Reachability analysis with Lebesgue-integrable time-varying uncertainties

François Bidet

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2022IPPAX060

Thèse de doctorat

Reachability analysis with Lebesgue-integrable time-varying uncertainties

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à l'École polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 06 septembre 2022, par

FRANÇOIS BIDET

Composition du Jury :

Pierre-Loïc Garoche Professeur, École nationale de l'aviation civile (LII)	Président
Thao Dang Directrice de recherche, CNRS (VERIMAG)	Rapporteure
Nacim Ramdani Professeur, Université d'Orléans (PRISME)	Rapporteur
Timothy Bourke Chargé de recherche INRIA, École normale supérieure - PSL (PARKAS)	Examinateur
Éric Goubault Professeur, École polytechnique (COSYNUS)	Directeur de thèse
Sylvie Putot Professeure, École polytechnique (COSYNUS)	Co-directrice de thèse
Marc Pouzet Professeur, École normale supérieure - PSL (PARKAS)	Invité

Notations

notation	**meaning**
capital letters, *e.g.* X, M	(depending on the context) sets, matrices, or set-valued functions
$A \setminus B$	(with two sets A and B) set of elements in A that are not in B: $x \in A \setminus B \Leftrightarrow \Big(x \in A \wedge x \notin B\Big)$
$A \subset B$	(with two sets A and B) the set A is a subset of the set B: $x \in A \implies x \in B$
$A \times B$	(with two sets A and B) the cartesian product of the sets A and B: $(a \in A \wedge b \in B) \Leftrightarrow (a, b) \in A \times B$
$\mathcal{P}(A)$	power set of the set A, *i.e.* set of all subsets of A: $B \subset A \Leftrightarrow B \in \mathcal{P}(A)$
$\mathrm{CH}(A)$	convex hull of the set A, *i.e.* smallest set such that $(A \subset \mathrm{CH}(A)) \wedge (\forall (a_1, a_2, \delta) \in \mathrm{CH}(A) \times \mathrm{CH}(A) \times [0,1], \delta a_1 + (1-\delta) a_2 \in \mathrm{CH}(A))$
$\mathrm{Int}(A)$	interior of the set A, *i.e.* biggest open set such that $\mathrm{Int}(A) \subset A$
$\overline{A}$	closure of the set A, *i.e.* smallest closed set such that $A \subset \overline{A}$
$\mathbb{N}$	set of all natural numbers
$\mathbb{N}^*$	set of all positive natural numbers, *i.e.* $\mathbb{N}^* = \mathbb{N} \setminus \{0\}$
$\mathbb{Z}$	set of all integers
$\mathbb{R}$	set of all real numbers
$\mathbb{R}_+^*$ or $\mathbb{R}_{>0}$	the set of positive real numbers
$\mathbb{R}_+$ or $\mathbb{R}_{\geq 0}$	the set of non-negative real numbers

notation	meaning
$\lceil v \rceil$	least integer bigger than the real number v: $\lceil v \rceil \in \mathbb{Z} \wedge v \leq \lceil v \rceil \wedge \big(\forall i \in \mathbb{Z},\ (v \leq i \implies \lceil v \rceil \leq i)\big)$
$[a, b]$	interval of real numbers bigger than a and smaller than b: $\forall x \in \mathbb{R},\ x \in [a, b] \Leftrightarrow a \leq x \leq b$
$[a, b[$	$[a, b] \setminus \{b\}$
$]a, b]$	$[a, b] \setminus \{a\}$
$]a, b[$	$[a, b] \setminus \{a, b\}$
$[\![a, b]\!]$	set of integers bigger than a and smaller than b: $i \in [\![a, b]\!] \Leftrightarrow (i \in \mathbb{N} \wedge i \in [a, b])$
$\lvert x \rvert$	absolute value of the real number x
$\lVert M \rVert$	(depending on the context) norm of the matrix (or vector) M
$\mathcal{C}(A, B)$	set of all the continuous functions from A to B
$\forall_{a.e.}\, t \in T$	for almost all t in the set T
$\int_a^b f(s)\, ds$	(Riemann- or Lebesgue-)integration of f over $[a, b]$ if $f(s)$ is a vector, the integration is componentwise
$\dot{f}$	derivative of the function f
$\dot{x}$	derivative with respect to the time of the function associated to the variable x
x'	value of the signal associated to the variable x after a jump
$f \circ g$	composition of a function f with a function g $\forall x,\ (f \circ g)(x) = f\big(g(x)\big)$
$\mathrm{width}(s)$	width of a unidimensional convex set $s \subset \mathbb{R}$: $\mathrm{width}(s) = \inf\{b - a \mid s \subset [a, b]\}$
$\mathit{interval}(s)$	smallest enclosing vector of intervals of the set $s \subset \mathbb{R}^n$: $\mathit{interval}(s) = \big([a_1, b_1], \ldots, [a_n, b_n]\big)^\top$ with $a_i = \inf(s_i)$ and $b_i = \sup(s_i)$
$\int p\, dt$	primitive of the function (or polynomial) p with respect to the variable t with null additive constant

Résumé

Cette thèse présente des méthodes pour effectuer l'analyse d'atteignabilité de systèmes dynamiques. Elle se concentre sur le calcul de sur-approximations de l'ensemble d'états atteignables en présence d'incertitudes intégrables au sens de Lebesgue ou de comportements Zénon.

Condition suffisante de sur-approximation dans le cas non-linéaire

Alors que plusieurs méthodes existent pour effectuer une analyse d'atteignabilité de systèmes continus définis comme systèmes d'équations différentielles ordinaires explicites, peu d'entre elles sont conçues pour gérer les incertitudes intégrables au sens de Lebesgue, ce qui est un des cas les plus généraux de signaux. Nous prouvons dans la section 3.2 que les méthodes basées sur des contractions d'ensembles par des versions ensemblistes des opérateurs de Picard associés aux problèmes produisent des sur-approximations valides même dans le cas d'incertitudes intégrables au sens de Lebesgue.

