisfied)

Moreover, for an LTL formula φ and a word w, we say that w satisfies φ if $w, 0 \models \varphi$, written $w \models \varphi$. We denote by L_φ the set of words w satisfying φ.

Definition 19. *Let $\mathcal{K}$ be a Kripke structure, and φ an LTL formula. We say that $\mathcal{K} \models \varphi$ if, and only if, for any execution σ of $\mathcal{K}$, the trace w of σ satisfies φ.*

Remark. *Said otherwise, $\mathcal{K} \models \varphi$ if and only if $\mathcal{L}(\mathcal{K}) \subseteq L_\varphi$.*

Similarly to regular languages (and finite automata), it is possible to characterise the language L_φ with a class of automata called Büchi automata.

Definition 20 (*Büchi Automaton*)

A Büchi automaton *is a 6-uple $\mathcal{B} = (S, T, I, F, \Sigma, L)$ with:*

- *S a set of states, and $T \subseteq S \times S$ a set of transitions;*
- *$I \subseteq S$ the set of initial states, $F \subseteq S$ the set of recurring states;*
- *Σ an alphabet and $L : S \to \Sigma$ a labelling function.*

Definition 21. *An execution σ of a Büchi automaton is an* infinite *sequence of states $s_0, \ldots, s_n, \ldots$ such that, for each $i \in \mathbb{N} : (s_i, s_{i+1}) \in T$. The trace of σ is the word $w = L(s_0)...L(s_n)....$ An execution is accepting if the set $\{i \in \mathbb{N} \mid s_i \in F\}$ is infinite (it goes infinitely often in a recurring state). A word w is recognized by $\mathcal{B}$ if there exists an accepting execution σ with trace w. We denote by $\mathcal{L}(\mathcal{B})$ the set of words recognized by $\mathcal{B}$.*

A usual question on languages and automaton is the emptiness problem.

Proposition 1. *Let $\mathcal{B}$ be a Büchi automaton. It is possible to check in linear time if the language recognized by $\mathcal{B}$ is empty. Furthermore, the emptiness problem is NLOGSPACE-complete.*

There exist several algorithms answering this question; a comparison between some of them has been published by Andreas Gaiser and Stefan Schwoon [41].

Let AP be a set of atomic propositions. We extend Definition 20 with a labelling function using the alphabet $\mathcal{P}(\mathcal{P}(AP))$, but where words are still considered using $\mathcal{P}(AP)$.

Definition 22 (Extension of definition 21)**.** *Let $\Sigma = \mathcal{P}(AP)$ and let $\mathcal{B}$ be a Büchi automaton over alphabet $\mathcal{P}(\Sigma)$. Let $w \in \Sigma^\omega$ and let σ be an accepting execution of $\mathcal{B}$. Then w is recognized by σ if and only if: $\forall i \in \mathbb{N} : w_i \in L(s_i)$.*

Using this extension, we can now state the following theorem, with L_φ the set of words w satisfying φ recognized by a Büchi automaton:

Theorem 2

Let φ be an LTL formula, over a set of atomic propositions AP. Then, we can build in exponential time a Büchi automaton $\mathcal{B}_\varphi$ over the alphabet $2^{2^{AP}}$ which exactly recognises the set of words satisfying φ: $\mathcal{L}(\mathcal{B}_\varphi) = \mathcal{L}_\varphi$.

Several algorithms perform the construction from φ to $\mathcal{B}_\varphi$, notably Paul Gastin and Denis Oddoux's one [42].

Remark. *In order to shorten state labelling notations and allow for the use of the transformation of [42], we use boolean formulas over AP: we recursively define a valuation function* $\mathrm{Val} : Bool(AP) \to 2^\Sigma = 2^{2^{AP}}$ *as follows:*

$$\begin{gathered}
\mathrm{Val}(\top) = 2^{AP} \\
\mathrm{Val}(p) = \Sigma_p = \{A \in 2^{AP} \mid p \in A\} \\
\mathrm{Val}(\varphi \wedge \phi) = \mathrm{Val}(\varphi) \cap \mathrm{Val}(\phi) \\
\mathrm{Val}(\varphi \vee \phi) = \mathrm{Val}(\varphi) \cup \mathrm{Val}(\phi) \\
\mathrm{Val}(\neg\varphi) = \Sigma \setminus \mathrm{Val}(\varphi)
\end{gathered}$$

For instance, with $AP = \{p, q, r\}$, we have $\mathrm{Val}(p \wedge \neg r) = \{\{p\}, \{p, q\}\}$. *We use this notation to define sets of $\mathcal{P}(AP)$.*

Example 11. Let p and q be atomic propositions. The Büchi automaton shown in Figure 1.12(a) recognizes $\Box \Diamond p$ (at each time, there exists a time in the future where p is satisfied), and the Büchi automaton shown in Figure 1.12(b) recognizes $p \,\mathsf{U}\, q$.

1.2.2 Computational Tree Logic

We now focus on CTL which considers the system execution tree, and is used to reason over branchings.

Definition 23 (Syntax of CTL). *Given a set AP of atomic propositions, CTL formulas are defined by the following grammar:*

$$\varphi ::= a \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \mathsf{EX}\,\varphi \mid \mathsf{AX}\,\varphi \mid \mathsf{E}\,\varphi_1 \,\mathsf{U}\, \varphi_2 \mid \mathsf{A}\,\varphi_1 \,\mathsf{U}\, \varphi_2 \ , \ a \in AP$$

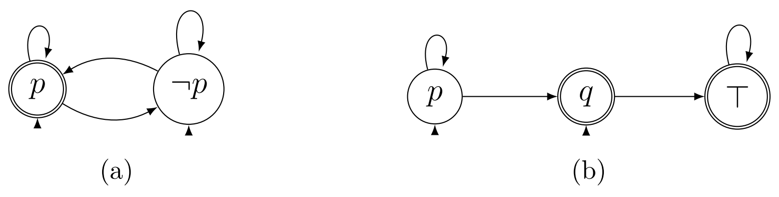


Figure 1.12: Two examples of Büchi automata

In this case, the temporal operators X and U are in the scope of a path quantifier, E being interpreted as *for some path* while A means *for all paths*.

A CTL formula is interpreted over a state of the execution tree of a Kripke structure $\mathcal{K} = (S, I, R, \Sigma, L)$. We write $\mathsf{Exec}(s)$ for the execution tree starting from state $s \in S$, and $\sigma = s_0 s_1 \ldots$ with $s_0 = s$ for a path in this tree, with $\sigma(i) = s_i$ for the state at position i.

Example 12.

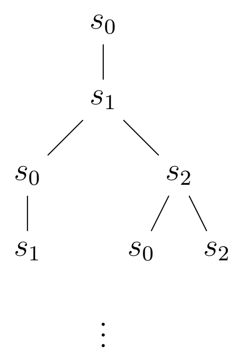


Figure 1.13: The beginning of an execution tree

Figure 1.13 shows the beginning part of the execution tree for the Kripke structure of Figure 1.2. The nodes are labelled by the corresponding states of the Kripke structure.

Definition 24 (*Semantics of CTL*)

The satisfaction of a CTL formula over a state is inductively defined by:

$$\begin{array}{rll}
s \models a \in AP & \text{if} & a \in L(s) \\
s \models \varphi_1 \vee \varphi_2 & \text{if} & (s \models \varphi_1) \text{ or } (s \models \varphi_2) \\
s \models \neg\varphi & \text{if} & s \not\models \varphi \\
s \models \mathsf{E}\,\mathsf{X}\,\varphi & \text{if} & \exists \sigma \in \mathsf{Exec}(s) : \sigma(1) \models \varphi \\
s \models \mathsf{A}\,\mathsf{X}\,\varphi & \text{if} & \forall \sigma \in \mathsf{Exec}(s) : \sigma(1) \models \varphi \\
s \models \mathsf{E}\,\varphi_1\,\mathsf{U}\,\varphi_2 & \text{if} & \exists \sigma \in \mathsf{Exec}(s), \exists j \geq 0 : \\
& & (\sigma(j) \models \varphi_2) \wedge (\forall 0 \leq k < j : \sigma(k) \models \varphi_1) \\
s \models \mathsf{A}\,\varphi_1\,\mathsf{U}\,\varphi_2 & \text{if} & \forall \sigma \in \mathsf{Exec}(s), \exists j \geq 0 : \\
& & (\sigma(j) \models \varphi_2) \wedge (\forall 0 \leq k < j : \sigma(k) \models \varphi_1)
\end{array}$$

Like for LTL, the standard operators $\Diamond$ and $\Box$ are defined by:

- $\mathsf{E}\,\Diamond\,\varphi \equiv \mathsf{E}\,\top\,\mathsf{U}\,\varphi$. A state s statisfies this formula if there exists a path starting from s where φ will be satisfied eventually ;
- $\mathsf{A}\,\Box\,\varphi \equiv \neg\,\mathsf{E}\,\Diamond\,\neg\varphi$. A state s statisfies this formula if φ always holds on all states

of all paths starting from s ;
- $\mathsf{A}\Diamond\varphi \equiv \mathsf{A}\top\mathsf{U}\varphi$. A state s statisfies this formula if φ holds eventually on all paths starting from s.

The Kripke structure K satisfies a CTL formula φ, written $K \models \varphi$ if its initial state satisfies φ, i.e. $s_0 \models \varphi$.

Example 13.

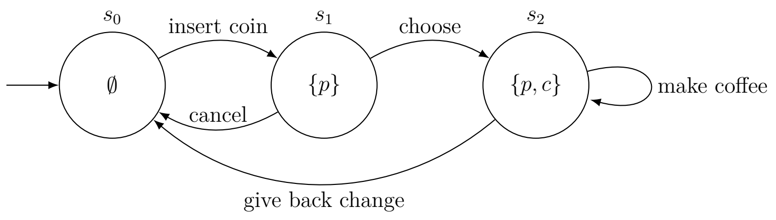


Figure 1.2: An example of Kripke structure : a coffee machine

We continue the example on the Kripke structure $\mathcal{K}$ shown in Figure 1.2. We have:
- $\mathcal{K} \not\models \mathsf{A}\Diamond c$: there exists a loop between states s_0 and s_1, which avoids the only state where c is satisfied, s_2;
- $\mathcal{K} \models \mathsf{AXEX}\, c$: the single successor of s_0 is s_1 which has for possible successor s_2 satisfying c;
- $\mathcal{K} \not\models \mathsf{EXAX}\, c$: state s_1 has also for possible successor s_0 not satisfying c.

1.2.3 Hybrid Automata Stochastic Logic

Logics LTL and CTL are appropriate for discrete event systems but cannot express usual quantitative timed requirements like a bounded response time for requests. In addition, they cannot express any satisfaction probability which is mandatory for risk evaluation. So we now present stochastic extensions to LTL and CTL that have mainly be introduced for numerical model checking (following the analyses of [12]) and we end this section by the HASL logic suited for statistical model checking of probabilistic hybrid systems and integrated in the tool Cosmos.

Continuous Stochastic Logic. In [7], Continuous Stochastic Logic (CSL) has been introduced and the decidability of the verification problem over a finite continuous-time Markov chain (CTMC) has been established. CSL extends the branching time reasoning of CTL to CTMC models by replacing the discrete CTL path-quantifiers All and Exists with a continuous path-quantifier that expresses that the probability of CTMC paths satisfying a given condition fulfils a given bound.

Several variants have been introduced and studied such as CSRL [10], that extends CSL to take into account Markov reward models, i.e. CTMCs with a single reward on states or possibly on actions [53], that is used to specify an interval (on the accumulated reward) on the `Until` or `Next` operator. asCSL, introduced in [9] replaces the interval time constrained `Until` of CSL by a regular expression with a time interval constraint. These path formulas can express elaborated functional requirements as in CTL* but the timing requirements are still limited to a single interval globally constraining the path execution. In the logic CSLTA [35], path formulas are defined by a single-clock deterministic timed automaton. This logic has been shown strictly more expressive than CSL and also more expressive than asCSL when restricted to path formulas.

DTA. In [27], deterministic timed automata with multiple clocks are considered and the probability for random paths of a CTMC to satisfy a formula is shown to be the least solution of a system of integral equations. The cost of this more expressive model is both a jump in the complexity as it requires to solve a system of partial differential equations, and a loss in guaranty on the error bound.

M(I)TL. Several logics based on linear temporal logic (LTL) have been introduced to consider timed properties, including Metric (Interval) Temporal logic in which the `Until` operator is equipped with a time interval. Chen *et al.* [28] have designed procedures to approximately compute desired probabilities for time bounded verification, but with complexity issues. The question of stochastic model checking on (a sublogic of) M(I)TL properties, has also been tackled see e.g. [76].

Observe that all of the above mentioned logics have been designed so that numerical methods can be employed to decide about the probability measure of a formula. This very constraint is at the basis of their limited expressive scope which has two aspects: first the targeted stochastic models are necessarily CTMCs; second the expressiveness of formulas is constrained by decidability/complexity issues. Furthermore the evolution of stochastic logics based on CTL seems to have followed two directions: one targeting *temporal reasoning* capability (in that respect the evolutionary pattern is: CSL $\to$ asCSL $\to$ CSLTA $\to$ DTA), the other targeting *performance evaluation* capability (evolutionary path: CSL $\to$ CSRL $\to$ CSRL+impulse rewards). A unifying approach is currently not available, thus, for example, one can calculate the probability for a CTMC to satisfy a sophisticated temporal condition expressed with a DTA, but cannot, assess performance evaluation queries at the same time (i.e. with the same formalism).

Hybrid Automata Stochastic Logic. The logic HASL (Hybrid Automata Stochastic Logic) describes properties via two elements:

- a Linear Hybrid Automaton (LHA) which contains numerical variables and is synchronised with the observations of the analyzed stochastic process;
- and an expression specifying the quantity to be evaluated.

Example 14.

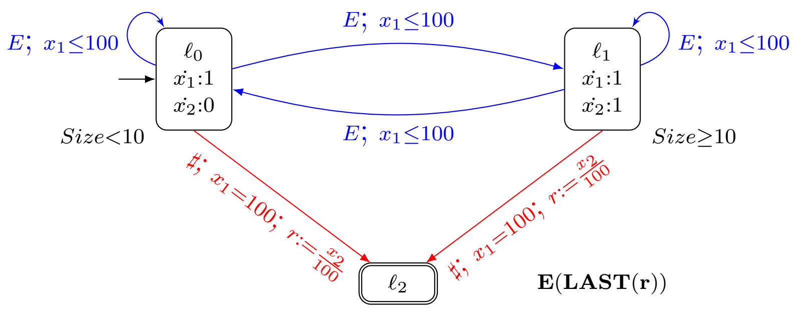


Figure 1.14: An example of Hybrid Automata Stochastic Logic (HASL) formula

A HASL formula is described on Figure 1.14 over the tandem queue system, represented by the DESP shown in Figure 1.6, equipped with the indicator $Size : (n, m) \to n + m$. This formula contains an automaton with three states (ℓ_0, ℓ_1, ℓ_2) and an expression $E(LAST(r))$ which describes the proportion of time during which too many people are in the queues (over 10 simultaneous clients), using a overall duration of 100 time units.
This automaton handles three variables:

- x_1 representing the elapsed time;
- x_2 representing the time during which the queues were too full;
- and r representing the time proportion over which the queues were too full.

The initial state ℓ_0 corresponds to the system running without overfilling. In this state, only x_1 evolves over time. In state ℓ_1, the variable x_2 also evolves, measuring the time during which the queues are overfilled. The measure ends in state ℓ_2, with the maximum simulation time.
There are two types of transitions: those who are fired by a process event, shown in blue, and the autonomous transitions fired by a condition over variables (in this case, the execution time) represented in red.
In the expression $E(LAST(r))$, the operator $LAST$ is used to evaluate r at the end of an execution, and E the expectation (over successful runs).

We now describe more formally the syntax and semantics of HASL. In the LHA, the transitions are equipped by *constraints* that form guards, and variable updates.

Definition 25. *A* constraint *is a boolean combination of comparisons of the form* $\sum_{1\leq i\leq n} \alpha_i x_i + c \bowtie 0$ *where* x_i *are variables,* α_i, c *are indicators and* $\bowtie \in \{=, <, >$

$, \leq, \geq\}$. *We denote by* Const *the set of constraints.*

Definition 26. *An* update *is a tuple of functions $u_1, \ldots, u_n$ such that every u_k is of the form $u_k = \sum_{1\leq i\leq n} \alpha_i x_i + c$ where α_i and c are indicators. We denote by* Up *the set of updates.*

In order to synchronize with the model of the system, we define *Linear Hybrid Automata*:

Definition 27

A Linear Hybrid Automaton (LHA) is a tuple $\mathcal{A} = (E, L, \Lambda, \mathrm{Init}, \mathrm{Final}, X, \mathrm{flow}, \to)$ where:

- *E is a finite alphabet of events;*
- *L is a finite set of locations;*
- *$\Lambda : L \to \mathrm{Prop}$ is a location labelling function;*
- *Init is a subset of L containing the initial locations;*
- *Final is a subset of L containing the final locations;*
- *$X = (x_1, \ldots, x_n)$ is a n-uple of data variables;*
- *$\mathrm{flow} : L \to \mathsf{Ind}^n$ is a function which associates with each location one indicator per data variable representing the evolution rate of the variable in this location. The projection of flow on its i^{th} component is denoted by flow_i;*
- *the transition relation $\to$ is a subset of $L \times \Big((2^E \times \mathrm{Const}) \uplus (\{\#\} \times \mathrm{Const})\Big) \times \mathrm{Up} \times L$, where $\uplus$ denotes the disjoint union. A transition $(\ell, E', \gamma, U, \ell') \in \to$ is written $\ell \xrightarrow{E',\gamma,U} \ell'$.*

  Furthermore, the following conditions are required:
- *<u>Initial determinism</u> : $\forall \ell_1 \neq \ell_2 \in Init, \Lambda(\ell_1) \wedge \Lambda(\ell_2) \Leftrightarrow \perp$;*
- *<u>Determinism on events</u> : $\forall E_1, E_2 \subseteq E$ st $E_1 \cap E_2 \neq \emptyset : \forall \ell, \ell_1, \ell_2$, si $\ell \xrightarrow{E_1,\gamma_1,U_1} \ell_1$ et $\ell \xrightarrow{E_2,\gamma_2,U_2)} \ell_2$ are two distinct transitions, then either $\Lambda(\ell_1) \wedge \Lambda(\ell_2) \Leftrightarrow \perp$ or $\gamma_1 \wedge \gamma_2 \Leftrightarrow \perp$;*
- *<u>Determinism on #</u> : $\forall \ell, \ell_1, \ell_2 \in L$ si $\ell \xrightarrow{\#,\gamma_1,U_1} \ell_1$ et $\ell \xrightarrow{\#,\gamma_2,U_2} \ell_2$ are two distinct transitions, then either $\Lambda(\ell_1) \wedge \Lambda(\ell_2) \Leftrightarrow \perp$ or $\gamma_1 \wedge \gamma_2 \Leftrightarrow \perp$;*
- *<u>No #-labelled loops</u>: for any sequence $\ell_0 \xrightarrow{E_0,\gamma_0,U_0} \ell_1 \xrightarrow{E_1,\gamma_1,U_1} \ldots \xrightarrow{E_{n-1},\gamma_{n-1},U_{n-1}} \ell_n$ such that $\ell_0 = \ell_n$, ithere exists $i \leq n$ such that $E_i \neq \#$.*

There are two kinds of transitions in this definition:

- transitions with label #, which are spontaneously fired with respect to the automaton's variable values;
- transitions with a label containing a subset of events, which are fired upon synchronisation with the model to evaluated: the occurence of a transition of the model "sends" the corresponding event to the automaton.

A *configuration* of the hybrid automaton consists of a state and a valuation of its variables.

The synchronised product of the stochastic process of the model and the hybrid automaton is also a stochastic process. The state of the synchronised product is a pair, composed of a configuration of the net and a configuration of the hybrid automaton.

A change of state in this synchronised product is produced either by the firing of a transition of the model (synchronised with an edge of the automaton), or by the firing of an autonomous transition from the current automaton state. As previously, the event to be considered is the earliest one. If a transition firing occurs in the model and cannot be synchronised with the automaton, the process ends in a failure state. The only other way to end this new process is to reach a final state of the automaton.

Example 15. In the example of figure 1.14, the set of events E is $\{in, move, out\}$. The two autonomous transitions are used to stop the synchronisation after 100 time units by reaching the final state ℓ_2, whatever the previous state. Conditions on state ℓ_0 and ℓ_1 distinguish the state in which tandem queues are in: either they are overfull ($Size \geq 10$) or there is a normal load.
Since $\dot{x_1} = 1$ during the whole run of the automaton, variable x_1 measures the elapsed time (and therefore stops the execution when needed). The variable x_2 is only modified in state ℓ_2, to measure the duration of overfull load in the tandem queues.

We now focus on the evaluation of a performance index over a trajectory of the synchronised product between the LHA and the model. This index is obtained via expressions over state variables:

Definition 28

HASL expressions are defined by the following grammar:

$$\begin{array}{rcl} Z & ::= & c \mid E(Y) \mid Z+Z \mid Z-Z \mid Z\times Z \mid Z/Z \\ Y & ::= & c \mid Y+Y \mid Y\times Y \mid Y/Y \mid \mathrm{LAST}(y) \mid \mathrm{MIN}(y) \mid \mathrm{MAX}(y) \mid \mathrm{INT}(y) \mid \mathrm{AVG}(y) \\ y & ::= & c \mid x \mid y+y \mid y\times y \mid y/y \end{array}$$

where x is a state variable (in the set X) of the automaton, and c a constant.

In this definition (from bottom to top):

- the y expressions are arithmetic expressions over state variables and constants;
- path operators are then applied on the previous expressions: minimum, maximum, integral with respect to time, last value. New arithmetic expressions are then built from these values. The result is a random variable Y associated with the random trajectory of the synchronised product.

- The expectation of these random variables form performance indices that we can then combine with arithmetic operations to define new indices.

These expectations are actually conditional ones since the trajectories have to be accepted. A variant for these expressions is to consider directly as a performance index, the probability that the trajectory is accepted.

Example 16. To continue again the example of figure 1.14, by construction trajectories always end in the final state ℓ_2 (after 100 time units), where we consider the last value of $r = \frac{x_2}{100}$, which is set by the transitions leading to ℓ_2. The HASL expression associated with this LHA is used to measure the fraction of time in which the queue was overloaded, for simulations of 100 time units.

Compared to the logics introduce above, HASL is the most expressive one thanks to (1) the selection of trajectories, (2) the existence of multiple indicators, and (3) the arithmetic operators.

1.3 Verification Methods

1.3.1 Verifying Kripke Structures

To check whether a Kripke structure $\mathcal{K}$ satisfies an LTL formula φ, a classical method consists in trying to exhibit a counter-example: more precisely, a particular execution σ, whose trace w satisfies $\neg\varphi$. We have seen in section 1.2.1 that a Büchi automaton can be used to generate the whole set of words (or executions) satisfying a LTL formula. Using this result, one only checks the emptiness of the synchronous product of $\mathcal{B}_{\neg\varphi}$ with the Kripke structure $\mathcal{K}$ to prove that there is no such counter example.

The synchronised product of a Kripke structure $\mathcal{K}$ and the Büchi automaton $\mathcal{B}_{\neg\varphi}$ is also a Kripke structure:

Definition 29 (*Kripke structure and Büchi automaton product*)

Let AP be a set of atomic propositions. Let $\mathcal{K} = (S_\mathcal{K}, I_\mathcal{K}, R_\mathcal{K}, L_\mathcal{K})$ be a Kripke structure (which states are labelled by elements of $\mathcal{P}(AP)$), and $\mathcal{B} = (S_\mathcal{B}, T_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B}, \mathcal{P}(\mathcal{P}(AP)), L_\mathcal{B})$ a Büchi automaton over the alphabet $\mathcal{P}(\mathcal{P}(AP))$. We define the product Büchi automaton $\mathcal{P}(S, T, I, F, \Sigma, L)$ as follows:

- *its set of states is*

$$S = \{(s, s') \in S_K \times S_\mathcal{B} \mid L_\mathcal{K}(s) \in L_\mathcal{B}(s')\}$$

  The states $(s, s') \in S$ are labelled by $L(s, s') = L_\mathcal{K}(s)$.
- *its initial states are $I = (I_\mathcal{K} \times I_\mathcal{B}) \cap S$, and the recurring states are $F = (S_\mathcal{K} \times F_\mathcal{B}) \cap S$;*

- *its alphabet Σ is $\mathcal{P}(AP)$;*
- *its set of transitions is defined using:*

$$((s_1, s'_1), (s_2, s'_2)) \in T \iff (s_1, s_2) \in R_{\mathcal{K}} \wedge (s'_1, s'_2) \in T_{\mathcal{B}}$$

We denote this product by $\mathcal{P} = K \otimes \mathcal{B}$.

Proposition 3. *A word is recognized by the Büchi automaton $\mathcal{K} \otimes \mathcal{B}$ if and only if it is recognized by both the Kripke structure $\mathcal{K}$ and the Büchi automaton $\mathcal{B}$,* id est*: $\mathcal{L}(\mathcal{K} \otimes \mathcal{B}) = \mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{B})$.*

Proof: Let $\Sigma = \mathcal{P}(AP)$ and w be a word recognized by the product Büchi automaton. There exists an accepting execution $\sigma = (s_0, s'_0) \dots (s_n, s'_n) \dots$ which recognizes w. We have, for every $i \in \mathbb{N} : ((s_i, s'_i), (s_{i+1}, s'_{i+1})) \in T$ and $w = L(s_i, s'_i) = L_{\mathcal{K}}(s_i)$. Furthermore, using the definition of S, we have $w \in L_{\mathcal{B}}(s_i)$.

- Using the definition of T, we have for every $i \in \mathbb{N} : (s_i, s_{i+1}) \in R_{\mathcal{K}}$ and by construction of S, $s_0, \dots, s_n, \dots$ is an infinite sequence of states of the Kripke structure $\mathcal{K}$, which means $\mathcal{K}$ recognizes w.
- Similary, for every $i \in \mathbb{N}$: $(s'_i, s'_{i+1}) \in T_{\mathcal{B}}$ and $s'_0, \dots s'_n \dots$ is an infinite sequence of states of the Büchi automaton. As σ is an accepting execution in the product automaton, the set $\{i \in \mathbb{N} \mid (s_i, s'_i) \in F\}$ is infinite. Moreover, as $F = (S_1 \times F_{\mathcal{B}})$ we have for all $t \in S_1 : (t, s'_i) \in F \Leftrightarrow s'_i \in F_{\mathcal{B}}$. We can deduce that $\{i \in N \mid s'_i \in F_{\mathcal{B}}$ is also infinite. Finally, using the definition of S, $w = L_{\mathcal{K}}(s_i) \in L_{\mathcal{B}}(s'_i)$ which means that w is recognized by the Büchi automaton $\mathcal{B}$

We can show similarly that if the word w is recognized by both automata, then it is recognized by the product automaton. ■

Proposition 4

Let $\mathcal{K}$ be a Kripke structure, and φ a LTL *formula. Then the problem: «$K \models \varphi$?» is decidable, with temporal complexity $|K| \times 2^{|\varphi|}$ and belongs to PSPACE.*

Proof: This proposition has been proved by Moshe Y. Vardi and Pierre Wolper in 1986 [72]. ■

Example 17. We want to check if $\varphi = \Box(p \Rightarrow \mathsf{X} \Diamond q)$ holds on the Kripke structure shown in Figure 1.3. We first compute $\neg\varphi \equiv \neg \Box(p \Rightarrow \mathsf{X} \Diamond q) \equiv \Diamond(\neg(p \Rightarrow \mathsf{X} \Diamond q)) \equiv \Diamond(p \wedge \mathsf{X}(\neg \Diamond q)) \equiv \Diamond(p \wedge \mathsf{X}(\Box(\neg q)))$. Then we build a Büchi automaton that recognizes $\neg\varphi$ and afterwards the product automaton. This operation is described in Figure 1.15.

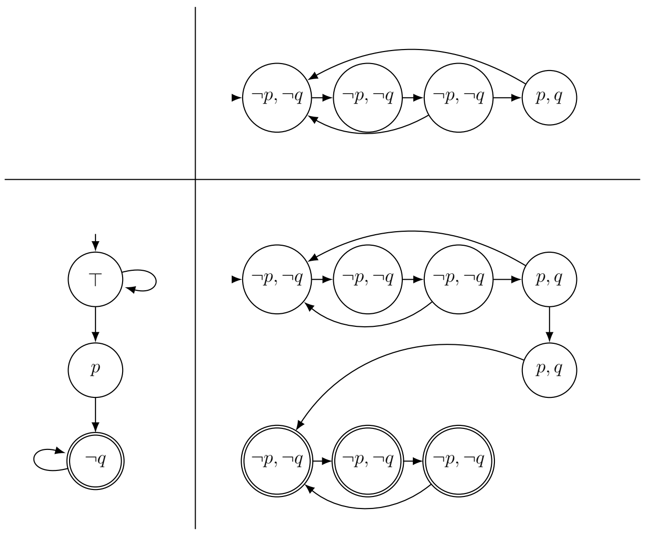


Figure 1.15: Product of the Kripke structure and the Büchi automaton

It is enough to read the product automaton: as it contains an accepting loop of recurring states reachable from the initial state, the formula φ is not satisfied over the Kripke structure. Even better, we immediately obtain on the Kripke structure a counter-example: reaching the p, q state and then looping over the three first states of the Kripke structure.

1.3.2 Numerical model-checking

Numerical model-checking of stochastic models has been intensively studied. Here we only give some relevant examples of these techniques and then discuss the limitations of numerical model checking. Let us start with time-bounded reachability for DTMC which requires to reach a subset of states S_+ in at most u steps. Such a property can inductively defined w.r.t. u as follows.

Definition 30 (*Time-Bounded Reachability for DTMC*)

Given a DTMC $\mathcal{C} = (S, S_0, \mathbf{P})$, a set of states $S_+ \subset S$ and a positive integer u, we define the vector of probabilities $\mu_u \in [0,1]^S$ as:

$$\mu_u(s) = \begin{cases} 1 & \text{if } s \in S_+ \\ 0 & \text{if } u = 0 \wedge s \notin S_+ \\ \sum_{s' \in S} \mathbf{P}(s,s')\mu_{u-1}(s') & \text{if } u > 0 \wedge s \notin S_+ \end{cases}$$

We now switch to CTMCs, and define μ_τ as the vector of probabilities to reach an absorbing state s_+ in τ time units by $\mu_\tau(s) = \pi_{\tau,s}(s_+)$. Using the uniformization method [50], one assumes w.l.o.g. that the exit rate is independent from the state and equal to some λ. Thus by conditionning on the number of steps during an interval $[0,\tau]$, the time-bounded reachability may be defined as follows.

Definition 31 (*Time-Bounded Reachability for CTMC*)

Given a CTMC $\mathcal{C}$, a state $s_+ \in S$ and a positive real τ, we define the vector of probabilities $\mu_\tau \in [0,1]^S$ by:

$$\mu_\tau(s) = e^{-\lambda\tau} \sum_{n\in\mathbb{N}} \frac{(\lambda\tau)^n}{n!} \mu_n(s)$$

In order to generalize it to a CTL-type formulas, one considers the formula as a tree, and evaluates those starting from the leaves. Given a sub-formula consisting of only one probabilistic operator at the root, and no nested probabilistic operators, the evaluation of U, X and $\Box$ formulas can be reduced to time-bounded or time unbounded reachability problems, as sub-formula are states formulas. Once sub-formulas have been evaluated on each state, they can be replaced by a proposition in the upper sub-formulas, until the root is reached. For CTL-type formulas, one proceeds similarly to section 1.3.1: the formula is translated into an automaton, then the synchronized product of the stochastic system with the automaton is built. This product is still a stochastic system, and the model-checking problem is thus reduced to time-unbounded reachability. However, if the formula is large, the size of the automaton and thus the size of the product may become problematic. The reader may refer to chapter 10 of *Principles of Model Checking* [11] for more details about the model-checking of probabilistic systems.

Most numerical model-checking algorithms rely on matrix vector multiplication, usually taking advantage of the sparsity of transition probability matrices. However, these computations are difficult to parallelize. The tool Marcie [46] features a parallel implementation of this method, but still a non negligible part of the computation cannot be performed in parallel.

In order to perform numerical model-checking, strong probabilistic hypotheses on the stochastic system to be analyzed are mandatory. The easiest case is to suppose that the system is *Markovian* and has a finite state space. Note that the system of linear equations produced during the computation can still be huge, and that the computation of solutions may be difficult and therefore require iterative methods. This implies that only approximate solutions of such systems can be obtained.

In the case of using a time automaton for the specification of the formula, there exist efficient techniques only if there is at most one clock in the automaton, that are described in [36] and [17]. More generally, numerical approaches can be adapted to deal with non determinism [71] and to tackle model-checking problems on Markov Decision Processes by alternating probabilistic transitions with non-deterministic ones.

This approach has been efficiently implemented in several tools with different types of Markovian systems, such as:

- Prism [52] deals with Discrete and Continuous Time Markov Chains, probabilistic automatas and Markov Decision Processes;
- Uppaal [20] deals with various types of automata;
- GreatSPN [8] and Marcie [46] deal with Stochastic Petri nets.

1.3.3 Statistical model-checking

The alternative to numerical approach when dealing with large systems is to use statistical methods. The statistical model checking relies on a *Monte Carlo* algorithm to estimate the probability of interest. More precisely, given a linear temporal logic formula, one defines a random variable X which takes 1 as value when the trajectory satisfies the formula and 0 otherwise. Thus, this variable follows a Bernouilli law, and one computes its *expected value* $E(X)$. The Monte Carlo algorithm simulates a large number N of trajectories, and counts the number of those trajectories satisfying the specification. An estimation of the probability is obtained as the ratio of the number of successful trajectories on the total number of trajectories. The random variable Z is defined as the mean of N independent copies of X: $Z = \frac{1}{N}\sum_{i=1}^{N} X_i$.

Example 18.

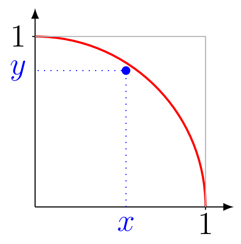


Figure 1.16: Does the realisation (x, y) belong in the quarter-circle?

Let X and Y be two random variables of distribution $\mathsf{UNIF}(0,1)$. Let Z be a random variable such that an observation z is 1 when the realisations x and y satisfy $x^2 + y^2 \leq 1$ and 0 otherwise. This definition means that z is equal to 1 if and only if (x, y) belong in the quarter-circle of radius 1 (see Figure 1.16), which has an area of $\frac{\pi}{4}$. Thus $E(Z) = \frac{\pi}{4}$.

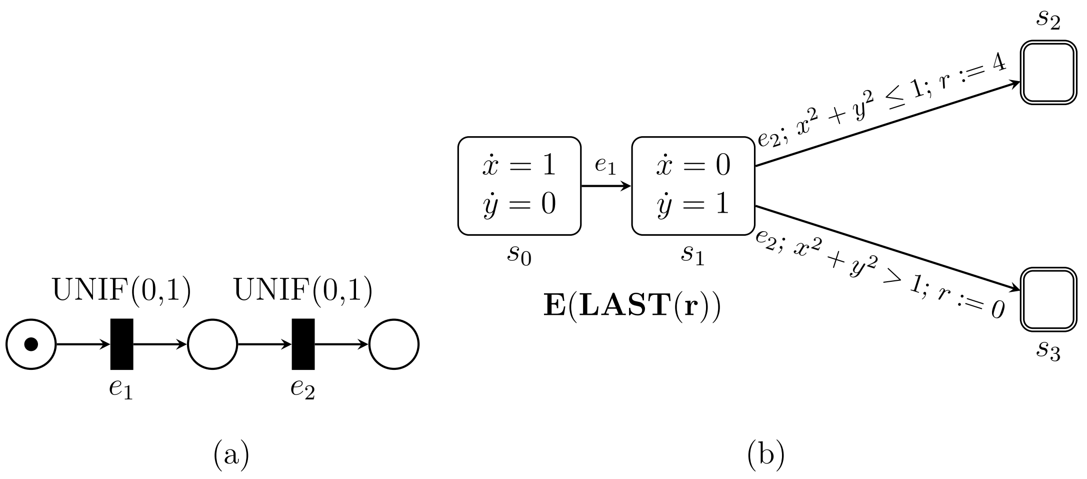


Figure 1.17: Computing π using a HASL formula and a stochastic net.

A trajectory is obtained by synchronizing the LHA shown in Figure 1.17(b) with the stochastic net shown in Figure 1.17(a), producing random values for x (the firing delay of e_1) and y (the firing delay of e_2) respectively. The last value r is 4 if $x^2 + y^2 \leq 1$, and 0 otherwise. In both case, the last synchronisation lead to a final state. The expression $E(LAST(r))$ corresponds to the expected value of r at the end of a trajectory, which is equal to π, four times the value of $E(Z)$.

It is easy to parallelize this method: it suffices to run several "simulators" of the system on different processes (either on different processors or machines), and take the mean result of trajectories of all simulators. The main difficulty comes from choosing the right random number generator so that all generated trajectories are independent one from each other. The only sequential operation being the computation of the mean value and the confidence interval. This implies there is almost no additional computing cost in using parallel computations. These statistical methods can be naturally extended to performance evaluation: instead of computing a probability, we compute the expected values of random variables whose values depend on trajectories of the systems.