En particulier, nous considérons le cas d'un problème avec une dérivée définie presque partout

$$\begin{cases} \forall_{a.e.} t \in [t_0, t_1],\ \dot{x}(t) = f\big(t, x(t), p, u(t)\big) \\ \forall t \in [t_0, t_1],\ u(t) \in U \\ x(t_0) = x_0 \in X_0 \subset \mathbb{R}^n \\ p \in P \end{cases} \tag{1}$$

avec f une fonction continue, U, X_0 et P des ensembles bornés. De plus, nous faisons l'hypothèse que pour chaque état initial $x_0 \in X_0$, paramètre $p \in P$ et incertitude intégrable au sens de Lebesgue u, le système admet une unique solution. Pour toute incertitude x_0, p et u respectant les contraintes, l'opérateur de Picard est défini par

$$\mathbb{P}_{x_0,p,u}(y) = t \mapsto x_0 + \int_{t_0}^{t} f\big(s, y(s), p, u(s)\big)\, ds \tag{2}$$

Nous en définissons une version sur l'ensemble des fonctions ensemblistes de $[t_0, t_1]$ dans l'ensemble des parties de $\mathbb{R}^n$:

$$\mathbb{P}_{x_0,p,u}(\varphi) = \Big\{ \mathbb{P}_{x_0,p,u}(y) \;\Big|\; y \in \mathcal{C}([t_0, t_1], \mathbb{R}^n) \wedge \forall t \in [t_0, t_1],\ y(t) \in \varphi(t) \Big\} \tag{3}$$

Finalement, nous définissons une version ensembliste qui regroupe toutes les possibilités d'incertitudes u :

$$\mathbb{P}_{x_0,p}(\varphi) = t \mapsto \bigcup_{\substack{u \text{ intégrable} \\ \forall t \in [t_0,t_1],\ u(t) \in U}} \mathbb{P}_{x_0,p,u}(\varphi)(t) \tag{4}$$

Une telle version de l'opérateur de Picard peut par exemple être obtenue avec l'arithmétique des intervalles (ou celle des modèles de Taylor) en remplaçant les occurrences de chaque composante de u par un intervalle sur-approximant son image [45, section 3.5].

Une fonction ensembliste φ de l'intervalle de temps $[t_0, t_1]$ dans l'ensemble $\mathcal{P}(\mathbb{R}^n)$ des sous-ensembles de $\mathbb{R}^n$ est contractée par $\mathbb{P}_{x_0,p}$ si

$$\forall t \in [t_0, t_1],\ \mathbb{P}_{x_0,p}(\varphi)(t) \subset \varphi(t) \tag{5}$$

Nous prouvons que si une fonction ensembliste $\varphi_{x_0,p}$ est contractée par $\mathbb{P}_{x_0,p}$, alors elle définit une sur-approximation de l'ensemble des solutions possibles du problème 1 avec l'état initial x_0 et le paramètre p. Autrement dit, pour toute incertitude u intégrable au sens de Lebesgue telle que $\forall t \in [t_0, t_1]$, $u(t) \in U$, l'unique solution x du problème 1 appartient à $\varphi_{x_0,p}$:

$$\forall t \in [t_0, t_1],\ x(t) \in \varphi_{x_0,p}(t) \tag{6}$$

Ainsi, une fonction ensembliste φ définie comme l'union de toutes les fonctions ensemblistes $\varphi_{x_0,p}$ possibles, *i.e.*

$$\forall t \in [t_0, t_1],\ \varphi(t) = \bigcup_{\substack{x_0 \in X_0 \\ p \in P}} \varphi_{x_0,p}(t) \tag{7}$$

est une sur-approximation de l'ensemble des états atteignables sur l'intervalle de temps $[t_0, t_1]$.

En particulier, alors que la méthode utilisée dans l'outil Flow* [45, 46] pour calculer des sur-approximations des ensembles d'états atteignables dans le cas des dynamiques continues n'a été prouvée correcte que pour des incertitudes continues, notre critère permet de prouver que cette méthode est également valide en présence d'incertitudes intégrables au sens de Lebesgue. Toutefois, les sur-approximations calculées par cette méthode sont souvent imprécises, *i.e.* les ensembles ainsi construits sont souvent bien plus larges que les ensembles d'états atteignables [49]. Cependant, notre critère prouve qu'un vecteur d'intervalles dans $\mathbb{R}^n$ contracté par cette version ensembliste de l'opérateur de Picard est une sur-approximation de l'ensemble des états atteignables, ce qui peut servir comme sur-approximation *a priori* dans certains algorithmes.

Cas des systèmes séparables par rapport aux incertitudes variant dans le temps

Dans le cas particulier de systèmes que nous appelons *systèmes séparables par rapport aux incertitudes variant dans le temps*, qui sont une généralisation des systèmes affines

par rapport aux entrées, nous proposons un nouvel algorithme basé sur une décomposition comme différence de fonctions positives des termes de droite des équations différentielles ordinaires pour convertir les problèmes avec incertitudes variant dans le temps en problèmes avec uniquement des incertitudes constantes. Cette conversion est définie de sorte à garantir une sur-approximation de l'ensemble des états atteignables, *i.e.* toute sur-approximation pour le problème converti avec des incertitudes constantes est une sur-approximation pour le problème d'origine avec des incertitudes variant dans le temps. Nous comparons les résultats produits par notre prototype avec ceux obtenus en utilisant des outils de l'état de l'art sur des exemples et nous remarquons que nos sur-approximations sont plus précises, quand bien même certains outils ne gèrent que des incertitudes intégrables au sens de Riemann, ce qui est un sous-ensemble de celles intégrables au sens de Lebesgue.

Nous appelons *système séparable par rapport aux incertitudes variant dans le temps*, ou simplement *système séparable*, un problème de la forme

$$\begin{cases} \forall_{ae.} t \in [t_0, t_1],\ \dot{x}(t) = g(u(t)) \cdot h(t, x(t), p) \\ \forall t \in [t_0, t_1],\ u(t) \in U \\ x(t_0) = x_0 \in X_0 \subset \mathbb{R}^n \\ p \in P \end{cases} \tag{8}$$

en utilisant les mêmes notations que pour le problème 1 et avec g une fonction continue de U dans $\mathbb{R}^{n \times k}$ et h une fonction continue de $[t_0, t_1] \times \mathbb{R}^n$ dans $\mathbb{R}^k$, avec k un entier naturel. Nous faisons toujours l'hypothèse que pour chaque état initial $x_0 \in X_0$, paramètre $p \in P$ et incertitude intégrable au sens de Lebesgue u, le système admet une unique solution. Cette classe de systèmes est un cas particulier du cas général non-linéaire mais elle permet d'exprimer toutes les dynamiques linéaires et celles avec des perturbations additives, *i.e.* de la forme $\dot{x}(t) = f(t, x(t), p) + u(t)$.

À partir d'un tel système, nous pouvons en définir un nouveau sans incertitude u variant dans le temps. Nous appelons *système auxiliaire* ce nouveau système défini par

$$\begin{cases} \forall_{ae.} t \in [t_0, t_1],\ \dot{x}(t) = Ah^+(t, x(t), p) - Bh^-(t, x(t), p) \\ A \in \mathit{interval}(g(U)) \\ B \in \mathit{interval}(g(U)) \\ x(t_0) = x_0 \in X_0 \subset \mathbb{R}^n \\ p \in P \end{cases} \tag{9}$$

avec h^+ et h^- deux fonctions continues positives dont la différence est égale à h (*i.e.* pour tout $t \in [t_0, t_1]$ et $x \in \mathbb{R}^n$, $h(t, x) = h^+(t, x) - h^-(t, x)$) et A et B deux matrices définies dans $\mathit{interval}(g(U))$, la plus petite boîte (vecteur d'intervalles) englobant l'image de g sur le domaine U. Ce système est construit de façon à garantir que toute solution du système séparable 8 est également solution du système auxiliaire correspondant. Ainsi, il suffit de calculer une sur-approximation de l'ensemble d'états atteignables du système 9 pour en obtenir une du système 8 correspondant.

Une telle sur-approximation de l'ensemble d'états atteignables peut être obtenue avec les méthodes classiques puisque le système auxiliaire est un système avec uniquement des incertitudes constantes. Toutefois, elle dépend fortement de la décomposition

$h = h^+ - h^-$. Nous discutons d'un caractère d'optimalité d'une telle décomposition dans la sous-section 3.3.2. Nous nous concentrons sur le cas des modèles de Taylor comme représentation des ensembles qui généralise le cas des intervalles ou des formes affines encodant les zonotopes, tout en pouvant représenter les mêmes ensembles que les zonotopes polynomiaux [6]. Nous proposons une décomposition affine qui est optimale si l'on se limite au cas des décompositions polynomiales.

Nous avons implémenté cette procédure de décomposition pour calculer une sur-approximation de l'ensemble d'états atteignables de système séparables. Nous avons comparé les sur-approximations obtenues avec celles obtenues par des outils de l'état de l'art et nous avons démontré expérimentalement que nous obtenions des sur-approximations plus précises malgré la prise en compte d'incertitudes intégrables au sens de Lebesgue.

Cas des systèmes commutés

Dans le cas de systèmes dynamiques hybrides, *i.e.* systèmes qui combinent des dynamiques continues et discrètes, certaines hypothèses simplificatrices comme l'instantanéité des réponses de contrôleurs peuvent engendrer des exécutions difficiles à interpréter. En particulier, les modèles peuvent admettre des solutions qui suivent un nombre infini d'évolutions discrètes dans un temps fini. De tels comportements sont appelés *comportements Zénon* et ils impliquent une singularité en temps : l'interprétation classique de ces systèmes empêche le temps de progresser au-delà d'un certain point. Toutefois, une interprétation classique des systèmes commutés (un cas particulier de systèmes hybrides) permet de définir des solutions au-delà de telles singularités en temps. Cette interprétation au sens de Filippov revient à considérer un système continu avec une incertitudes intégrable au sens de Lebesgue contrainte par les valeurs possibles des dérivées. Ainsi, nous proposons dans la section 3.4 un algorithme qui exploite notre résultat sur les systèmes continus pour gérer le cas des systèmes commutés.

Un *système commuté* est un système dont les dynamiques sont de la forme suivante :

$$\begin{cases} \dot{x}(t) = f_i\big(t, x(t), p, u(t)\big) \\ i \in G\big(t, x(t), p, u(t)\big) \end{cases} \tag{10}$$

avec f_i des fonctions continues donc l'indice i appartient à un ensemble fini I, G une fonction ensembliste hémicontinue supérieurement avec valeurs dans l'ensemble des parties de I, et p et u ayant les mêmes contraintes que dans le cas général des problèmes non-linéaires 1. Comme précédemment, nous faisons l'hypothèse que pour tout $i \in I$, la dynamique $\dot{x}(t) = f_i\big(t, x(t), p, u(t)\big)$ admet une unique solution pour des incertitudes données (état initial, paramètre et entrée). Nous faisons également l'hypothèse que l'ensemble sur lequel l'image de G n'est pas un singleton est de mesure nulle, *i.e.* l'image de G est presque partout un singleton.

Pour pouvoir calculer une sur-approximation de l'ensemble d'états atteignables au-delà de tout comportement Zénon, nous utilisons l'inclusion différentielle introduite par

Filippov [63] :

$$\begin{cases} \dot{x}(t) \in F_J(t, x(t), p, u(t)) \\ J = G(t, x(t), p, u(t)) \end{cases} \tag{11}$$

avec F_J l'enveloppe convexe des valeurs de toutes les fonctions f_j telles que j appartient à l'image de G. Puisqu'une inclusion différentielle peut être interprétée comme une incertitude intégrable au sens de Lebesgue, nous pouvons utiliser les résultats précédents. En particulier, si les fonctions f_j ne dépendent pas des incertitudes u, alors l'inclusion différentielle peut être interprétée comme un système séparable.

Ainsi, nous proposons un algorithme pour calculer une sur-approximation de l'ensemble d'états atteignables de systèmes commutés en exploitant cette interprétation au sens de Filippov. Il est similaire aux algorithmes classiques pour les automates hybrides [99] à l'exception des changements de modes : si l'image de G n'est pas un singleton, nous essayons de raffiner sa sur-approximation, *e.g.* en utilisant une subdivision de son domaine, puis nous calculons l'évolution suivant l'enveloppe convexe des dynamiques correspondantes.

De la simulation au calcul ensembliste pour les systèmes hybrides

Finalement, en collaboration avec Marc Pouzet, nous nous intéressons dans le chapitre 5 à des cas de modèles hybrides qui ne sont pas formalisés comme des automates hybrides. En effet, les logiciels permettant de concevoir de tels systèmes et de simuler les modèles créés autorisent la définition de systèmes comme assemblages de sous-systèmes, par exemple des composants électroniques ou mécaniques, et les actions discrètes peuvent être définies par des programmes complexes, par exemple des contrôleurs numériques avec mémoires. Les modèles ainsi définis par les utilisateurs sont ensuite traduits par les logiciels en modèles intermédiaires adaptés à la simulation.

Nous nous concentrons sur le cas du langage Zélus[1] qui est basé sur la programmation synchrone, notamment le langage Lucid Synchrone[2][131] qui est lui-même une extension du langage Lustre[81], à laquelle sont ajoutées des équations différentielles ordinaires pour définir des dynamiques continues et des événements de *zero-crossing* pour déclencher les dynamiques discrètes. Le principal avantage de ce langage est que son code nous est accessible permettant sa modification si nécessaire. De plus, il est suffisamment expressif pour implémenter beaucoup de composants élémentaires comme ceux proposés dans la bibliothèque standard de l'outil Simulink[3] [40] et, contrairement à ce dernier, Zélus a une sémantique bien définie [31].

Toutefois, contrairement aux automates hybrides, les représentations des états des systèmes peuvent être des structures complexes. Cela est notamment le cas des automates hiérarchiques dont les composants ont eux-mêmes des états. Les représentations des

[1] `https://zelus.di.ens.fr/`

[2] `https://www.di.ens.fr/~pouzet/lucid-synchrone/`

[3] `https://fr.mathworks.com/products/simulink.html`

états sont différentes d'un modèle à l'autre et ne peuvent être manipulées qu'à travers des fonctions définissant les modèles intermédiaires associés. Ainsi, les algorithmes de simulation ou d'analyse d'atteignabilité n'ont qu'une connaissance partielle du modèle manipulé.