Statistical procedures We now detail the different proccedures that are available in Cosmos for the evaluation of HASL expressions:

- **Sequential hypothesis testing [73].** This procedure checks whether a probability is above a threshold. Parameters of this procedure are the probability of an error for a positive answer and a negative answer and the width of the indifference region. When the value of the probability is outside the indifference interval, the probability of an error is bounded by the parameter corresponding to the answer.

- **Chernoff-Hoeffding bounds [48].** This static method requires three related parameters, each of them can be determined by the two others. These parameters are the interval width, the confidence level and the number of samples. It outputs a confidence interval whose width satisfies the requirement and where the probabilistic guarantee is exact. It applies to estimate the expectation of a bounded random variable.
- **Chow-Robbins bounds [32].** This sequential method requires two parameters: the interval width and the confidence level. It outputs an interval whose width satisfies the requirement and where the probabilistic guarantee is asymptotic w.r.t. the width of the interval. It applies to estimate the expectation of a random variable, when no known bound is available.
- **Gaussian approximation.** This static method requires two parameters. The number of samples has to be given. The second parameter is either the confidence level or the interval width, one of these determining the other one. It ouputs an interval whose width satisfies the requirement and where the probabilistic guarantee is asymptotic w.r.t. the number of samples. It applies to estimate the expectation of a random variable. It is based on the central limit theorem.
- **Clopper-Pearson bound [33]** This static method computes confidence intervals for binomial distributions. It takes as input three parameters, the total number of samples, the confidence level and the number of successful samples and outputs a confidence interval for the probability of a sample to be successful.

Comparison of numerical and statistical methods. The statistical methods have several advantages compared to the numerical methods:

- the memory required to simulate one trajectory of the system is usually very small (and the possible values of the random variables are rarely large), making the memory requirement of a statistical model-checker very low. A contrario, since models *and* formulas may introduce a large size increase of either the probability matrix or the model/formula product, memory is a bottleneck of numerical methods;
- we have seen in section 1.3.2 that numerical methods are only usable on a very restricted set of models. A contrario, statistical methods only need an operational semantic of the stochastic model;
- as already mentioned, it is straightforward to parallelize the simulation process and it comes with no meaningful additional computation cost. A contrario, parallelization is much more difficult for numerical model-checking;
- complex linear time properties, expressed for instance using the HASL logic (see section 1.2.3), can be efficiently evaluated using statistical model-checkers while numerical model checkers require restricted temporal logics.

However, statistical methods have several drawbacks such as:

- the computation time may be a problem in the case of tight confidence intervals, that one needs for precise results. In general, dividing by two the width of the confidence interval requires performing four time more simulations;
- it is not suited for the evaluation of logics based on state formulas;
- as it is possible to simulate only finite trajectories, a statistical model-checking procedure cannot evaluate the unbounded until operator $\phi \, \mathsf{U} \, \psi$; a workaround is described in [45];
- a crude Monte carlo estimator is unsuitable to compute very small probabilities, such as the probability of a *rare event*; this specific problem is widely discussed in [14].

1.3.4 Tools for statistical model-checking

In this section, which is only a slight improvement of the *Related Tools* and *Tool Evaluation* sections of [12], we list and compare a certain number of tools that can be used for statistical model-checking. Figures of this section are taken directly from the journal article.

Related tools. Most of these tools have been briefly mentioned with respect to their numerical model-checking features.

COSMOS[1] [12] is a statistical model-checker for the Hybrid Automata Stochastic Logic over high-level stochastic Petri nets. It has been developed first during Hilal Djafri's PhD [34] at LSV, ENS Cachan. It was improved, adding HASL support in [13], and rare event handling in Benoît Barbot's PhD [14] still at LSV, ENS Cachan. As several improvements have been made during this PhD, the tool is presented more thoroughly in Chapter 4. It is currently maintained by Benoît Barbot.

PRISM[2] [52] is a tool for performing model checking on probabilistic models, that has been used for numerous applications [58]. It can perform numerical model-checking on discrete and continuous-time Markov chains, Markov decision processes and probabilistic timed automata. Its statistical part only deals with Markov chains. The PRISM language defines these probabilistic systems with a synchronised product between reactive modules and can, in this manner, describe large systems in a compact way. The property language subsumes several well-known probabilistic temporal logics, including CSL and PCTL.

UPPAAL[3] [20] is a verification tool that includes many formalisms, such as timed automata, or timed games. It supports automata-based and game-based

[1]COSMOS is available at `http://cosmos.lacl.fr/`.
[2]PRISM is available at `https://www.prismmodelchecker.org/`.
[3]UPPAAL is available at `http://www.uppaal.org/`

verification techniques, and has shown its ability to analyse large scale applications. A statistical-model checker engine has been added in 2011 in the form of an extension[4] (here called Uppaal-smc), enriching Uppaal with the ability to verify timed systems with stochastic semantics. The specification language, PLTL, is an adaptation of LTL with path operators substitued for quantifiers and with bounded Until (U) operator.

Plasma[5] [49] is a platform dedicated for statistical model-checking. It accepts the Prism language for its models, extended with more general distributions and a dedicated biological language. Its property specification language is a restricted version of PLTL, with a single threshold operator. Furthermore, it is built with a plugin system giving the ability for developers to extend it. It can also be integrated, via a library, to another software.

Ymer[6] [74] is a statistical model checker for CTMCs and generalised semi-Markov processes described using the PRISM language. Its property specification language is a fragment of CSL without the steady-state operator but including the unbounded Until (U) operator. It seems to be no more maintained.

Marcie[7] [46] is a tool for qualitative and quantitative analysis of generalised stochastic Petri nets, that relies on Interval Decision Diagrams to represent symbolically the state space of the Petri net. It also has a simulation engine for the model checking of PLTL formulas. Marcie has been mainly developed for the study of chemical reaction networks.

Tool Evaluation. The experiments were run with Cosmos, Prism (version 4.0.2), Uppaal-smc (version 4.1.13), Plasma (version 1.1.4), Ymer (version 3.1) and Marcie (version 1178M). Two models were considered:

- the first model is a Tandem Queuing Network (TQN) taken from [47]. The interarrival time of clients in the tandem queue follows an exponential distribution, the service of the first queue follows a Cox_2 distribution and the service of the second queue follows an exponential distribution. A state of this system is composed of the number of clients in each queue and the state of service of the first queue when non empty.
- the second model is a model of dining philosophers.

Parameters. For the experiments, the following parameters were used: the queue capacity $N = 5$, the arrival rate $\lambda = 20$, the service rates on the first phase in the

[4]This extension is available at `http://people.cs.aau.dk/~adavid/smc/`

[5]Plasma is available at `https://project.inria.fr/plasma-lab/`

[6]Ymer is available through a git repository at `https://github.com/hlsyounes/ymer`

[7]Marcie is available at `http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Marcie`

first queue $\mu_1 = 0.2$ and $\mu_1' = 1.8$ (the latter corresponding to clients without a second service phase) and the service rate of the second phase in the first queue is $\mu_2 = 2$. The service rate of the second queue is $\kappa = 4$.

The dining philosopher model is a mutual exclusion problem where N philosophers are sitting around a table. They can decide to eat by taking two forks, that are shared with their right and left neighbours. A contention problem may arise due to the sharing of resources. For the experiments, the rate of all exponential distribution is chosen equal to 10.

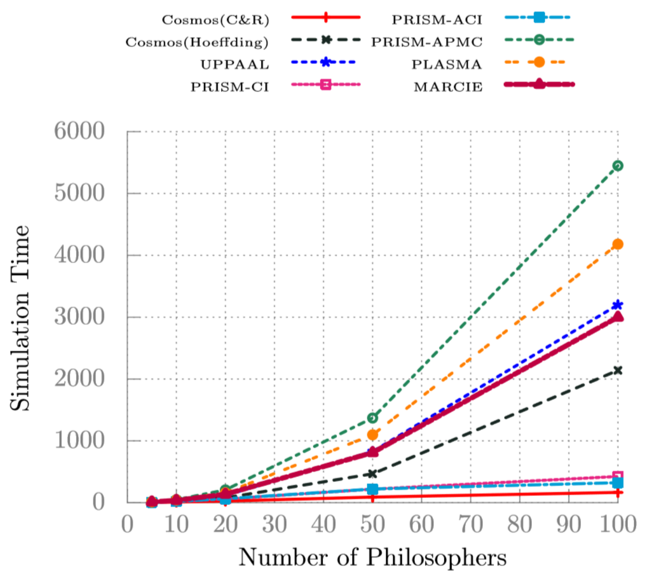
Performance indices. For the tandem queue system, we consider the following time-bounded reachability measure: the probability ϕ_{TQS} that the first queue of the tandem gets full within time T. For the dining philosophers, it is the probability ϕ_{DQM} of reaching a deadlock state before N philosophers eat. Such a deadlock occurs when all philosophers have taken one fork. Those properties can be straightforwardly encoded in CSL and in HASL: equivalent verification experimentations can be performed by all tools.

Experiment settings. The following statistical parameters have been set: the confidence level is 0.95, and the width of the confidence interval 0.005. For the hypothesis testing, the probability of error is 0.005 and the width of the indifference region is set to 0.001. The other parameters have been set to default values. The tools must generate a large number of trajectories to be able to fulfill these parameters. Most tools can take advantage of parallelisation; however the experiments have been executed on a single core of a 2.4 GHz Intel Core 2 Duo.

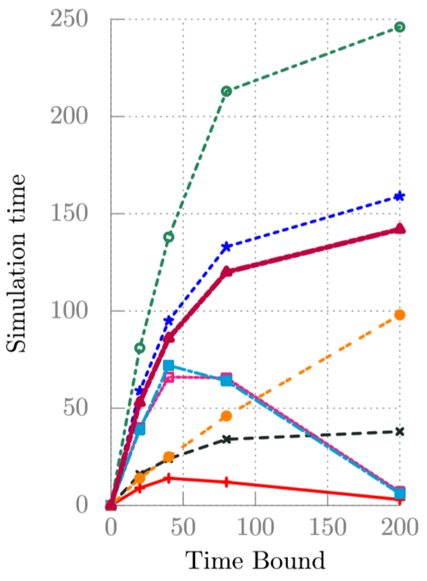
Comments. Figure 1.18(a) shows the runtime for the dining philosopher models as a function of the number of philosophers. At the time of [12], Cosmos is the fastest tool using Chernoff-Hoeffding bounds. For a hundred philosophers, Marcie is 1.4 times slower, Uppaal-smc is 1.5 times slower, Plasma is 1.9 times slower, and Prism using APMC[8] is 2.5 times slower. Among the tools using sequential procedures, the two versions of Prism have similar run times, Cosmos being up to 1.9 times faster.

Figure 1.18(b) shows the runtime comparison with different time bounds T for the tandem queue system experiment. Two kinds of behaviours for tools can be distinguished, depending on the applied statistic method. For the first one, that corresponds to the Chernoff-Hoerding method, the simulation time is increasing with the time bound T. About 295 000 trajectories are required to obtain the specified confidence interval. For the second one, corresponding to sequential confidence interval methods, the required number of sampling decreases when

[8]Approximate Probabilistic Model Checking



(a) The philosophers model



(b) The TQS model

Figure 1.18: Comparison of simulation time for probability measures

Time	NumValue	$p \geq ?$	UPPAAL	PRISM	YMER	COSMOS
10	0.17505	0.17	4.78	12.02	3.29	3.36
20	0.33574	0.33	11.54	23.78	7.48	5.08
40	0.56931	0.564	21.23	46.47	14.00	7.78
80	0.81894	0.814	20.10	43.46	13.01	7.60
200	0.98655	0.981	2.81	8.10	1.92	2.74

Table 1.1: Runtime comparison for the TQS for Sequential Testing

the time bound increases. It is a consequence of the evolution of the satisfaction probability of ϕ_{TQS}, which goes to 1 when T goes to infinity. COSMOS is again the fastest of the tools using a Chernoff-Hoerding method. When the time bound is $T = 200$, PLASMA is approximately 2.6 times slower, MARCIE 3.7 times slower, UPPAAL 4.2 times slower, and PRISM using APMC 6.5 times slower. Among the tools using sequential procedures, when the time bound is 40, COSMOS is still the fastest and the two versions of PRISM have similar runtime, but 2.8 times slower than COSMOS.

Sequential hypothesis testing for Tandem Queue System. The results on hypothesis testing are reported in Table 1.1. Each value is the mean over 100 experiments. The threshold value for the hypothesis is always very close to the

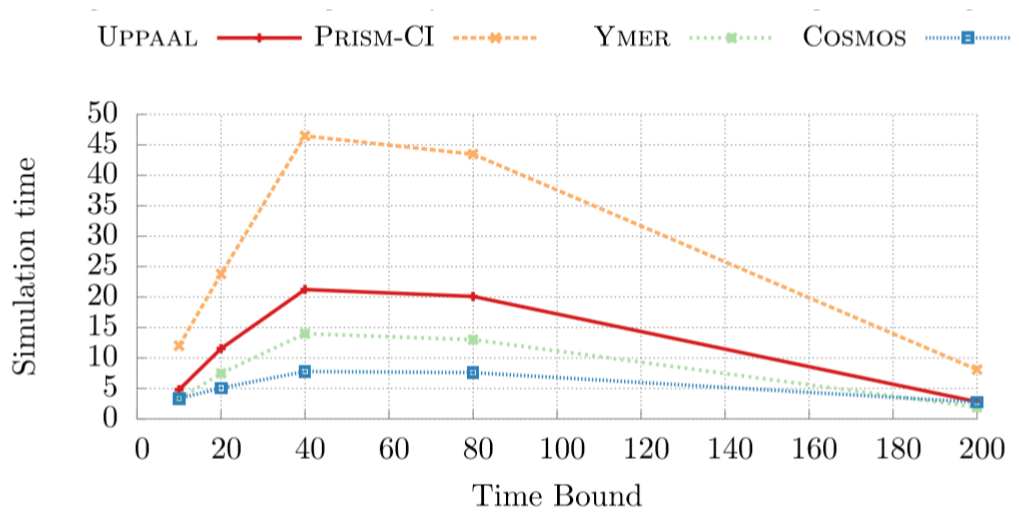


Figure 1.19: Runtime for a probability measure of the TQS model for Sequential Testing

numerical value, in order to increase the number of trajectories that tools have to perform. This number of trajectory is similar for each tool compared. Figure 1.19 shows the runtime comparison with different Time Bounds, restricted to sequential testing. In most cases, Cosmos is the fastest, Ymer is up to 1.8 times slower, Uppaal 2.8 times slower and Prism 6 times slower.

Accuracy comparison. Finally, to assess the accuracy of Cosmos, its output has been compared to the one produced by Prism via both its numerial engine and its statistical engine (with confidence level 99.99% and 0.01 interval width). The results indicate that Cosmos and the statistical engine of Prism are comparably accurate with the estimated intervals. Both always contains the value obtained using the numerical engine of Prism. This numerical value is also used to perform a coverage test of Cosmos: the ratio of simulation that return a confidence interval containing the real value is always close to the confidence level.

CHAPTER 2

APPROACHES FOR THE CONTROL OF AUTONOMOUS VEHICLES

Abstract. The control of autonomous vehicles triggers numerous issues: how to deal with the mechanical constraints of the vehicles? how to take into account the uncontrolled behaviour of other vehicles or pedestrians? how to specify several road situations such as crossing, or entrance ramps? what are the relevant safety indices? etc. Given the huge scope of these tasks, researchers have tackled subproblems, by identifying significant case studies. The goals related to these subproblems can be classified in two different ways:

- must the controller be automatically generated (*synthesised*) or manually designed and then verified?
- are the involved methods based on geometry or logic?

Given the diversity and the number of related publications, we only present a selective bibliography which provides relevant examples for each kind of approach. We emphasize that most of the methods do not support probabilities and, as such, cannot take uncertainties into account and quantify risk.

Contents

2.1 Geometry-based approaches

2.1.1 Verifying Coordinated Evasive Maneuvers [3]

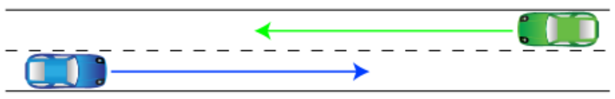
Objective. The goal is to check whether the trajectories given by another module (the planner) are robust with respect to noise and vehicle deviations.

Assumptions. In this communication, one assumes that checking evasive maneuvers is done in two steps:

- every vehicle computes periodically its spatiotemporal trajectory in the current situation, and broadcasts it to other vehicles;
- then, every controller checks whether this set of trajectories is collision-free. In this case, these trajectories are selected. In the other case, the process is iterated with additional information.

The study is restricted to the analysis of this second phase.

While broadcasting full trajectories may be a time and space-consuming task, it provides a more accurate information than the one only given by the position and speed of other vehicles (see Figure 2.1, all figures from this section are extracted from the communication). Indeed, the latter case would require a probabilistic model of the behaviour of these vehicles. In Figure 2.1(b), the support of random trajectories is illustrated by the colored cones, and it results in a probability of collision. This is addressed in [37] and [4], and induces a more costly and conservative controller design.


(a): Known trajectories

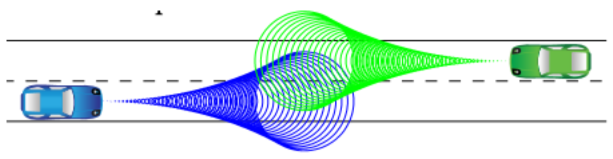
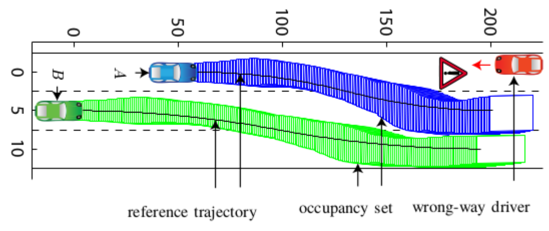
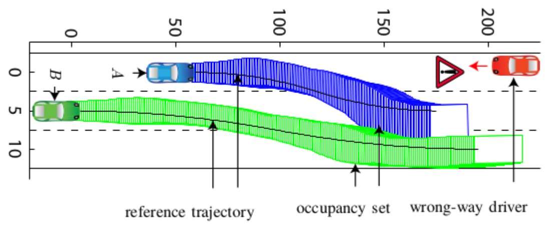

(b): Unknown trajectories

Figure 2.1: Two controller models



(a) Evasion manoeuver with $a^A_{lat} = 0.4g$



(b) Evasion manoeuver with $a^A_{lat} = 0.6g$

Figure 2.2: Occupancy sets for evasion manoeuvers

Computing trajectories with disturbances. In order to check for robustness, the model is enlarged with perturbations, leading to a deviation from the reference trajectory, which is specified by the following equation:

$$\dot{x} \in \mathcal{A}x + \mathcal{B}u \tag{2.1}$$

where:

- x is the state vector, containing the deviations to the reference trajectory and their derivatives;
- u is the input vector, containing the curvature of the reference trajectory and the steering angle of the vehicle;
- $\mathcal{A}$ and $\mathcal{B}$ are interval matrices that represent the uncertainty over the trajectory.

The actual trajectory is described by:

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{2.2}$$

where A and B are matrices compatible with $\mathcal{A}$ and $\mathcal{B}$, and $u(t) \in U$.

A temporal discretisation with time parameter r is used to solve this equation system. It consists in iteratively building the set of possible positions in the time interval $[kr, (k+1)r]$ based on the set of possible positions in $[(k-1)r, kr]$. Once this discretisation is done, the solution of the homogenous equation is computed then the inhomogenous part is added. Such a process provides a sequence of polytopes whose union is the occupancy set over the trajectory.

If the occupancy sets do not intersect over a time interval $[0, t_f]$ then the manoeveurs of all vehicles are safe.

Case of wrong-way driver. The usefulness of this method is illustrated with the case study of a vehicle going the wrong way. This vehicle puts at risk two autonomous vehicles, on a three-lane road. To reduce the risk of accident, both vehicles plan a lane-changing manoeuver, as shown in Figure 2.2.

Conclusion. This approach has been validated by several case studies. It can be generalised to the case of multiple trajectories provided by the planner. In this case, the controller selects the one minimising intersection.

2.1.2 Time-Memory Tradeoff for Collision Detection [63]

Objective. The objective is to provide an approach to reduce the computation time of collision detection in order to match human reaction delay.

Assumptions. There are two vehicles: the controlled one *ego* and another vehicle *other* of fixed sizes, both represented by rectangles located on a given area.

We suppose that the generated trajectory is represented by a sequence of connected segments of fixed length $\delta > 0$. Moreover, for two adjacent segments, the difference between their orientations is assumed lower than $\phi_{max} \in [0, \frac{\pi}{2}[$. Furthermore, the time evaluation interval $[t_0, t_n]$ is splitted in intervals $[t_{k-1}, t_k]$ for $k \in [\![1, n]\!]$.

Database. The database consists of a single relation indicating whether the two rectangles intersect or not. The value domain is finite, due to two factors: the boundedness of the area, and the discretisation of the positions. As the size of the vehicle is supposed fixed, the fields of the relation are the center and orientation of each vehicle, and a boolean indicating if there is an intersection.

Verification of trajectories. Two trajectories are considered: the trajectory of the controlled vehicle, and the center of the lane on which the other vehicle is located. The safety verification of the trajectory repeats the following steps, until t_n:

1. compute reachability and occupation sets of the vehicles, taking into account their dynamics and the uncertainties over $[t_{k-1}, t_k]$;
2. cover the occupation set of the vehicles by rectangles;
3. checks if there is a collision using the database.

Conclusion. A benchmark of the procedure has been done using a computer equipped with a 2.80GHz i5-4330M processor, 12GB of RAM and the following librairies and softwares:

- for the construction of the database, MATLAB 2014a (and it required 23.37MB of memory);
- for time measures, the Windows API (QueryPerformanceCounter),
- for collision detection algorithms, a C++ program compiled using Microsoft Visual C++ 2013 (with the O2 optimisations).

In the case of the collision detection over random rectangles, the proposed approach is faster by a factor 16.04. For verifying planned trajectories, the factor is 8.9.

2.2 Logic-based approaches

2.2.1 Control for Collision Avoidance [25]

Objective. The goal is to build a robust controller on a motorway segment.

Assumptions. Every vehicle of the segment is controlled. During the control phase, at most one vehicle may lose control and behave in a random way.

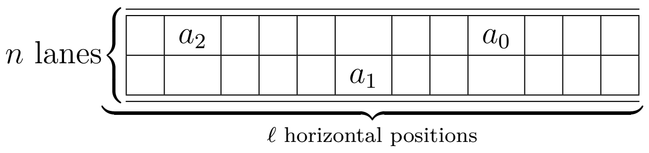


Figure 2.3: A motorway segment

A *segment* of the motorway consists of $n\ell$ positions (as shown in Figure 2.3). A *position* is defined by its coordinates (x, y) with $y \in [\![0, n-1]\!]$ the lane and $x \in [\![0, \ell-1]\!]$ the horizontal position.

Physical parameters (such as the maximal speed *vmax*, maximal acceleration *amax*, minimum entry speed *vmin* and minimum delay between two entering vehicles) are assumed to be related to the motorway section. Moreover, it is supposed that state and position changes are synchronous. We give below a specification for relevant information of a vehicle and the configuration of a section:

A *vehicle* present in the section is a tuple $a = \langle a.x, a.y, a.v, a.c\rangle$ where: $(a.x, a.y)$ is the position of a; $a.v \in [\![0, vmax]\!]$ is its current horizontal speed; and $a.c \in \mathbb{B}$ a boolean denoting whether the vehicle is controllable. A *configuration* of the section is a tuple $s = \langle d, A\rangle$ where A is a finite set of vehicles, and d is an integer array indexed by the lanes such that $d[i]$ denotes the time elapsed since the last entrance on lane i when $d[i] < dmin$, and $d[i] = dmin$ otherwise.

A possible configuration is shown in Figure 2.3. We consider a motorway segment of one kilometer, with two lanes and $\ell = 100$ positions. There are at most 25 vehicles on each lane, due to the safety distance. The speed values are in $[\![0, 4]\!]$, which corresponds to a maximum speed of 40m.s^{-1}. The possible acceleration values are -1, 0 and 1.

Principle. This situation can be seen as a turn-based game between two players, *environment* and *controller*. It is a zero-sum game as the environment tries to force a collision and the controller wants to avoid it.

Description of the game. A *configuration* of the game is a tuple $s = \langle d, A\rangle$ where A is a finite set of vehicles, and d is an integer array indexed by the lanes such that $d[i]$ denotes the time elapsed since the last entrance on lane i when $d[i] < dmin$, and $d[i] = dmin$ otherwise.

The next paragraphs describe the behaviour of the system in a formal way.

Vehicle movement constraints. Let $a = \langle a.x, a.y, a.v, a.c\rangle$ be a vehicle, its next state is written as $\langle a.x', a.y', a.v', a.c\rangle$, where:

- $a.x' = a.x + a.v$ (and if $a.x' > \ell$ then the vehicle is removed);
- $a.y' \in [a.y - 1, a.y + 1] \cap [0, n - 1]$ if $a.v > 0$ ($a.y' = a.y$ otherwise);
- $a.v' \in [a.v - amax, a.v + amax] \cap [0, vmax]$.

The controllability status is not modified by a vehicle move. The speed of the vehicle can change in a non-deterministic manner within the requirements of the motorway segment. If the vehicle has a positive speed, it can also change lane.

The system evolves from a state $s = \langle d, A\rangle$ to a state $s' = \langle d', A'\rangle$ in two steps:

- the *environment* step: a vehicle can be marked as non-controlled (if there is no such one), some vehicles may be added to the segment, and the non-controlled vehicle is moved (if it exists);
- the *controller* step: all controlled vehicles are moved.

Let $s = \langle d, A\rangle$ be a configuration. Then $s_1 = \langle d_1, A_1\rangle$ can be reached by a *environment transition* by the following operations:

1. If no vehicle in A is uncontrollable, then the environment may select $a \in A$ and set $a.c = \perp$;
2. For every lane i, the value $d_1[i] = \min(d(i) + 1, dmin)$ is computed and a new vehicle ($a.x = 0, a.y = i, a.v \in \{vmin, \ldots, vmax\}, a.c = \top$ may be inserted if $d_1[i] = dmin$; the delay must then be reinitialised: $d_1[i] = 0$;
3. If there is a non-controlled vehicle a, it can be moved according to vehicle movement constraints.

Let $s_1 = \langle d_1, A_1\rangle$ be a configuration (obtained after an environment step), then $s' = \langle d', A'\rangle$ can be reached in one *controller transition* by moving each controlled vehicle of A_1 according to vehicle movement constraints.

Let $s = \langle d, A\rangle$ be a configuration. Then $s' = \langle d', A'\rangle$ can be reached by a *complete* transition if there exists an intermediate configuration $s_1 = \langle d_1, A_1\rangle$ such that an environment transition leads from s to s_1 and a controller transition leads from s_1 to s'. Moreover, for every $a_1 \neq a_2 \in A \cap A'$:

1. $a_1.x' \neq a_2.x' \vee a_1.y' \neq a_2.y'$ (two distinct vehicles do not share position);
2. If $a_1.x < a_2.x$, then either $a_1.x' < a_2.x'$ or ($a_2.x' \leq a_1.x'$ and $a_1.y = a_1.y'$ and

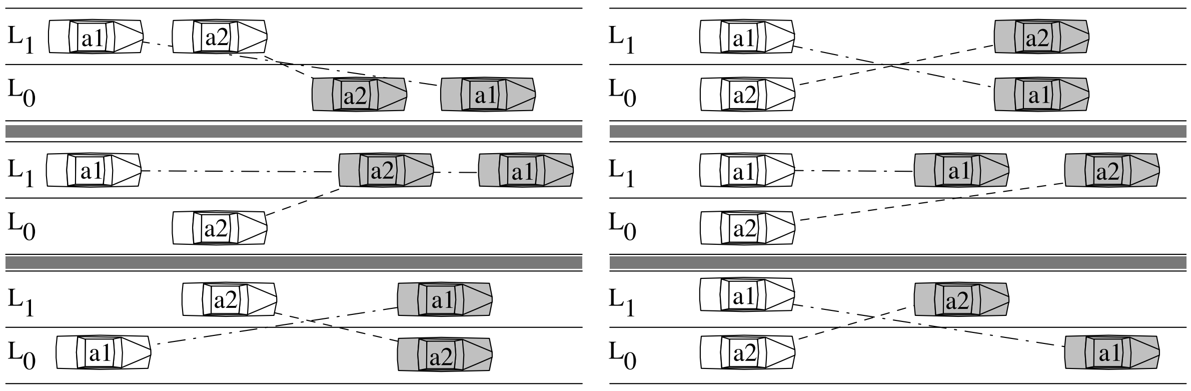


Figure 2.4: Risky behaviours to be avoided

$a_2.y = a_2.y'$ and $a_1.y \neq a_2.y)$: if a_1 is behind a_2, then either a_1 stays behind or the two vehicles stay in their respective lanes (only if they were in separate lanes);

3. If $a_1.x = a_2.x$ then $a_1.y = a_1.y'$ and $a_2.y = a_2.y'$: if the two vehicles start from the same horizontal position, they stay in their respective lanes.

These additional conditions help to avoid risky behaviours, as shown in Figure 2.4 extracted from [25].

It is well known that for such games, there exist positional winning strategies. If the initial state is winning for the controller, its strategy gives us an implementation of the controller; in the other case, the environment strategy provides a failure scenario that may help to modify the settings of the problem in such a way that the controller is winning. Reachability games are solved in polynomial time by a saturation algorithm, that we now describe.

Let S be the set of all configurations and let s_0 be the initial configuration, in which the motorway section is empty. For $s \in S$ we define:

- $Succ_e(s)$ the set of successors of s by a environment transition;
- $Succ_c(s)$ the set of successors of s by a controller transition.

The set $S_{fail} = \{s \mid Succ_c(s) = \emptyset\}$ denotes the set of configurations in which the controller can not avoid a collision and we define $S^* = S \setminus S_{fail}$.

A *strategy* is a mapping $f : S^* \to S$ such that $f(s) \in Succ_c(s)$. The corresponding set of states $G_f(s_0) = G^c_f(s_0) \cup G^e_f(s_0)$ is defined inductively by:

- $s_0 \in G^c_f(s_0)$;
- if $s \in G^c_f(s_0)$ then $\forall s' \in Succ_e(s), s' \in G^e_f(s_0)$
- if $s \in G^e_f(s_0) \cap S^*$ then $f(s) \in G^c_f(s_0)$.

It is a *winning* strategy if $G^e_f(s_0) \cap S_{fail} = \emptyset$.

Informally, the standard algorithm searching for a winning strategy is as follows.

Let $Sbad_c$ (resp. $Sbad_e$) be the set of configurations from which the controller will lose (resp. the environment will win) if it its turn to play.

1. These sets are initially defined by: $Sbad_e = S_{fail}$ and $Sbad_c = \emptyset$;
2. These two sets are enriched using the following rules:
   a Let s be a configuration such that $\forall s' \in Succ_c(s), s' \in Sbad_e$, then $Sbad_c = Sbad_c \cup \{s\}$;
   b Let s be a configuration such that $\exists s' \in Succ_e(s), s' \in Sbad_c$, then $Sbad_e = Sbad_e \cup \{s\}$.
3. The algorithm stops if either $s_0 \in Sbad_c$ (in which case there is no winning strategy for the controller), or when no rule can be applied anymore.

This algorithm is polynomial with respect to the size of the system. However, such a system is generally very big, which requires to define adapted data structures for the implementation.

Example 19.

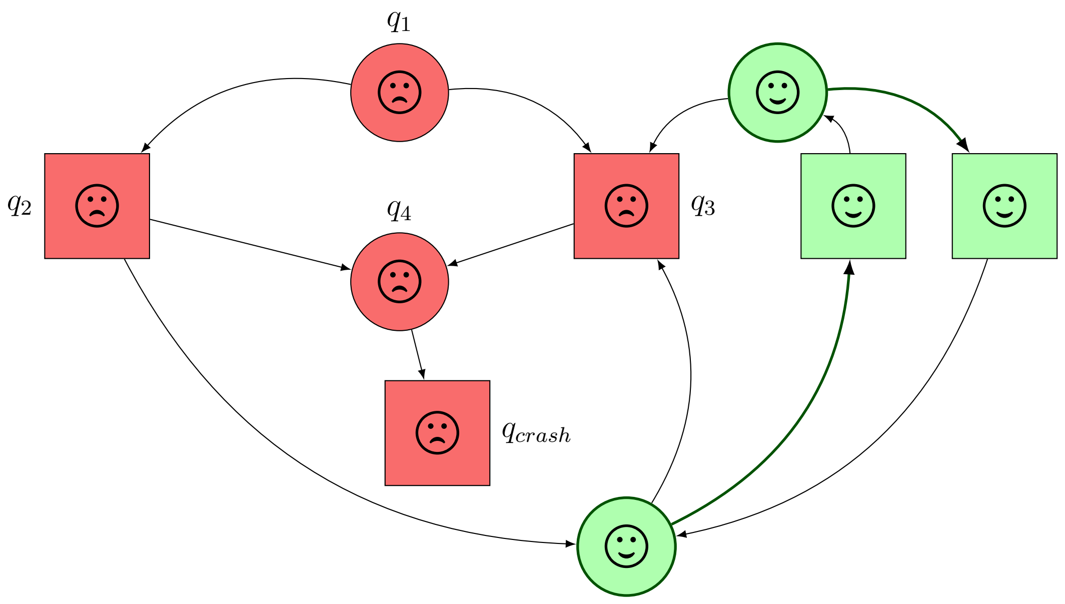


Figure 2.5: Illustrating the saturation algorithm

In Figure 2.5, we present a simplified version where the states are partitioned into two sets: the states where the environment can play (represented with squares), and those where the controller can play (represented by circles). The states that lead to a collision are represented in red.

First, we have $Sbad_e = S_{fail} = \{q_{crash}\}$ and $Sbad_c = \emptyset$. During the first step, the controller state q_4 that can only lead to a crash is added to $Sbad_c$. Then, the environment states q_2 and q_3 leading to $q_4 \in Sbad_c$ are added to $Sbad_e$. Finally, the controller state q_1 which can only lead to $Sbad_e$ is added to $Sbad_c$. The green states are safe states.

Conclusion. Other works have implemented the algorithm described above. In particular, in [75] the objective is to generate a controller for a system described by a Java program.

2.2.2 Verification of Smart Intersections [59]

Objective. The case study presented in the chapter 6 of this PhD thesis focuses on Cooperative Intersection Collision Avoidance Systems (CICAS) that would enhance intersections with controllers that communicate with “smart” vehicles. More specifically, it focuses on the CICAS for Stop-Sign Assist implementation (CICAS-SSA), where a driver crossing the road needs to decide when it enters in the crossing, depending on the gaps between other vehicles and the crossing. The verification objective is to ensure that, if the *subject vehicle*[1] driver follows the system advice, there is no collision.

Assumptions. The other vehicles behave according to the expected behaviour on a motorway.

Principles. The input of the method is the strategy of the intersection controller, which is specified as follows: the intersection controller allows the subject vehicle to enter the crossing only if all the oncoming vehicles are far enough away (beyond a fixed distance) from the intersection. Note that it is an advice, and the vehicle may wait until the next one.

A hierarchical heterogeneous tree (see Figure 2.6 extracted from [59]) is built: the idea is to decompose the problem into subproblems. The basic subproblems (i.e. located at the leaves of the tree) are modelled with an appropriate formalism and solved by a dedicated method. The other subproblems combine subproblems already handled and are then solved by compositional methods. More specifically, the compositional methods are either conjunctive, disjunctive, or disjunctive with switching (see below, node $3j$).

We only present some subproblems of the case study, one for each kind of composition.