En particulier, pour faire l'analogie avec les automates hybrides, les variables encodant le mode dans lequel se trouve un système hybride ne peuvent pas être manipulées indépendamment du reste des états des modèles. Cela peut rendre l'exploration de différentes branches en parallèle plus compliquée. Par exemple, une dynamique discrète pourrait résulter en deux modes différents (*e.g.* la réponse d'un thermostat numérique), ce qui impliquerait de retourner deux états lors d'une analyse d'atteignabilité. Contrairement aux automates hybrides, le mode résultant d'une exécution discrète est déterminé pendant cette exécution et non au moment de son activation.

Pour effectuer une analyse d'atteignabilité (ou plus généralement des simulations ensemblistes) de tels modèles intermédiaires, *i.e.* dont seules les interfaces sont connues, il faut pouvoir en définir des versions ensemblistes. Dans le cas de Zélus, ces modèles intermédiaires sont implémentés en OCaml comme des collections de fonctions qui modifient un état donné par une structure. Nous montrons que de tels modèles peuvent être convertis en des versions ensemblistes en les paramétrant par une représentation d'ensembles. Nous avons implémenté un prototype qui montre cela en utilisant les foncteurs d'OCaml.

Toutefois, cette méthode ne permet pas de gérer efficacement les structures conditionnelles car celles-ci peuvent propager des contraintes à l'ensemble des signaux. Pour propager de telles contraintes entre les signaux alors qu'une évaluation d'une expression n'en concerne qu'une partie, il faut manipuler l'ensemble de signaux comme un tout, éventuellement via une structure globale enregistrant les dépendances entre eux. Cette méthode est mise en uvre par les analyseurs statiques (*e.g.* Astrée[4] ou Fluctuat[5]) qui manipulent généralement le code source de programmes. La paramétrisation des modèles intermédiaires que nous proposons échoue actuellement à gérer de telles structures conditionnelles dans le cas général. Une adaptation de notre méthode pour gérer de telles dynamiques discrètes est laissée pour un travail ultérieur visant à réunir les techniques d'analyse d'atteignabilité pour les systèmes hybrides et les techniques d'analyse de programmes par interprétation abstraite.

[4] `https://www.astree.ens.fr/`

[5] `https://www.lix.polytechnique.fr/Labo/Sylvie.Putot/fluctuat.html`

Remerciements

Je tiens tout d'abord à remercier mes deux directeurs de thèse, Sylvie Putot et Éric Goubault, pour m'avoir donné l'opportunité d'effectuer ce doctorat à la suite de ma formation d'ingénieur, pour m'avoir guidé, encouragé et motivé tout au long de ces années afin de produire un travail que j'espère de qualité.

Je remercie également Marc Pouzet pour tous les encouragements et pour les nombreux échanges que nous avons eus à propos de simulations ensemblistes de systèmes hybrides : confronter nos différents points de vue m'a beaucoup aidé à mieux comprendre les méthodes que j'ai expérimentées.

Je tiens aussi à remercier Pierre-Loïc Garoche, Thao Dang, Nacim Ramdani et Timothy Bourke pour avoir accepté de faire partie de mon jury de soutenance, pour les questions qu'ils m'ont posées à l'issue de ma présentation, pour le temps qu'ils ont passé à lire ce manuscrit et pour tous les retours qu'ils m'ont faits afin de l'améliorer. Merci également à Nacim Ramdani et Timothy Bourke pour avoir constitué mon comité de suivi pendant la durée de mon doctorat.

Je souhaite remercier toutes celles et ceux avec qui j'ai pu discuter, les collègues de l'équipe Cosynus, de l'École polytechnique, mais aussi de tous les autres établissements, que ce soit lors de séminaires pour échanger sur nos travaux ou lors de moments informels : tous ces échanges ont alimenté mes réflexions et m'ont souvent permis d'appréhender les problèmes avec de nouvelles approches.

Bien évidemment, tout ce travail n'aurait pas été possible sans toutes les équipes administratives du laboratoire, de l'école doctorale et de l'École polytechnique que je tiens à remercier pour tout leur travail m'ayant permis de me focaliser sur mon doctorat.

Il en est de même pour ma famille et mes amis qui m'ont soutenu jusqu'au bout, s'assurant de mon bien-être, me forçant à faire des pauses lorsqu'elles devenaient nécessaires. Leur attention à mon égard m'a certainement beaucoup aidé moralement et c'est pour cela que je tiens à les remercier ici.

Enfin, je tiens tout particulièrement à remercier Claire-Sophie qui a été à mes côtés pendant toutes ces années, à me supporter dans mon acharnement mais surtout à m'encourager et à me soutenir au quotidien.

Contents

Chapter 1

Introduction

1.1 Context

1.1.1 Modeling of systems

Before complex systems are manufactured, they are most of the time simulated in order to validate their behaviors. For example, instead of constructing a whole car and replacing every component until it works as expected, it is much more efficient to model every component and to simulate its behavior. Then, these components may be combined with a model of their environment to obtain a model of a complex system. When such a model is defined, it is convenient to test different controllers without imperiling a physical device or users, *e.g.* pilots in an aircraft or nuclear plants. Such systems that gather continuous dynamics (*e.g.* evolution of the temperature) and discrete ones (*e.g.* numerical controllers) are called *hybrid systems*.

In that case, the physical system is considered as part of the environment in which a program is executed. This approach of modeling the environment in order to simulate the behavior of a program is at the base of the Model-Based Design method (MBD). Due to the complexity of the environment, such models are intentionally simplifications of the reality in order to be simulated. For example, the computation time of numerical controllers may be neglected with respect to the physical dynamics, *i.e.* it may be assumed instantaneous.

Such simplifications may result in complex behaviors. For example, consider a naive thermostat that turns the heater on if the current temperature is below the expected one, and that turns it off otherwise. If the modeled dynamics of the temperature has no inertia, *i.e.* the temperature increases (*resp.* decreases) has soon as the heater is on (*resp.* off), then the thermostat controller should switch an infinite number of times as soon as the temperature reaches the expected one. Such a behavior is called a *Zeno behavior*.

In the case of *switched systems* (*cf.* Subsection 2.1.2), we can define a natural evolution of such models beyond Zeno behaviors using differential inclusions instead of differential equations. Differential inclusions introduce an uncertainty on the derivatives,

which can be interpreted as an unknown Lebesgue-integrable function with values in a closed bounded convex set [63].

Moreover, the original models, *i.e.* even without considering any extension beyond Zeno behaviors, may also contain uncertain values. The environment may include time-varying perturbations (*e.g.* the wind blowing on a vehicle), but the system itself may also be defined with some partially known parameters (*e.g.* the length of a part defined with some tolerance).

1.1.2 Proof of properties on the models

Despite the uncertainties, we want to be able to prove some properties of the models. We are interested in *safety properties* which express that the models stay in safe states. For example, a safety property of a network of trains could be that the distance between two consecutive trains is always bigger than a given threshold.