Node 01. The root presents a case of conjunction. The first level nodes correspond to independent aspects of the system:

- in Node 11, the goal is to estimate the driver’s response time;

[1]the vehicle at the stop-sign, served by CICAS-SSA

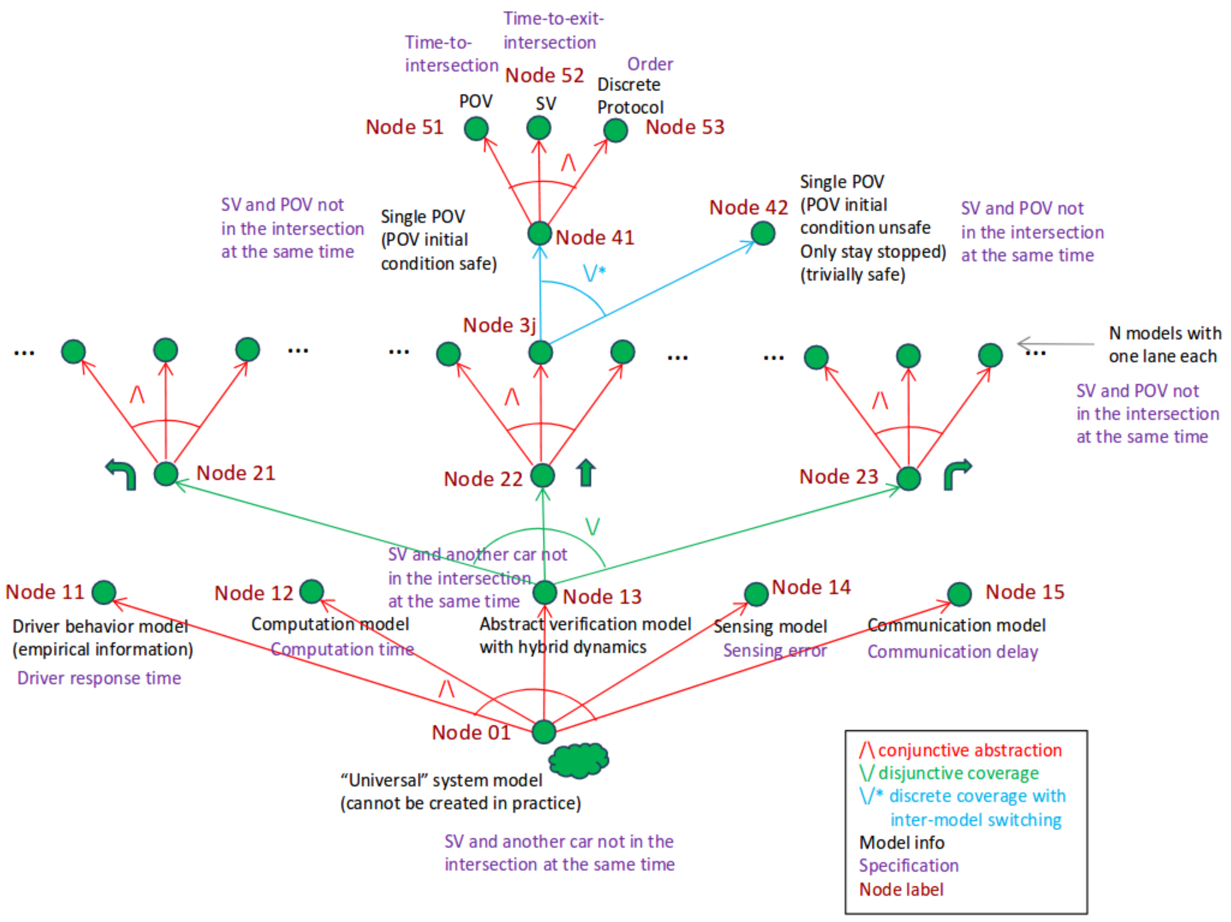

Time-to-exit-intersection
Time-to-intersection
Node 52
Order
Discrete Protocol
POV
SV
Node 51
Node 53
/\
SV and POV not in the intersection at the same time
Single POV (POV initial condition safe)
Node 41
Node 42
Single POV (POV initial condition unsafe Only stay stopped) (trivially safe)
SV and POV not in the intersection at the same time
\/*
Node 3j
N models with one lane each
SV and POV not in the intersection at the same time
Node 21
Node 22
Node 23
\/
SV and another car not in the intersection at the same time
Node 11
Node 12
Node 13
Node 14
Node 15
Driver behavior model (empirical information)
Driver response time
Computation model
Computation time
Abstract verification model with hybrid dynamics
Sensing model
Sensing error
Communication model
Communication delay
Node 01
"Universal" system model (cannot be created in practice)
SV and another car not in the intersection at the same time
/\ conjunctive abstraction
\/ disjunctive coverage
\/* discrete coverage with inter-model switching
Model info
Specification
Node label

Figure 2.6: CICAS-SSA hierarchical heterogeneous verification tree ([59], p. 99)

- in Node 12, the goal is to do worst-case execution time analysis on the computational model – estimating the delay in which the response of the intersection controller is given;
- in Node 14, the goal is to make sure that the error of the sensing subsystem is always bounded, within a given range of positions and velocities;
- in Node 15, the goal is to find the upper bound on the communication delay between the sensing subsystem and the computation subsystem
- in Node 13 (see below), the goal is to establish that the subject vehicle is not in the intersection at the same time as another vehicle, for given values on computation time, communication time, sensing errors and driver response time.

The Node 01 is a general case of the Node 13, that is completed with the analysis from Node 11, 12, 14 and 15. Therefore, this conjunctive abstraction completes the verification task of Node 01 (if the verification tasks of all first level nodes have been handled).

Node 13. This node corresponds to a case of disjunction. The goal described in this node is to ensure that the subject vehicle is not in the intersection at the same time as another vehicle. The verification model M_{13} (shown in Figure 2.7) is consists of two hybrid automata components:

- the **Major Road** component models the dynamics of oncoming vehicles, where x represents the position of the nearest[2] oncoming vehicle from the intersection (called the *principal other vehicle*). The value of x is always negative, by the definition of the intersection (which is located at 0, and x is increasing). Note that when a vehicle reaches the intersection, another vehicle becomes this principal other vehicle and x decreases to its position.
- the **SV** components models the subject vehicle, which has to commit to a direction: either going straight, or turning left or right. It also represents the controller, through the conditions to move from a `conflict` state to a `clear` state. The position variables are distinguished with respect to the decision of the vehicle.

The collision-freedom specification S_{13} can be defined by the temporal logic formula 2.3 where constants are chosen based on a typical highway intersection geometry.

$$\Box\neg((x = 0 \wedge 0 < y_s < 4.5) \vee (x = 0 \wedge 0 < y_r < 170) \vee (x = 0 \wedge 0 < y_l < 180) \quad (2.3)$$

Models M_{21}, M_{22} and M_{23} are constructed from M_{13} by enforcing the choice of the vehicle: in M_{21} it has to go left, in M_{22} to go straight, and in M_{23} it has to go right. These are similar to the model shown in Figure 2.7 except we use only one of the three right branches, and enforce its variable name on the `waiting` state.

[2]nearest by *time-to-intersection*, determined from positions and approach velocities

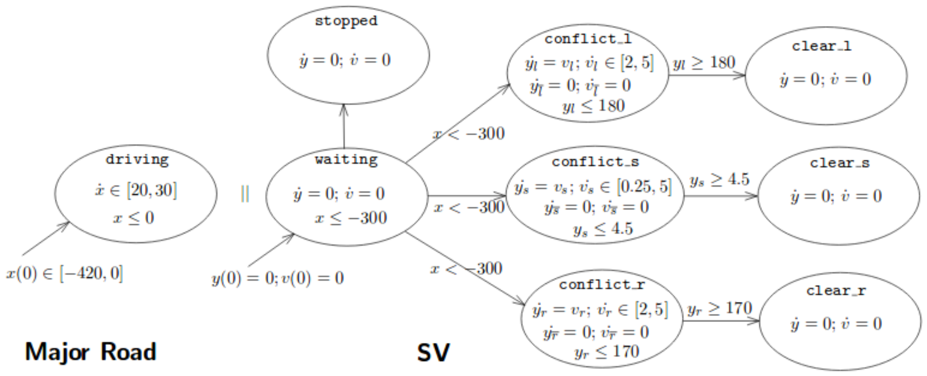

stopped
$\dot{y} = 0;\ \dot{v} = 0$
conflict_l
$\dot{y_l} = v_l;\ \dot{v_l} \in [2,5]$
$\dot{y_{\overline{l}}} = 0;\ \dot{v_{\overline{l}}} = 0$
$y_l \leq 180$
$y_l \geq 180$
clear_l
$\dot{y} = 0;\ \dot{v} = 0$
$x < -300$
driving
$\dot{x} \in [20,30]$
$x \leq 0$
||
waiting
$\dot{y} = 0;\ \dot{v} = 0$
$x \leq -300$
$x < -300$
conflict_s
$\dot{y_s} = v_s;\ \dot{v_s} \in [0.25,5]$
$\dot{y_{\overline{s}}} = 0;\ \dot{v_{\overline{s}}} = 0$
$y_s \leq 4.5$
$y_s \geq 4.5$
clear_s
$\dot{y} = 0;\ \dot{v} = 0$
$x(0) \in [-420,0]$
$y(0) = 0; v(0) = 0$
$x < -300$
conflict_r
$\dot{y_r} = v_r;\ \dot{v_r} \in [2,5]$
$\dot{y_{\overline{r}}} = 0;\ \dot{v_{\overline{r}}} = 0$
$y_r \leq 170$
$y_r \geq 170$
clear_r
$\dot{y} = 0;\ \dot{v} = 0$
Major Road
SV

Figure 2.7: Verification model M_{13} ([59], p. 101)

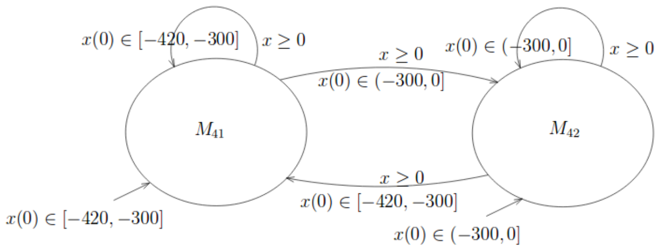


Figure 2.8: Inter-model switching that covers M_{3j} ([59], p. 106)

Those verification models correspond to nodes 21, 22 and 23 of the verification tree of Figure 2.6.

Using behaviour relations to ensure that there is no variable conflicts between the three submodels, the authors show that there are heterogeneous implications and that finally if all these submodels verify their respective specifications, M_{13} satisfies S_{13}, as depicted in the verification tree shown in Figure 2.6.

Node $3j$**.** This node corresponds to a case of a disjonction with mode switching. The goal described by this node is to ensure that the subject vehicle and a specified car are never in the intersection at the same time (in the case of a single lane). Figure 2.8 shows an inter-model switching between:

- M_{41}, initially safe, where the incoming vehicle is far enough from the intersection, and the subject vehicle can cross the intersection if it chooses to;
- M_{42}, initially unsafe, where the incoming vehicle is too close and the subject vehicle must stop.

Observe that the designer (or the safety engineer) must provide this inter-model switching. This reduces the dimension (number of vehicles involved) to one of the single-lane multi-vehicle Major Road models to one for single-lane single-principal other vehicle. The different subject vehicle actions for the two models can be analyzed independently for safety while the system is in those modes.

Conclusion. This application chapter tackles a high-level specification (the subject vehicle must not collide with other vehicles) by splitting it into subproblems, and applying the various methods developed earlier in the thesis. This early separation of various aspects of the verification model is useful in keeping the analysis complexity at check. This proof of concept may be applied to other objectives, including the vehicle industry, which heavily uses model-based designs. However, this kind of methods is limited: it is not always possible to do such a decomposition, and the mode switching operator requires additional modeling.

2.2.3 Formal Approach to Vehicle Coordination [6]

Objective. The goal of this paper is to provide a formalism describing coordination protocols between autonomous vehicles such that it can be encoded as a constraint satisfaction problem and solved by standard tools, like Z3.

Assumptions. Each vehicle is equipped with an autonomous controller and can communicate with other vehicles.

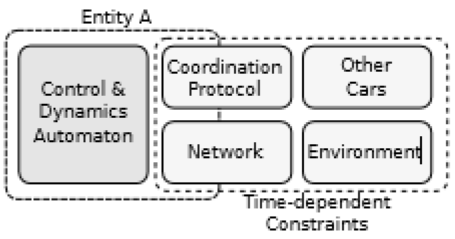


Figure 2.9: System Architecture

Principle. The system consists of (see Figure 2.9, all figures are extracted from [6]) a control module represented by an automaton, the coordination protocol, other cars, the network and of the environment. This work adopts the point of view of a single vehicle. The behaviour of the other vehicles is modelled by a set of constraints overapproximating their possible trajectories. This set of constraints, with the property, is then given to a solver (such as Z3).

Case Study: Simple Intersection. The case study consists of a four way intersection. The entity A is approaching the intersection and is assumed not to turn. There are four regions for this entity in relation to the intersection: "far away", "close", "in intersection" and "passed" (see Figure 2.10).

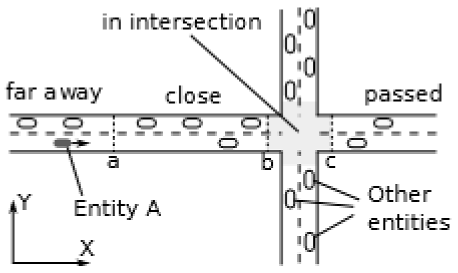


Figure 2.10: Intersection

Using this decomposition, an automaton describing the behaviour of the vehicle is built (see Figure 2.11), considering the need of a minimal speed for the crossing. The intersection is considered as a resource. The goal is to check that every reachable state satisfies a safety property, such as safety distance preservation.

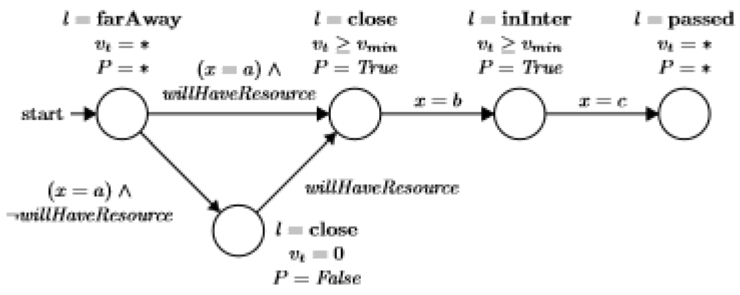


Figure 2.11: Vehicle Control Automaton

Conclusion. The solver approach has been validated by the case study, partly solved using Z3. The entire model is composed of 825 lines of SMT-lib code (including comments), and the verification by Z3 took 14 seconds on a Dell optiplex 990 with a 3.4 GHz Intel Core i7 processor, using 109MB of memory and generating 965 000 equations. There are limitations on the expressivity of the formulas (euclidian distance has not been implemented by the authors). Moreover, some features of the approach are not sufficiently described in order to convince the reader that it may be applied to a more general context.

Part II

Contributions

CHAPTER 3

AN OPERATIONAL SEMANTICS FOR SIMULINK

Abstract. We briefly mentioned Simulink® in section 1.1 as a more complex model than hybrid automata. This tool from MathWorks implements a modelling language and simulation engine, largely used in the French automotive industry for the design of controllers for autonomous vehicles. However, the model behaviours are not defined formally. For this reason, several works proposed formal translations from (subsets of) Simulink blocks to other models like hybrid automata [69, 2], or languages like Lustre [70]. Other works directly define exact semantics [21] or operational semantics [24] for Simulink. We follow the latter approach and propose semantics for a significant fragment of Simulink. We proceed in two steps: we first develop an exact version, and then enrich it with effective procedures with the aim to integrate it into the model-checker Cosmos.

Contents

3.1 Syntax

We first introduce a formal syntax for Simulink block diagrams, that we call here *Simulink models* or, shortly, *SK*-models. An *SK*-model consists of a set of blocks connected by wires supporting signals. More precisely, blocks act as *operators* that transform their input signals into output signals.

3.1.1 Types, signals and operators

Types are associated with signals and operators. We denote by $\mathcal{T}ype$ the set of types used in *SK*-models.

Definition 32 (*Basic types and constructors*)

Basic types *are a subset of* $\mathcal{T}ype$ *containing subsets of the set* $\mathbb{R}$ *of real numbers:*

- *booleans* $\mathbb{B} = \{0, 1\}$*;*
- *(un)signed integers, either represented with varying number of bits (`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`) or with usual sets* $\mathbb{N}$ *or* $\mathbb{Z}$*;*
- *floating numbers (`double`, `single`) or the entire set of real numbers* $\mathbb{R}$.

A constructor *is a function* $\mathcal{T}ype^n \to \mathcal{T}ype$*, where* $n \in \mathbb{N}$ *is the* arity *of the constructor.*

Remark. *This definition covers both exact (*$\mathbb{N}, \mathbb{Z}, \mathbb{R}$*) and operational (other types) semantics.*

Example 20. The constructor $Tuple_3$ multiplexes three signals into one, like in $Tuple_3(\mathbb{N}, \mathbb{R}, \mathbb{R})$.

Signals will be evaluated over a time domain, which will usually be a finite interval of $\mathbb{R}$ denoted by $\mathbb{T} = [t_{\text{init}}, t_{\text{end}}]$ and called the *simulation interval.* Except for a finite number of discontinuities, the signals are smooth functions because they are solutions of differential equations over subintervals:

Definition 33 (*Signals*)

A signal *of type* $\mathcal{T}p \in \mathcal{T}ype$ *is a right-continuous piecewise smooth function* $s : \mathbb{T} \to \mathcal{T}p$*, admitting a left limit denoted by* $s(t^-)$ *for each* $t \in \mathbb{T}$.
We denote by $\mathsf{type}(s)$ *the type of* s *and by* $\mathcal{S}ig_{\mathbb{T}}(\mathcal{T}p)$ *the set of signals of type* $\mathcal{T}p$ *defined on* $\mathbb{T}$.

Discontinuities are handled like in discrete event systems by applying instantaneous operations to the left-limit $s(t^-)$ to obtain the value $s(t)$.

Definition 34 (*Operators*)

An operator *is a function* $\mathrm{op} : \mathcal{S}ig_{\mathbb{T}}(\mathcal{T}p_1) \times \ldots \times \mathcal{S}ig_{\mathbb{T}}(\mathcal{T}p_m) \to \mathcal{S}ig_{\mathbb{T}}(\mathcal{T}p)$ *such* *that for each* $t \in \mathbb{T}$, $\mathrm{op}(s_1, \ldots, s_m)(t)$ *only depends on the restriction over* $[t_{\mathrm{init}}, t]$ *of the signals* $(s_i)_{1\leq i\leq m}$.

This restriction is natural as an operator cannot predict the future values of its input signals.

3.1.2 Blocks

A block contains a tuple of operators generating output signals from input signals. Blocks are classified using three criteria:

(i) Whether they are continuously evaluated. If not, they are called *discrete*, either asynchronous or synchronous. In the latter case, a *sampling delay* must be provided;

(ii) Whether there is a *latency* for the evaluation of inputs. A block without latency is called *immediate* and a non zero latency is either *positive* or *infinitesimal*, where the output value at time t only depends on the inputs over $[t_{init}, t[$ (which is the case for integration);

(iii) Whether the output value depends on threshold crossing by an input signal, called *critical input* and denoted by i_c. In this case, the threshold values $(v_i)_{i\in I}$ must be specified, as a countable increasing sequence without accumulation points. Such blocks are called *threshold blocks.*

Definition 35 (*Block type*)

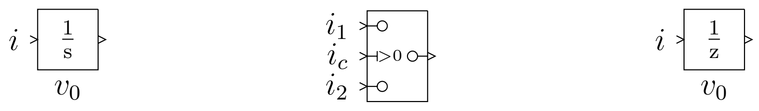
A block type *is a tuple* $\mathsf{BT} = (n, m, (\mathrm{op}_i)_{1\leq i\leq n}, b_c, b_l, b_i, b_s, \mathrm{Param})$ *where:*

- n *and* m *are the numbers of output and input signals respectively;*
- $(\mathrm{op}_i)_{1\leq i\leq n}$ *is a tuple of operators, one for each output signal;*
- b_c *is a boolean indicating if the block is discrete or continuous;*
- $b_l \in \{\mathsf{imm}, \mathsf{inf}, \mathsf{pos}\}$ *indicates if the block is immediate, with an infinitesimal or a positive latency;*
- b_s *is a boolean indicating if the block is a threshold block;*
- Param *is a set of additional parameters depending on the block type, including a sampling delay* δ *for discrete synchronous blocks and a value* $r > 0$ *for blocks with a positive latency.*

A generic block type *over a set* $\mathcal{G}$ *is a family* $(\mathsf{BT}_g)_{g\in\mathcal{G}}$ *of block types.*

For instance, the generic block type Add can be defined with $(Add_i)_{i\in\mathbb{N}}$ where i is the arity.

Example 21. We illustrate the definition of block type with standard examples:



(a): Integrator (b): Switch (c): Unit Delay

Figure 3.1: Three classical block types

The *Integrator* block (Figure 3.1(a)) computes the integral of its input: $\mathrm{op}(i)(t) = v_0 + \int_{t_{\text{init}}}^{t} i(\tau)\mathrm{d}\tau$ where $v_0 \in \mathrm{Param}$ is the single parameter. It is a continuous block ($b_c = \top$) with an infinitesimal latency ($b_l = \mathsf{inf}$), but not a threshold block ($b_s = \bot$).
The *Switch* block (Figure 3.1(b)) is a *threshold block* (over its input i_c) with exactly one discontinuity point. Its operator is defined by: $\mathrm{op}(i_1, i_c, i_2)(t) = \mathsf{if}\ i_c(t) \rhd v\ \mathsf{then}\ i_1(t)\ \mathsf{else}\ i_2(t)$ where the comparison operator $\rhd$ and the threshold value v are the additional parameters. It is a continuous and immediate block. On the figure, $\rhd$ is $>$ and $v = 0$.
The *Unit Delay* block (Figure 3.1(c)) samples its input every δ, with a δ latency. It is a discrete block with initial value v_0 as additional parameter.

3.1.3 Block list

We now give a more extensive description of the set of blocks considered in this work.

a) Blocks without inputs

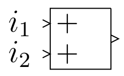
(a): Constant (b): Sine Wave

Figure 3.2: Blocks without inputs

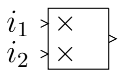
Block **Constant** is a continuous immediate block with one parameter k, its value. Its operator is defined by $\mathrm{op}_1()(t) = k$.

Block **Sine Wave** is a continuous immediate block with four parameters: A is the amplitude of the Sine Wave, f the frequency, K the phase and b the bias. Its operator is defined by $\mathrm{op}_1()(t) = A\sin(\frac{t}{\mathsf{f}} + K) + b$.

b) Immediate blocks



(a): Add



(b): Product

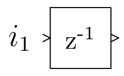
Figure 3.3: Immediate blocks

Block **Add** (Figure 3.3(a)) is an immediate block adding its two inputs: $\text{op}_1(i_1, i_2)(t) = i_1(t) + i_2(t)$. It corresponds to the instance Add_2 of the generic block type Add introduced earlier.

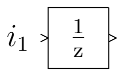
This generic block type can be extended to $(Add)_{\mathsf{sign}}$ where sign is a tuple of signs in $\{+, -\}$. For this new generic block type, the operator will be defined by $\text{op}_1((i_k)_{1 \leq k \leq m})(t) = \sum_{k=1}^{m} \mathsf{sign}_k \cdot i_k(t)$.

Block **Product** (Figure 3.3(b)) is an immediate block that computes the product of its two inputs: $\text{op}_1(i_1, i_2)(t) = i_1(t) \times i_2(t)$. It can be extended in a similar way to Add.

c) Sampling blocks



(a): Delay



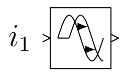
(b): Unit Delay

Figure 3.4: Sampling blocks

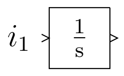
Block **Delay** is a synchronous discrete block, with three parameters: the sampling delay δ, the delay length n and the initial value v_0. It samples and delays its input signal and its behaviour is represented by the operator $\text{op}_1(i_1)(t) = i_1(t_{\text{init}} + \lfloor \frac{t - t_{\text{init}}}{\delta} - n \rfloor \delta)$. This block usually has a positive latency, unless $n = 0$ in which case it is immediate.

Block **Unit Delay** is particular case of the Delay block, where the delay length is fixed to $n = 1$. Hence, the operator is defined by $\text{op}_1(i_1)(t) = i_1(t_{\text{init}} + \lfloor \frac{t - t_{\text{init}}}{\delta} - 1 \rfloor \delta)$.

d) Continuous blocks



(a): Transport Delay



(b): Integrator

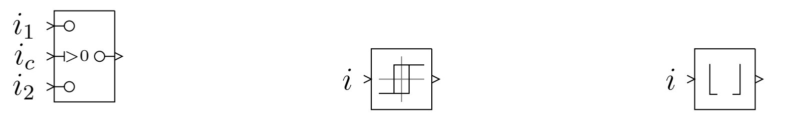
Figure 3.5: Continuous blocks

Block **Transport Delay** is a continuous block with positive latency. It has two parameters: the time delay r and the initial value v_0. It applies a latency of r to its input signal. Its operator is defined by:

$$\mathrm{op}(i_1)(t) = \begin{cases} v_0 & \text{if } t - t_{\mathrm{init}} < r \\ i(t-r) & \text{otherwise.} \end{cases}$$

Block **Integrator** is a continous block with infinitesimal latency. It has one parameter: the initial value v_0. Its operator is $\mathrm{op}(i_1)(t) = v_0 + \int_{t_{\mathrm{init}}}^{t} i_1(\tau)\mathrm{d}\tau$.

e) Threshold blocks



(a): Switch (b): Relay (c): Floor

Figure 3.6: Threshold blocks

Block **Switch** is an immediate continuous block with two parameters $\rhd \in \{>, \geq\}$ and $v \in \mathbb{R}$ (with $\rhd =>$ and $v = 0$ on Figure 3.6(a)). It compares the value of the critical input i_c with v according to the chosen comparison $\rhd$, then returns the value of i_1 or i_2 accordingly. Its operator is $\mathrm{op}(i_1, i_c, i_2)(t) = \mathsf{if}\ i_c(t) \rhd v\ \mathsf{then}\ i_1(t)\ \mathsf{else}\ i_2(t)$.

More generally, we define a generic block type $Switch_{\mathcal{T}p_1,\mathcal{T}p_2}$ with $\mathcal{T}p_1$ the type of the input signals i_1, i_2 and the output signal, and $\mathcal{T}p_2$ the type of the conditional input. If the output type $\mathcal{T}p_1$ is a boolean type, then the block is an asynchronous discrete block.

Block **Relay** (Figure 3.6(b)) is an immediate asynchronous discrete block with four parameters: v_{on}, v_{off}, s_{on}, and s_{off}. It represents a bang-bang controller which outputs v_{on} when it is activated and v_{off} otherwise. It has one input and one output and its current state can be viewed as an internal signal. Its unique operator is defined as follow:

$$\mathrm{op}(i)(t) = \begin{cases} v_{\mathsf{on}} & \text{if } i(t) > s_{\mathsf{on}} \\ v_{\mathsf{off}} & \text{if } i(t) < s_{\mathsf{off}} \\ \mathrm{op}(i)(t^-) & \text{if } t > t_{\mathrm{init}} \\ v_{\mathsf{off}} & \text{otherwise.} \end{cases}$$

Block **Floor** (Figure 3.6(c)) is an immediate asynchronous discrete block without parameter. It represents the exact floor function with the countable set $\mathbb{Z}$ of threshold points, with:

$$\mathrm{op}(i)(t) = n \text{ if } n \leq i(t) < n+1.$$

f) (Stateflow) Chart

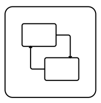


Figure 3.7: Block Chart

A *(Stateflow) Chart* is a discrete asynchronous block describing a deterministic state machine, where transitions are equipped with guards and behaviours are triggered by *events*. An *event* is defined by an input signal and an activation mode:

Definition 36. *An event e is a pair (x, tr) composed of a boolean input signal $x = \mathrm{src}(e)$ and a trigger type $\mathsf{tr} = \mathrm{trig}(e) \in \{\uparrow, \downarrow\}$. We denote by E the set of events.*

An event operates as a block with the exception that its output is not a signal as it does not fulfill the right-continuity condition.

Definition 37. *The semantics of an event e is the mapping $[\![e]\!] : \mathbb{T} \to \mathbb{B}$ defined by: $[\![e]\!](t) = \neg\,\mathrm{src}(e)(t^-) \wedge \mathrm{src}(e)(t)$ if $\mathrm{trig}(e) = \uparrow$ and $[\![e]\!](t) = \mathrm{src}(e)(t^-) \wedge \neg\,\mathrm{src}(e)(t)$ if $\mathrm{trig}(e) = \downarrow$.*

An *activation* of E is an element $b = (b_e)_{e \in E}$ of $\mathbb{B}^E$. Anticipating on the definition of Chart activation times for a more intuitive point of view, the meaning of b_e is that it holds at time t if $[\![e]\!](t)$.

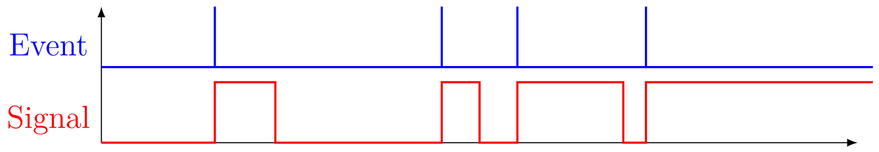


Figure 3.8: A signal and its $\uparrow$ event

The input and output signals of a Chart are considered as variables of the state machine, respectively denoted by X_I and X_O. We write $X = X_I \uplus X_O$ for the set of variables and $\mathsf{type}(X) = \prod_{x \in X} \mathsf{type}(x)$. A *valuation* is an element v of $\mathsf{type}(X)$, also written as a tuple $v = (v(x))_{x \in X}$.

In order to specify transitions, we need a syntax to define their guards:

Definition 38. *For the set X of variables and the set E of events, the set $\mathsf{G}(X, E)$ of guards is defined by the following grammar:*

$$\begin{array}{rcl} ag & ::= & F \mid x \bowtie c \mid x \bowtie x \mid \tau \\ g & ::= & ag \mid g \wedge g \mid g \vee g \end{array}$$

where $F \subseteq E$ is a subset of events, $\tau \in \mathbb{Q}_+$ represents a time constraint, $x \in X, \bowtie \in \{<, \leq, =, \geq, >, \neq\}$.

Variable updates are generally specified by MATLAB code. Therefore, in order to avoid *ad hoc* notations, we define them in a semantical way. An update modifies the values of output variables according to the current values of all variables.

Definition 39. *An update is a mapping* $a : \mathsf{type}(X) \to \mathsf{type}(X)$ *such that for every* $x \in X_I : a(v)(x) = v(x)$*. The set of updates is denoted by* $\mathsf{Up}(X)$*.*

We now have all the ingredients necessary to define a Chart:

Definition 40 (*(Stateflow) Chart*)

A Chart *is a tuple* $\mathcal{S} = (Q, q_0, X, E, \Delta, \mathrm{Pri}, \mathrm{init})$ *where:*

- Q *is a finite set of states and* $q_0 \in Q$ *is the initial state;*
- $X = X_I \uplus X_O$ *is a finite set of variables composed of a set* X_I *of input variables and a set* X_O *of output variables;*
- E *is a finite set of events;*
- $\Delta \subseteq Q \times \mathsf{G}(X, E) \times \mathsf{Up}(X) \times Q$ *is the transition function;*
- $\mathrm{Pri} : \Delta \to \mathbb{N}$ *is an injective priority function;*
- $\mathrm{init} \in \mathsf{Up}(X)$ *is the initial action.*

A transition $\delta = (q_1, g, a, q_2) \in \Delta$ is written $q_1 \xrightarrow{g,a} q_2$.

Example 22.

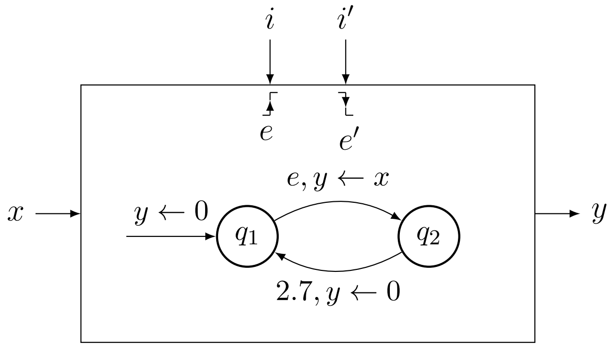


Figure 3.9: A Chart

A Stateflow Chart is depicted in Figure 3.9. It has two states $Q = \{q_1, q_2\}$, two variables $X = \{x, y\}$, two events $E = \{e, e'\}$ with $\mathrm{trig}(e) = \uparrow$ and $\mathrm{trig}(e') = \downarrow$, and an initial action $y \leftarrow 0$. For this Stateflow Chart, no priority function is required as there is only one possible transition at each state. Finally, while event e' is not linked to any transition, it can be used to activate the Stateflow Chart.

In order to evaluate guards and updates, we need the notion of environment. An environment is composed of a valuation for every signal, an activation condition and the time elapsed since the last state change:

Definition 41. *An* environment *of a Chart $\mathcal{S}$ is a tuple* $\text{env} = (v, b, d)$ *where v is a valuation, b an activation and d a duration. A* configuration *of a Chart is a pair* (q, env) *where q is a state of $\mathcal{S}$ and* env *an environment.*

Note that q and d are stored as *internal* signals of the Chart block.

Definition 42 (Evaluation of a guard)**.** *Given an environment* $\text{env} = (v, b, d)$ *and a guard g, we define* $\text{env} \models g$ *inductively by:*
- $\text{env} \models F$ *if* $\exists e \in F : b_e$*;*
- $\text{env} \models x \bowtie c$ *if* $v(x) \bowtie c$*;*
- $\text{env} \models x \bowtie y$ *if* $v(x) \bowtie v(y)$*;*
- $\text{env} \models \tau$ *iff* $d \geq \tau$*;*
- $\text{env} \models g_1 \wedge g_2$ *iff* $\text{env} \models g_1$ *and* $\text{env} \models g_2$*;*
- $\text{env} \models g_1 \vee g_2$ *iff* $\text{env} \models g_1$ *or* $\text{env} \models g_2$*.*

Definition 43 (Evaluation of an update)**.** *Let* $a \in \mathsf{Up}(X)$ *and* $\text{env} = (v, b, d)$*. We define:* $a(\text{env}) = (v', b, 0)$ *where* $\forall x \in X : v'(x) = a(v)(x)$*.*

Example 23. In the Chart shown in Figure 3.9, the update function defined by the transition from q_1 to q_2 is $a : \mathsf{type}(x) \times \mathsf{type}(y) \to \mathsf{type}(x) \times \mathsf{type}(y)$ such that $a((\alpha, \beta)) = (\alpha, \alpha)$. Note that if a variable is not specified on the transition, its value is unchanged.

Definition 44. *A transition* $\delta = q_1 \xrightarrow{g,a} q_2 \in \Delta$ *is enabled in a configuration* (q_1, env) *if* $\text{env} \models g$*. The configuration reached by firing δ is* $(q_2, a(\text{env}))$*. The global transition is written* $(q_1, \text{env}) \xrightarrow{\delta} (q_2, a(\text{env}))$*.*
We have $(q, \text{env}) \overset{\delta}{\Rightarrow} (q', \text{env}')$ *if:*
- $(q, \text{env}) \xrightarrow{\delta} (q', \text{env}')$*;*
- *there is no* $\delta' \in \Delta$ *such that* $\text{Pri}(\delta') < \text{Pri}(\delta)$ *enabled in* (q, env)*.*

We define $\Rightarrow = \cup_{\delta \in \Delta} \overset{\delta}{\Rightarrow}$ *and* $\Rightarrow^*$ *the transitive reflexive closure of* $\Rightarrow$*.*

We now explain how, on activation, a Chart block updates its output and internal signals. We adopt here the maximal step semantics meaning that all possible transitions are fired to obtain the next configuration. We assume here that every loop between states contains either at least one time constraint or exclusive conditions on input signals.

Definition 45. *We define* $(q, \text{env}) \overset{\max}{\Longrightarrow} (q', \text{env}')$ *if and only if* $(q, \text{env}) \Rightarrow^* (q', \text{env}')$ *and there is no enabled transition from* (q', env')*. The resulting configuration* (q', env') *is denoted by* $\text{SuccMax}((q, \text{env}))$*.*

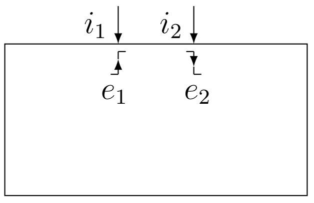
Let $(q', \text{env}') = \text{SuccMax}((q, \text{env}))$ with $\text{env}' = (v', b', d')$. We write:

- $q' = \text{SuccMax}_{\text{st}}(q, \text{env})$;
- $d' = \text{SuccMax}_{\text{dur}}(q, \text{env})$;
- $v'(x) = \text{SuccMax}_{x}(q, \text{env})$ for all $x \in X_O$.

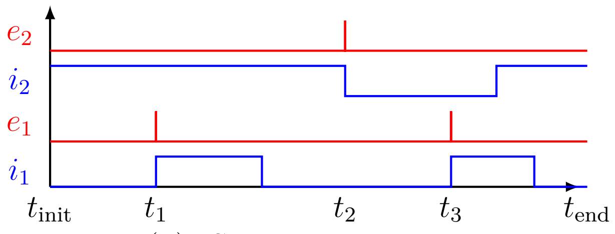
In order to completely specify the semantics of a Chart, we introduce the sequence $(t_i)_{i\geq 0}$ of *activation* times containing (in increasing order) the initial time and the times where at least one event is activated.

The time sequence can be determined from the specification of the input signals, as illustrated by the following example:

Example 24.



(a): Stateflow block



(b): Signals and events

Figure 3.10: Sequence of activation times

The event signals e_1 and e_2 in Figure 3.10 are computed respectively from input signals i_1 and i_2. The sequence of activation times, in this example, is $(t_{\text{init}}, t_1, t_2, t_3)$.

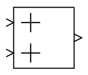
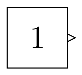
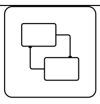
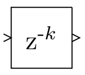
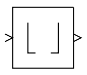
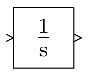
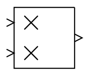
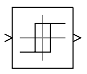
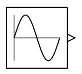
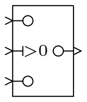
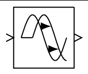
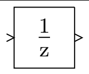
From the time sequence $(t_i)_{i\geq 0}$, the operator of the Chart provides the corresponding sequence of configurations $(q_i, (v_i, b_i, d_i))_{i\geq 0}$. Given $(q_i, (v_i, b_i, d_i))$ at time t_i, the configuration at time t_{i+1} is defined as follows:

- First, the configuration at t_{i+1}^- is built without changing the inputs: $(q_i, (v_i, b_i, d_i + t_{i+1} - t_i))$
- Then, the environment for t_{i+1} is obtained by copying output signals and updating the valuation of inputs signals alongside with their events: $(q_i, (v'_i, b_{i+1}, d_i + t_{i+1} - t_i))$
- This finally yields the configuration: $(q_{i+1}, (v_{i+1}, b_{i+1}, d_{i+1})) = \text{SuccMax}((q_i, (v'_i, b_{i+1}, (d_i + t_{i+1} - t_i))))$

Note that we only have considered a subset of the capabilities of Stateflow Charts, which includes their main features. For instance, we do not consider nested Stateflow Charts.

g) Summary

We give a summary of blocks in alphabetical order:

 Add p.59	 Constant p.58	 Chart p.61–64	 Delay p.59	 Floor p.60
 Integrator p.60	 Product p.59	 Relay p.60	 Sine Wave p.58	 Switch p.60
 Transport Delay p.60	 Unit Delay p.59			

3.1.4 *SK*-models

An *SK*-model defines an architecture where blocks are *instances* of block types with their parameter values.