While an execution of a model may identify a reachable unsafe state, *i.e.* a state in which a safety property is violated, it cannot prove that a safety property is satisfied for all executions. Indeed, due to the uncertainties, an uncountable number of executions may be possible and it is infeasible to compute each of them.

Thankfully, multiple methods exist to prove safety properties. First, we can try to write a formal proof for each safety property [129, 130]. If such a proof exists, then the corresponding property is true. With such a method, we have to construct a proof for each property, even if multiple properties are gathered using conjunctions. For example, instead of proving a property P_1 then a property P_2, we can directly try to prove the property $P_1 \wedge P_2$, but it requires to prove both operands of the conjunction. So, we can only expect to factorize the two proofs.

We focused on another method called *reachability analysis*. It consists in identifying the set of states that can be reached by the model under study and checking that none of them are unsafe, *i.e.* none of them violate a safety property. This method allows to prove multiple properties with only one computation of the set of reachable states. Alternatively, we can compute the set of states from which the model can reach some unsafe ones [142, 141]. Then, instead of checking that a reachable state is unsafe, we have to check whether an initial state belongs to the set of states that can reach an unsafe one. It is called *backward reachability analysis* [116]. In this thesis, we focus on computing the set of reachable states from a set of possible initial states, which is called *forward reachability analysis* [116].

1.1.3 Reachability analysis with uncertainties

Computing the exact set of reachable states is often impossible. First, a solution of an ordinary differential equation may not exist in a closed-form. Moreover, in case of hybrid automata (*cf.* Section 2.1.1), the computation of the set of reachable states is undecidable [15, 84]. So, we compute an *over-approximation* (or *outer-approximation*) of such a set, *i.e.* a set that contains all the reachable states but that can also contain other ones [133, 3, 45, 36]. Such an over-approximation allows to prove safety properties: if

none of the contained states are unsafe, then none of the reachable states are unsafe and the safety property is proven. On the contrary, if such an over-approximation contains an unsafe state, it does not prove that the safety property is false, because such an unsafe state may not be reachable. Figure 1.1 illustrates it: the over-approximation (in orange) proves that the reachable states are all below a given threshold (the dashed green line), but it does not prove that they are all below another threshold (the dashed red line), while the exact set of reachable states (in hatched blue) actually satisfies this property. A violation of a safety property may be proved computing an *under-approximation* (or *inner-approximation*) of the reachable set, *i.e.* a set whose all elements are reachable states [78, 73, 75, 36]. If an unsafe state belongs to such an under-approximation, this proves that the safety property is false.

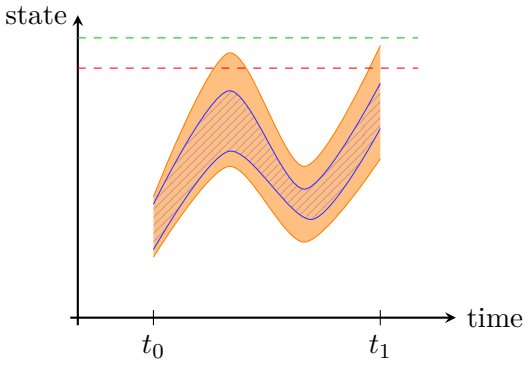


Figure 1.1 – Over-approximation (in orange) of the set of reachable states (in hatched blue) over the time interval $[t_0, t_1]$ that proves the satisfaction of a safety property (below the dashed green line) but not the satisfaction of another one (below the dashed red line)

Multiple methods exist to compute such over-approximations. Their precisions depend on the chosen representations of the sets (*cf.* Subsection 2.4.1) but also on the hypotheses about the models. For example, some methods are specially designed for models with only affine explicit ordinary differential equations [17, 100, 12, 65], while others may handle nonlinear ones [4, 45, 72]. While all these methods handle uncertain initial states or parameters, *i.e.* constant uncertainties, some methods handle continuous time-varying uncertainties [14, 45], piecewise continuous (or even Riemann-integrable) uncertainties [12, 4] or Lebesgue-integrable uncertainties [90, 72]. Other kind of continuous dynamics may be handled (*e.g.* delay differential equations [79]), but we focus on nonlinear explicit ordinary differential equations as continuous dynamics in this thesis.

Finally, the methods also depend on the type of dynamics in the models. The previous methods can handle models without discrete dynamics such as state jumps. On the contrary, a model composed of only discrete dynamics is just a program and abstract interpretation [52, 51] is a classical way to compute over-approximations of the set of reachable states [51, 37, 77, 69]. A hybrid model, *i.e.* a model that mixes discrete and continuous dynamics, can be handled by dedicated algorithms [83, 99, 50, 66, 45, 91].

As far as we know, while methods have been developed in order to simulate hybrid systems beyond Zeno behaviors [88, 148], only few of them have been designed to automatically compute an over-approximation of the set of reachable states in presence of possible Zeno behaviors [95, 94].

1.2 Contributions

We identify four points of contribution in this thesis:

1. We provide a criterion to check if a set-valued function is an over-approximation of the set of reachable states of a nonlinear explicit ordinary differential equation with bounded Lebesgue-integrable uncertainties and uncertain initial states over a closed bounded time interval. The proof is inspired by the one by Martin Berz and Kyoko Makino [34]. It does not provide any methods to construct such a set-valued function, but it proves in particular that the method presented by Xin Chen in [45] for bounded continuous uncertainties is correct even in the case of bounded Lebesgue-integrable ones. We later exploit it to compute quickly a rough initial over-approximation of the reachable set.

2. For a particular class of systems of nonlinear explicit ordinary differential equations that we call *separable systems with respect to the time-varying uncertainties* and that are a generalization of the *input affine systems*, we provide an algorithm to reduce the problem of computing an over-approximation despite Lebesgue-integrable uncertainties into a problem of computing an over-approximation despite only constant uncertainties. We compare the produced over-approximations with the ones returned by state-of-the-art tools on examples and we demonstrate that our over-approximations are indeed tighter than the others.

3. We propose an algorithm exploiting our previous result in order to compute an over-approximation of the set of reachable states of *switched systems*, *i.e.* hybrid automata without jumps. We justify its correctness without proving its termination. We illustrate its application on a motivating example.

4. We present a reflection about the reachability analysis of alternative models than hybrid automata. We focus on models written in Zélus[1], which is an extension of the synchronous data-flow language Lustre. Those models are compiled into intermediate models written in OCaml and, contrarily to hybrid automata, entire synchronous programs (possibly with memories) can be executed during discrete transitions. So, we have to mix classical reachability analysis of hybrid systems and abstract interpretation of programs. We present needed adaptations of the interfaces between the models, the simulation engine and the different external solvers in order to compute reachability analyses. We also propose a method to lift such models that use by default floating-point numbers as representation of

[1] `https://zelus.di.ens.fr/`

real numbers into equivalent models parameterized by a representation of sets that guarantee the over-approximation of the computed signals. This work is still in progress, but we illustrate an expected behavior using a prototype written in OCaml.

1.3 Outline

This thesis is organized as follows.

In Chapter 2, we introduce the basic definitions and theorems used in the rest of the document. We give definitions of hybrid automata, switched systems, Zeno behaviors and differential inclusions in the sense of Filippov. Then, we introduce the synchronous language Lustre and its extension Zélus to model hybrid systems. We also present basic results of the theories of ordinary differential equations. Finally, we define the reachability analysis and we present different representations of sets, different methods to handle continuous systems and a classic method to handle hybrid systems.

In Chapter 3, we present our theoretical contributions on the reachability analysis of continuous systems with Lebesgue-integrable uncertainties and on the reachability analysis of switched systems presenting possible Zeno behaviors. In both cases, the application of the algorithms are detailed on examples.

In Chapter 4, we present an implementation of the proposed algorithm to compute an over-approximation of the set of reachable states of *separable systems* with Lebesgue-integrable uncertainties. We also compare the results with the ones of state-of-the-art tools on examples.

Finally, in Chapter 5, we present our reflections about the application of reachability analysis (and more generally, set-valued computations) to hybrid systems defined as programs written in dedicated languages such as Zélus. We start with high level considerations about interfaces of involved solvers (solver of continuous dynamics and detector of events) adapted to the reachability analysis. We also present a method based on a parametrization of the models by a representation of sets in order to guarantee over-approximations of the computed values and to allow computation of reachability analyses. We illustrate the feasibility of this method with a prototype written in OCaml, using functors.

Chapter 2

Background

In this chapter, we introduce the basic knowledge required for our work and its motivation, which is the proof of safety properties. We first present the notion of *hybrid systems* and *hybrid automata* and discuss *Zeno behaviors* and their interpretation. Then, we present some results about ordinary differential equations and the theory of integration in the sense of Lebesgue. Finally, we introduce the notion of *reachability analysis* and some representations of sets and algorithms designed for continuous or hybrid dynamics.

2.1 Hybrid systems

Most of the objects around us are controlled by some software. Such software has a direct influence on its physical environment. Think about a domestic water heater, a boiler, needed to warm up a house. It could be set with the desired power to heat the water forever. Hopefully, the heat exchanges between the radiators, the house and the exterior will be balanced. However, the temperature of the exterior is much likely non-constant, so will be the temperature of the room. That is why most houses are equipped with a thermostat that controls the boiler. The thermostat can be a numerical device with sensors to acquire the current temperature of the house, with actuators to control the boiler, and with a software to adapt its behavior depending on the temperature.

The combination of the house, the boiler and the thermostat is a dynamical system, whose temperature evolves over the time, and it involves software.

Such systems typically have two kinds of dynamics: *continuous dynamics* for most of their physical part and *discrete dynamics* for the change of behaviors. In the case of the previous example, the evolution of the temperature of the house has continuous dynamics (it evolves at every instant) while the control of the boiler is a discrete dynamics (turning the boiler on is done only at specific instants, depending on the temperature and the frequency of the numerical thermostat). We call *hybrid dynamical systems*, or simply *hybrid systems*, the systems that combine these two kinds of dynamics[71]. These hybrid systems are a sub-class of the *cyber-physical systems*, which gather computational, physical and communication capacities.

It is often needed to check that the behaviors of such systems satisfy some specifications. For example, a specification for the thermostat could be an admissible upper bound of the difference between the temperature of a house and some target, given the specifications of the environment (the thermal isolation of the house, the maximal power of the heaters, *etc.*). Those specifications about the behaviors of the systems can refer to the efficiency of the device, but they can also refer to some safety properties. For example, to ensure the safety of the passengers of a train, we could specify a lower bound of the distance between two trains at all instants.

In order to check the satisfaction of the specifications, we can perform experiments on the actual system, but it can be expensive, time consuming or even dangerous (*e.g.* a stress test of a nuclear power plant). Moreover, problems detected at the conception time, before doing the actual implementation, are far less costly to correct. So, a convenient way to validate the specifications is to reason on a model of the system.

2.1.1 Hybrid automaton

Multiple ways to model hybrid systems exist. A classical one consists in modelling the discrete dynamics with a finite-state automaton, whose states are called *modes*, while the continuous dynamics are modeled as finitary relations between multiple states of the system in each mode of the automaton. We called such models *hybrid automata*. While [82, Definition 1.1] uses predicates to encode the possible evolutions, we use functions depending on the current state, parameters and inputs to define the next states. Our definition is then an intermediate between the one in [82, Definition 1.1] and the one in [104, Definition 2].

Definition 1 (Hybrid automaton)
A hybrid automaton H consists of the following components.

State Variables. A finite set $X = \{x_1, \ldots, x_n\}$ of real-numbered variables called *state variables*. The number n is called the *dimension* of H. We write $\dot{X}$ for the set $\{\dot{x}_1, \ldots, \dot{x}_n\}$ of dotted variables (that represent the first derivatives during continuous changes), and we write X' for the set $\{x'_1, \ldots x'_n\}$ of primed variables (that represent the values at the conclusion of discrete changes).

Parameters. A finite set $P = \{p_1, \ldots, p_k\}$ of real-numbered variables called *parameters*.

Input variables. A finite set $U = \{u_1, \ldots, u_m\}$ of real-numbered variables called *input variables*.

Control graph. A finite directed multigraph (V, E). The vertices in V are called *control modes* or simply *modes*. The edges in E are called *control switches*, *switches* or *transitions*.

Initial set. A set $Init \subset V \times \mathbb{R}^n \times \mathbb{R}^k$ of possible initial control modes and values of

the state variables and the parameters.

Invariant conditions. A vertex labeling function *inv* that assigns to each control mode $v \in V$ a predicate $inv(v)$ whose free variables are from $X \cup P \cup U$.

Flow equations. A vertex labeling function *flow* that defines for each control mode $v \in V$ the first order derivative of each state variable depending on the values of the state variables, of the parameters and of the input variables, *i.e.* $\dot{x} = flow(v)(x, p, u)$.

Guards. An edge labeling function *guard* that assigns to each control switch $e \in E$ a predicate $guard(e)$ whose free variables are from $X \cup P \cup U$.

Jumps. An edge labeling function *jump* that assigns to each control switch $e \in E$ the value of x' depending on the state variables, of the parameters and of the input variables, *i.e.* $x' = jump(e)(x, p, u)$.

The values of the input variables are assumed to be defined by continuous functions, *i.e.* the function u from time to input domain is continuous. Parameters can be seen as constant inputs, *i.e.* $p_i \in \mathbb{R}$ can be replaced by an input variable $u_{m+i} : \mathbb{R} \to \mathbb{R}$ such that for all times $t \in \mathbb{R}$, $u_{m+i}(t) = p_i$. So, we often ignore the parameters in the following to reduce the notations.