Definition 46 (*SK-Model*)

An SK-model $\mathcal{M} = (\mathcal{B}, L)$ consists of:

(1) A set $\mathcal{B}$ of blocks, defined by their respective type and parameter values. We denote by $\langle B, o\rangle$ an output o of block B, and $\langle i, B\rangle$ an input i of B.

(2) A set L of links *of the form $(\langle B', o\rangle, \langle i, B\rangle)$ satisfying: for any input $\langle i, B\rangle$, there is exactly one output $\langle B', o\rangle$ such that $(\langle B', o\rangle, \langle i, B\rangle) \in L$.*

Example 25. An example of *SK*-model is shown on Figure 3.11. It describes a hybrid system with two Integrator blocks and a threshold block. This toy example already requires complex evaluation since even without block B_1 it cannot be directly written as a differential equation due to block B_3 which features a positive latency.

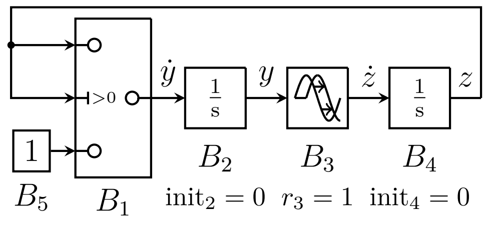


Figure 3.11: An *SK*-model

In this figure (and later on) we use intuitive abbreviations: init_2 is the initial value of block B_2, r_3 the latency of block B_3 and init_4 the initial value of block B_4.

Simulink models have an underlying graph representation which has a semantic interest.

Definition 47

The graph $\mathcal{G}_{\mathcal{M}}$ of the SK-model $\mathcal{M} = (\mathcal{B}, L)$ is a labelled directed (multi-)graph defined by:
- *the set $\mathcal{B}$ of nodes*
- *for every link $(\langle B', o\rangle, \langle i, B\rangle) \in L$ an edge from B' to B labelled by (o, i).*

Example 26. The graph of the *SK*-model from Figure 3.11 is given in Figure 3.12:

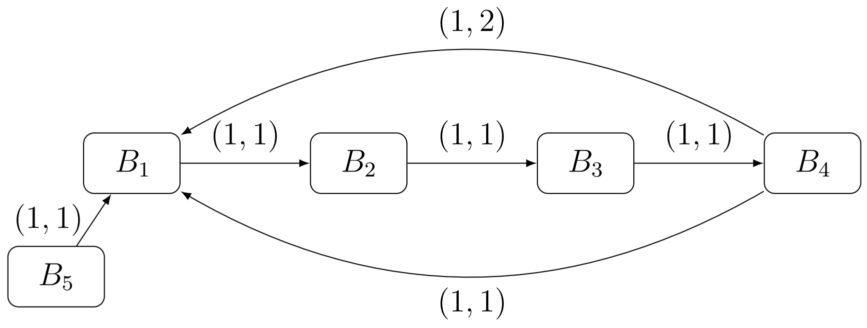


Figure 3.12: Graph of the *SK*-model of Figure 3.11

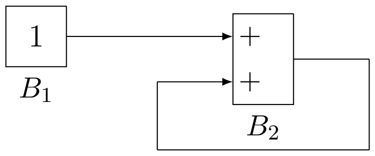
Anticipating on section 3.2, we informally define a *trajectory* of an *SK*-model $\mathcal{M}$ as a vector $\vec{w}$ of values for all output signals over $\mathbb{T}$.

The application of block operators in order to obtain a trajectory requires temporal dependency constraints between blocks and more specifically between immediate blocks. The following definition allows us to identify incorrect dependencies:

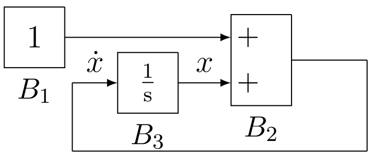
Definition 48

An immediate cycle *of an SK-model $\mathcal{M}$ is a cycle in $\mathcal{G}_{\mathcal{M}}$ along immediate blocks. An SK-model is* correct *if it does not contain any immediate cycle.*

Example 27. We illustrate the notions above with the *SK*-model shown in Figure 3.13(a), which contains an immediate cycle with immediate block B_2 linked to itself. It would represent the equation $x(t) = x(t) + 1$ for all $t \in \mathbb{T}$, which is impossible.



(a): An *SK*-model with an immediate cycle



(b): A correct *SK*-model

Figure 3.13: Illustrating the correctness of *SK*-models

In the *SK*-model shown in Figure 3.13(b), an Integrator block has been added to break the immediate loop. With this addition, it now represents the differential equation $\dot{x} = x + 1$.

As shown in the next proposition, the correctness property is easily decided.

Proposition 5. *SK-model correctness is decidable in linear time.*

Proof: It is done with a topological sort on its graph restricted to immediate blocks. ■

The decision procedure could also use a depth-first search algorithm but the topological sort is used later (definition 54) in the notion of block ordering. Like in Simulink, from now on, we only consider correct models.

3.2 Exact semantics

Since an *SK*-model represents a deterministic hybrid system, its semantics should be a unique trajectory. It should be clear that such a semantics cannot be operational since the trajectory is evaluated over an interval in $\mathbb{R}$. Moreover, even with discrete sampling points, the possible presence of an Integrator block forbids the exact computation of these values. Furthermore, due to the general form of implicit differential equations, there is no guarantee that a trajectory exists.

Principle. The trajectory of an *SK*-model over its simulation interval $\mathbb{T} = [t_{\text{init}}, t_{\text{end}}]$, if it exists, can be obtained informally as follows. The interval $[t_{\text{init}}, t_{\text{end}}]$ is split into a finite sequence of contiguous subintervals. The intermediate interval bounds correspond to either threshold crossing or discrete block sampling. Within each interval, the trajectory is the solution of a differential equation system depending on the signal values obtained until then.

3.2.1 Differential equations of an *SK*-model

In this section, we first define the backward graph of a block in an *SK*-model. We also define the notions of *mode* (for threshold blocks) and *history* (for blocks with latency). We use these definitions to explain how to obtain the differential equations that represent the behaviour of the *SK*-model.

Example 28. Using Add, Product and Integrator blocks, it is easy to represent polynomial differential equations systems by *SK*-models. It requires exactly as many Integrator block as variables in the system.
For instance the system $\{\dot{x} = x^2 + xy, \dot{y} = xyz, \dot{z} = y^2\}$ might be described as shown in Figure 3.14:

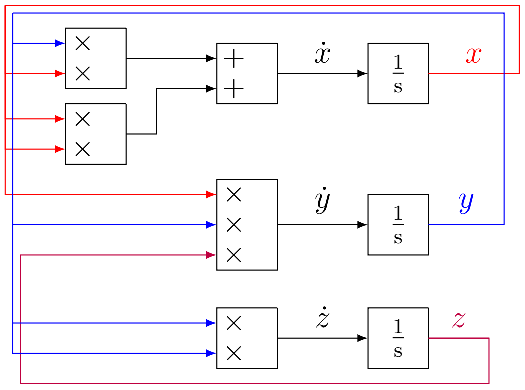


Figure 3.14: A polynomial differential equation system encoded into a *SK*-model

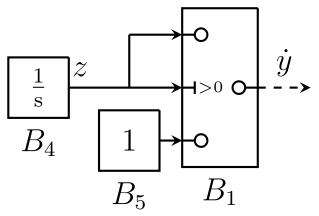
To specify the differential equation of an Integrator block B, a backward exploration of $\mathcal{G}_{\mathcal{M}}$ is first performed from a block, denoted by B^-, for which one of the outputs, denoted by o^-, is connected to the input of B. This exploration stops when meeting a non-immediate block. Subgraphs obtained by backward explorations are defined as follows:

Definition 49 (*Backward graph*)

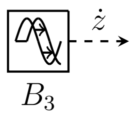
Let $\mathcal{M}$ be an SK-model. The backward graph $\mathcal{G}_B$ *of a block B is a directed acyclic graph of root B, defined inductively by:*

- *if B is a non-immediate block, then $\mathcal{G}_B$ is the block B without any edge;*
- *otherwise, let $B_1, \ldots, B_m$ be the blocks with outputs linked to input signals of B; then $\mathcal{G}_B$ is obtained by adding B and the links from $B_1, \ldots, B_m$ to B to $\cup_{i=1}^m \mathcal{G}_{B_i}$.*

Example 29. For the *SK*-model of Figure 3.11, the backward graphs of blocks B_1 and B_3, connected to Integrator blocks B_2 and B_4 respectively, are given below:



(a): Backward graph of block B_1



(b): Backward graph of block B_3

Figure 3.14: Some backward graphs of the model of Figure 3.11

The specification of differential equations also depends on the *mode* of threshold blocks. We denote by $\mathsf{Th}(\mathcal{M})$ the set of threshold blocks of $\mathcal{M}$.

Definition 50 (*Mode*)

Given a threshold block with threshold values $\{v_i\}_{i\in I}$, these discontinuities define a partition of $\mathbb{R}$ into contiguous intervals. A mode *of $\mathcal{M}$ is a mapping associating an interval with each block in $\mathsf{Th}(\mathcal{M})$.*

To complete the specification of the differential equation system, we still have to define which functions should be substituted for each non-immediate block. There are three types of non-immediate blocks B to be considered:

- B is a block without inputs. In this case, the function associated with each output of B is substituted in the differential equation;
- B is an integration block. In this case, the integration variable associated with the output of B is substituted in the differential equation;
- B is a positive latency block. In this case, the substituted function should be given by an external mechanism, which requires the following definition.

Let $\mathsf{Lat}(\mathcal{M})$ be the set of positive latency blocks of $\mathcal{M}$.

Definition 51 (History)**.** *A* history *of $\mathcal{M}$ is a mapping associating a function $h^{(B,o)}$ with each output o of a block B of $\mathsf{Lat}(\mathcal{M})$.*

We can now define the differential equation system obtained once a mode and a history are fixed.

Definition 52 (*Differential equations of a model*)

Let m be a mode, and h an history. The differential equation associated with an integration block B, for which the variable is denoted by x_B, is obtained by associating with each output o of each block B' of $\mathcal{G}_{B^-}$ an expression $y_{B',o}$ defined inductively over the blocks of $\mathcal{G}_{B^-}$ following the reverse topological order:

- *if B' is a block without input, then $y_{B',o} = \mathrm{op}_{B',o}()$;*

- *if B' is an integration block, then $y_{B',o} = x_{B'}$;*
- *if B' is a positive latency block, then $y_{B',o} = h^{(B',o)}$;*
- *if B' is an immediate block with inputs $(B_1, o_1), \ldots, (B_m, o_m)$ then $y_{B',o} = \mathrm{op}_{B',o}(y_{B_1,o_1}, \ldots, y_{B_m,o_m})$.*

The differential equation is then defined by $\dot{x_B} = y_{B^-,o^-}$.

Example 30.

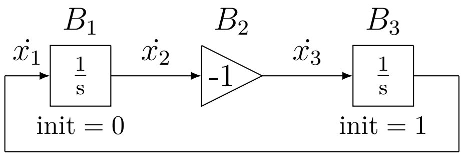


Figure 3.15: A model for a sinusoidal function

For the model depicted in Figure 3.15, the equations are $\dot{x_1} = x_3$, $\dot{x_3} = -x_1$, which lead to the characteristic equation of a sinusoidal function $\ddot{x_1} = -x_1$.

Example 31.

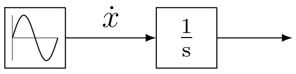


Figure 3.16: Another model for a sinusoidal function

In the model depicted in Figure 3.16, the equation is $\dot{x}(t) = A\sin(\frac{t}{\mathsf{f}} + K) + b$. Observe that here the time t appears explicitely.

Example 32. Recall the *SK*-model of Figure 3.11, for which the necessary backward graphs were given in Figure 3.14. For this model, the equations are $\dot{z} = h^{(3)}$, the output of block B_3, and either $\dot{y} = 1$ if *mode* $= \{] - \infty, 0[\}$ or $\dot{y} = z$ if *mode* $= \{[0, +\infty[\}$.

3.2.2 Interval Partitioning

Several factors may impact the partition of the simulation interval $\mathbb{T} = [t_{\text{init}}, t_{\text{end}}]$:

- Sampling blocks modify their output instantly at each sampling step, hence creating a discontinuity which requires splitting the simulation interval at their sampling times;
- Positive latency blocks signals must be known during subinterval where differential equations are fixed. Hence the length of a subinterval cannot be greater than the minimal positive latency.

- Signals on critical input of threshold blocks should satisfy the mode specified by the differential equation to solve. A new mode corresponds to a new splitting point of the simulation interval.

We introduce the *static next step* function next_s in order to deal with the two first factors, as well as $\delta_{\max}$, a bound on the length of the subintervals, which corresponds to the delay specified by an *SK*-model. Given t the lower bound of a subinterval, $\mathrm{next}_s(t)$ gives the (maximal) upper bound of the subinterval that satisfies the first two conditions above.

Definition 53 (*Static Next Step*)

Let $\mathcal{M}$ be a SK-model, let Δ be the set of its sampling steps and let R be the set of its positive latencies. We set $\delta_{\mathrm{lat}} = \min(\delta_{\max}, \min(R))$, and we define the static next step *value $\mathrm{next}_s(t)$ for time $t < t_{\mathrm{end}}$ by:*

$$\begin{array}{rcl} \delta_{\mathrm{samp}}(t) & = & \min\{p\delta \mid \delta \in \Delta, p \in \mathbb{N}, p\delta > t\} \\ \mathrm{next}_s(t) & = & \min(\delta_{\mathrm{samp}}(t), t + \delta_{\mathrm{lat}}, t_{\mathrm{end}}) \end{array}$$

When the interval changes, we need to reevaluate the output signals of some of the sampling blocks. This requires an order for the evaluation of blocks, which is also needed to evaluate signals which are not output of an Integrator block after differential equations have been solved.

Definition 54 (*Block Ordering*)

The block ordering BO *of a (correct) SK-model $\mathcal{M}$ is a total order which extends the following order on blocks:*

1. *blocks without input;*
2. *blocks with positive latency, by increasing latency;*
3. *blocks with infinitesimal latency;*
4. *immediate blocks in topological order.*

3.2.3 Trajectories

We now give detailed explanations about trajectories, beginning with conditions under which a trajectory exists for an *SK*-model.

Definition 55 (*Trajectory*)

The vector $\vec{w}$ is a trajectory of the SK-model $\mathcal{M}$ over $\mathbb{T} = [t_{\mathrm{init}}, t_{\mathrm{end}}]$ if there exists (i) a sequence $(t_i)_{i \in [\![0,N]\!]}$ with $t_0 = t_{\mathrm{init}}$ and $t_N = t_{\mathrm{end}}$ and for each i: $t_{i+1} \leq \min_{\tau > t_i} (\tau \mid \exists j \leq i, r \in R : \tau = t_j + r)$, (ii) a minimal sampling delay $\varepsilon_{\mathbb{T}} > 0$, which will be a lower bound on the length of sub-intervals, and (iii) a sequence $(m_i, h_i)_{i \in [\![0,N-1]\!]}$ of modes and histories, such that:

0. *The initial mode m_0 and history h_0 are those of the initial values of the model. Moreover, for $0 < i < N$, h_i on $[t_i, t_{i+1}[$ agrees with $\vec{w}$ on $[t_0, t_i[$.*

1. *For all $i < N$, the differential equation system $\dot{\vec{x}}(t) = F_{m_i,h_i}(t, \vec{x}(t))$ admits a solution on the interval $[t_i, t_{i+1} + \varepsilon_{\mathbb{T}}]$ which coincides with $\vec{w}$ on $[t_i, t_{i+1}[$ and is consistent with m_i on $]t_i, t_{i+1}[$. The vector $\vec{w}$ agrees with the operations of the remaining blocks.*

2. *For all $0 \leq i < N - 1$, $\vec{w}(t_{i+1})$ corresponds to the application to $\vec{w}(t^-_{i+1})$ of the operators of* active *discrete blocks (i.e. such that t_{i+1} is a sampling time). The solution of $\dot{\vec{x}}(t) = F_{m_i,h_{i+1}}(t, \vec{x}(t))$ on $[t_{i+1}, t_{i+1} + \varepsilon_{\mathbb{T}}]$ is consistent with m_{i+1}.*

In this definition, we use the sampling delay $\varepsilon_{\mathbb{T}}$ to deal with mode changes. Another approach, described in [21], uses non-standard analysis which implies handling infinitesimals.

Example 33. Returning to the example of Figure 3.11, the history is $h_1^{(3)}(t) = y(t-1) = t - 1$ on $[1, 2]$, and the mode $m_1^{(1)} =]0, +\infty[$ must agree with the value of z at time $1 + \varepsilon_{\mathbb{T}}$. Hence with $t_2 = 2$, the differential equations are $\dot{z}(t) = t - 1$ and $\dot{y} = z$, which yields $z(t) = (t-1)^2/2$ and $y(t) = (t-1)^3/6 + 1$.

In some cases, there is no trajectory:

Example 34. The differential equation $\dot{x} = 1 + x^2$ with $x(0) = 0$, easily represented with a Simulink model, has the unique maximal solution $\tan(x)$ over $]-\frac{\pi}{2}, \frac{\pi}{2}[$. Hence, there is no trajectory over any interval strictly containing $]-\frac{\pi}{2}, \frac{\pi}{2}[$.

Unicity of a trajectory. With suitable hypotheses on the operators, we prove that if a trajectory exists, it is unique. Unfortunately, as is often the case for hybrid systems, the existence of a trajectory is an undecidable problem.

This semantic looks like it is non-deterministic as it seems to depend on how the interval is splitted, the value of $\varepsilon_{\mathbb{T}}$ and the solutions of the succesive differential equations. However, with the hypothesis made on operators (which are piecewise smooth), we have:

Proposition 6

An SK-model has at most one trajectory.

The proof of this proposition is based on the following theorem ([26] theorem 1.8.2 (page 121)):

Theorem 7. *Let $f : (\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$ be a locally Lipschitz function with respect to its second variable and let I be an interval of $\mathbb{R}$. If there are two exact solutions φ_1*

and $\varphi_2 : I \to \mathbb{R}$ of the differential equation $\dot{x} = f(t, x)$, and if they are equal at a point $t_0 \in I$, then φ_1 and φ_2 are equal into the whole interval I.

A trajectory is defined by several differential equations, for which we need the local-Lipschitz condition to hold:

Lemma 8

Let $\vec{w}$ be a trajectory with its sequences $(t_i), (m_i)$ and (h_i), then for every $i \in [\![0, N-1]\!]$: F_{m_i,h_i} is locally Lipschitz with respect to the second variable.

Proof: From definition 52 we remark that all the intermediate functions (including the history functions) are C^∞ since they are defined inductively by solutions of previous differential equations, except in the initialisation where the history functions are constants and for blocks without input.

Then, the derivative of the solutions are bounded and F_{m_i,h_i} are locally Lipschitz with respect to the second variable. ■

Proof of proposition 6: Let us consider two trajectories $\vec{w}$ and $\vec{w}'$ of the *SK*-model, with $(t_i)_{i\in[\![0,N]\!]}$ and $(t'_i)_{i\in[\![0,N']\!]}$ the two sequences characterising the underlying time interval partitions and $\varepsilon_{\mathbb{T}}$, $\varepsilon'_{\mathbb{T}}$ the respective minimal sampling delays. We define $(u_i)_{i\in[\![0,M]\!]}$ as the time sequence obtained by merging these sequences with $u_0 = t_{\text{init}} = t_0 = t'_0$. Note that (u_i) still satisfies for each i: $u_{i+1} \leq \min_{\tau > u_i} (\tau \mid \exists j \leq i, r \in R : \tau = u_j + r)$. We denote by $(m_i, h_i)_{i\in[\![0,M-1]\!]}$ and $(m'_i, h'_i)_{i\in[\![0,M-1]\!]}$ the modes and histories over the intervals $[u_i, u_{i+1}[$ induced by the two original trajectories.

We prove by induction that, for all $i \in [\![0, M-1]\!]$, $m_i = m'_i$, $h'_i = h_i$, and $\vec{w}(u_i) = \vec{w}'(u_i)$.

Initialisation. By definition, as they are defined using the initial values of the model, we have $m_0 = m'_0$, $h'_0 = h_0$ and, $\vec{w}(u_0) = \vec{w}'(u_0)$.

Induction. Let $k \in [\![1, M-1]\!]$. Using the induction hypothesis, we have $m_{k-1} = m'_{k-1}$ and $h'_{k-1} = h_{k-1}$ and $\vec{w}(u_{k-1}) = \vec{w}'(u_{k-1})$. Thus, the differential equation $\dot{\vec{x}}(t) = F_{m_{k-1},h_{k-1}}(t, \vec{x})$ is the same for both trajectories over $[u_{k-1}, u_k[$, hence, using lemma 8 and theorem 7, those trajectories coincide over this interval. This implies that $h_k = h'_k$ and that the limits $\vec{w}(u_k^-) = \vec{w}'(u_k^-)$. The values $\vec{w}(u_k)$ and $\vec{w}'(u_k)$ are defined by application of the operator of active discrete blocks, hence they are equal. Let $\varepsilon = \min(\varepsilon_{\mathbb{T}}, \varepsilon'_{\mathbb{T}})$. Since the modes m_k and m'_k are determined using the solution of $\dot{\vec{x}} = F_{m_{k-1},h_k}(t, \vec{x})$ over $[u_{k-1}, u_k + \varepsilon]$, they are equal. Therefore, we have: $h_k = h'_k$, $m_k = m'_k$ and $\vec{w}(u_k) = \vec{w}(u_k)$.

We have shown that for every $i \in [\![0, M-1]\!]$, the trajectories are equal over $[u_i, u_{i+1}[$. The trajectories are thus equal over $[t_{\text{init}}, t_{\text{end}}[$. We also have $\vec{w}(u_M) = \vec{w}'(u_M)$ using the same argument as in the recursion part of the proof.

Thus, the two trajectories $\vec{w}$ and $\vec{w}'$ are equal on the whole interval $[t_{\text{init}}, t_{\text{end}}]$ and unicity holds. ■

Failure cases. There are several cases where a trajectory does not exist:

- as seen above, if the model contains a differential equation without solution over its time interval;
- if the solution violates the mode constraints related to some threshold block (see example 35);
- or if there are an infinite number of discontinuities (see the model Figure 3.18, and proposition 10).

Example 35.

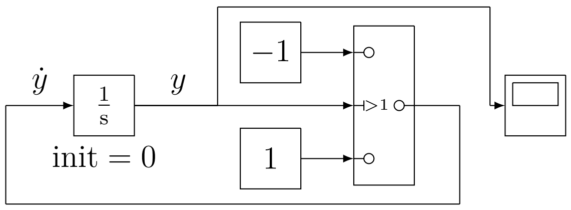


Figure 3.17: A model for which constraint at $t = 1$ cannot be solved

Let $\mathbb{T} = [0, 2]$; we have $t_0 = 0$. The mode at t_0 is $mode_0 = \{] - \infty, 1]\}$. The differential equation corresponding to that mode is $\dot{y} = 1$. We choose $t_1 = 1$ based on this differential equation and the condition. The solution $y(t) = t$ satisfies this mode for $]0, 1[$; we can easily deduce the output values of each blocks on $[0, 1[$. On $[1, 1 + \varepsilon_{\mathbb{T}}[$, $y(t) = t$ is still a solution of the differential equation $\dot{y}(t) = 1$. However, for any chosen value of t_2, this solution does not satisfy mode $\{] - \infty, 1]\}$ over $]1, t_2[$. The model semantics is fail over $[0, 2]$.

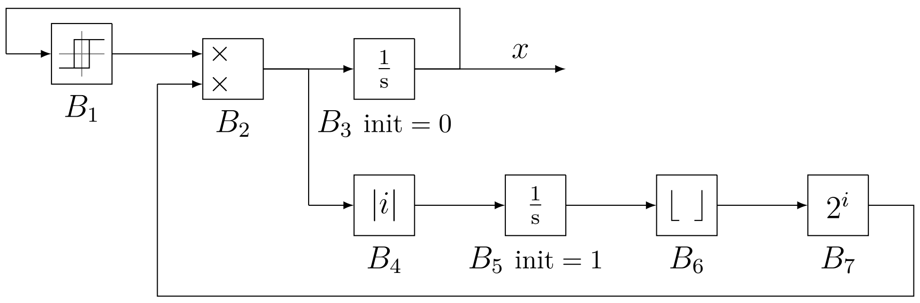


Figure 3.18: A model with an infinite number of discontinuities over $[0, 1]$

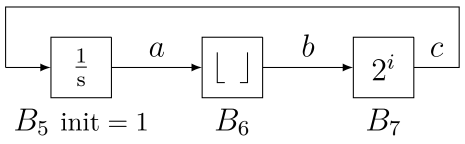


Figure 3.19: The (fast) increasing loop

Let us consider the Simulink model shown Figure 3.18 with, for block B_1, $s_{\mathsf{on}} = 1$, $s_{\mathsf{off}} = 0$, $v_{\mathsf{on}} = -1$, $v_{\mathsf{off}} = 1$.

Seeing as B_1 always output 1 or -1 and B_4 is the absolute value of the output of B_1 multiplied by B_7 (which is always positive), computing the value of B_7 requires to study the behaviour of the mode shown Figure 3.19.

Lemma 9

Let $\mathcal{M}$ be the model shown Figure 3.18, and let $N \in \mathbb{N}$. Let $t_k = 1 - \frac{1}{2^k}$ for all $k \in [0, N[$. Then,

$$\begin{array}{rcl} a(t) & = & (k+1) + 2^{k+1}(t - t_k) \\ b(t) & = & k+1 \\ c(t) & = & 2^{k+1} \end{array}$$

Proof: We proceed by induction. Let $\varepsilon_{\mathbb{T}} = \frac{1}{2^{N+2}}$

Initially (at $t_0 = t_{\text{init}} = 0$) we have $a(t_0) = 1$ and by direct implication $b(t_0) = 1$ and $c(t_0) = 2 = 2^1$. Over $[0, \frac{1}{2}[$ $a(t) = 1 + 2t$, $b(t) = 1$ and $c(t) = 1$. There is no crossing of the thresholds of the block B_6. However at $t_1 = \frac{1}{2}$, $a(t_1) = 2$ corresponds to some crossing of B_6. The next step of the solution stays consistent over $[\frac{1}{2}, \frac{1}{2} + \varepsilon_{\mathbb{T}}]$ with the mode of B_6. Also, this means that $a(t_1) = 2$, $b(t_1) = 2$, and $c(t_1) = 2^2$.

We assume the proposition true for $k \in [0, N-1[$. In that case, $a(t_{k+1}) = k+2$, $b(t_{k+1}) = k+2$ and $c(t_{k+1}) = 2^{k+2}$. For any $t \in [t_{k+1}, t_{k+2}[$ we have $a(t) = k + 2 + 2^{k+2}(t - t_{k+1})$, which implies $b(t) = k+2$ and $c(t) = 2^{k+2}$. There is no crossing over this interval of the threshold of the block B_6. At t_{k+2}, $a(t_{k+2}) = k+3$ and we have directly $b(t_{k+2}) = k+3$ and $c(t_{k+2}) = 2^{k+3}$. Similarly, the next step of the solution stays consistent over $[t_{k+2}, t_{k+2} + \varepsilon_{\mathbb{T}}]$. Hence, the proposition is true for $k+1$. ■

Finally, the input of B_5 is the absolute value of the input of block B_3. However, the input of B_3 changes signs whenever B_3 reaches 1 (with increasing values) or 0 (with decreasing values). Hence,

Proposition 10

For all $k \in \mathbb{N}$: over $[t_k, t_{k+1}[$:

$$x(t) = (k \mod 2) + (1 - 2(k \mod 2))2^{k+1}(t - t_k)$$

Note that there is an infinite number of discontinuities from the threshold crossings of blocks B_6 and B_1, hence the model does not have an exact trajectory over $[0, 1]$. A trajectory would start as shown on Figure 3.20 if it was calculated over $[0, v]$ with $v \in [0, 1[$.

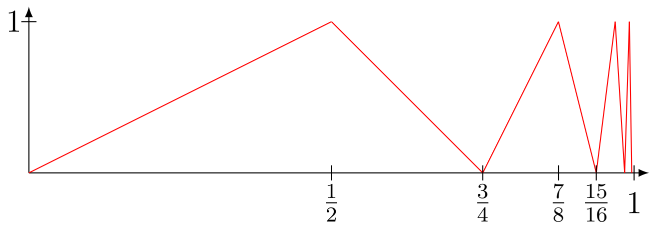


Figure 3.20: The start of a unfinished trajectory of model Figure 3.18

We used a discrete controller operating in a continuous environment to define the exact semantics of SK-models. This controller is limited by its reaction capacities, that are implicitly specified by $\varepsilon_{\mathbb{T}}$. Hence, when going from an interval of the simulation to another, the controller needs to keep a mode to decide whether it applies to the next differential equation.

It is well-known that questions on hybrid systems trajectories are usually undecidable (*e.g.* threshold crossing). Here, even the problem of existence of a trajectory over a finite interval is undecidable.

Proposition 11

Trajectory existence problem for SK-models is undecidable.

Proof: The proof is a reduction of the halting problem for two-counter machines (which is undecidable [56]) to the trajectory existence problem for SK-models.

Recall that a two-counter machine consists of a finite sequence of labelled instructions, which handle two counters c and d and end with a special instruction with label Halt. The other instructions have one of the two forms below with $x \in \{c, d\}$ representing one of the two counters:

1. $\ell : x := x + 1; \mathsf{goto}\ \ell'$
2. $\ell : \mathsf{if}\ x = 0\ \mathsf{then\ goto}\ \ell'\ \mathsf{else}\ x := x - 1\ \mathsf{goto}\ \ell''$

Since a (long) sequence instructions of the first type can set the counters at any initial values, we assume that the counters have initial value zero. The behaviour of such a machine is described by a (possibly infinite) sequence of configurations: $< \ell_0, 0, 0 >< \ell_1, c_1, d_1 > \cdots < \ell_i, c_i, d_i > \cdots$ where c_i and d_i are the respective values of counters c and d, and ℓ_i the label of the current instruction after the i^{th} instruction.

The reduction is done by encoding a two-counter machine as a Stateflow Chart, which will be activated by the signal x of the model shown in Figure 3.18. Then, by adding an output to this Chart which is connected to the enabler model, we ensure that the final model has a trajectory if, and only if, the two-counter machine halts.

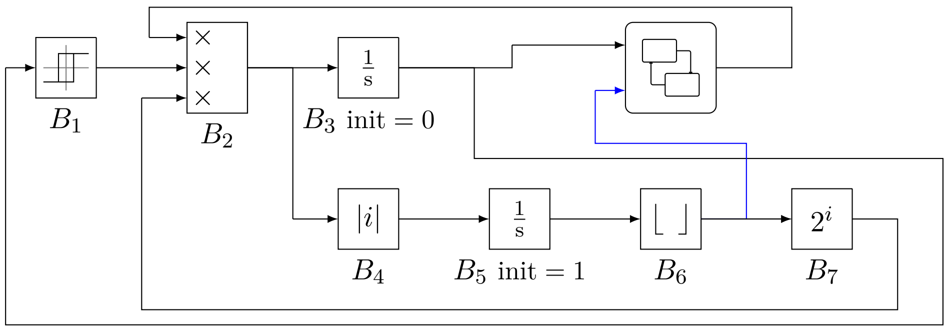


Figure 3.21: The resulting model

The model built by this operation is depicted in Figure 3.21. The output of block B_6 is also connected to the Chart to ensure that only one step of the two-counter machine is computed at each activation.

The Stateflow Chart is constructed as follows:

- for each instruction label ℓ, the Chart has a corresponding state $\mathsf{state}(\ell)$;
- for each counter c and d, a variable named similarly is chosen;
- two additional variables are added: s indicating whether the Chart reached the Halt instruction (which is initialised to 1 and is an output of the Chart) and n a variable that will be used to prevent the Chart from doing more than one step at each activation;
- two inputs, e being the event input (which is the output of block B_3 above) and m the output of the B_6 block which will be used to check the current iteration;
- for each instruction $\ell : x := x + 1\,\mathsf{goto}\,\ell'$ an arc $\mathsf{state}(\ell) \xrightarrow{m>n,\{x\leftarrow x+1; n\leftarrow m\}} \mathsf{state}(\ell')$;
- for each instruction $\ell : \mathsf{if}\, x = 0\,\mathsf{then\,goto}\,\ell'\,\mathsf{else}\, x := x - 1\,\mathsf{goto}\,\ell''$, two arcs $\mathsf{state}(\ell) \xrightarrow{m>n\wedge x=0, n\leftarrow m} \mathsf{state}(\ell')$ and $\mathsf{state}(\ell) \xrightarrow{m>n,\{x\leftarrow x-1; n\leftarrow m\}} \mathsf{state}(\ell'')$;
- for each arc leading into the state $\mathsf{state}(\mathsf{Halt})$, we append $s \leftarrow 0$ to its list of instructions.

Now, it has been seen (proposition 10) that the block B_3 generates a peak at each $1 - \frac{1}{2^(2n+1)}, n \in \mathbb{N}$. This sequence has an infinity of values in $[0, 1]$ meaning that the Stateflow Chart will be activated as many times as possible. When the

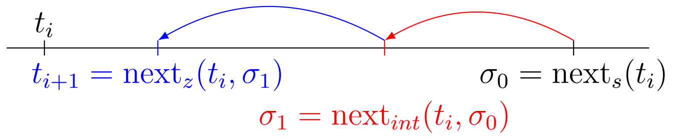

t_i
$t_{i+1} = \text{next}_z(t_i, \sigma_1)$
$\sigma_0 = \text{next}_s(t_i)$
$\sigma_1 = \text{next}_{int}(t_i, \sigma_0)$

Figure 3.22: Searching for the next step t_{i+1}

Chart reaches the state(Halt) state, the output B_2 is forced to 0 and this stops the output of B_3 from changing further. The model Figure 3.21 has a trajectory if, and only if, the Chart halts, which is if and only if the two-counter machine halts. Thus, the trajectory existence problem for *SK*-models is undecidable. ■

Observe that this proof uses discrete asynchronous blocks.

3.3 Operational semantics

The exact semantics relies on solving differential equations and determining threshold crossings. It is well-known that such operations are not effective. We now define an *approximate semantics* that will be the base of a simulation engine for *SK*-models. Moreover, it should be possible to give guarantees on the approximation towards exact semantics.

General principle. The approximate semantics is based on the iterative construction of a subinterval partition of $[t_{\text{init}}, t_{\text{end}}] = \bigcup_{i=0}^{N-1}[t_i, t_{i+1}]$. In order to control errors induced by approximations, we again define $\varepsilon_{\mathbb{T}}$ as the *minimal stepsize* and add an accuracy parameter ε_V satisfying: $|x| < \varepsilon_V \Rightarrow x \simeq 0$. The approximate semantics will replace complete trajectories by an array mapping, for each output o of each block B, its values $W_{B,o}[i]$ at each step t_i, $0 \leq i \leq N$.

Iteration step. Let $t_0, \dots, t_i$ be known, as well as the values $W_{B,o}[j]$ for each output o of each block B, and $0 \leq j \leq i$. The first step is to determine t_{i+1}. This value is obtained by applying the next step function next_s: $\sigma_0 = \text{next}_s(t_i)$, and then possibly lowering this value:

1. First, by applying a variable-step integration method, such as Runge-Kutta-Felhberg[1] [39], between t_i and σ_0 over all Integration blocks, giving $\sigma_1 = \text{next}_{int}(t_i, \sigma_0)$;
2. Second, by detecting the mode change of threshold blocks, by linear approximations, producing $t_{i+1} = \text{next}_z(t_i, \sigma_1)$.

In both cases, it is necessary to compute the values corresponding to backwards graphs of concerned blocks at intermediate points generated by these methods.

[1]also called ODE45

Variable-step Integration in *SK*-models. The chosen method, ODE45, is based on the simultaneous computation of two approximations by fourth and fifth-order Runge-Kutta methods, as described in [39]. If the difference between the two approximations is above ε_V, then the stepsize is reduced with a minimum value of $\varepsilon_{\mathbb{T}}$ and the procedure repeated.

However, in order to use Runge-Kutta methods, we need to explicitly know differential equations, while here they are implicitly specified by the *SK*-model. This requires to adapt these methods, evaluating backward graph blocks to the intermediate times of the Runge-Kutta method.

Example 36. In order to give a general idea, we show an application of the classical fourth-order Runge-Kutta method on a simple Simulink model. This can be adapted to any other Runge-Kutta method.