For any mode $v \in V$ and for any edge $e \in E$, the predicates $inv(v)$ and $guard(e)$ define subsets of the space $\mathbb{R}^n \times \mathbb{R}^k \times \mathbb{R}^m$ on which they are satisfied. We use the same notations to represent the predicates and the corresponding sets.

We may interpret the time t as an implicit state variable in $\mathbb{R}_{\geq 0}$ with a constant associated flow equation: $\forall v \in V$, $\dot{t} = 1$, *i.e.* t belongs to X. However, in this work, we consider t as a special variable that is not part of the state variables, but that can be a free variable of the predicates $inv(v)$ and $guard(e)$ or a dependency of the functions $flow(v)(t, x, p, u)$ and $jump(e)(t, x, p, u)$. Moreover, the initial set *Init* is then a subset of $\mathbb{R}_{\geq 0} \times V \times \mathbb{R}^n \times \mathbb{R}^k$.

In this work, we focus on hybrid automata with flow conditions defined as *explicit ordinary differential equations of the first order*, *e.g.* $\dot{x}(t) = f\big(t, x(t), p, u(t)\big)$ with f a continuous function. The restriction to *ordinary differential equations* avoids having to handle *partial differential equations*, *e.g.* the heat equation $\frac{\partial x}{\partial t} = \Delta x$ with the Laplacian operator Δ. The restriction to *explicit* differential equations allows us to easily compute the value of the derivative given the current time and the current state of the system. However, the focus on first order equations is not a restriction, because we can add extra variables to a system of order n to obtain a system of explicit first order differential equations:

$$x^{(n)}(t) = f\big(t, x(t), \dot{x}(t), x^{(2)}(t), \ldots, x^{(n-1)}(t), p, u(t)\big)$$

can be translated into

$$\begin{cases} \dot{x}_{n-1}(t) &= f(t, x_0(t), x_1(t), \ldots, x_{n-1}(t), p, u(t)) \\ \dot{x}_{n-2}(t) &= x_{n-1}(t) \\ & \vdots \\ \dot{x}_0(t) &= x_1(t) \end{cases}$$

Moreover, we often do not specify the discrete changes when they are identity functions $x' = \mathit{jump}(v)(t, x, p, u) = x$.

Finally, we can graphically represent a hybrid automaton: we use the representation of the finite directed multigraph, whose vertices contain the differential equations and the invariants. The guard and jump conditions are denoted on the corresponding directed edges. The initial conditions are represented as arrows from no vertices to the vertices of the condition (usually a unique arrow to a vertex).

Example 1 (Thermostat as hybrid automaton)
We present here a simple model of a thermostat as a hybrid automaton that turns the heaters on if the temperature is lower than a reference and before it becomes lower than the reference minus a threshold and that turns them off if the temperature is bigger than the reference and before it becomes bigger than the reference plus the threshold:

State variables: $X = \{x\}$ with x the value of the temperature

Parameters: $P = \{p\}$ with p the threshold of the hysteresis

Input variables: $U = \{u\}$ with u the time-varying target temperature

Control graph: $V = \{\mathit{On}, \mathit{Off}\}$ and $E = \{(\mathit{On}, \mathit{Off}), (\mathit{Off}, \mathit{On})\}$

Initial set: $\mathit{Init} = \{(\mathit{On}, x, p) \mid x \in [10, 15] \wedge p \in [0.5, 1]\}$

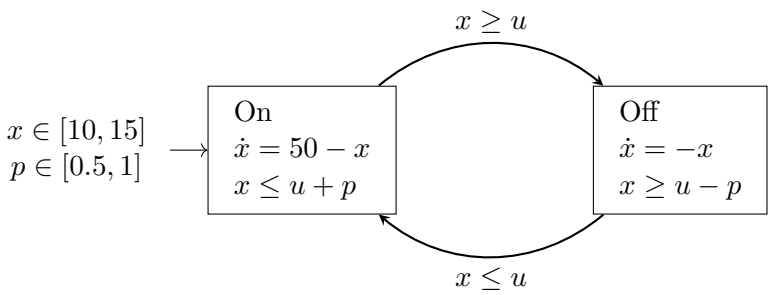
Invariant conditions: $\mathit{inv}(\mathit{On}) = (x \leq u + p)$ and $\mathit{inv}(\mathit{Off}) = (x \geq u - p)$

Flow equations: $\mathit{flow}(\mathit{On})(t, x, p, u) = 50 - x$ and $\mathit{flow}(\mathit{Off})(t, x, p, u) = -x$

Guards: $\mathit{guard}((\mathit{On}, \mathit{Off}))(t, x, p, u) = (x \geq u)$
and $\mathit{guard}((\mathit{Off}, \mathit{On}))(t, x, p, u) = (x \leq u)$

Jumps: There are no jumps, *i.e.* $\forall e \in E$, $\mathit{jump}(e)(t, x, p, u) = x$

Finally, we draw here the graphical representation of this hybrid automaton



The interpretation of hybrid automata is given by the *transition semantics* [82, Definition 1.3] and the *trace semantics* [82, Definition 1.5], but we present here a simpler version without labeling the transitions and inspired by the definition of *trajectories* in [80].

A *trajectory*, or *trace*, is in our case a sequence of events of the system (the mode and the values of the state variables) with a non-negative real number to encode the elapsed time since the previous event.

In order to simplify the notation, we use the same notation to represent the variables and their values.

Definition 2 (Trajectory of a hybrid automaton)
Let H be a hybrid automaton with n state variables.

A *trajectory* of H, or *trace*, is a sequence of triples $\left((\delta_i, v_i, x_i)\right)_{i \in I}$, called *events*, with

- $I = \mathbb{N}$ or $I = [\![0, k]\!]$ the set of indices (with $k \in \mathbb{N}$),
- for all $i \in I$, $\delta_i \in \mathbb{R}_{\geq 0}$ the elapsed time since the last event,
- for all $i \in I$, $v_i \in V$ the current mode,
- for all $i \in I$, $x_i \in \mathbb{R}^n$ the current values associated to the variables.

The initial elapsed time δ_0 is the initial time.

This definition allows to define infinitely many trajectories, because the set of possible values of the variables $\mathbb{R}^n$ is infinite and even uncountable. However, all those trajectories do not represent valid evolutions of the system. So, we define the set of *admissible trajectories* as the set of trajectories that satisfy all the constraints of the hybrid automaton: all the events satisfy the corresponding invariant conditions, the first event is in the initial set and a pair of two successive events satisfies a transition (guard and jump) or a flow equation.

Definition 3 (Admissible trajectory of a hybrid automaton)
Let H be a hybrid automaton and let $\tau = \left((\delta_i, v_i, x_i)\right)_{i \in I}$ be a trajectory of it.