Classical fourth-order Runge-Kutta method. We consider the differential equation $\dot{y} = f(t, y)$ with initial value $y_0 = y(t_0)$. The classical RK4 method of stepsize h iteratively computes the values $y_n = y(t_0 + nh)$ values:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where $\begin{cases} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\ k_3 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\ k_4 &= f(t_n + h, y_n + hk_3) \end{cases}$

At each step, the output signal of an Integration block is estimated using initial value and an elapsed time multiplied by the derivative. To simplify notations, we will symbolically denote $k_j = f(\tau_j, z_j)$ the equation corresponding to step j.
Note that this example is simplified as the block outputs never directly depends on the current time (except for integrals) and there is no delay functions nor threshold block.
In a Simulink model, the function represented by f results from the computation of the backward graph of the input block of an Integrator block. We consider the Simulink model of Figure 3.23 as an example. Initial values of B_1 and B_3 are respectively 1 and 0. We will simulate the model over interval $[0, 10]$ with a maximal stepsize 0.5. The exact semantics of output signals are $\cos(t)$ for B_1, and $-\cos(t)$ for B_2 and $-\sin(t)$ for B_3.

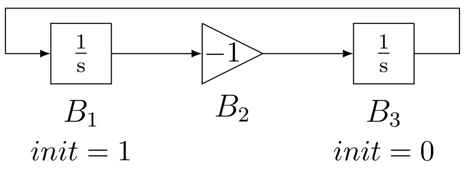
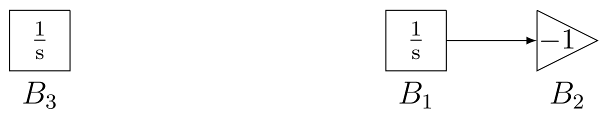


Figure 3.23: A Simulink model computing an approximation of $\cos(t)$

We compute the backward graphs for blocks B_3 and B_2 (inputs of B_1 and B_3), that are respectively shown in figures 3.24(a) and 3.24(b). We denote $k_{j,i}$ the coefficient k_j associated to block B_i. Similar notations will be used for f and other blocks.



(a): Backward graph of B_3, input of B_1 (b): Backward graph of B_2, input of B_3

Figure 3.24: Backward graphs of the Figure 3.23 Simulink model

Using those backwards graphs, we can unwrap the definition of equations $k_{j,i} = f(\tau_j, z_{j,i})$. We then have $k_{j,1} = z_{j,3}$ for block B_1 and $k_{j,3} = z_{j,1}$ for block B_3. We can then apply the RK4 method to the Simulink model.
We then describe the calculations for $t_0 = 0$ and $h = \frac{1}{2}$. For the first step, we have $k_{1,1} = f(t_0, z_{1,1}) = W_{3,0} = 0$. Likewise, $k_{1,3} = f(t_0, z_{1,3}) = -W_{1,0} = -1$.
The other steps are described in the following table:

1	$k_{1,1} = f_1(0, y_{0,3}) = y_{0,3} = 0$	$k_{1,3} = f_3(0, y_{0,1}) = -y_{0,1} = -1$
2	$k_{2,1} = f_1(\frac{1}{4}, 0 + \frac{1}{4}k_{1,3}) = \frac{1}{4}k_{1,3} = \frac{-1}{4}$	$k_{2,3} = f_3(\frac{1}{4}, 1 + \frac{1}{4}k_{1,1}) = -1 - \frac{1}{4}k_{1,1} = -1$
3	$k_{3,1} = f_1(\frac{1}{4}, 0 + \frac{1}{4}k_{2,3}) = 0 + \frac{1}{4}k_{2,3} = \frac{-1}{4}$	$k_{3,3} = f_3(\frac{1}{4}, 1 + \frac{1}{4}k_{2,1}) = -1 - \frac{1}{4}k_{2,1} = \frac{-15}{16}$
4	$k_{4,1} = f_1(\frac{1}{2}, 0 + \frac{1}{2}k_{3,3}) = \frac{1}{2}k_{3,3} = \frac{-15}{32}$	$k_{4,3} = f_3(\frac{1}{2}, 1 + \frac{1}{2}k_{3,1}) = -1 - \frac{1}{2}k_{3,1} = \frac{-7}{8}$
-	$y_{1,1} = y_{0,1} + \frac{1}{12}(k_{1,1} + 2k_{2,1} + 2k_{3,1} + k_{4,1})$ $= 1 + \frac{1}{12}(0 + \frac{-1}{2} + \frac{-1}{2} + \frac{-15}{32})$ $= \frac{337}{384} \simeq 0,877$	$y_{1,3} = y_{0,3} + \frac{1}{12}(k_{1,3} + 2k_{2,3} + 2k_{3,3} + k_{4,3})$ $= 0 + \frac{1}{12}(-1 - 2 - \frac{15}{8} - \frac{7}{8})$ $= \frac{-23}{48} \simeq -0,479$

Figure 3.25: Applying RK4 method to Simulink model of Figure 3.23

Final values are approximations of $\cos(0.5)$ and $-\sin(0.5)$ exactly to 10^{-3}.

Remark. *It is possible that a threshold block changes* mode *between two intermediate steps of the Runge-Kutta methods. As we tackle the threshold crossing immediately after, we apply Runge-Kutta method with fixed modes (the mode of time t_i).*

Threshold crossing. Let $\sigma_1 = \text{next}_{int}(t_i, \sigma_0)$. We determine if there is a mode change between t_i and σ_1. In this case, we compute the minimal instant of mode

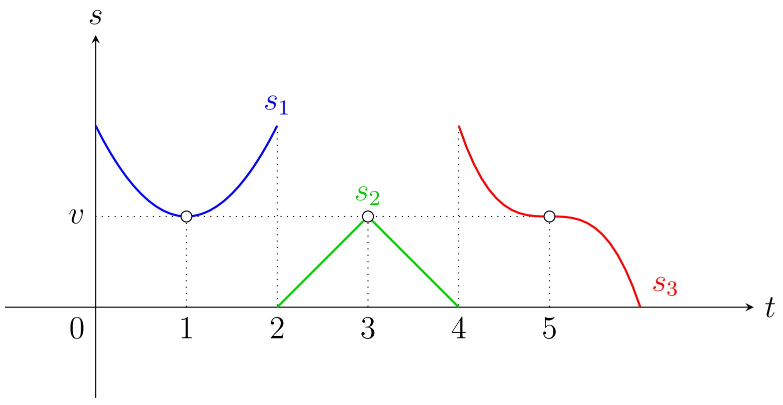


Figure 3.26: Unsafe crossings of value v

change between all threshold blocks, by doing a linear interpolation over all critical inputs. This linear interpolation is evaluated with a maximal precision given by ε_V.
Computing new values. The following step $t_{i+1} = \text{next}_z(t_i, \text{next}_{int}(t_i, \text{next}_s(t_i)))$ given by the procedures described above, we compute output of all blocks at this given time in the block order. For Integration blocks, as the ODE45 method has already given a guarantee over precision, computing their new value with RK4 is good enough. For other blocks, we apply operators with respect to the current mode.

Then, similarly to the exact semantics, we compute the new values at $t_{i+1} + \varepsilon_{\mathbb{T}}$ to deduce the mode at t_{i+1}. If a mode change occurs in $]t_{i+1}, t_{i+1} + \varepsilon_{\mathbb{T}}[$ then the result is fail.
Approximation conjecture. This operational semantics is proposed with the aim to provide an accurate approximation of the exact semantics. This is formally expressed as follows:

Definition 56 (*Approximation*)

Let $\vec{w}$ be the trajectory of an SK-model $\mathcal{M}$. Then, the set of values W produced by the operational semantics accurately approximates *$\vec{w}$ if, for any ε, there exist values of $\varepsilon_{\mathbb{T}}$ and ε_V such that:*

$$\forall t_i, \forall k, \forall o, |W_{k,o}[i] - w_{k,o}(t_i)| < \varepsilon.$$

Such a result would require reasonable hypotheses on the model. We now propose a semantical condition on signals, called *Safe Crossing*, under which we conjecture that the result holds:

Definition 57 (*Safe Crossing*)

An SK-model $\mathcal{M}$ satisfies the Safe Crossing condition *if there exists d_{min} and*

$\alpha > 0$ such that, for any output signal s of $\mathcal{M}$, for any threshold value v, if $s(t_c) = v$ for some $t_c \in \mathbb{T}$, then:

1. *s is continuous over the interval $[t_c - \alpha, t_c + \alpha]$;*
2. *For any $\beta \leq \alpha$, $(s(t_c - \beta) - v)(s(t_c + \beta) - v) < 0$ (the threshold value is effectively crossed);*
3. *$|\frac{ds}{dt}(t_c^+)| \geq d_{min}$ and $|\frac{ds}{dt}(t_c^-)| \geq d_{min}$ (the right and left derivatives of the signal on the crossing are above the minimal value).*

This condition precludes classical problematic situations like those illustrated in Figure 3.26, where signals s_1, s_2 and s_3 have an unsafe crossing of value v for $t_c = 1$, 3 and 5 respectively.

We conclude this chapter with the conjecture:

Conjecture 12

Under the Safe Crossing condition, if a trajectory $\vec{w}$ exists, then the operational semantics is an accurate approximation of $\vec{w}$.

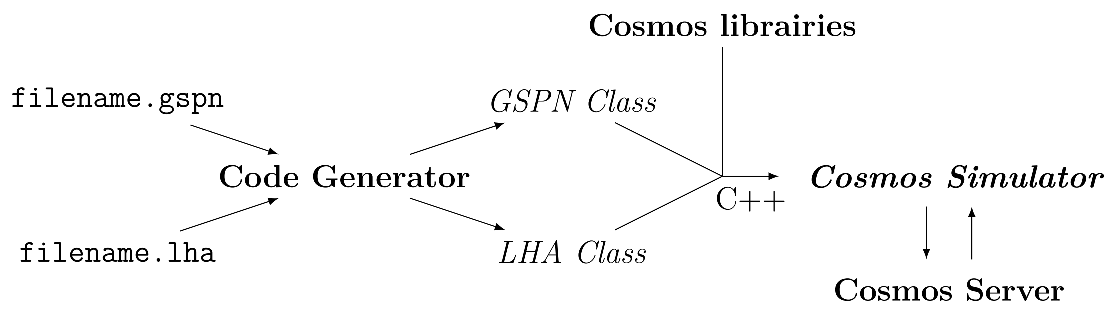
CHAPTER 4

EXTENSIONS TO COSMOS

Abstract. Cosmos is a statistical model-checker for the Hybrid Automata Stochastic Logic over high-level stochastic Petri nets. It has been developed first during Hilal Djafri's PhD [34]. It was improved, adding HASL support in [13], and rare event handling in Benoît Barbot's PhD [14]. In the first section, we describe the former version of Cosmos. Then, we present our contributions: an improvement of the implementation for high-level Petri nets concerning token handling, a simulator for Simulink models, and support for multi-model simulation.

Contents

4.1 Technical Description of Cosmos



bold	Pre-compiled files	`tt`	Input files
it	Generated C++ code	***boldit***	Generated executable

Figure 4.1: Overview of the architecture of Cosmos

Overview. Cosmos is built around three main parts:

- a *Code Generator*, which transforms the model files and the description of the HASL formula into C++ code that will be included in the Simulator;
- the *Simulator*, which is built using a library (that we describe shortly after) and the C++ code generated by the Code Generator;
- the *Server*, which runs in parallel several instances of the Simulator then aggregates their results and computes the statistical evaluation, based on the chosen statistical method.

General notations. In this chapter, code listings are used to present the data structures and the class templates. Functions will be presented using readable pseudo-code.

Code optimization. In the actual code, several optimization techniques are implemented, some of them using C++ templates: in particular the *curiously recurring template pattern*[1] which implements static polymorphism.

We will not describe in detail the use of templates in the code, but will present the functional aspects of the classes.

For instance, we present the two classes `SPN` and `SKModel` as fulfilling the `DEDS` interface which communicates with the Simulator, as shown in Figure 4.2.

[1]`https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern`

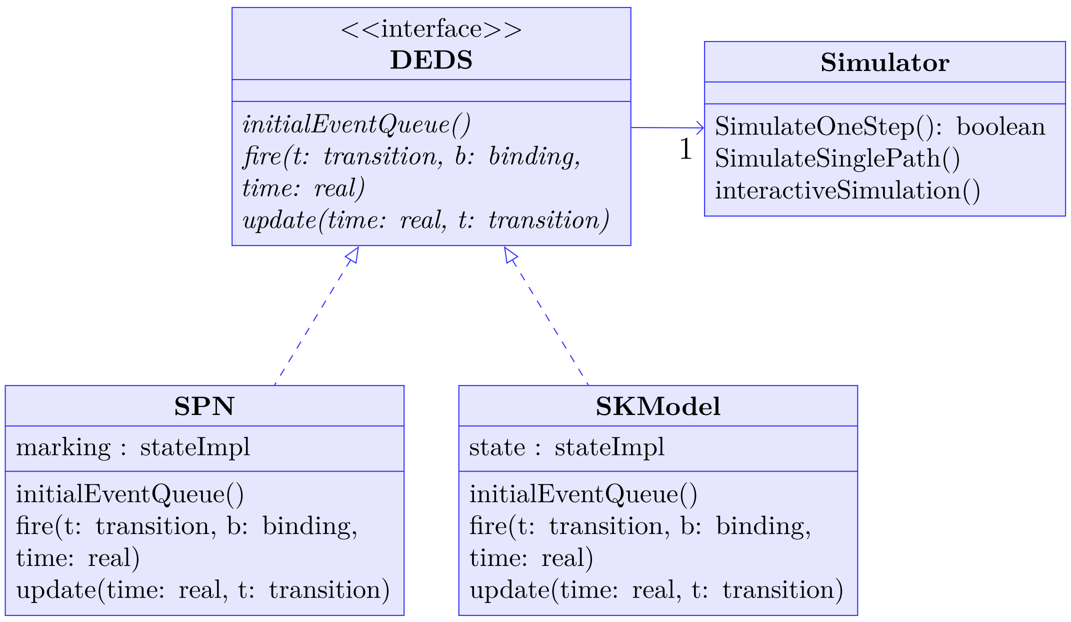

<<interface>>
DEDS
initialEventQueue()
fire(t: transition, b: binding, time: real)
update(time: real, t: transition)
1
Simulator
SimulateOneStep(): boolean
SimulateSinglePath()
interactiveSimulation()
SPN
marking : stateImpl
initialEventQueue()
fire(t: transition, b: binding, time: real)
update(time: real, t: transition)
SKModel
state : stateImpl
initialEventQueue()
fire(t: transition, b: binding, time: real)
update(time: real, t: transition)

Figure 4.2: Relation of `DEDS` classes and the Simulator

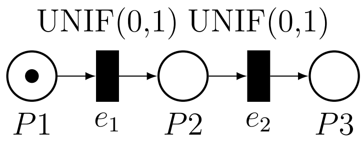
4.1.1 Code Generator

In this section, we first present the `Marking` class used to store the current state of a model. Then, we describe the `DEDS` class that will be instantiated on code generation. We then show the version of this class used in the library for Petri nets, reducing the number of elements generated by the *Code Generator*. Usually, the generalized stochastic Petri nets are given to Cosmos in the `.grml` format, that can be produced using the GreatSPN Editor[2].

`DEDS` class. As shown in algorithms 2 and 3, the Event Queue management is handled by a class corresponding to the `DEDS` interface (that we shorten to `DEDS` class). To this aim, two functions will be generated by the Code Generator: `initialEventQueue` which initialises the event queue at simulation start, and `update` which updates the event queue after the firing of a transition. This generic structure is currently instantiated for Petri nets and Simulink models.

The `DEDS` class handles the current state, stored into `DEDState` which contains, in the case of Petri nets, an instance of the `Marking` class. This state is updated by the `fire` function according to the specification of the transition. The initial value of the state is filled through the constructor of the class.

[2]`http://www.di.unito.it/~amparore/mc4cslta/editor.html`



(a) A Petri net

```
class abstractMarkingImpl {
public:
    int _PL_P1;
    int _PL_P2;
    int _PL_P3;
};
```

(b) Its Marking class

Figure 4.3: A Marking structure example

`Marking` class. It is used to represent a model state. For Petri nets, it contains as many variables as there are places. The `Marking` class in Figure 4.3(b) corresponds to the net in Figure 4.3(a), which has been used previously to estimate π using the Monte-Carlo method in section 18. The `abstractMarking` class is an instance for the `DEDState` used in class templates. The `abstractMarkingImpl` class is used to define model-dependant variables and circumvent C++ limitations.

Petri Net models. In the case of a Petri Net, the `update` and `initialEventQueue` functions are implemented as a part of the library, through `SPNBase.cpp`:

- the `initialEventQueue` iterates over all transitions, checking whether each transition is enabled and, in that case, adding it to the event queue with a random delay corresponding to the specified distribution (using *generateEvent*).
- the `update` function, called after the firing of transition T, shown in Algorithm 1, updates the Event Queue with an incremental method:
  - first, it checks whether transition T is still enabled, and removes or updates it accordingly;
  - then, it checks each possibly enabled transition, and add new possible events. It uses the `PossiblyEnabled` vector, generated by the Code Generator, that maps each transition t to the list of transitions t' for which there exists a place p with $t \to p \to t'$;
  - then, it checks each possibly disabled transition and remove events that have been disabled. It uses the `PossiblyDisabled` vector that maps each transition t to the list of transitions t' for which there exists a place p with $p \to t$ and $p \to t'$ and $t \neq t'$.
  - finally, each transition for which the time distribution depends on the value of the current marking (stored into the `MarkingDependant` vector) is checked and updated accordingly.

Example 37.

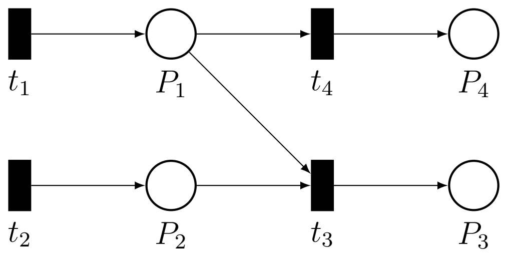


Figure 4.4: A toy Petri net

In the net shown in Figure 4.4, we have:
- `PossiblyEnabled`$(t_1) = \{t_3, t_4\}$;
- `PossiblyEnabled`$(t_2) = \{t_3\}$;
- `PossiblyEnabled`(t_3) = `PossiblyEnabled`$(t_4) = \emptyset$;
- `PossiblyDisabled`(t_1) = `PossiblyDisabled`$(t_2) = \emptyset$;
- `PossiblyDisabled`$(t_3) = \{t_4\}$;
- and `PossiblyDisabled`$(t_4) = \{t_3\}$.

In the case if t_2 is fired, `PossiblyEnabled`(t_2) is checked and tells that we need to check if t_3 is enabled. Then, we check the number of tokens of places P_1 and P_2. If there is at least one token in each of these places, we know that t_3 is enabled. We then check if this transition is already in the Event Queue. If not, this transition is added to the Event Queue (the time and weight are drawn through `generateEvent`).

There is a more clever way to perform this update function with a better incremental update where a value is associated with each transition giving the number of conditions that are not yet satisfied. However, simulations using this approach are only faster for ordinary Petri nets. The advantage of the chosen algorithm here is its flexibility and, in particular, its adaptability to high-level Petri nets.

Generating a model class. The generalized stochastic Petri nets are given to Cosmos in the `.grml` format, and then transformed into a C++ code that will have the behavior of the net. For example, the Petri net shown in Figure 4.4 would have the `fire` function shown in Figure 4.5. For each transition t, this function applies the token changes for the impacted places. Other model functions are generated in a similar way: `IsEnabled` would return the result of the test associated to the places of the transition (in this case, for t_3, it would be `Marking.P->_PL_P1 > 0 && Marking.P->_PL_P2 > 0`).

Algorithm 1: Function updating the Event Queue for a Petri Net

```
Data: the Event Queue $EQ$, a Time Generator $TG$
1 Function update($ctime$,$lastTr$,$EQ$,$TG$)
      Input : the current time $ctime$, the last activated transition $lastTr$,
              the Event Queue $EQ$ and Time Generator $TG$
2     for $t \in PossiblyEnabled(lastTr)$ do
3         if $IsEnabled(t)$) then
4             if $\neg EQ.isScheduled(t)$ then
5                 $generateEvent(F)$; $EQ.insert(F)$
6     for $t \in PossiblyDisabled(lastTr)$ do
7         if $\neg IsEnabled(t)$ then $EQ.remove(t)$;
8     for $t \in MarkingDependant$ do
9         if $EQ.isScheduled(t)$ then $generateEvent(F)$; $EQ.replace(F)$ ;
```

```cpp
void SPN::fire(TR_PL_ID t, const abstractBinding &b,REAL_TYPE time
   ){
    switch (t){
        case 0: //t1
            Marking.P->_PL_P1 += 1;
            break;
        case 1: //t2
            Marking.P->_PL_P2 += 1;
            break;
        case 2: //t3
            Marking.P->_PL_P1 -= 1;
            Marking.P->_PL_P2 -= 1;
            Marking.P->_PL_P3 += 1;
            break;
        case 3: //t4
            Marking.P->_PL_P1 -= 1;
            Marking.P->_PL_P4 += 1;
    }
}
```

Figure 4.5: The `fire` function of the toy Petri net

```
class LHA {
public:
    int CurrentLocation;
    double CurrentTime;

    vector<double> FormulaVal;
    vector<bool> FormulaValQual;

protected:
    vector <LhaEdge> Edge;
    vector <int> EdgeCounter;
    set <int> InitLoc;
    vector<bool> FinalLoc;
    vector < set <int> > Out_A_Edges;

    Variables *Vars;
    Variables *tempVars;
    vector<double> LinForm;
    vector<double> OldLinForm;
    vector<double> LhaFunc;
    vector<double> LhaFuncDefaults;

    void resetVariables();
    void DoElapsedTimeUpdate(double, const DEDState&);
    double GetFlow(int, const DEDState&)const;
    bool CheckLocation(int,const DEDState&)const;
    bool CheckEdgeContraints(int,size_t, const abstractBinding&,
       const DEDState&)const;
    t_interval GetEdgeEnablingTime(int,const DEDState&)const;
    void DoEdgeUpdates(int, const DEDState&, const abstractBinding
       &);
    void UpdateLinForm(const DEDState&);
    void UpdateLhaFunc( double&);
    void UpdateFormulaVal(const DEDState&);
};
```

Figure 4.6: A simplified version of `LHA.hpp`

```
template<class DEDState>
class LHA_orig : public LHA<DEDState>  {
public:
    void copyState(LHA_orig*); // Copy state of another LHA
    void fireAutonomous(int,const DEDState&);
    virtual int synchroniseWith(size_t, const DEDState&,const
       abstractBinding&);
    virtual AutEdge GetEnabled_A_Edges(const DEDState&);
    virtual void updateLHA(double DeltaT, const DEDState &);
    virtual bool isFinal();
    virtual void reset(const DEDState&);
    virtual void getFinalValues(const DEDState&,vector<double>&,
       vector<bool>&);

protected:
    void fireLHA(int,const DEDState&, const abstractBinding&);
    virtual void setInitLocation(const DEDState&);
    int GetEnabled_S_Edges(size_t, const DEDState&,const
       abstractBinding&);
    void resetLinForms();
};
```

Figure 4.7: A simplified version of `LHA_orig.hpp`

HASL and LHA class. The LHA is also defined as a class (see Figure 4.6), generated at runtime. It maintains informations about the global state of the automaton, among them:

- `CurrentLocation` and `CurrentTime` are respectively an integer representing the current node of the automaton, and a double representing the current time;
- `FormulaVal` is a vector of doubles containing the current value of each path expression;
- `FormulaValQual` is a vector of booleans indicating if a path expression, used for binomial distribution evaluation, is currently satisfied.

It also contains a set of protected functions, which will be used by the class LHA_orig (described in the next paragraph) that will serve as an interface for the Simulator. Among them:

- `DoElapsedTimeUpdate` updates the variables of the LHA according to the given elapsed time and the current marking of the model;
- `GetEdgeEnablingTime` gets the earliest time at which an edge of the LHA will be enabled (when its guard becomes true), by solving a linear equation system;
- `DoEdgeUpdates` updates the LHA variables after the firing of an edge of the LHA, according to the current marking of the model.

Complete LHA class. The final class for the Simulator is another class, called LHA_orig (see Figure 4.7), that inherits from the LHA class described in the previous paragraph. It contains mainly public functions:

- `fireAutonomous` firing an autonomous transition of the LHA;
- `updateLHA` which updates all the values of the LHA and HASL path expressions according to an elapsed time whose value is indicated by `DeltaT` and to the current state of the model;
- `synchroniseWith` checks if there is an enabled synchronisation edge (with `GetEnabled_S_Edge`) and fires it, and returns whether there was such an edge.

Some of them rely on protected functions. Among them:

- `fireLHA` fires a transition of the LHA, by using the `DoEdgeUpdates` function to update the LHA variables according to the current marking of the model, then changes the current state.
- `GetEnabled_S_Edge` checks all LHA transitions starting from the current LHA state and which are labelled by the model transition; for each of these transitions, it then checks whether the guard of the destination and the edge are fulfilled, in which cases it returns the corresponding LHA transition.

4.1.2 Simulator

We describe here the `Simulator` class template which handles calls for model simulation and automaton synchronisation. The algorithms will then be presented, followed by the `Event` structure used in the `EventQueue` class.

`Simulator` class. The essential functions of the header file of the simulator are shown in Figure 4.8. As can be seen, the Simulator is a template that extends the `SimulatorBase` template. This `SimulatorBase` template has three generic types:

- the Simulator `S` itself;
- the structure `EQT` for an Event Queue;
- and the simulated model `DEDS`.

It implements several functions, the essential ones are:

- `SimulateOneStep` which fires autonomous edges of the LHA until the next activation time of the model, and then performs the corresponding step. It returns whether the model has a next simulation step.
- `SimulateSinglePath` which simulates the model until:
    1. a final state of the LHA is reached;
    2. a model transition cannot be synchronised with the LHA;
    3. the Event Queue is empty.
- `interactiveSimulation` which implements a command-line interface where the user can control the simulation by choosing to execute exactly one step, or execute the model for a chosen delay, or firing one of the enabled transitions.

```
template <class S, class EQT, class DEDS>
class SimulatorBase:public timeGen {
public:
    SimulatorBase(DEDS& N,LHA_orig<typename DEDS::stateType>&);
    ~SimulatorBase();
    void SetBatchSize(const size_t);

    DEDS &N;
    LHA_orig<typename DEDS::stateType> &A;
    EQT* EQ;

    bool SimulateOneStep();
    void SimulateSinglePath();
    void interactiveSimulation();

    void reset();
};

template <class EQT, class DEDS>
class Simulator:public SimulatorBase<Simulator<EQT, DEDS>, EQT,
   DEDS>{
public:
    Simulator(DEDS& deds,LHA_orig<typename DEDS::stateType>& lha):
       SimulatorBase<Simulator,EQT,DEDS>(deds, lha){};
};
```

Figure 4.8: A simplified version of `Simulator.hpp`

Note that unlike the model which needs a template as its structure might be quite generic, the `LHA_orig` class is fully instantiated.

Simulator algorithms. The `SimulateOneStep` function is described by Algorithm 2. This function first defines τ_{next} the time of the next event ($+\infty$ if there is no such event) and stores the next event of the model into a variable E. Then, while there is an autonomous edge of the LHA for which condition is fulfilled before τ_{next}, it fires it and updates the LHA variables accordingly. In the case where a final state is reached during this loop, the simulation is accepted and the function returns false as there are no more possible simulation steps. Once this loop is finished, it checks whether there is a possible event in the model. If there is none, the simulation is rejected and the function returns false. Finally, it updates the LHA variables for their new value at τ_{next} and fires the event E. If the transition of event E cannot be synchronised with the LHA, it rejects the simulation and returns false. Otherwise, the function performs the synchronisation and checks whether the LHA has reached a final state. In that case, it accepts the simulation and returns false. In the other case, it finally updates the Event Queue (using the model's `update` function) and returns true.

The `SimulateSinglePath` function is built directly using `SimulateOneStep`. It is described by Algorithm 3. It first calls the `initialEventQueue` of the model, initialising the Event Queue, then runs the `SimulateOneStep` function in a loop as long as the simulation can progress.

The procedure for interactive simulation is also based on `SimulateOneStep`, and is described by Algorithm 4. It is similar to the single path function, except that it maintains a variable *minInteractive* which contains the minimum time at which control is given back to the user. When the user has control, he may:

1. either fire a specific transition, which gives full priority to the event associated with this transition (executed at current time, with highest priority and lowest weight);
2. or execute a single step of the model;
3. or execute steps of the models until a given time (stored into *minInteractive*).

Other features are implemented into the interactive simulation, such as waiting for a specific transition to be fired, or drawing the current Petri net state (through Graphviz's DOT language[3]).

A verbose level can also be given when running Cosmos, which is useful for debugging. Depending on the verbose level, additional information will be provided about the current state of the model, the Event Queue, etc. This information will be printed by `SimulateOneStep`.

[3]see `https://www.graphviz.org/doc/info/lang.html`

Algorithm 2: The `SimulateOneStep` function

```
Data: the Event Queue $EQ$, the LHA $A$, and the model $N$
1  Function SimulateOneStep(EQ,A,N)
       Data: An (possibly empty) event $E$
       Output: a boolean that indicates whether the simulation can be
               continued
2      $E \leftarrow$ null; $\tau_{\text{next}} \leftarrow +\infty$ ;
3      if $EQ \neq \emptyset$ then  $E \leftarrow EQ.first$; $\tau_{\text{next}} \leftarrow E.time$ ;
       /* Execute all autonomous transitions that happen before
          $\tau_{\text{next}}$                                        */
4      while $AE = A.GetEnabled\_A\_Edge$ and $AE.time < \tau_{\text{next}}$ do
5          $S.time \leftarrow AE.time$ ;
6          $A.updateVariables(AE.time)$ ;
7          $A.fire(AE)$ ;
8          if $A.final$ then  Halt with acceptance. return false. ;
       /* If there is no possible event and the automata is not in
          a final state, halt                                     */
9      if $E =$ null then  Halt with rejectance. return false. ;
       /* In other cases, update variables in LHA and fire
          transition.                                             */
10     $A.updateVariables(\tau_{\text{next}})$; $N.fire(E.Trans, \tau_{\text{next}})$ ;
11     if no possible synchronisation then  Halt with rejectance. return false.
        ;
12     $A.synchronise$ ;
13     if $A.final$ then  Halt simulation with acceptance. return false. ;
14     $N.update()$ // This updates the Event Queue
15     return true.
```

Algorithm 3: The `simulateSinglePath` function

```
Data: a Event Queue structure $EQ$, the LHA $A$, the model $N$
Result:
1  $N.initialEventQueue(EQ, S)$ ;
2  $cont \leftarrow$ true ;
3  while $cont$ do  $cont \leftarrow simulateOneStep()$ ;
```

Algorithm 4: The `interactiveSimulation` function

```
   Data: a Event Queue structure $EQ$, the LHA $A$, the model $N$
   Result:
1  $N.initialEventQueue(EQ, S)$ ;
2  $cont \leftarrow$ true ;
3  $minInteractive \leftarrow 0$ ;
4  while $cont$ do
5      $checkinput \leftarrow$ true ;
6      while $checkinput \&\& S.time \geq minInteractive$ do
7          $input \leftarrow getline(cin)$ // Reads the last line of the (shell)
              input
           /* Fire a specific transition:                              */
8          if $input.substr(0,5).compare = "fire "$ then
9              $trans \leftarrow input.substr(5, input.size() - 5)$ ;
               /* Find the id of the transition named $trans$:         */
10             for $tid = 0; tid < transLabels.size()$ && $transLabels[tid]! =$
                $trans; tid++$ ;
               /* If there is such a transition, then force its
                  execution                                            */
11             if $tid < transLabels.size()$ then
12                 if $EQ.isScheduled(tid)$ then
13                     $EQ.prioritise(F)$ ;
14                     $checkinput \leftarrow$ false ;
           /* Execute exactly one step:                                 */
15         if $input = "step"$ || $input = "s"$ then $checkinput \leftarrow$ false ;
           /* Execute simulation until a specified time:                */
16         if $input.substr(0,5).compare = "wait "$ then
17             $until \leftarrow input.substr(5, input.size() - 5)$ ;
18             $minInteractive \leftarrow until$ ;
               // This requires a cast from string to float (and
                  maybe some safety checks...)
19     $cont \leftarrow simulateOneStep()$
```

`Event` class. It is used to represent events of a model and consists of five variables:

- an integer `transition`: the identifier of the transition;
- a double `time`: the time at which the event will happen;
- a double `priority`: the priority of the event (for example, the priority of the transition);
- a double `weight`: the weight of the event;

A fifth variable `binding` corresponds to the binding (of the variables of the transition) of the event. It is used in high-level Petri nets and its type depends on the structure of the net (this will be explained in section 4.2).

This class also implements several functions:

- update functions `setTime`, `setPriority`, and `setWeight` that update the corresponding variables with the parameter of each function;
- a function `isPriorer` which takes another event e_2 as parameter and returns `true` if e_2 should be fired before this event (see section 1.1.3)

`EventQueue` class. The event queue is defined as a class (see Figure 4.9) that maintains:

- a vector of events `evtTbl` which maps each possible (transition, binding) pair to an event;
- a vector `evtHeap`, which is a heap of enabled (transition, bindings) pairs in the order of priority of their events;
- and a vector `evtHeapIndex` which maps each possible (transition, binding) pair to the position of the corresponding event in the heap (or -1 if the event is not in the queue).

Note that the heap (`evtHeap`) refers to each event through its (transition, binding) pair. The event is then found in the `evtTbl` vector. In order for the model to update its Event Queue, several standard functions are available:

- `insert` which inserts a new `Event` in the Event Queue;
- `replace` which replaces the `Event` of the same (transition,binding) pair by a new element, to update time or priority;
- `pause` and `restart` which allows to pause the time of an event, and reschedule the event with the remaining time, in case of an *age memory* policy.
- `isScheduled` which checks whether there is a scheduled `Event` of the (transition, binding) couple;
- `remove` which removes of the queue the `Event` corresponding to the given (transition,binding) pair.

These functions use the classical heap functions `siftUp`, `siftDown` and `swapEvt` described below.

```
class EventsQueue {
private:
    vector<vector< long int >> evtHeapIndex;
    vector<vector< Event >> evtTbl;
    vector<sizeSq> evtHeap;
public:
    EventsQueue(const std::vector<size_t> &sizes);
    void insert(const Event &);
    void replace(const Event &);
    void pause(double,size_t,size_t);
    bool restart(double,size_t,size_t);
    void remove(size_t,size_t);
    bool isScheduled(size_t,size_t) const;
    Event& getEvent(size_t,size_t);
    void reset();
    size_t getSize();
    const Event& InPosition(size_t)const ;
private:
    void siftUp(size_t);
    void siftDown(size_t);
    void swapEvt(size_t,size_t);
}
```

Figure 4.9: A simplified version of `EventQueue.hpp`

Event Queue Algorithms. The heap is built as a binary tree, implemented by the functions described in this section. The vectors are initialised through the class initialisation function: for that, it checks the number of possible bindings for each transition. The following functions are defined to ease further definitions: `getLeftChildIndex`$(k) = 2k + 1$, `getRightChildIndex`$(k) = 2k + 2$ and `getParentIndex`$(k) = \lfloor (k-1)/2 \rfloor$. We first define the basic functions:

- `swapEvt` exchanges the positions of the events of nodes i and j of the heap, and modify the vector `evtHeapIndex` accordingly.
- `siftUp` first checks if the node is the root node, in which case it does nothing. Otherwise, it gets the parent `Event` and checks whether the current event (in the i^{th} node) has a higher priority (with respect to the `isPriorer` function). In that case, it swaps both events (with `swapEvt`) and performs a recursive call on the new index (basically, $\lfloor (i-1)/2 \rfloor$).
- `siftDown` first finds the highest priority child (after checking whether there are zero, one, or two childs), then checks if the current event has a higher priority; In that case, it swaps both events and performs a recursive call on the new index.

The following functions are then defined:

- `isScheduled` gets the transition and binding identifiers, checks the value in `evtHeapIndex`, and returns `false` if the corresponding value is -1 and `true` otherwise;
- `InPosition` simply reads the current value in `evtHeapIndex`;
- `getEvent` takes a (transition, binding) pair and returns the corresponding event using `evtTbl`;
- `insert` inserts an event in the queue by putting it at the highest index, and then using `siftUp` until stabilisation.

Using the $(transition, binding)$ pair as an identifier prohibits the multiple (or infinite) server policy, since in this case an additional indexing should be necessary to distinguish between several instances of the same pair.

`TimeGen` class. This class handles the generation of the delays for newly-enabled events, according to the distribution associated with their transition. It implements a function called `GenerateTime` which uses the library Boost and the Mersenne Twister random number generator.

4.1.3 Server

We now describe the functionalities of the server which include:

- launching simulators working in parallel;
- aggregating the simulation results;
- computing the values of HASL expressions;

```
class BatchR {
public:
    BatchR(size_t,size_t);

    unsigned long int I;
    unsigned long int Isucc;
    double simTime;

    std::vector<bool> IsBernoulli;
    std::vector<double> Mean;
    std::vector<double> M2;
    std::vector<double> M3;
    std::vector<double> M4;
    std::vector<double> Min;
    std::vector<double> Max;
    std::vector< unsigned long int > bernVar;

    void addSim(const SimOutput&);
    void unionR(const BatchR&);
    void outputR(std::ostream &f);
    bool inputR(FILE* f);
};
```

Figure 4.10: A simplified version of `BatchR.hpp`

- and deciding the termination of the computation.

Result batch. The `BatchR` class (see Figure 4.10) is used to manage results from batches of simulations. It maintains different values of path expressions, such as the sum of results (in `Mean`), minimum and maximum values. The following functions are available:
- `addSim` adds the results of a single simulation;
- `unionR` adds the results of another batch;
- `outputR` prints the current batch results into a file and `inputR` reads a file that has been generated by `outputR`.

Server. The server algorithm is shown in algorithm 5. It requires the number of parallel simulators to launch, the list of HASL formulas, and the statistical parameters. When `KillClient` is called, it also erases all the current batches made by running simulators.

Statistical methods. The `evaluateHASL` function of the server will compute the new interval confidence for each HASL formula, using the *Boost.Math* library

Algorithm 5: The server algorithm

```
  Data: a $Result$
1 $LaunchClients(P)$ ;
2 while $Result.continueSim()$ do
3     Wait for any simulator $i$ to return its result ;
4     BatchR $batchResult(P.nbAlgeb, P.nbQual)$ ;
5     $batchResult.inputR(i)$ ;
6     $Result.unionR(batchResult)$ ;
7     $Result.evaluateHASL(batchResult)$ ;
8 $KillClient()$ // Clean up after simulation is finished
9 $printResults()$ // Output the simulation results
```

heavily. For example:

- in the case of a $E(f)$ HASL formula, it computes the mean value by dividing `Mean` by the number of successful simulations. Then, it uses the `M2` value to obtain the variance ($V = m2 - mean^2 + \frac{1}{Succ}$), using also the number of succesful simulations (needed in the Chows-Robbin algorithm). The width of the confidence interval is then obtained using the variance and the wanted confidence level;
- in the case of a sequential hypothesis testing, it constructs a confidence interval (to be handled by the other functions of Cosmos). It uses a logarithmic version of the likelihood ratio of the Wald method, for probabilities $v1$ and $v2$. If this logarithmic value is negative, then returns the confidence interval $[0, v2]$. In the other case, $[v1, 1]$ is returned.

Using these confidence intervals, an approximation on the progress of the simulation is made, using the ratio between the current width of the interval and the target width. Note that it is possible to use a relative width. Finally, `continueSim` checks whether the simulation should be continued:

- if the number of simulation exceeds the maximum number of simulations, then it returns false;
- otherwise, if there is no target width and no sequential testing, then it returns true;
- otherwise, if the the minimum value of all progress ratios is smaller than 1, then it returns true;
- otherwise, it returns false.

Note that there is a slight difference in sequential hypothesis testing, as we collect results by batch of simulations instead of one simulation at a time. However, one may consider that this generated bias is small enough in most case, the size of a batch being small enough compared to the final number of simulations.

Possible Improvements. Cosmos is currently designed to work on a single computer, and can benefit of multiple cores of a processor by launching several clients in parallel. The current implementation can however be transformed for multi-machine simulation by:

- transforming UNIX files (used for communication between processes) to sockets;
- broadcasting the Simulator binary to every machine.

This raise the problem of having a similar infrastructure so that the generated binary can actually run on each of the machines. The current implementation is not flexible enough to support binaries generated on different infrastructures: there is no way to certify that the format of batch results will be the same on each machine.

4.2 Improving the Binding Mechanism for high-level nets

The original implementation for high-level Petri nets has two limitations: it cannot handle infinite color classes (like the integers), and it was inefficient for large color classes. Here, we describe our first implementation contribution: an alternative data structure for markings on which we developed a more efficient search for enabled bindings, in particular in the case where color classes are very large or even infinite. This alternative data structure is toggled via a command-line option by the user.

4.2.1 Standard Implementation

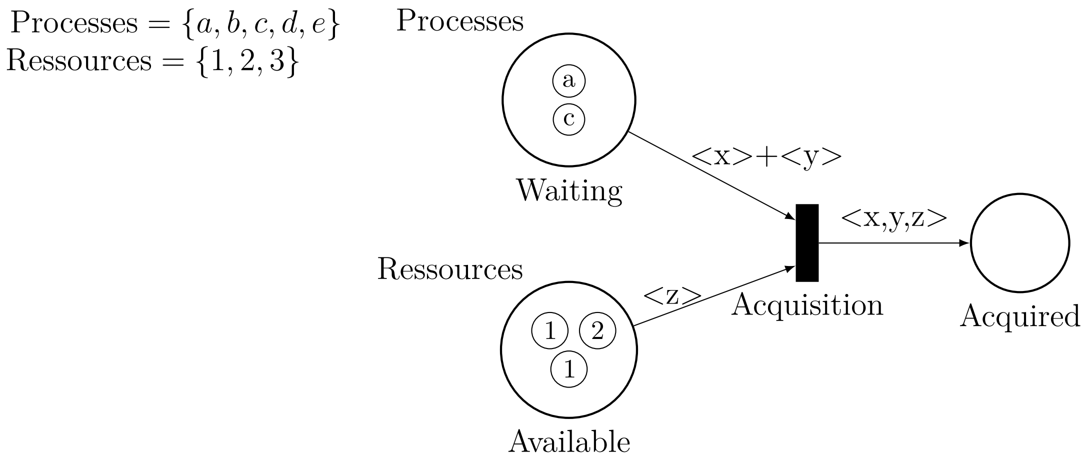


Figure 4.11: An example of high-level Petri net

High-level Petri nets. As described in section 4.1.1, Cosmos generates a file `markingImpl.hpp` to handle marking of Petri nets. It is enhanced to handle colors at the code generation.

In this file, each basic (finite) color domain `dom` is represented with:

- an enumeration of its colors (`dom_color_Classe`) with an element `Color_dom_dom_IC_`*i* for each possible *i* a possible value of the color), `Color_dom_Total` used to end iterations and `Color_dom_All` a special value used to represent all colors of a domain in an edge;
- a structure `dom_Token` with two variables, one for its class `c0` and the other for its multiplicity `mult`. This structure will handle the iteration of the enumeration of color classes (with functions `iter`, `next(int i)`, and `islast()`);
- a structure `dom_Domain` with one variable `mult` which is a vector of the multiplicity of each possible color class. This structure will be used to handle the marking of each place.

Finally, the class `abstractMarkingImpl` is defined, creating a C++ variable for each place (which contains the marking of the place) and each variable of the net (used to enumerate all the possible bindings). These C++ variables are typed with the Domain structure described above.

Example 38. For the model shown in Figure 4.11, the initial marking is represented in the following way:

- Processes: $\begin{array}{|c|c|c|c|c|}\hline 1 & 0 & 1 & 0 & 0 \\ \hline\end{array}$;
- Ressources: $\begin{array}{|c|c|c|}\hline 2 & 1 & 0 \\ \hline\end{array}$;
- Acquired: $\begin{array}{|c|c|c|}\hline 0 & \quad\dots\quad & 0 \\ \hline\end{array}$ (with $5 \times 5 \times 3 = 75$ null entries).

The model of the net is similar to previously explained:

- the `InitialEventQueue` iterates over all transitions, then over bindings of the variables of the transitions, then checks whether the transition is enabled using this marking. If the transition is enabled, it is added to the event queue;
- the `update` function, shown in Algorithm 6 updates the event queue with the same incremental method as before, but this time iterating over bindings when needed.

Binding Iteration. As shown in Algorithm 6 describing the `update` function for high-level Petri nets, the function `nextPossiblyEnabledBinding` is called to iterate over bindings of a transition. This is done by iterating over the color domains of each variable.

Example 39. Continuing example 38 : finding a binding for the Acquisition transition requires iterating twice over the Processes and once over the Ressources, for a total of $5 \times 5 \times 3$ iterations.

Algorithm 6: Function updating the Event Queue for a high-level Petri Net

```
Data: the Event Queue $EQ$, a Time Generator $TG$
1  Function update($ctime$,$lastTr$,$EQ$,$TG$)
       Input : the current time $ctime$, the last activated transition $lastTr$,
               the last associated binding $lb$, the Event Queue $EQ$ and Time
               Generator $TG$
2      for $t \in PossiblyEnabled(lastTr)$ do
3          while $b = nextPossiblyEnabledBinding(t, lb)$ do
4              if $IsEnabled(t, b)$ then
5                  if $\neg EQ.isScheduled(t)$ then
6                      $generateEvent(F, b)$; $EQ.insert(F)$
7      for $t \in PossiblyDisabled(lastTr)$ do
8          while $b = nextPossiblyDisabledBinding(t, lb)$ do
9              if $\neg IsEnabled(t, b)$ then $EQ.remove(t)$;
10     for $t \in MarkingDependant$ do
11         for $b \in t.bindingList$ do
12             if $EQ.isScheduled(t, b)$ then
13                 $generateEvent(F)$; $EQ.replace(F)$
```

Note that the number of possible bindings can increase exponentially with respect to the size of the expression labelling the edge. This was already observed and lead to some optimization when the color functions are simple (like single variables).

4.2.2 Alternative Implementation

Changes to the Marking class. Instead of using a `map` in `dom_Domain`, we now use an ordered set of (`dom_Token`,`mult`) pairs. In C++, the set data structure is implemented with red-black trees.

Example 40. For the model shown in Figure 4.11, the initial marking would now be represented this way:
- Processes: $\{(a, 1), (c, 1)\}$;
- Ressources: $\{(1, 2), (2, 1)\}$;
- Acquired: $\emptyset$.

In this new setting, the memory used to store the marking is linear with respect to the number of tokens.

The `EventQueueSet` class. This class (see Figure 4.12) is similar to the `EventQueue` class as it still maintains three tables:
- `evtTbl` maps a transition and a binding hash to an Event;
- `evtHeap` is now a vector of Events, organized as a heap;
- `evtHeapIndex` maps a transition and a binding hash to the position of the corresponding event in the heap (and -1 if there is no corresponding event).

Note that these definitions are different from those presented in section 4.1.2.

Binding Iteration. This time, the function `nextPossiblyEnabledBinding` tries to find values for each variable by iterating over the content of each input place. For remaining variables, it iterates over their color domain.

4.2.3 Benchmarks

In this section we describe two models, one of the patient flow in a hospital (with one token per patient, each having a different color) and the other of a small section of a motorway (with one token per vehicle). On these models, we run both implementations on one core of an *AMD Turion(tm) II Ultra Dual-Core Mobile M640*. This choice of a single core compared to a parallel implementation has no impact on the relative performances. The simulation parameters have been chosen so that all the simulation runs in less than 30 minutes. Moreover, the memory consumption has been tested on the simulator alone (using `/usr/bin/time -v`).

```
class EventsQueueSet {
private:
    vector<map<size_t, long int>> evtHeapIndex;
    vector<map<size_t, Event>> evtTbl;
    vector<Event*> evtHeap;
public:
    void insert(const Event &);
    void replace(const Event &);
    void pause(double,size_t,size_t);
    bool restart(double,size_t,size_t);
    void remove(size_t,size_t);
    bool isScheduled(size_t,size_t) const;
    Event& getEvent(size_t,size_t);
}
```

Figure 4.12: A simplified version of `EventQueueSet.hpp`

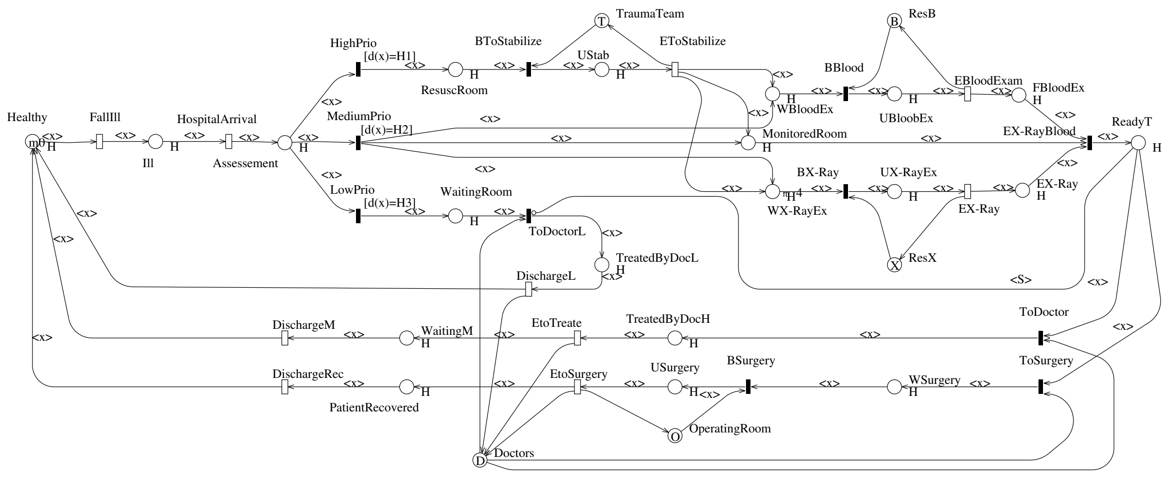


Figure 4.13: The `Hospital` model

	Standard		**Alternative**	
	Time	**Memory**	**Time**	**Memory**
20 patients	81.6 s	3.96MB	115.6 s	3.90MB
100 patients	902.0 s	4.16MB	1184.6 s	3.85MB

Table 4.1: Benchmarks for the `Hospital` model

		Standard		**Alternative**	
Cells	*Vehicles*	**Time**	**Memory**	**Time**	**Memory**
50	10	418.9 s	2.60GB	4.09 s	4.06MB
50	100	1 610 s	2.60GB	9.27 s	4.07MB
500	10	*Out of Memory*		140.14 s	5.42MB
500	100	*Out of Memory*		77.93 s	5.57MB

Figure 4.14: Benchmarks for the `SIA` model

The Hospital model. The Petri net from [5] shown in Figure 4.13 describes the flow of patients in a hospital emergency department. This net has been modified to be able to scale the number of patients, and the diseases (and the patient priority) are now chosen through stochastic transitions (instead of a static relation between patient and disease priority). There is a single color class, representing the set of patients.

All patients, represented by their tokens, start in the *Healthy* place. They can *FallIll*, and are assessed at their arrival. Depending on the priority of their illness, they go through different examinations and treatments.

Benchmarks for the Hospital model. We have run 5 000 trajectories of 200 time units with both implementations of the high-level models, for 20 or 100 patients. Results are shown in Table 4.1. The standard implementation runs this model 25% faster than the alternative implementation, due to having the same magnitude of tokens than of colors. In this case, the alternative implementation does not have significant impact on the memory used.

The SIA model. The Petri net shown in Figure 4.2 is used to model vehicles in a jammed motorway. It has been first presented in [15] and will be described in more details in Chapter 5. It has six color classes: `PosX`, `PosY` (positions in both axes), `VitX`, `VitY` (speed in both axes), `AccX` and `AccY` (acceleration in both axes). The controlled vehicle is represented using the color domain `PosX` $\times$ `PosY` $\times$ `VitX` $\times$ `VitY` $\times$ `AccX` $\times$ `AccY` and the other vehicles using the color domain `PosX` $\times$ `PosY` $\times$ `VitX` $\times$ `VitY`. Note that the transition *collision* has a lot of different

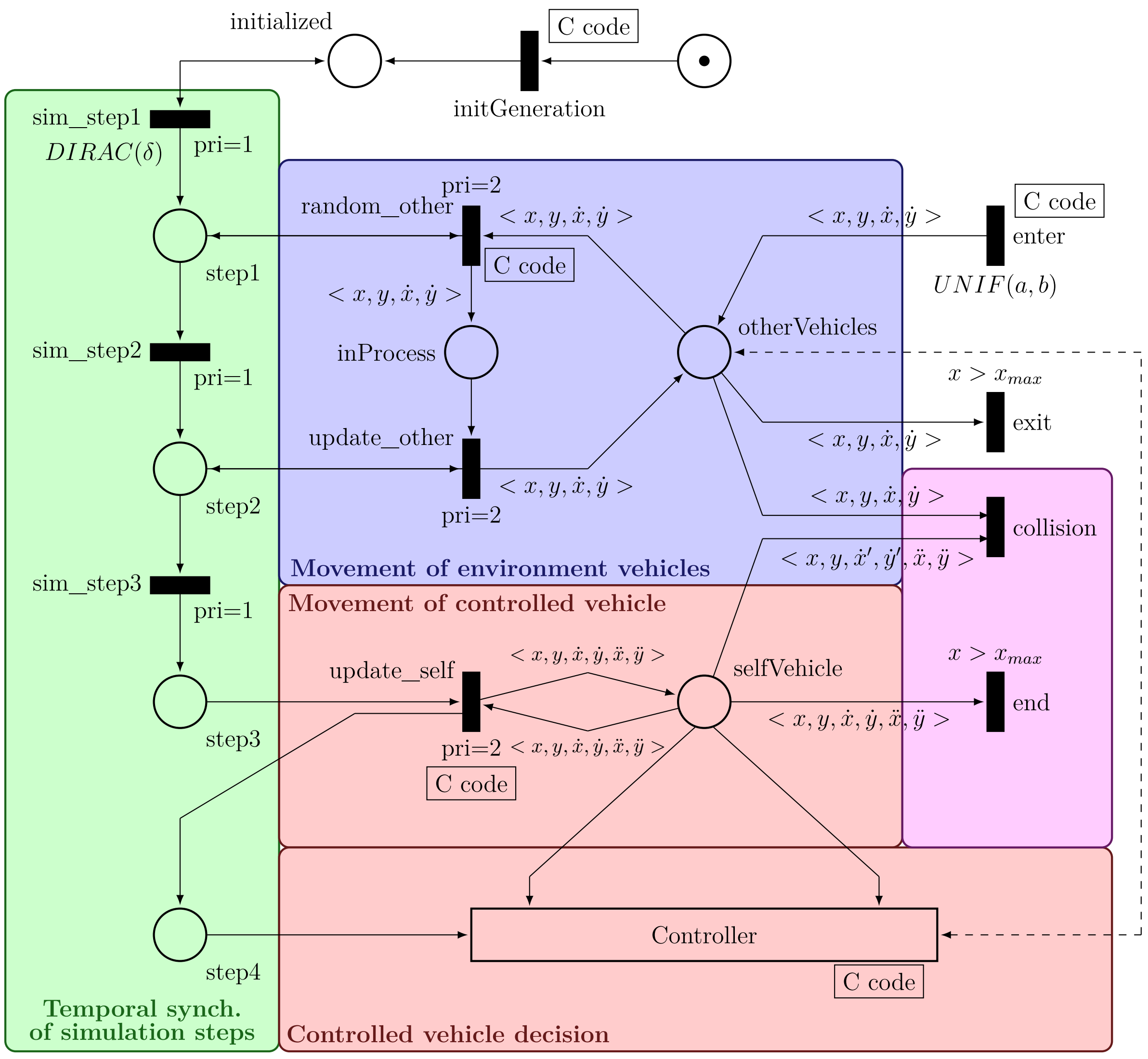

initialized
C code
initGeneration
sim_step1
$DIRAC(\delta)$
pri=1
step1
pri=2
random_other
$< x, y, \dot{x}, \dot{y} >$
C code
$< x, y, \dot{x}, \dot{y} >$
C code
enter
$UNIF(a, b)$
$< x, y, \dot{x}, \dot{y} >$
inProcess
otherVehicles
sim_step2
pri=1
$x > x_{max}$
exit
update_other
$< x, y, \dot{x}, \dot{y} >$
step2
$< x, y, \dot{x}, \dot{y} >$
pri=2
$< x, y, \dot{x}, \dot{y} >$
collision
$< x, y, \dot{x}', \dot{y}', \ddot{x}, \ddot{y} >$
Movement of environment vehicles
sim_step3
pri=1
Movement of controlled vehicle
$< x, y, \dot{x}, \dot{y}, \ddot{x}, \ddot{y} >$
update_self
selfVehicle
$x > x_{max}$
end
step3
$< x, y, \dot{x}, \dot{y}, \ddot{x}, \ddot{y} >$
pri=2
$< x, y, \dot{x}, \dot{y}, \ddot{x}, \ddot{y} >$
C code
step4
Controller
C code
Temporal synch. of simulation steps
Controlled vehicle decision

Table 4.2: The `SIA` model

possible bindings. The behaviour of the different vehicles are modelled through C code added to the transitions.

Benchmarks for the SIA model. We have run 60 trajectories of the model until either the controlled vehicle exits the section of the motorway or a collision happens. We do these simulations with different values of cells (number of possible `PosX` values) and vehicles. Results are shown in Table 4.14. Note that the standard implementation is far slower than the alternative implementation, and not even running for high number of cells (because the size of the binding vector is higher than the available memory). Finally, the alternative implementation runs faster with more vehicles in the case of 500 cells, due to a higher collision rate. Moreover, this alternative implementation uses negligible memory space compared to the first color implementation in this example.

Conclusion. In the standard implementation, the memory space used to store a marking is exponential with respect to the number of class occurrences in the domain definition. For a transition, in the worst case, the binding enumeration time is exponential with respect to the size of the expression labelling the edge. Due to subtle optimisations of the rules, in some cases (as in the Hospital model), the binding can be performed in constant time.

In the alternative implementation, the memory space used to store a marking is linear with respect to the number of tokens. For a transition, in the worst case, the binding is polynomial in the number of tokens in each place.

4.3 Simulink Integration

The `Model` class has been introduced in section 4.1.1 as a generalization of the structure used for Petri nets. In this section, we describe how Cosmos transforms Simulink models into this structure, with the approximate semantics described in section 3.3.

The `SKTime` class. In order to avoid undesirable approximations when dealing with time operations, we chose to represent time by the `SKTime` class. This class consists of two integers: `time` and `precision`. A (t, p) `SKTime` would represent a precise time of $t.10^{-p}$. Usual arithmetic operations and relations have been implemented in this class. Furthermore, cast must be done explicitly using the `getDouble` function. This is particularly useful when dealing with latencies.

Example 41. The code shown in Figure 4.15 represent time calculations, using a simple model with a static step (0.2) and a block of delay 0.4. The expected

```c
#include <stdio.h>
int main() {
    double time = 0;
    double target = 0;

    for (int i=0;i < 6;i++) {
        time = time + 0.2;
        target = time - 0.4;
    }
    if (time < 1.2) { printf("Smaller than 1.2\n"); }
    else { printf("Greater or equal to 1.2\n"); }

    if (target < 0.8) { printf("Smaller than 0.8\n"); }
    else { printf("Greater or equal to 0.8\n"); }
    return 0;
}
```

Figure 4.15: A toy example showing floating point approximation limits

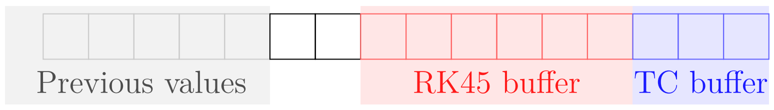


Figure 4.16: Buffer zones in Simulink vectors

behavior is that both `time` and `target` values are equal to respectively 1.2 and 0.8 after the 6 steps of the loop. However, running the executable generated from this code prints `Greater or equal to 1.2` and `Smaller than 0.8`. This is explained by the fact 0.2 has no finite representation as a binary floating-point value.

Model state. The state of a Simulink model in Cosmos is represented by a class `stateImpl` using vectors indexed by the simulation step. There is one such vector for the time values of the simulation (in the `SKTime` class), and one vector by output signal for every block. The integer `lastPrintEntry` stores the current step of simulation. Note that this class can be used, like `markingImpl`, as an input for any function requiring a `DEDState`.

In order to deal with two particular features, these vectors have more cells than the number of simulation step. Cosmos has nine additional cells:

- the buffer for application of the Runge-Kutta-Fehlberg method and its intermediate values requires six cells;
- the computation of the time of threshold crossings requires three cells: current lower and upper bounds, and new candidate value.

Model functions. As seen in section 4.1.1 that a `Model` class has to implement three functions: `initialEventQueue`, `update` and `fire`. A Simulink model comes with a unique transition, called `SimulinkTransition`, which remains in the Event Queue at a time corresponding to the next simulation step. These functions are defined as follows:

- `initialEventQueue` inserts the `SimulinkTransition` in the Event Queue at time 0 with a minimal priority;
- `fire` computes the current values of each block at given time t, according to the block ordering, then computes the value at $t + \varepsilon_{\mathbb{T}}$ where $\varepsilon_{\mathbb{T}}$ is the minimum increment of the `SKTime` value.
- `update` which computes the next simulation step, starting with the default step size, and refining it due to the Runge-Kutta-Felhberg integration method and the threshold crossing search.

These functions rely on basic ones:

- `executeBlock` evaluates the value of the outputs signals of the given block at the given simulation step. It calls another function, `findLatencyIndex` which returns the index corresponding to the delay of blocks with latency;
- `computeBkwds` computes the output values of the backward graph of the given block at the given simulation step;
- `estimateIntegrators` performs the computation of the Runge-Kutta-Fehlberg integrator over the whole model, using the dedicated cells. It also generates a new approximation of step size if necessary.

Code generation. The simulink models, given in the default `.slx` format, are a set of XML files describing the models and the architecture. The transformation of these models into the C++ code implementing their behaviour through the functions described in this section is done via OCaml. The information from the XML tree is parsed using the standard Xml-light library and converted into a graph which is represented by a pair:

- a list of `blocks` (the edges), which are 4-uplets (a string `blocktype`, an integer `blockid`, a string `name` and a list of (string,string) pairs `values`;
- a list of `simulinkLink` values which are 4-uplets of integers (`fromblock`, `fromport`, `toblock`, `toport`.

The default block values, provided separately in the Simulink model, are joined to this graph representation as a list of (string `blocktype`,(string,string) `values`) pairs. They are then used to complete the values of each block, not overwriting any of the provided values of the block. The latency time of each block is then computed from the given block values, as the Simulink value can be in a relative format.

At this step, the blocks are usually only ordered by their identifier. It is now necessary to reorder them, using the Block Ordering described in section 3.2.2. It is done the following way, using a hash table `c` mapping each block ID to a number, and a set `s` of blocks that can be added to the final sorted list (decreasing corresponding `c` values and adding it to the list):

- first, fill this hash table by getting the number of incoming edges of each block;
- initialise `s` by adding all blocks with no incoming edges;
- add blocks not using any of their inputs to the final list of blocks;
- order blocks by increasing latency, then add blocks with non-infinitesimal latency to the final list;
- then process the infinitesimal latency blocks (which is the Integrator blocks, plus the DiscreteIntegrator blocks using the Forward-Euler method);
- finally, perform the topologic sort by dealing with each element of `s`.

The list of non-processed links between blocks is kept during this whole procedure, and an error is returned if this list is not empty at the end.

Finally, the code is generated from this final list of blocks using each block's definition.

4.4 Multi-model Simulation

The requirements for automotive transports has led to deal with two different formalisms: one for the probabilistic aspects used to model vehicular environments (a high-level Petri net), and the other for the control aspects (a Simulink model). This multi-model phenomenon tends to increases, which led us to to propose a generic approach for multi-formalism modelisation.

In this section, we describe how the simulation with two models is handled, including how communication is done between the two models. It is still a work in progress, with the multi-model simulation currently only possible between Petri nets and Simulink models.

4.4.1 Generic multi-model loop

Two models into one. In order to reuse most of the code handling the simulation in the case of two interacting models, we embed them into one single model. This, however, requires to dispatch the event handling between models. The `MultiModel` class has two variables $m1$ and $m2$, each corresponding to one of the two models. An additional *shift* parameter is added to the `DEDS` specification which should be given when creating an instance of the corresponding class. The `DEDS` functions are adapted as follows:

- `initialEventQueue` calls the function of both models;

- `fire` calls the appropriate fire function, by checking whether the transition ID tr is greater or equal to $m2.\mathit{shift}$, in which case it calls the $m2$ `fire` function with the appropriate ID ($tr - m2.\mathit{shift}$). In the other case, it calls the $m1$ `fire` function with the appropriate ID (tr);
- `update` calls the appropriate update function with the same parameters, except for the transition ID.

This way, the Event Queue can be shared for both models. The only modifications to model functions consist in dealing correctly with *shift* in calls of functions of the Event Queue. This `MultiModel` is defined as a library, for any kinds of models.

Interface for communication. In the implementation of the `MultiModel` class, we further add two functions to the MultiModel specification. These functions are specific to the pair of considered models. They are generated when parsing both models. These are:

- `synchronize_fire` which, after a transition is fired into a model, updates the state of the other model considering as input the new state of the first model;
- and `synchronize_update` which fires the appropriate synchronisation transition before calling the `update` function of the corresponding model.

Both functions are called in the respective `fire` and `update` functions, after the operations described in the previous paragraph are performed.

Future works. As seen in the previous paragraphs, while the simultaneous simulation of multiple models can be done in a generic way, the communication between these models is done in a *ad hoc* manner. One possible way to generalize the construction of communication between models would be introducing a formal language to describe the relation between each model. This language would be parsed and the appropriate code generated. Another improvement in the case of multiple models would be to use communication paradigms of distributed models, such as broadcast. One may also need to have informations shared between different models.

4.4.2 Simulink and Petri Net Simulation

As seen in Chapter 1, it would be interesting to combine stochastic high-level Petri nets and hybrid models since they present complementary features (randomness versus continuous time and space). Hence, they are good candidates for instanciating the generic multi-model loop presented in the previous section.

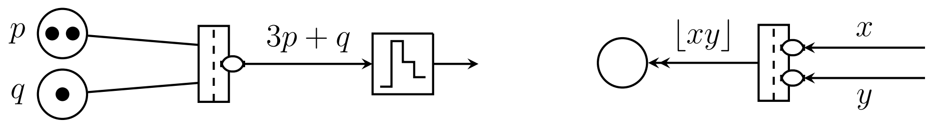
Interface transitions. In this paragraph, we detail the communication interface between Simulink models and stochastic high-level nets.

There are two kinds of interface transitions: *SK*-in transitions, directed from the net to the *SK*-model and *SK*-out transitions in the other direction.

The input arcs of a *SK*-in transition (see Figure 4.17(a)) are *test arcs* (explained with the firing) connected to places of the net. Any output arc is connected to the input of a Simulink block. An *SK*-in is enabled when the content of at least one of its input place is modified. The firing of such a transition proceeds as follows: a function is associated with each output arc, taking as parameters the contents of the input places. When the firing takes place, the function is evaluated. This function can be specified by a multiset of tokens as illustrated, or by a C code associated with the arcs.

The input arcs of an *SK*-out transition (see Figure 4.17(b)) are output signals of an *SK*-model and the output arcs are *overwriting* arcs connected to places that can only be connected to ordinary transitions of the net by *read* arcs. Similarly to *SK*-in transitions, the output arcs are labelled by functions of the incoming signals. Such a transition is activated at every sampling time of the *SK*-model. Upon firing, it rewrites the contents of the output places according to the evaluation of the function.

Parsing these transitions leads to the generation of the the `synchronize_fire` and `synchronize_update` functions.



(a) An *SK*-in transition (b) An *SK*-out transition

Figure 4.17: Petri net/Simulink Interface transitions

Target Simulation loop. We now instanciate the multi-model simulation loop for Simulink models and stochastic high-levels nets. All enabled transitions are stored into an *event* queue implemented as a binary heap, with their time of occurrence, their priority and weight. The next Simulink step is added as a possible event. At each simulation step, the earliest event is chosen. Among simultaneous events, the (decreasing) priority order is the following: *1.* *SK*-out firings, *2.* ordinary transition firings, *3.* *SK*-in firings, *4.* *SK*-event. In case of equal priorities, the choice is randomized according to the weights. Once an event is selected:

- If it is an ordinary transition firing, the marking is updated, the associated C code is executed; transitions that are newly enabled trigger new events while events corresponding to disabled transitions are removed.
- If it is an *SK*-in firing, the Simulink signals are updated and the time of the Simulink event is set to the current time.

- if it is the *SK*-event, all output signals are updated and the time of the Simulink event is updated as presented in Section 3.3. Finally, the *SK*-out transitions are added to the event queue with the current time.
- if it is a *SK*-out firing, the contents of output places are updated.

To simulate a discrete event system, at each step, one only has to compute what the next event will be and increase the simulation time to the time of this event. This is how ordinary Petri net transitions are fired in Cosmos. This leads to an efficient simulation of such system as the time to compute a simulation depends on the number of events and not on the simulated time. Unfortunately, this property is lost when simulating hybrid systems: the *SK*-event is triggered at least at a fixed frequency (δ_{max}).

Implementation of the simulation loop. In practice, this simulation loop is done using the `synchronize_fire` and `synchronize_update` functions we have described in the previous paragraph. These functions are generated when processing the Petri net. The generated functions work as follows:

- whenever the Simulink transition is called (which is the only case a *SK*-out transition exists in the queue), the corresponding `synchronize_fire` updates the corresponding markings in the net. Then, the corresponding `synchronize_update` fires the *SK*-out transition described in the net before updating the Event Queue using the net `update` function;
- the *SK*-in transitions appear in the net `PossiblyEnabled` vectors, which means that whenever a transition changes the entry place of a *SK*-in transition, then the *SK*-in transition is added to the event queue. The `synchronize_fire` corresponding to this *SK*-in transition then use the `setInput` function of the Simulink model, updating the corresponding Inport block. The `synchronize_update` then calls `updateInput` which advances the SimulinkTransition to the current time.

4.4.3 Use Case: Double Thermostat

Among the multiple systems that can be modeled using this enhanced version of Cosmos, we choose a well known toy example (which is presented in our Petri Nets 18 article [16]): a device with two heaters prone to faults, and using bang-bang controllers to keep the temperature in a room between $20°C$ and $25°C$. The system is modeled by a stochastic Petri net (Figure 4.18) with randomized faults and repairs. The evolution of room temperature and heater behaviours are hybrid and thus are modeled in Simulink (Figure 4.19). The fault transitions of the net have an exponential time distribution (with different rates). The repairman, initially at the Idle state, randomly chooses which (faulty) heater he will repair, then proceeds

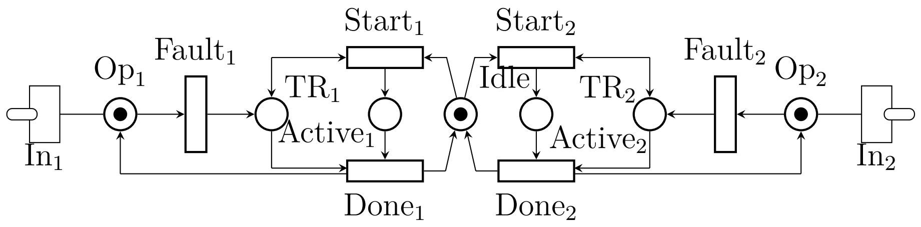


Figure 4.18: Petri net handling faults and repairs of a double heater

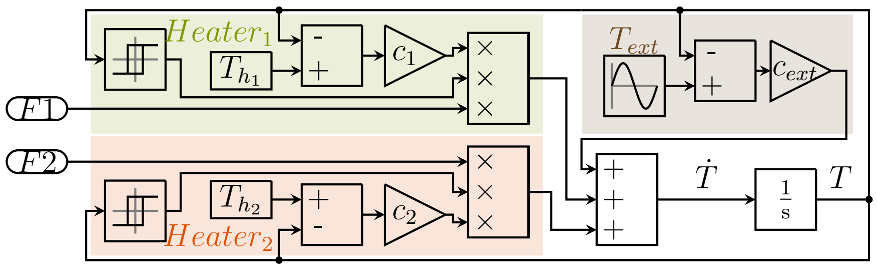


Figure 4.19: A Simulink model computing differential equations for the double heater. It contains four parts: the two heater temperatures, the external temperature and the block completing the differential equation

in fixed time and goes back to the Idle state. By default, both heaters are working (places Op_1 and Op_2 have a token).

The Simulink model handles the differential equations for both heaters, and for the outside temperature which is modelled by a sine wave (T_{ext}). The differential equation is : $\dot{T} = \mathbb{1}_{On_1} c_1(T_{h_1} - T) + \mathbb{1}_{On_2} c_2(T_{h_2} - T) + c_{ext}(T_{ext} - T)$ where c_1, c_2, and c_{ext} are the respective thermal conductivity coefficients, T_{h_1} and T_{h_2} are the respective temperatures at which each heater functions, and On_1 and On_2 are the respective states of each bang-bang controller which should maintain the temperature between $T_{min} = 20°C$ and $T_{max} = 25°C$. A bang-bang controller is a very simple hysteresis controller where the heater is switched on ($On_i = 1$) when the temperature decreases to T_{min} and switched off ($On_i = 0$) when the temperature increases to T_{max}. The inputs $F1$ and $F2$ receive respectively the content of places Op_1 and Op_2.

Figure 4.20 shows a simulation of the system. In the first period there is only a small failure of heater 2, and we can observe the bang-bang behaviours of the system. In the second period both heaters fail at the same time while the outside temperature is low, thus the temperature quickly drops to $13°C$ before the first heater is repaired.

We are interested in several performance indices. The first type concerns the reliability of the model measured by two indices: *the minimal temperature observed*

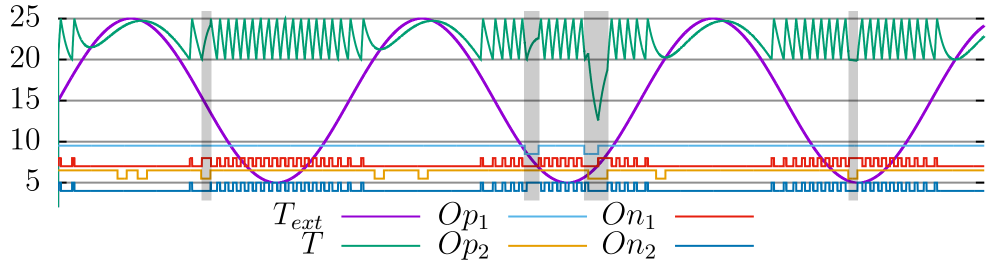


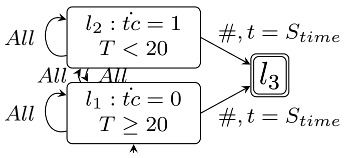
Figure 4.20: A trace of simulation: T and T_{ext} represent the inside and outside temperatures, Op_i corresponds to heater i being operational and On_i corresponds to heater i being switched on. Gray areas highlight failure of at least one heater when $T_{ext} < T_{min}$.

along a trajectory (I_1) and *the time spent in a state where the temperature is below* 20°C (I_2). The second type concerns the average behaviour: *the average temperature* (I_3), *the average number of switched-on heaters* (I_4), which is correlated with the energy consumption of the system, and *the average time during which the repairman is idle* (I_5).

These indices are specified in HASL with an LHA (Fig. 4.21) which accepts the trajectories after S_{time} time units. The LHA contains a hybrid variable tc with derivative 1 when the temperature is below 20°C and 0 otherwise. The HASL expressions start with a *probability operator*: here AVG is used for all indices to specify the average value over all trajectories; then a *path operator* (Min, Last, Mean) which is defined along each path. Path operators take as parameters algebraic expressions over the Petri net places and the LHA variables. For example, I_1 specifies the minimal temperature along a trajectory and then the average value over all trajectories.

I_1 : AVG(Min(T))
I_2 : AVG(Last(tc))
I_3 : AVG(Mean(T))
I_4 : AVG(Mean($Active_1 + Active_2$))
I_5 : AVG(Mean($Idle$))

(a) HASL formulas



(b) LHA

Figure 4.21: HASL specification for performance indices

The model as it is described above is referred to as M_0. In order to study the overhead of integral computations over the stochastic simulation, we build two additional alternative models. The first one M_1 is a model where the integration block in the Simulink diagram has been replaced by a discrete time integrator. In the model M_2, the simulink part is omitted, keeping only events that are transition firings.

Indices	M_0	M_1
I_1	[18.698 ; 18.716]	[18.272 ; 18.289]
I_2	[66.344 ; 67.217]	[92.921 ; 93.927]
I_3	[22.439 ; 22.442]	[22.485 ; 22.488]
I_4	[0.4999 ; 0.5006]	[0.4884 ; 0.4890]
I_5	[0.9239 ; 0.9242]	[0.9239 ; 0.9241]

Models	Build time	Sim. time
M_0	5.74s	6 885s
M_1	5.73s	1 145s
M_2	1.31s	1.810s

Table 4.3: Simulation results

Each model was run for 500 000 simulations of 2 000 seconds, with $\varepsilon_V = 0.01$ and $\delta_{max} = 1$. The sine wave frequency was 0.01 and oscillating between 5°C and 25°C, and the time step of the discrete-time integrator was δ_{max} (1 second). We used $T_{h_1} = 55$°C, $T_{h_2} = 65$°C, $c_1 = 0.02$, $c_2 = 0.013$ and $c_{ext} = 0.04$. Results are reported in Table 4.3. The left table reports the computed confidence interval for the different indices, the right one reports simulation and building times.

Tool analysis. The build time is always less than the simulation time and becomes negligible when models include a Simulink part. The critical factors for simulation time are: (i) the speed of step firing, about 10^{-7} sec. for net firing compared to 10^{-6} sec. for Simulink steps, and (ii) the number of steps per trajectory, about 40 for M_2 vs. 2000 for M_1. As expected, the use of a discrete-time Integrator yields a faster simulation, albeit still far longer than the net alone, while it affects the accuracy of the index values and more precisely triggers a larger variation of temperature over time.

Property analysis. We focus on the most pertinent model M_0, with two antagonist goals: minimizing the installation cost (depending on the parameters of heaters and repairman), and maximizing the comfort of the user (depending on the temperature evolution). With the current parameters, each heater is active about 1/4 of the time and the repairman is idle 92% of the time. The average temperature is about 22°C, reaching the objective, while the minimal temperature is slightly above 18°C.

CHAPTER 5

CASE STUDIES

Abstract. The increasing development of autonomous vehicles leads to safety challenges for the involved companies like Waymo and Tesla. The goal of the SVA[1] industrial project [68] hosted by IRT SystemX is to provide methods, based on simulation, for assessing the safety of autonomous vehicles. The members of the project work in cooperation with major french car manufacturers such as Renault or PSA Peugeot Citroën, but also suppliers of vehicle components such as Continental or Valeo. In this section, we apply the earlier chapter developments to two relevant case studies. In the first one, we describe the study of a simple motorway segment, published in an industrial conference [15]. The second case concerns an entrance ramp. It is more sophisticated since it requires a more complex decision-making algorithm for the autonomous vehicle.

In order to provide an accurate modeling, we have identified the quantitative and qualitative caracteristics of the scenarios: *1.* the granularity of the representation of the lane(s), *2.* the random behaviour of environment vehicles, *3.* the information given to the controller, and *4.* the controller algorithms to be analysed.

Contents

[1]Simulation for the safety of Autonomous Vehicles

5.1 Motorway Segment

Here, we consider a one-way motorway segment, with one controlled vehicle and other vehicles with random behavior. We want to evaluate the given controller with various performance indices, such as:

- the *collision rate*: probability of a collision before the vehicle exits the section;
- the *distance to collision*: expected travelled distance before collision, if any.

5.1.1 Model of the Motorway

The situation is depicted in Figure 5.1. The controlled vehicle, shown in red, and the environment vehicles, shown in white rectangles, are handled separately. Their position is characterized by the cell coordinates (i, j) representing position i on lane j. The lateral and longitudinal speeds are also tracked for all vehicles, while the accelerations on both axes are only kept for the controlled vehicle. Indeed, the behaviour of the other vehicles is modeled by a random choice of speed every time unit, while the acceleration of the controlled vehicle is produced by the controller and the new speed is then computed in a standard way.

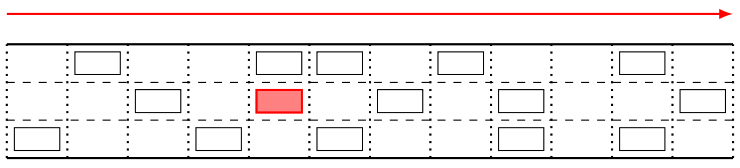


Figure 5.1: A motorway segment

Parameters, color classes and domains. The stochastic high-level Petri net depicted in Figure 5.2 models our case study. It requires to fix some parameters:

- length N_{pos} of the section;
- number N_ℓ of lanes;
- maximal speed $v_{\max}$;
- and maximal number N_{veh} of vehicles.

The following color classes are defined using these parameters:

- $\texttt{PosX} = [\![0, N_{pos}]\!]$ the longitudinal positions;
- $\texttt{PosY} = [\![0, N_\ell]\!]$ the lateral positions;
- $\texttt{VitX} = [\![0, v_{\max}]\!]$ the longitudinal speeds;
- $\texttt{VitY} = [\![-1, 1]\!]$ the lateral speeds;
- $\texttt{AccX} = [\![-1, 1]\!]$ the longitudinal accelerations;
- $\texttt{AccY} = [\![-1, 1]\!]$ the lateral accelerations.

From these color classes, the following color domains are defined:

- `SelfVehicle` $=$ `PosX` $\times$ `PosY` $\times$ `VitX` $\times$ `VitY` $\times$ `AccX` $\times$ `AccY` representing the data of the controlled vehicle;
- `OtherVehicle` $=$ `PosX` $\times$ `PosY` $\times$ `VitX` $\times$ `VitY` representing the data of any environment vehicle.

The following variables are then defined: x is in `PosX`, y in `PosY`, $\dot{x}$ and $\dot{x}'$ in `VitX`, $\dot{y}$ and $\dot{y}'$ in `VitY`, $\ddot{x}$ in `AccX` and $\ddot{y}$ in `AccY`.

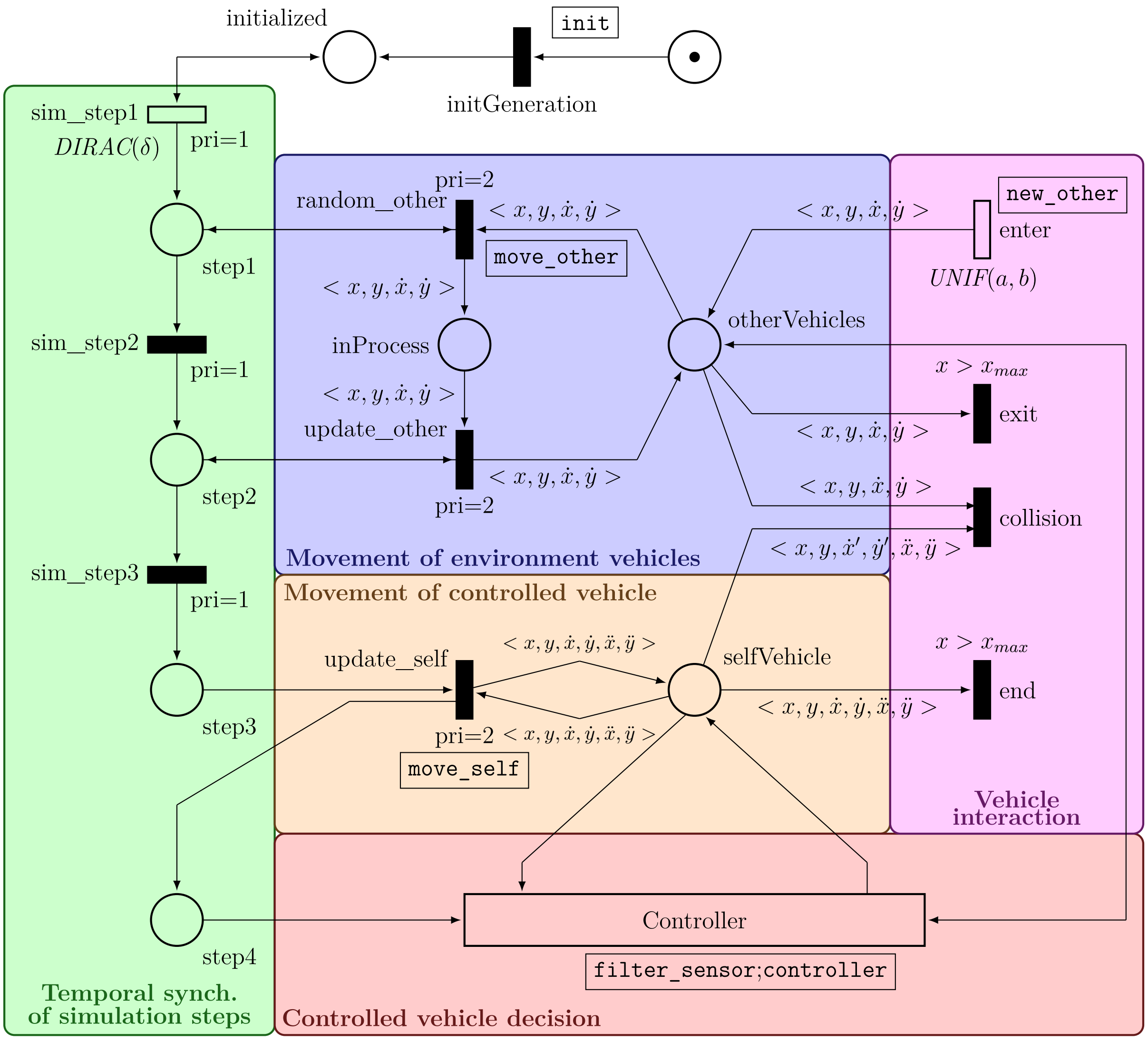


Figure 5.2: Motorway model

Slightly abusing notations since the variables are local to the transitions, we describe a token of color domain `SelfVehicle` by the tuple $\langle x, y, \dot{x}, \dot{y}, \ddot{x}, \ddot{y}\rangle$ and tokens of color domain `OtherVehicle` by $\langle x, y, \dot{x}, \dot{y} \rangle$.

General structure. An ideal model would use the formalism of Cosmos as a specification language, all the features of the case study being specified only using high-level Petri nets. However, some features are currently not easily expressible with such nets, for instance the random generation over numbers inside a token. Furthermore, the controller algorithm is more naturally specified by a C code.

The model is shown in Figure 5.2 and the corresponding GrML files can be found in the examples provided with Cosmos (in the folder `Examples/domainset/SIA`). The model is composed of several parts:
- the green part schedules the simulation steps performed by the other parts;
- the brown part controls the movement of the controlled vehicle;
- the blue part controls the movement of the other vehicles;
- the unique transition of the red part specifies the decision of the controlled vehicle;
- the pink part concerns vehicle interactions.

Synchronisation of simulation steps. Transitions `sim_step1`, `sim_step2`, `sim_step3` act as schedulers for the simulation parts: `sim_step1` has a Dirac distribution with a parameter corresponding to the time between two steps, while `sim_step2` and `sim_step3` are immediate, thus enforcing the synchronous behaviour of the simulation. The priorities of the transitions enforce the order of the simulation. There must be a token in `step1` for `random_other` to be fired, and this token will not leave `step1` until all tokens of `otherVehicles` have been processed. Then, this token is consumed by `sim_step2`, producing a token in `step2` which allows `update_other` to be fired. Similarly, the `step2` token cannot leave the place until all tokens of `inProcess` have been processed. Then, this token is consumed by `sim_step3` which produces a token in `step3`, allowing `update_self` then `Controller` to be fired in this order.

Note that, as `enter`, `exit`, `collision` and `end` have a zero priority, those transitions cannot be fired during the synchronous step of the simulation, but only once all steps have been completed.

Movement of other vehicles. The other vehicles are described using tokens of color domain `OtherVehicle`. Due to the C function `move_other`, each firing of the transition `update_other` produces an updated token with new speed and position in place `inProcess`. Then, in the next simulation step, all tokens from place `inProcess` are moved back to `otherVehicles`. As the definition of the behaviour of other vehicles is part of the definition of the scenario, it is described in section 5.1.3.

```
int newposition = ((int) (b.P->x).c0) + ((int) (b.P->dx).c0);
// The next lines define the value of "newway" and "newspeed",
   that depend on the scenario choices and are described in
   section 5.1.3
...
Marking.P->_PL_inProcess += (Vehicle_Token( (PosX_Color_Classe)
   newposition, (PosY_Color_Classe) newway, (VitX_Color_Classe)
   newspeed, b.P->dy));
```

Figure 5.3: Function `move_other`

Movement of controlled vehicle. The controlled vehicle is described using a token of color domain `SelfVehicle`. Transition `update_self` updates the speed and position of the controlled vehicle via the `move_self` C function.

```
int newposition = ((int) (b.P->x).c0) + ((int) (b.P->dx).c0);
int newspeed = ((int) (b.P->dx).c0) + (((int) (b.P->ddx.c0)) - 1);
Marking.P->_PL_selfVehicle += (SelfVehicle_Token( (
   PosX_Color_Classe) newposition, b.P->y, (VitX_Color_Classe)
   newspeed, b.P->dy, b.P->ddx, b.P->ddy));
```

Figure 5.4: Function `move_self`

Controlled vehicle decision. This single transition represents the call to the controller, including the modeling of the vehicle sensors. As the vehicle does not communicate with the motorway, it only relies on its own sensors to detect the other vehicles. We assume that the vehicle can only see vehicles which are close enough (see Figure 5.5). The C function `filter_sensor` updates a vector `data` of vehicles corresponding to the sensor view. Then, the evaluated `controller` is called with this `data` vector and the current position and speed of the controlled vehicle.

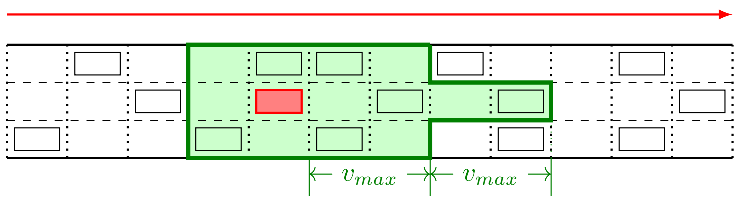


Figure 5.5: The model of the sensors

```
int seenVehicles=0;
int dbx=max(0,px-1);
int fbx=max((int) px+2*vMax,(int) Color_PosX_Total);
int dby=max(0,py-1);
int fby=max((int) py+1,(int) Color_PosY_Total);
for (int i0=dbx;i0 < fbx;i0++) {
    if (i0>px+vMax) { dby=py; fby=py; }
    for (int i1=dby;i1 < fby;i1++) {
    for (int i2=0;i2 < (int) Color_VitX_Total;i2++) {
    for (int i3=0;i3 < (int) Color_VitY_Total;i3++) {
        if (contains(
            Marking.P->_PL_otherVehicles,
            Vehicle_Token(
                (PosX_Color_Classe) i0,
                (PosY_Color_Classe) i1,
                (VitX_Color_Classe) i2,
                (VitY_Color_Classe) i3)
        )) {
            vehicle = {i0, i1, i2, i3};
            data[seenVehicles] = vehicle;
            seenVehicles++;
        }
    } } }
}
```

Figure 5.6: Function `filter_sensor`

Vehicle Interaction. This part of the net contains the transitions removing the vehicles from the simulation in two situations:

- Transition `exit` (respectively `end`) is fired when the vehicle longitudinal position is higher than $N_{pos} - v_{max}$, and removes the environment (respectively controlled) vehicle from the simulation.
- Transition `collision` is fired when some environment vehicle has the same (x, y) position as the controlled vehicle.

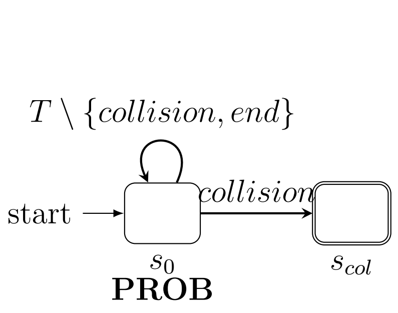
Transition `enter` corresponds to the asynchronous part of the simulation, adding a vehicle in the first v_{max} positions of a lane.

5.1.2 Specification of the Performance Indices

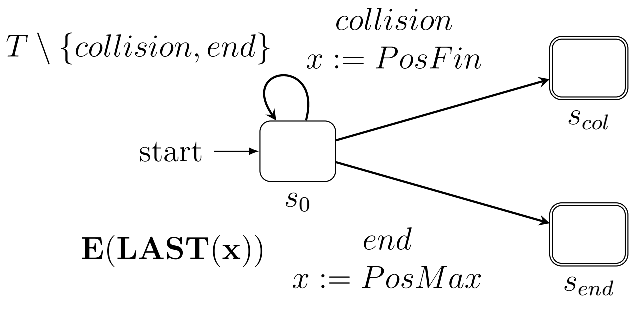
Observe that a trajectory ends when either `end` or `collision` is fired. The firing of `end` cannot be synchronised with any of the HASL automata (see below), and leads to an unsuccessful trajectory.

The HASL formula depicted in Figure 5.7 (a) is used to evaluate the probability of collision, where T denotes the set of all transitions of the Petri net in Figure 5.2. The synchronization implies that state s_{col} is reached exactly when transition `collision` is fired in the simulation and the expression **PROB** means that the value returned is the probability of reaching s_{col}.

The HASL formula depicted in Figure 5.7 (b) is used to evaluate the expected distance covered by the controlled vehicle before collision (if it occurs). With respect to the previous automaton, we add a new transition reaching state s_{end} that will be synchronized with the transition *end* of the Petri net, and a variable x that will take the current value of the horizontal position when the transition is fired. The expression $\mathbf{E}(\mathbf{LAST}(\mathbf{x}))$ (expected value of the last value of x) yields the desired index.



(a) I_1



$T \setminus \{collision, end\}$
$collision$
$x := PosFin$
start
s_0
s_{col}
$\mathbf{E}(\mathbf{LAST}(\mathbf{x}))$
end
$x := PosMax$
s_{end}

(b) I_2

Figure 5.7: Two HASL formulas (a) for the probability of collision and (b) for expected travelled distance

	Smooth velocity change				**Aggressive velocity change**			
	Basic		**Lane-changing**		**Basic**		**Lane-changing**	
I_1	0.838 ± 0.005		0.023 ± 0.005		0.847 ± 0.005		0.022 ± 0.005	
	468.51s	36 120	153.41s	6 620	459.67s	34 520	147.94s	6 020
I_2	267.44 ± 6.68		102.61 ± 2.56		282.23 ± 7.06		106.22 ± 2.66	
	71.48s	5 400	4 147 s	169 640	74s	5 480	3 766 s	154 120

Table 5.1: Evaluation of controllers. For each measure, controller and other vehicle behaviours, we provide a 99% confidence interval of the measure, the time in seconds required to perform the simulation and the number of simulated trajectories.

5.1.3 Evaluation

The first objective of this benchmark is to compare the performances of different controllers in various situations. As explained above, we measure the probability of a collision (I_1) and the expected covered distance before collision if any (I_2). The second objective is to quantify the influence of these parameters on the simulation time.

Scenarios. To evaluate our approach we have modelled simple controllers. We first study a basic controller (see Figure 5.8) which only aims at monitoring the vehicle ahead (l.12–20) and aligns its velocity to avoid collision with it. When there is no vehicle ahead or it is too far away, this basic controller speeds up until it reaches a maximal velocity v_{max} (l.22). Then we study a more advanced controller (see Figure 5.9) which is able to pass the vehicle ahead. In order to change its lane the controller monitors vehicles in the two adjacent lanes (l.23–25). The controller initiates a take over only when the speed of the vehicle in front of him is slower than the speed v_{max} and when there is no vehicle in the two adjacent lanes (l.28–39).

We have evaluated the controllers in the environment described in section 5.1.1 with two kinds of behaviours for other vehicles. In one set of experiments, all speed changes are smooth: at each step of simulation vehicles either stay at the same speed, moderately accelerate (the speed is increased by 1) or moderately decelerate (the speed is decreased by 1). This is done through `move_other` function as shown in Figure 5.10. Those behaviours produce a highly predictive environment making the controller job easier. In contrast the second set of experiment involves vehicles with unpredicables speed changes: at each step of simulation, every vehicle randomly selects a new speed in a predefined interval. The `move_other` function of this scenario is shown in Figure 5.11.

```
egoVehicle controller(int nb, radarData& data, egoVehicle& v) {
    // Target data:
    int nposx = v.posx; int nposy = v.posy;
    int nvitx = v.vitx; int nvity = v.vity;
    int naccx = v.accx; int naccy = v.accy;
    // Looking for the vehicle ahead:
    int opx = 0; int opy = 0;
    int ovx = 0; int ovy = 0;
    bool seenother = false;
    int posbest = INT_MAX;
    othVehicle veh;
    for (int i=0; i < nb; i++) {
        veh = donnees[i];
        opx = veh.px; opy = veh.py;
        ovx = veh.vx; ovy = veh.vy;
        if (opy == posy) { // If vehicle in same lane,
            seenother = true;
            // Check whether it is closer than the last chosen
               vehicle:
            if (opx < posbest) { posbest = opx; nvitx = ovx; }
        }
    }
    if (not seenother) { nvitx = Vmax; }
        // New vehicle:
    egoVehicle valeurs = { nposx, nposy,
                   nvitx, nvity,
                   naccx, naccy};
    return valeurs;
}
```

Figure 5.8: Basic controller

```
egoVehicle controller(int nb, radarData& data, egoVehicle& v) {
  // Target data:
  int nposx = v.posx; int nposy = v.posy;
  int nvitx = v.vitx; int nvity = v.vity;
  int naccx = v.accx; int naccy = v.accy;
  // Looking for the vehicles ahead:
  int opx = 0; int opy = 0;
  int ovx = 0; int ovy = 0;
  bool seenother = false; int posbest = 0;
  int oupx = 0; int oupy = 0; int ouvx = 0; int ouvy = 0;
  int odpx = 0; int odpy = 0; int odvx = 0; int odvy = 0;
  bool seenup = false; bool seendown = false;
  int bestup = 0; int bestdown = 0;
  int fvy = 0; int uvy = 0; int dvy = 0;
  int fvx = 0; int uvx = 0; int dvx = 0;
  int fpx = 0; int upx = 0; int dpx = 0;
  othVehicle veh;
  for (int i=0; i < nb; i++) {
    veh = donnees[i];
    opx = veh.px; opy = veh.py;
    ovx = veh.vx; ovy = veh.vy;
    // Similarly as the basic controller (l.20--24):
    // Put the target vehicle of current lane in (posbest, nvx)
    // Put the target vehicle of above lane in (bestup, uvx)
    // Put the target vehicle of below lane in (bestdown,uvx)
  }

  if       (not seenother) { nvitx = vMax; }
  else if (not seenup   ) { nposy = v.posy+1; nvitx = vMax; }
  else if (not seendown ) { nposy = v.posy-1; nvitx = vMax; }
  else { // Try to adjust speed to the fastest vehicle ahead and
     go to its lane if possible
        nvitx = fvx;
        if (dvx > nvitx && dpx > v.posx) {
          nvitx = dvx; nposy=posy-1;
        }
        if (uvx > nvitx && upx > v.posx) {
          nvitx = uvx; nposy=posy+1;
        }
  }
  // Output the new values:
  egoVehicle valeurs = { nposx, nposy,
                nvitx, vity,
                naccx, naccy};
  return valeurs;
}
```

Figure 5.9: Simple lane-changing controller

```
int newposition = ((int) (b.P->x).c0) + ((int) (b.P->dx).c0);
int deltaspeed = (rand() % 3) - 1;
int newway = ((int) (b.P->y).c0);
int newspeed = max(minSpeed,min(maxSpeed,((int) (b.P->dx).c0)+
    deltaspeed);
Marking.P->_PL_inProcess += (Vehicle_Token( (PosX_Color_Classe)
    newposition, (PosY_Color_Classe) newway, (VitX_Color_Classe)
    newspeed, b.P->dy));
```

Figure 5.10: Function `move_other` with smooth speed changes

```
int newposition = ((int) (b.P->x).c0) + ((int) (b.P->dx).c0);
int deltaspeed = rand() % (2*maxSpeed+1);
deltaspeed = deltaspeed - maxSpeed;
int newspeed = max(minSpeed,min(maxSpeed,((int) (b.P->dx).c0)+
    deltaspeed);
int newway = ((int) (b.P->y).c0);
Marking.P->_PL_inProcess += (Vehicle_Token( (PosX_Color_Classe)
    newposition, (PosY_Color_Classe) newway, (VitX_Color_Classe)
    newspeed, b.P->dy));
```

Figure 5.11: Function `move_other` with unpredictable speed changes

Simulation parameters. The results of our simulations are collected in Table 5.1. For these computations we simulate a motorway with 2 lanes and 1000 cells. The controlled vehicle is surrounded by 20 other vehicles for which speed varies between 3 and 5 cells per step. More precisely, all vehicles (including the controlled one) are generated randomly, with a uniform distribution on both lanes and cells (up to the 200th). If a vehicle is generated on a cell where there is another vehicle, we generate new values for both lane and cell. The I_1 performance index is estimated with a fixed confidence interval width of 0.01, and the I_2 performance with a relative confidence interval width of 5%. All simulations were performed on a single machine with 12 Intel Xeon cores using 8 parallel threads.

Controller evaluation. Comparing the probability of collision between the two controllers, we observe that the lane changing controller is far better than the basic one: 2.3% probability of collision vs 83.8% in the smooth environment and 2.2% vs 84.7% in the more aggressive one. We notice that the behaviours of other vehicles do not have a major impact on the probability of collisions. Similarly, the average travelled distance before collision is highly different between the two controllers.

In Table 5.1 we also observe that most collisions occurred at the beginning of the simulation. They occur around cell 104 for the lane changing controller and around cell 275 for the basic one. This shows that the simulation is biased by its initial state. More reliable results could be obtained by simulating a sliding window of the motorway around the controlled vehicle. Instead of simulating the controlled vehicle on the entire motorway we could make the controlled vehicle static in the middle of a short motorway and simulate all vehicles relatively to the controlled one.

Furthermore, in real life, the other vehicles do not behave randomly but follow an implicit control law thus reducing again this probability. In summary, this benchmark could be improved by adding these elements in order to obtain more accurate results (see section 5.2).

Scalability evaluation. We finally observe in Table 5.1 that when the probability of collisions becomes small, the number of simulations required to obtain relevant estimates becomes large. This can be seen for instance when comparing the required time to estimate the average distance before collision for the two controllers (71s vs 4147s and 74s vs 3766s). Thus the simulation could be intractable when the probability of collisions becomes close to zero. This is a well-known difficulty in presence of rare events for which several theoretical techniques and practical tools have been developed in the literature [64, 18] and can be used to improve this simulation.

Possible Improvements. We have identified other limits of our simulation which should be overcome to produce more precise results. Compared to real vehicles which feature turn signals and braking lights, there is no communication between vehicles in our simulation. This prevents the controller from synchronising its action with those of other vehicles. The behaviours of vehicles on the motorway should be enhanced to include some communication.

Currently, a synchronous simulation step is done in linear time with respect to the number of vehicles. A modeling integrating such synchronisation between vehicles would induce a quadratic blowup of this synchronous step and would require the introduction of the sliding window to overcome this complexity.

5.2 Entrance Ramp

We now consider an entrance ramp into a motorway segment, again with one controlled vehicle. A given controller will be evaluated with various performance indices, such as:

- the *vehicle collision rate*, *i.e.* the probability of a collision with another vehicle before the controlled vehicle exits the section;

- the *barrier collision rate*, *i.e.* the probability of the vehicle not entering the motorway segment at the end of the entrance ramp (and colliding with the barrier);
- the *severity of collision*, related to the velocities during a collision.

5.2.1 Model

Our model is based on an official French document [55]; the situation is depicted in Figure 5.12, using a longitudinal cell size of 7 meters. The model is similar to the case described in the previous section, except that the lanes are cut into 5 lateral sections. The lateral granularity is finer since we need to take into account the progressive entrance of the vehicle. The choices of 40 pre-entrance cells and 40 entrance cells are based on a 100km/h motorway segment according to [55]. This speed is called the *expected speed of the segment*, which is below the maximal speed. We add 20 cells to the section to check whether the vehicle is correctly inserted in the section. We also suppose that the controlled vehicle has all traffic information and tracks vehicles with unique identifiers. This choice is justified with respect to the capacities of sensors and the length of the section. With this size of cell and time step, the minimum positive acceleration is already above the standard acceleration.

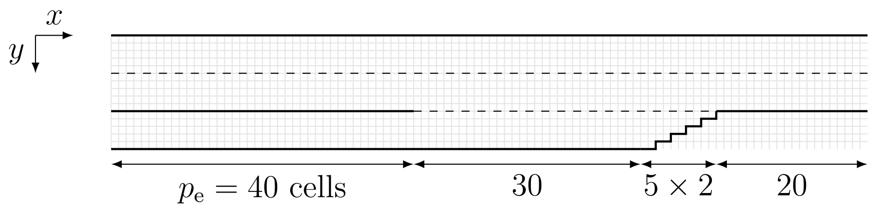


Figure 5.12: An entrance ramp to a motorway segment

Parameters, color classes and domains. Similarly to the previous section, the stochastic high-level Petri net depicted in Figure 5.13 models this case study, with the following parameters to be fixed:
- initial number N_{veh} of vehicles;
- maximal speed v_{max} (with a default value of 5);

Note that we have a fixed number of vehicles in the simulation: no vehicle enters.
The following color classes are defined:
- `PosX` $= [\![0, 100]\!]$ the longitudinal positions;
- `PosY` $= [\![0, 15]\!]$ the lateral positions, corresponding to three lanes, centered in positions 2, 7 and 12;
- `VitX` $= [\![0, v_{\max}]\!]$ the longitudinal speeds;
- `VitY` $= [\![-1, 1]\!]$ the lateral speeds;

- `AccX` $= [\![-1,1]\!]$ the longitudinal accelerations;
- `AccY` $= [\![-1,1]\!]$ the lateral accelerations;
- `VehId` $= \mathbb{N}$ the vehicle identifiers.

From these color classes, the following color domains are defined:

- `SelfVehicle` = `PosX` × `PosY` × `VitX` × `VitY` × `AccX` × `AccY` × `VehId` representing the state of the controlled vehicle;
- `OtherVehicle` = `PosX` × `PosY` × `VitX` × `VitY` × `VehId` representing the state of any environment vehicle.

The following variables are then defined: x and x' whose type is `PosX`, y and y' whose type is `PosY`, $\dot{x}$ and $\dot{x}'$ whose type is `VitX`, $\dot{y}$ and $\dot{y}'$ whose type is `VitY`, $\dot{x}$ whose type is `AccX`, $\dot{y}$ in `AccY`, m and k in `VehId`.

With the same notations as in the previous section, we describe a token of color domain `SelfVehicle` by the tuple $\langle x, y, \dot{x}, \dot{y}, \ddot{x}, \ddot{y}, k\rangle$ and a token of color domain `OtherVehicle` by $\langle x, y, \dot{x}, \dot{y}, m\rangle$.

General structure. Based on the motorway segment model from section 5.1.1, the new model is composed of the following parts:

- the green part schedules the simulation steps performed by the other parts;
- the brown part controls the movement of the controlled vehicle;
- the blue part controls the movement of the other vehicles;
- the unique transition of the red part specifies the decision process of the controlled vehicle;
- the pink part concerns vehicle interactions.

Synchronisation of simulation steps. This part is identical to the one in the motorway segment model. Transitions `sim_step1`, `sim_step2` and `sim_step3` act as schedulers for the simulation parts: `sim_step1` has a Dirac distribution with a parameter corresponding to the time between two steps, while `sim_step2` and `sim_step3` are immediate, thus enforcing the synchronous behaviour of the simulation. The priorities of the transitions enforce the order of the simulation. There must be a token in `step1` for `random_other` to be fired, and this token will not leave `step1` until all tokens of `otherVehicles` have been processed. Then, this token is consumed by `sim_step2`, producing a token in `step2` which allows `update_other` to be fired. Similarly, the `step2` token cannot leave the place until all tokens of `inProcess` have been processed. Then, this token is consumed by `sim_step3` which produces a token in `step3`, allowing `update_self` then `Controller` to be fired in this order.

Movement of other vehicles. The other vehicles are described using tokens of color domain `OtherVehicle`. Due to the C function `move_other`, each firing of the

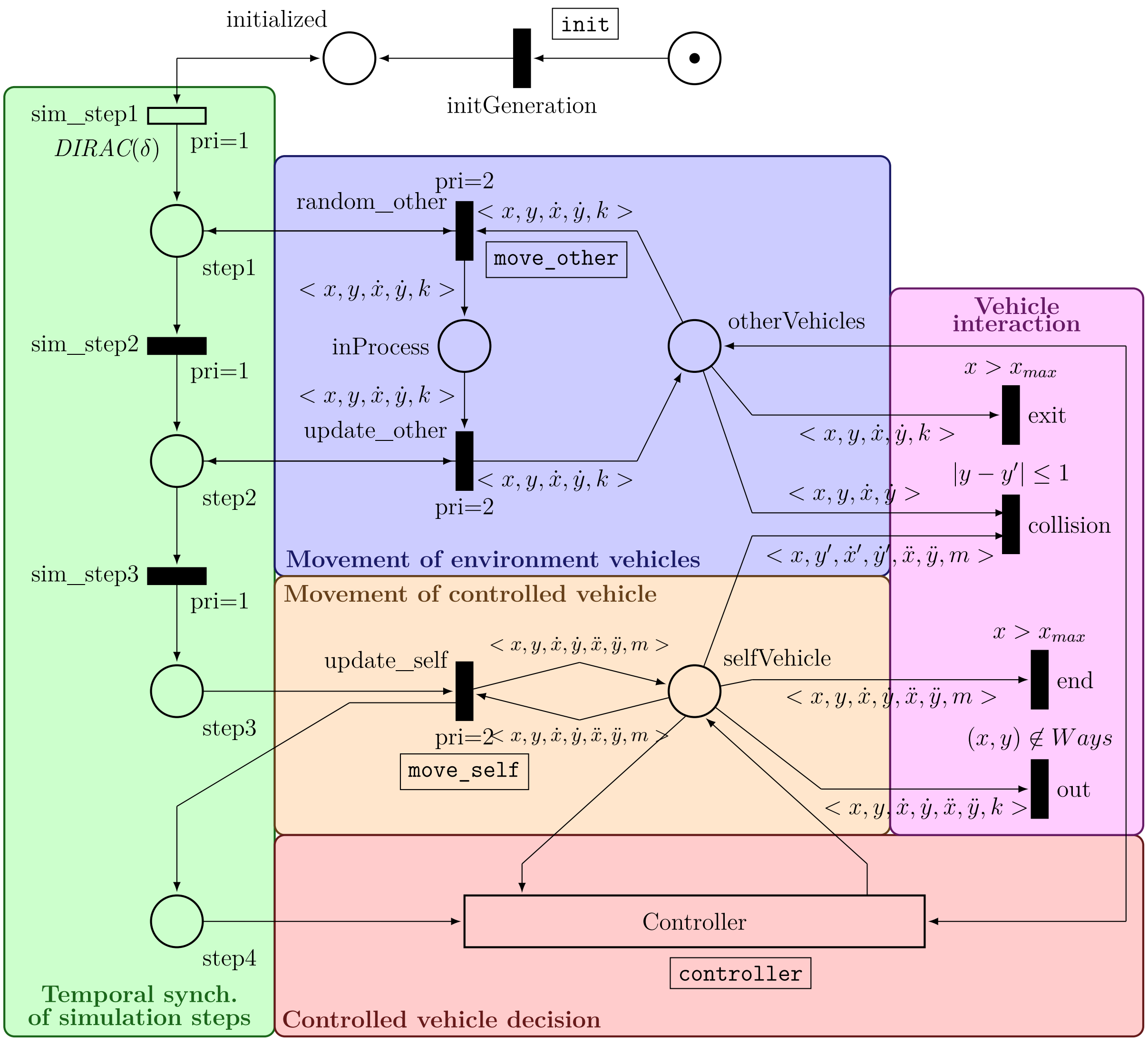

initialized
init
initGeneration
sim_step1
$DIRAC(\delta)$
pri=1
pri=2
random_other
$< x, y, \dot{x}, \dot{y}, k >$
move_other
step1
inProcess
otherVehicles
sim_step2
update_other
step2
Movement of environment vehicles
sim_step3
Movement of controlled vehicle
update_self
$< x, y, \dot{x}, \dot{y}, \ddot{x}, \ddot{y}, m >$
selfVehicle
move_self
step3
step4
Controller
controller
Temporal synch. of simulation steps
Controlled vehicle decision
Vehicle interaction
$x > x_{max}$
exit
$|y - y'| \leq 1$
collision
$< x, y, \dot{x}, \dot{y} >$
$< x, y', \dot{x}', \dot{y}', \ddot{x}, \ddot{y}, m >$
end
$(x, y) \notin Ways$
out
$< x, y, \dot{x}, \dot{y}, \ddot{x}, \ddot{y}, k >$

Figure 5.13: Entrance ramp model

transition `update_other` produces an updated token with new speed and position in place `inProcess`. Then, in the next simulation step, all tokens from this place are moved back to `otherVehicles`. As the definition of the behaviour of other vehicles is part of the definition of the scenario, it is described in Section 5.2.3.

Movement of controlled vehicle. The controlled vehicle is described using a token of color domain `SelfVehicle`. Transition `update_self` updates the lateral and longitudinal speed and position of the controlled vehicle via the `move_self` C function shown Figure 5.14.

```
int newposition = ((int) (b.P->x).c0) + ((int) (b.P->dx).c0);
int newspeed = ((int) (b.P->dx).c0) + (((int) (b.P->ddx.c0)) - 1);
Marking.P->_PL_selfVehicle += (SelfVehicle_Token( (
    PosX_Color_Classe) newposition, b.P->y, (VitX_Color_Classe)
    newspeed, b.P->dy, b.P->ddx, b.P->ddy));
```

Figure 5.14: Function `move_self` for the Entrance Ramp

Controlled vehicle decision. This single transition represents the call to the controller. We now consider that the vehicle can either see all the vehicles of the section via, for instance, a radar (as the section is fairly small) or get this information via communication with the motorway. The `controller` can obtain through the `orderedVehicle` function a vector of vehicles that are ordered by increasing lateral position, increasing longitudinal position, then incresing lateral and longitudinal speed. The `controller` function being the target of the evaluation, it is described in Section 5.2.3.

Vehicle Interaction. This part of the net contains the transitions removing the vehicles from the simulation in two situations:

- Transition `exit` (respectively `end`) is fired when the vehicle longitudinal position is higher than $N_{pos} - v_{max}$ and removes the environment (respectively controlled) vehicle from the simulation;
- Transition `out` is fired when the position of the controlled vehicle is outside the segment (in the case it does not leave the entrance ramp on time);
- Transition `collision` is fired when any the environment vehicle has the same longitudinal position and either the same lateral position, or a lateral position that is one higher or lower.

5.2.2 Specification of the Performance Indices

A trajectory ends when either `end`, `out` or `collision` is fired. The three HASL formulas shown in Figure 5.15 are based on the single automaton, which sets the appropriate variable to `true`, according to the outcome of the simulation.

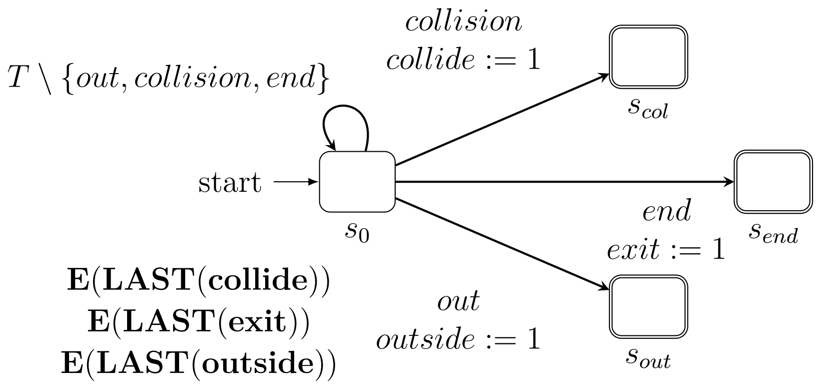

$T \setminus \{out, collision, end\}$
start
s_0
$collision$
$collide := 1$
s_{col}
end
$exit := 1$
s_{end}
out
$outside := 1$
s_{out}
E(**LAST**(**collide**))
E(**LAST**(**exit**))
E(**LAST**(**outside**))

Figure 5.15: The HASL formulas for the probabilities of each outcome

These three performance indices only measure the kind of collision (if any) that has happened. A fourth performance index, **E**(**LAST**(`CollForce`)) is added, measuring the severity of the collision. The value of `CollForce` is defined as follows:

- if the controlled vehicle reaches the end of the motorway section without collision, `CollForce` is set to 0;
- if the controlled vehicle goes outside the motorway segment, `CollForce` is set to $1 + v_x^2$ whwere v_x is the longitudinal speed of the vehicle;
- if the controlled vehicle collides with another vehicle w, `CollForce` is set to $1 + |w_x - v_x|^2$ where v_x and w_x are their respective longitudinal speeds.

This choice of values should give a decent approximation: the collision is not too dangerous at a low speed, but the higher the relative speed, the (quadratically) greater the damages are.

5.2.3 Evaluation

The first objective of this benchmark is to compare the performances of two controllers in various situations. As explained above, we measure the probability of a collision (I_1), the probability of leaving the entrance ramp unproperly (I_2), the probability the entering motorway properly (I_3) and the severity of a collision (I_4). We also measure the probability of a timeout (TO) where the controlled vehicle fails to enter the motorway into the 500 simulation steps. The second objective is to quantify the influence of some parameters on the simulation time, in order to check the scalability of our approach. These parameters are: the number of vehicles on the motorway, the presence of vehicles on the entrance ramp, and whether a new environment vehicle is inserted after an exit.

Controller. Let us now detail the two controllers. The first one tracks two vehicles, V_{front} and V_{behind}, the vehicle directly behind V_{front} on the target lane. Then, it compares their positions and speeds, before either choosing a new *front* vehicle or starting the entrance process. This entrance process also requires that the controlled vehicles accelerates or decelerates up to the front vehicle speed. It is initiated if the front vehicle is ahead of the controlled vehicle and the speed is reached. These tasks are described more thoroughly later on.

The second controller is built over the first one, trying to avoid collisions with other vehicles in the Entrance Ramp by accelerating if the vehicle behind is too fast and close, and decelerating if the vehicle ahead is too slow and close. Moreover, when the barrier is too close, it changes its strategy and avoids colliding with a barrier by slowing down to zero.

Environment vehicles. The behaviour of the environment vehicles is more refined here than in the first case study. We assign a *mode* to each of these vehicles. Let us describe these modes:

0. In this (default) mode, the vehicle adjusts its speed to the *target* vehicle (the vehicle ahead). If this target vehicle is far away, then it adjusts its speed to the expected speed of the segment. It only does so by unit increments or decrements. This models the expected behaviour on the motorway.
1. In this mode, the vehicle does not check anything and continues at its current speed. This models, for example, a micro-sleep of its driver.
2. In this mode, the vehicle decelerates if its speed is above two cells per time unit. This could model the driver being interrupted by a phone call.
3. In this mode, the vehicle changes lane to the right lane of the motorway segment. This is used in the two following cases: either the vehicle is on the left lane and is preparing for the next exit, or the vehicle is also in the entrance ramp and is preparing to enter the motorway.

The choice of the current mode is randomized (see Figure 5.16) in the following way:

- The starting mode is mode 0 (speed adjustment);
- If the vehicle is currently in the entrance ramp at position x, then it switches to mode 3 (lane change) with probability $\frac{(x-c_1+1)}{c_2-c_1+1}$ (meaning that the further it is to the end of the entrance ramp, the higher is the chance of going into the motorway segment). We set c_1 to 40 (the first position where the vehicle may

```
if (cmode == 0) {
    if (voie > 9) {
        rdmval=40 + rand()%41; /* random between 0 and 40 */
        if (position > rdmval) { } else { cmode=3; }
    } else {
        rdmval=rand()%200;
        if (rdmval==0) { cmode=2; }
        else if (rdmval < 5) { cmode=1; }
    else if (rdmval < 7 && voie!=7) { cmode=3; }
    }
} else if (cmode == 1) { // Sleep
    rdmval=rand()%4;
    if (rdmval==0) { cmode=0; }
} else if (cmode == 2) { // Phone call
    rdmval=rand()%5;
    if (rdmval==0) { cmode=0; }
} else if (cmode == 3) { // Lane Change
    if (voie==7) { cmode=0; }
}
```

Figure 5.16: Mode change function

enter) and c_2 to 80 (the last position of the full entrance ramp), according to the parameters shown in Figure 5.12;

- If the vehicle is currently in mode 3 (lane change), it deterministically goes back to the mode 0, only if it has reached the destination lane;
- If the vehicle is in mode 1 (micro-sleep), there is at each time step a probability 25% to return in default mode. Similarly, in mode 2 (phone call), we assign a probability 20%.
- If the vehicle is in mode 0, it can change its mode either to mode 1 (with probability 2.5%), mode 2 (with probability 0.5%), or mode 3 (with probability 1%, only if it is not yet in the middle of the right lane).

Moreover, we consider two possible situations for the other vehicles:

- In the first situation, a vehicle leaving the motorway from the main lanes is replaced by another one entering it (with the same ID, same speed, and same lane). This keeps the number of vehicle constants. This situation is depicted with the ○ pictogram in the tables;
- In the second case, a vehicle leaving the motorway from the main lanes is simply removed.

Detailed controller specification. The controllers perform four consecutive tasks, the current one being stored in the `currstate` variable as described below:

0. accelerating in the entrance ramp before the entrance location;

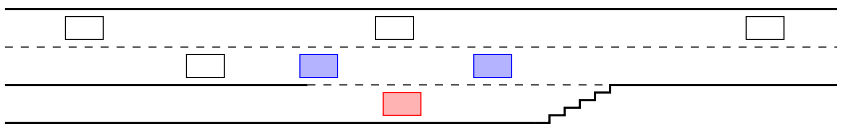


Figure 5.17: The two target vehicles

1. synchronising with the vehicules inside the segment;
2. entering into the motorway segment;
3. driving in the motorway segment.

In order to perform these tasks, the controller maintains two variables: `targetspeed` the current target longitudinal speed, and `targetway` the current target lateral position. These tasks rely on finding two vehicles between which the controlled vehicle will enter the motorway segment (see Figure 5.17); the two target vehicles identifiers will be stored in `targetfront` and `targetbehind`. The procedure is described in Algorithm 7. The controller enumerates the vehicles with respect to the decreasing longitudinal position, until the vehicle A is behind the controlled vehicle and there is enough space between A and B the vehicle in front of A. The variables `targetbehind` and `targetfront` are respectively set to A and B.

Algorithm 7: The `targetChoice` procedure

```
Data: the ordered vehicle vector vehicles
Result: The two target vehicles targetfront and targetbehind
1  firstvehicle ← 1 ;
2  targetfront ← 0 ;
3  targetbehind ← 0 // Id of controlled vehicle
4  while there is another vehicle vehicle in vehicles do
5      if vehicle.py ∈ [5;10] then
           // The vehicle is at least partly in target lane
6          if firstvehicle then
7              targetbehind ← vehicle
8          else
9              targetfront ← targetbehind ;
10             targetbehind ← vehicle ;
               // We are checking the vehicles in decreasing px
                  order:  the new vehicle is behind the previous.
11             if targetfront.px − targetbehind.px ≥ 3 and
                targetbehind.px < px then  return;
               // The distance is good enough for an insertion.
```

Algorithm 8: Tasks of the controller

```
   Data: The target vehicles targetfront and targetbehind
   Result: A target longitudinal speed targetspeed and lateral position
           targetway
 1 if currstate == 0 then
       // Acceleration
 2     targetspeed ← targetfront.px ;
 3     targetway ← py // The current lateral position
 4     if px > $p_{\text{entrance}}$ then currstate ← 1 ;
 5 if currstate == 1 then
       // Synchronisation
 6     targetspeed ← targetfront.px ;
 7     targetway ← py ;
 8     if targetfront.px > px and vx = targetspeed then
        currstate ← 2 ;
       // If the front target is in front of the controlled
          vehicle and we have reached its speed, we can continue.
 9 if currstate == 2 then
       // Entrance
10     targetspeed ← targetfront.px ;
11     targetway ← 7 // The middle of the lane next to the ramp
12     if py == targetway then currstate ← 3 ;
13 if currstate == 3 then
       // Driving, similar to the Motorway Section.
14     targetspeed ← targetfront.px ;
15     targetway ← 7 ;
```

	Controller Version 1			
	10	10 $\nearrow$	10 $\bigcirc$ $\nearrow$	20 $\bigcirc$ $\nearrow$
I_1	$[0.016, 0.019]$	$[0.043, 0.047]$	$[0.041, 0.046]$	$[0.126, 0.133]$
I_2	$[0.053, 0.057]$	$[0.067, 0.073]$	$[0.070, 0.076]$	$[0.035, 0.039]$
I_3	$[0.925, 0.929]$	$[0.881, 0.887]$	$[0.880, 0.887]$	$[0.829, 0.838]$
I_4	$[0.016, 0.019]$	$[0.100, 0.112]$	$[0.100, 0.111]$	$[0.184, 0.197]$
TO	▨	▨	▨	▨
Simulations	151 000	81 000	83 000	71 000
Time	1 672s	1 014s	1 519s	3 601s

	Controller Version 2			
	10	10 $\nearrow$	10 $\bigcirc$ $\nearrow$	20 $\bigcirc$ $\nearrow$
I_1	$[0.080, 0.089]$	$[0.144, 0.155]$	$[0.182, 0.185]$	$[0.202, 0.220]$
I_2	▨	▨	▨	▨
I_3	$[0.793, 0.805]$	$[0.770, 0.783]$	$[0.815, 0.817]$	$[0.780, 0.798]$
I_4	$[0.259, 0.286]$	$[0.466, 0.504]$	$[0.601, 0.607]$	$[0.486, 0.537]$
TO	$[0.112, 0.121]$	$[0.069, 0.078]$	$[8.1 \cdot 10^{-4}, 9.3 \cdot 10^{-4}]$	🟧 $[0, 3.4 \cdot 10^{-4}]$
Simulations	37 000	34 000	2 000 000*	16 000
Time	515s	486s	39 041s	828s

🟧 Non meaningful ▨ Zero by construction

N^* Reached maximum simulation number

Table 5.2: Evaluation of controllers (with 99% confidence interval and 10% relative width)

The different tasks are described in Algorithm 8. During the acceleration phase, the controller synchronises its longitudinal speed to the front target longitudinal speed. It only moves to the next task after reaching the longitudinal position p_e.

Simulation parameters. The two previously described controllers are evaluated through four different scenarios, presented in order of increasing difficulty for the controller:

- the first one has 10 vehicles in the motorway segment;
- the second one has additional vehicles in the entrance ramp;
- in the third one, the enivronment vehicles are added back to the beginning of the motorway section when they have reached the end;
- in the fourth one, there are 20 vehicles in the motorway segment that are still added back, and the vehicles on entrance ramp are kept.

In Table 5.2, these scenarios are represented using symbols: the number indicates the number of vehicles in the motorway segment, the $\bigcirc$ symbol indicates that exiting vehicles are replaced and the $\nearrow$ symbol indicates that some environment vehicles are present in the entrance ramp.

Let us recall the performance indices:

- I_1 the probability of collision with another vehicle;
- I_2 the probability of collision with a barrier of the entrance ramp or the motorway segment (when the vehicle did not enter properly);
- I_3 the probability of a successful entrance in the motorway;
- I_4 the average severity of the collision as described in section 5.2.2;
- and TO the probability of a timeout (in the case where the vehicle stays at null velocity).

We have chosen to perform these evaluations with a 10% relative width, a 99% confidence interval, and a limit of 2 000 000 simulations for a single scenario, on a single machine with 12 Intel Xeon cores, using 8 parallel threads.

Controller evaluation. The results are shown in Table 5.2. On the easiest scenario, the first controller already fails to enter almost 8% of the time and often collides with the barrier. The examination of Algorithm 8 reveals that no check is performed in the synchronisation state on either the vehicle behind, or the closeness of the end of the entrance ramp. It only checks whether the front target is indeed in front of the vehicle.

The chance of collision increases as expected with respect to the scenario difficulty. More interestingly, the main factor is the number of the vehicles either in the main lane or the entrance ramp. For instance, the number of collisions more than doubles between the easiest and most difficult scenarios.

However, adding the possibility for vehicles to loop in the section does not increase the chances of a collision, except *slightly* increasing a barrier collision chance, because less place is available for the vehicle to enter. It is only when a lot more vehicles are added to the simulation that the chance of success drops to around 83% (17% of failure).

For the second controller, designed to avoid any barrier collision and reduce the collisions in the entrance ramp, the results are surprisingly worse than those of the first controller. Indeed, the controlled vehicle completely avoids barriers but this cost of a greatly increased collision probability. In addition, with a significant probability, the vehicle fails to enter the motorway segment within the 500 time units. Whatever the scenario, the probability of entrance success is now between 77% and 82%.

These results show that to avoid collisions with vehicles in the entrance ramp, or the barrier, requires more attention for the design of the controllers. Besides, the overall collision rates are far higher than the realistic ones. It remains to understand whether the scenarios are too pessimistic or the controllers are too basic.

Scalability evaluation. We have chosen a relative width since we expected that some performance indices could be very small. In fact, this occurs in two cases, corresponding to the two hardest scenarios for the second controller. In the first case (10 $\bigcirc$ $\nearrow$) we stop without reaching the 10% relative width because of the 2 000 000 simulation limit. This case illustrates the rare event phenomenon.

In the other case, the simulation stops when reaching the 10% relative width on the three other performance indices, because Cosmos entirely ignores, for its termination condition, a performance index whose simulated mean is zero. It is very likely that *rare events* are associated with this performance index: the lower bound is zero, since no simulation has timed out; however, this simulation cannot guarantee that it never happens. This would require a further study.

With only 10 environment vehicles, 90 simulations are performed per second, which is more than in the Motorway Segment case study. Progressively adding difficulties, it drops to 79 per second with up to two vehicles in the entrance ramp, then to 54 per second with the vehicles being allowed to loop. The worst case scenario has only 19 simulation per second, performed with 6 cores, which is only a third of the simulation speed of the first case study.

Possible Improvements. The chances of collision are really high for the case study: over such a segment of 120 cells (which is around 840 meters) the collision rate is approximately 10^{-9}. None of the simulations have produced a better entrance rate than 93%. We have already dicsussed that the first controller was rough, not even trying to notice whether other vehicles are interacting in the entrance ramp. The discussion could be extended to the second controller, for which the additional checks to avoid barrier collisions lead to a greater vehicle collision rate than the sum of collision rates of the first controller. However, the design of controllers is outside the range of the thesis, and we must limit to some improvement ideas: a new controller could improve its estimation of the next positions of the target vehicles, and allow for acceleration *during* a manoeuver.

We have only roughly modeled the behaviour of the other vehicle: in real situation, a vehicle on the motorway might decide to slightly go on the left to ease the entrance of a vehicle in the entrance ramp. This, with the controller slightly enforcing its entrance before the end of the entrance ramp, could help reducing the number of collisions.

CONCLUSION AND PERSPECTIVES

Summary

Based on several case studies of autonomous vehicle situations, we have identified the need for an expressive formalism of probabilistic hybrid systems. Rather than building it from scratch, we decided to combine stochastic high-level Petri nets with Simulink. The interest of high-level stochastic Petri nets comes from the fact that it is supported by the performant tool COSMOS, while Simulink is a *de facto* standard for the design of vehicle controllers. Furthermore, the features of the two models are complementary. Generalising our objectives, we proposed to follow a multi-model approach to combine several formalisms. In order to empirically validate this approach by simulations over examples, this multi-model formalism had to be integrated into COSMOS. This required several steps:

- providing a formal syntax and exact semantics for (a significant fragment of) Simulink. The block-diagram models are transformed into a set of differential equations over subintervals of the simulation time, taking into account the modes related to threshold crossings and the delays of blocks with latencies;
- providing an operational semantics intended to accurately approximate the exact semantics, when the trajectory exists. Our approach relies on several approximation methods, such as Runge-Kutta integration method, zero-crossing detection, and linear interpolation for latencies;
- implementing three important upgrades in COSMOS:
  - adding a new binding mechanism for high-level Petri nets, that largely improves the performance for nets with large color classes but small number of tokens;
  - implementing the operational semantics of Simulink;
  - adding hooks for multi-model simulation giving a communication framework between different models. In the case of stochastic net and Simulink co-simulation, these hooks were instanciated as special transitions in the net.

  All these upgrades were illustrated and tested on several toy examples.

We finally proposed two significant case studies for the validation of this approach, in the context of the autonomous vehicle. The first one is a motorway with heavy traffic, and the second one is an entrance ramp to a motorway.

Perspectives

Our perspectives are organised from short to long term.

Short term perspectives. On the theoretical side, we plan to establish the approximation conjecture possibly with modifications of the current hypotheses. On the implementation side, while we have formally described Stateflow Charts in our Simulink semantics, they are not yet integrated into Cosmos. Concerning case studies, we plan to extend our models to other interesting simple situations: for instance, the presence of pedestrians or lane reductions.

Middle term perspectives. We have defined the communication between models in an *ad hoc* way, which so far only applies to stochastic high-level nets and Simulink. We plan to define, or use, a *pivot formalism* to combine several models that have been designed using various formalisms. This could be based on the FMI[2] [40] standard of the Modelica Association, in a manner similar to its application to Uppaal [51].

Cosmos has been previously used for the verification of DNA Walkers Circuits [19]. So the work performed during this thesis could be also used in bioinformatic for more involved case studies. It would require to introduce a dedicated formalism for designing, for example, regulation circuits, and instanciate them in the multi-model framework.

Long term perspective. In the IRT SystemX project SVA [68], Wei Chen started in september 2016 a thesis called *Formal Models for the Conceptualization and Caracterisation of Use Cases for the Autonomous Vehicle* to provide a formalism to generate a large number of situations to be tested [29]. The creation of a tool for generating scenarios from this new formalism, and the analysis of such scenarios, could be the subject of another thesis.

[2]Fonctional Mock-Up Interface

BIBLIOGRAPHY

[1] Parosh Aziz Abdulla and Aletta Nylén. *Timed Petri Nets and BQOs*, pages 53–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[2] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electr. Notes Theor. Comput. Sci.*, 109:43–56, 2004.

[3] Matthias Althoff, Daniel Althoff, Dick Wollherr, and Matthias Buss. Safety verification of autonomous vehicles for coordinated evasive maneuvers. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 1078–1083, June 2010.

[4] Matthias Althoff, Olaf Stursberg, and Martin Buss. Model-based probabilistic collision detection in autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 10(2):299–310, June 2009.

[5] Elvio Gilberto Amparore, Benoit Barbot, Marco Beccuti, Susanna Donatelli, and Giuliana Franceschinis. Simulation-based verification of hybrid automata stochastic logic formulas for stochastic symmetric nets. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, pages 253–264, New York, NY, USA, 2013. ACM.

[6] Mikael Asplund, Atif Manzoor, Mélanie Bouroche, Siobhán Clarke, and Vinny Cahill. A formal approach to autonomous vehicle coordination. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2012.

[7] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Model-checking continuous-time markov chains. *ACM Trans. Comput. Logic*, 1(1):162–170, July 2000.

[8] Soheib Baarir, Marco Beccuti, Davide Cerotti, Massimiliano De Pierro, Susanna Donatelli, and Giuliana Franceschinis. The greatspn tool: Recent enhancements. *SIGMETRICS Perform. Eval. Rev.*, 36(4):4–9, March 2009.

[9] C. Baier, L. Cloth, B. Haverkort, M. Kuntz, and M. Siegle. Model checking action- and state-labelled Markov chains. *IEEE Trans. on Software Eng.*, 33(4):701–710, 2007.

[10] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. On the logical characterisation of performability properties. In *ICALP'00*, LNCS 1853, pages 780–792, 2000.

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

[12] Paolo Ballarini, Benoît Barbot, Marie Duflot, Serge Haddad, and Nihal Pekergin. HASL: A new approach for performance evaluation and model checking from concepts to experimentation. *Performance Evaluation*, 90:53–77, August 2015.

[13] Paolo Ballarini, Hilal Djafri, Marie Duflot, Serge Haddad, and Nihal Pekergin. COSMOS: A statistical model checker for the hybrid automata stochastic logic. In *Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, Aachen, Germany, 5-8 September, 2011*, pages 143–144. IEEE Computer Society, 2011.

[14] Benoît Barbot. *Acceleration for statistical model checking*. Theses, École normale supérieure de Cachan - ENS Cachan, November 2014.

[15] Benoît Barbot, Béatrice Bérard, Yann Duplouy, and Serge Haddad. Statistical Model-Checking for Autonomous Vehicle Safety Validation. In *SIA Simulation Numérique*, Montigny-le-Bretonneux, France, March 2017. Société des Ingénieurs de l'Automobile.

[16] Benoît Barbot, Béatrice Bérard, Yann Duplouy, and Serge Haddad. Integrating simulink models into the model checker cosmos. In *Application and Theory of Petri Nets and Concurrency - 39th International Conference, PETRI NETS 2018, Bratislava, Slovakia, June 24-29, 2018, Proceedings*, pages 363–373, 2018.

[17] Benoît Barbot, Taolue Chen, Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre. Efficient ctmc model checking of linear real-time objectives. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 128–142, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[18] Benoît Barbot, Serge Haddad, and Claudine Picaronny. Coupling and importance sampling for statistical model checking. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[19] Benoît Barbot and Marta Kwiatkowska. On quantitative modelling and verification of dna walker circuits using stochastic petri nets. In Raymond Devillers and Antti Valmari, editors, *Application and Theory of Petri Nets and Concurrency*, volume 9115 of *Lecture Notes in Computer Science*, pages 1–32. Springer International Publishing, 2015.

[20] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 years. *Softw. Pract. Exper.*, 41(2):133–142, February 2011.

[21] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78(3):877 – 910, 2012.

[22] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[23] Bernard Berthomieu and Michel Diaz. Modeling and Verification of Time Dependent Systems using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, Mar 1991.

[24] Olivier Bouissou and Alexandre Chapoutot. An operational semantics for simulink's simulation engine. In *Proceedings of the 13th ACM SIGPLAN/SIGBED*, LCTES '12, pages 129–138, New York, NY, USA, 2012. ACM.

[25] Béatrice Bérard, Serge Haddad, Lom-Missan Hillah, Fabrice Kordon, and Yann Thierry-Mieg. Collision avoidance in Intelligent Transport Systems: towards an application of control theory. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 346–351, May 2008.

[26] H. Cartan. *Calcul différentiel.* Collection Méthodes. Hermann, 1971.

[27] T. Chen, T. Han, J.-P. Katoen, and A. Mereacre. Quantitative model checking of CTMC against timed automata specifications. In *Proc. LICS'09*, pages 309–318, 2009.

[28] Taolue Chen, Marta Diciolla, Marco an Kwiatkowska, and Alexandru Mereacre. Time-bounded verification of ctmcs against real-time specifications. In *9th International Conference, FORMATS 2011*, Lecture Notes in Computer Science, pages 26–42. Springer, 2011.

[29] Wei CHEN and Leila Kloul. An Ontology-based Approach to Generate the Advanced Driver Assistance Use Cases of Highway Traffic. In *10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management.*, Seville, Spain, September 2018.

[30] Cécile Chevallier. Essonne : bientôt des navettes sans chauffeur entre Massy et le plateau de Saclay. *Le Parisien*, 2018.

[31] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Trans. Comput.*, 42(11):1343–1360, November 1993.

[32] Yuan S Chow and Herbert Robbins. On the asymptotic theory of fixed-width sequential confidence intervals for the mean. *The Annals of Mathematical Statistics*, pages 457–462, 1965.

[33] CJ Clopper and Egon S Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, pages 404–413, 1934.

[34] Hilal Djafri. *Numerical and statistical approaches for model checking of stochastic processes.* Theses, École normale supérieure de Cachan - ENS Cachan, June 2012.

[35] S. Donatelli, S. Haddad, and J. Sproston. Model checking timed and stochastic properties with CSL^{TA}. *IEEE Trans. on Software Eng.*, 35:224–240, 2009.

[36] S. Donatelli, S. Haddad, and J. Sproston. Model Checking Timed and Stochastic Properties with $\mathrm{CSL}^{\mathrm{TA}}$. *IEEE Trans. Software Eng.*, 35(2):224–240, 2009.

[37] Andreas Eidehall and Lars Petersson. Statistical threat assessment for general road scenes using monte carlo sampling. *IEEE Transactions on Intelligent Transportation Systems*, 9(1):137–147, March 2008.

[38] EUROpean New Car Assessment Programme. `https://www.euroncap.com/en`, 1997.

[39] E. Fehlberg. *Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems.* NASA technical report. National Aeronautics and Space Administration, 1969.

[40] Functional Mock-up Interface. `https://fmi-standard.org/`, 2010.

[41] Andreas Gaiser and Stefan Schwoon. Comparison of algorithms for checking emptiness on buechi automata. *CoRR*, abs/0910.3766, 2009.

[42] Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 53–65, London, UK, UK, 2001. Springer-Verlag.

[43] R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors. *Hybrid systems*, volume 736 of *LNCS*. Springer, 1993.

[44] Andrew J. Hawkins. Waymo is first to put fully self-driving cars on US roads without a safety driver. *The Verge*, 2017.

[45] Ru He, Paul Jennings, Samik Basu, Arka P. Ghosh, and Huaiqing Wu. A bounded statistical approach for model checking of unbounded until properties. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 225–234, New York, NY, USA, 2010. ACM.

[46] Monika Heiner, Christian Rohr, and Martin Schwarick. Marcie – model checking and reachability analysis done efficiently. In José-Manuel Colom and Jörg Desel, editors, *Application and Theory of Petri Nets and Concurrency*, pages 389–399, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[47] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[48] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[49] Cyrille Jegourel, Axel Legay, and Sean Sedwards. A PLatform for High Performance Statistical Model Checking – PLASMA. In *Proceedings of the*

18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'12, pages 498–503, Berlin, Heidelberg, 2012. Springer-Verlag.

[50] A. Jensen. Markov chains as an aid in the study of markov processes. *Skand. Aktuarietidskrift*, 3:87–91, 1953.

[51] P. G. Jensen, K. G. Larsen, A. Legay, and U. Nyman. Integrating Tools: Co-simulation in UPPAAL using FMI-FMU. In *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 11–19, Nov 2017.

[52] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic Symbolic Model Checker. In *Computer Performance Evaluation: Modelling Techniques and Tools*, 2002.

[53] L.Cloth, J.-P. Katoen, M. Khattri, and R. Pulungan. Model checking Markov reward models with impulse rewards. In *Proc. DSN'05*, 2005.

[54] Milecia Matthews, Girish Chowdhary, and Emily Kieson. Intent communication between autonomous vehicles and pedestrians. *CoRR*, abs/1708.07123, 2017.

[55] Ministère de l'Équipement, des Transports, du Logement, du Tourisme et de la Mer. *Conception des accès sur Voies Rapides Urbaines de Type A (VRU A).* CERTU, 2003.

[56] Marvin L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[57] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Universität Hamburg, 1962.

[58] Prism – case studies. `https://www.prismmodelchecker.org/casestudies/`.

[59] Akshay H. Rajhans. *Multi-Model Heterogeneous Verification of Cyber-Physical Systems.* PhD thesis, Carnegie Mellon University, 2013.

[60] R. H. Rasshofer, M. Spies, and H. Spies. Influences of weather phenomena on automotive laser radar systems. *Advances in Radio Science*, 9:49–60, 2011.

[61] Reuters. Tesla's Autopilot to get 'full self-driving feature' in August. *Reuters*, 2018.

[62] RFI. France: Emmanuel Macron veut faciliter les tests de véhicules autonomes. *RFI*, 2018.

[63] Albert Rizaldi, Sebastian Sontges, and Matthias Althoff. On time-memory trade-off for collision detection. In *2015 IEEE Intelligent Vehicles Symposium, IV 2015, Seoul, South Korea, June 28 - July 1, 2015*, pages 1173–1180. IEEE, 2015.

[64] Gerardo Rubino and Bruno Tuffin. *Rare Event Simulation Using Monte Carlo Methods*. Wiley Publishing, 2009.

[65] Yassmina Saadna and Ali Behloul. An overview of traffic sign detection and classification methods. *International Journal of Multimedia Information Retrieval*, 6(3):193–210, Sep 2017.

[66] M. L. Sichitiu and M. Kihl. Inter-vehicle communication systems: a survey. *IEEE Communications Surveys Tutorials*, 10(2):88–105, Second 2008.

[67] Zehang Sun, G. Bebis, and R. Miller. On-road vehicle detection: a review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(5):694–711, May 2006.

[68] Simulation of Autonomous Vehicle Safety – IRT SystemX. `https://www.irt-systemx.fr/en/project/sva/`, 2015.

[69] Ashish Tiwari. Formal Semantics and Analysis methods for Simulink Stateflow Models. Technical report, SRI, 2002.

[70] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embedded Comput. Syst.*, 4(4):779–818, 2005.

[71] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 327–338, Washington, DC, USA, 1985. IEEE Computer Society.

[72] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[73] A. Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 06 1945.

[74] Håkan L. S. Younes. Ymer: A statistical model checker. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 429–433, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[75] Yan Zhang. *Conception Semi-automatique de Contrôleurs avec VeriJ*. PhD thesis, Sorbonne Université, 2013.

[76] Paolo Zuliani, André Platzer, and Edmund M. Clarke. Bayesian statistical model checking with application to stateflow/simulink verification. *Formal Methods in System Design*, 43(2):338–367, 2013.

Titre : Application des méthodes formelles au contrôle du véhicule autonome

Mots clés : Simulink, véhicule autonome, méthodes formelles, model-checking statistique

Résumé : Cette thèse s'inscrit dans le cadre de la conception de véhicules autonomes, et plus spécifiquement de la vérification de contrôleurs de tels véhicules. Nos contributions à la résolution de ce problème sont les suivantes : (1) fournir une syntaxe et une sémantique pour un modèle de systèmes hybrides, (2) étendre les fonctionnalités du model checker statistique Cosmos à ce modèle et (3) valider empiriquement la pertinence de notre approche sur des cas d'étude typiques du véhicule autonome.
Nous avons choisi de combiner le modèle des réseaux de Petri stochastiques de haut niveau (qui était le formalisme d'entrée de Cosmos) avec le formalisme d'entrée de Simulink afin d'atteindre un pouvoir d'expression suffisant. En effet Simulink est très largement utilisé dans le domaine automobile et de nombreux contrôleurs sont spécifiés avec cet outil. Or Simulink n'a pas de sémantique formellement définie. Ceci nous a conduit à concevoir une telle sémantique en deux temps : tout d'abord en introduisant une sémantique dite exacte mais qui n'est pas opérationnelle puis en la complétant par une sémantique approchée intégrant le facteur d'approximation recherché.
Afin de combiner le modèle à événements discrets des réseaux de Petri et le modèle continu spécifié en Simulink, nous avons proposé au niveau syntaxique une interface reposant sur de nouveaux types de transitions et au niveau sémantique une extension de la boucle de simulation. L'évaluation de ce nouveau formalisme a été entièrement implémentée dans Cosmos.
Grace à ce nouveau formalisme, nous avons développé et étudié les deux cas d'étude suivants : d'une part une circulation dense sur une section d'autoroute et d'autre part l'insertion du véhicule dans une voie rapide. L'analyse des modélisations correspondantes a démontré la pertinence de notre approche.

Title : Applying Formal Methods to Autonomous Vehicle Verification

Keywords : Simulink, Autonomous Vehicle, Formal Methods, Statistical Model-Checking

Abstract : This thesis takes place in the context of autonomous vehicle design, and concerns more specifically the verification of controllers of such vehicles. Our contributions are the following : (1) give a syntax and a semantics for a hybrid system model, (2) extend the capacities of the model-checker Cosmos to that kind of models, and (3) empirically confirm the relevance of our approach on typical case studies handling autonomous vehicles.
We chose to combine high-level stochastic Petri nets (which is the input formalism of Cosmos) with the input formalism of Simulink, to obtain an adequate expressive power. Indeed, Simulink is largely used in the automotive industry and numerous controllers have been specified using this tool. However, there is no formal semantics for Simulink, which lead us to define such a semantics in two steps : first, we propose an exact (but not operational) semantics, then we complete it by an approximate semantics that includes the targeted approximation level.
In order to combine the discrete event model of Petri nets and the continous model specified in Simulink, we define a syntactic interface that relies on new transition types ; its semantics consists of an extension of the simulation loop. The evaluation of this new formalism has been entirely implemented into Cosmos.
Using this new formalism, we have designed and studied the two following case studies : on one hand, a heavy traffic on a motorway segment, and on the other hand the insertion of a vehicle into a motorway. Our approach has been validated by the analysis of the corresponding models.