Let $\Delta_i = \sum_{k=0}^{i} \delta_k$ be the sum of all the previous elapsed time, *i.e.* the duration from the origin of time.

τ is an *admissible trajectory* of H, or H *admits* the trajectory τ, if and only if

there exists a constant value $p \in \mathbb{R}^k$ of the parameters and a function $u : \mathbb{R}_{\geq 0} \to \mathbb{R}^m$ encoding the values of the inputs depending on the time, such that the following conditions are satisfied:

1. The first triple is a valid initial state, *i.e.* $(v_i, x_i, p) \in \mathit{Init}$ and $\mathit{inv}(v_0)$ are satisfied for the values $x := x_0$, p and $u(\Delta_0)$.
2. There exists a valid evolution between two successive states, *i.e.* for all $i \in I \backslash \{0\}$:
   - if $\delta_i = 0$, then there exists an edge $e \in E$ from v_{i-1} to v_i and the discrete evolution of the state is valid according to the guard and the jump, *i.e.* $\mathit{guard}(e)$ and $x' = \mathit{jump}(e)(x, p, u(\Delta_i))$ are true with $x := x_{i-1}$, $x' := x_i$, p and $u(\Delta_i)$;
   - otherwise $v_i = v_{i-1}$ and there exists a differentiable function f over $[0, \delta_i]$ such that $f(0) = x_{i-1}$, $f(\delta_i) = x_i$ and for all positive reals $\xi \in [0, \delta_i]$, $\dot{x} = \mathit{flow}(v_i)(x, p, u(\Delta_{i-1} + \xi))$ and $\mathit{inv}(v_i)$ are satisfied with the values $x := f(\xi)$, $\dot{x} := \dot{f}(\xi)$, p and $u(\Delta_{i-1} + \xi)$.

In the following, we refer to Δ_0 or δ_0 as t_0, which corresponds to the smallest time of a given trajectory.

Example 2 (Admissible trajectory of the thermostat)
We present an admissible trajectory of the hybrid automaton of the thermostat described in Example 1.

We assume that for all time t, $u(t) = 20$ and $p = 1$.

We also assume that the initial time is equal to 0, *i.e.* $t_0 = \Delta_0 = \delta_0 = 0$. Moreover, the initial condition can only be true in mode *On* with $x \in [10, 15]$. So we arbitrarily set $(\delta_0, v_0, x_0) = (0, \mathit{On}, 10)$. Notice that this initial event satisfies the invariant condition $x \leq 21$ in the mode *On*.

While the value of x stays below 20, the only admissible evolution is a continuous one constrained by the flow condition $\dot{x} = 50 - x$. So we have to find a duration $\delta > 0$ and a function f such that $f(0) = 10$ (the value of x in the previous event) and for all time $t \in [0, \delta]$, $\dot{f}(t) = 50 - f(t)$ and $f(t) \leq 21$. We notice that $\delta = \ln(40/29)$ and $f : t \mapsto 50 - 40\mathrm{e}^{-t}$ satisfy those conditions. We can set the next event of the sequence to $(\delta_1, v_1, x_1) = \big(\ln(40/29), \mathit{On}, 21\big)$, because $f(\delta) = 21$.

Now, only a discrete evolution through the edge $(\mathit{On}, \mathit{Off})$ is possible, because any continuous one would violate the invariant condition $\mathit{inv}(\mathit{On})$. So, we can define the next state of the sequence as $(\delta_2, v_2, x_2) = (0, \mathit{Off}, 21)$ such that the jump condition $x \geq 20$ of the edge $(\mathit{On}, \mathit{Off})$ and the invariant conditions $x \leq 21$ and $x \geq 19$, respectively of the modes *On* and *Off*, are satisfied.

Finally, we can repeat this procedure until x reaches 19 via a continuous evolution in the mode *Off*, then compute the discrete one through the edge $(\mathit{Off}, \mathit{On})$, and repeat the previous steps from the mode *On*. We get the definition of an admissible

trajectory $\tau = \left((\delta_i, v_i, x_i)\right)_{i\in\mathbb{N}}$ such that

$$\forall i \in \mathbb{N},\ (\delta_i, v_i, x_i) = \begin{cases} (0, On, 10) & \text{if } i = 0 \\ (\ln(40/29), On, 21) & \text{if } i = 1 \\ (0, Off, 21) & \text{if } \exists n \in \mathbb{N},\ i = 4n+2 \\ (\ln(21/19), Off, 19) & \text{if } \exists n \in \mathbb{N},\ i = 4n+3 \\ (0, On, 19) & \text{if } \exists n \in \mathbb{N},\ i = 4n+4 \\ (\ln(31/29), On, 21) & \text{if } \exists n \in \mathbb{N},\ i = 4n+5 \end{cases} \tag{2.1}$$

However, multiple other trajectories are admissible, for example with another initial state that satisfies the initial condition $x \in [10, 15]$.

We presented here a simple definition of hybrid automata that allows to model hybrid systems. Some variations of such a definition exist. For example, we can add a set of variables as *outputs* of the system. It allows easier composition of hybrid automata as presented in [104].

2.1.2 Zeno behaviors

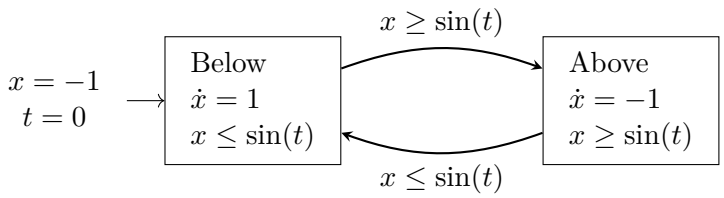
A hybrid automaton is a model that sometimes simplifies the actual system. For example, a bouncing ball may be modeled by a hybrid automaton that simplifies the bounce replacing a complex nonlinear dynamics by a discrete behavior that changes the sign of the velocity.

Having simpler models allows easier comprehension by humans, which is important for team work, and easier verification. The cost of it is a slight difference between the set of admissible trajectories of a simple model and a more precise one. This difference can be negligible, for example between a differentiable admissible trajectory and a piecewise constant one with a very high sampling frequency, but it can also be important for instance if it introduces *Zeno behaviors*.

A *Zeno behavior* is part of an admissible trajectory that contains an infinite number of jumps on all neighborhoods of a time: given a trajectory $\tau = \left((\delta_i, v_i, x_i)\right)_{i\in I}$, a *Zeno behavior* occurs if there exist $i \in I$ and $M \in \mathbb{N}$ such that $\sum_{k=i}^{\infty} \delta_k \leq M$. In that case, time cannot progress after the instant $\sum_{k=i}^{\infty} \delta_k$, which is in contradiction with physical reality. We call the time $\sum_{k=i}^{\infty} \delta_k$ a *Zeno point*.

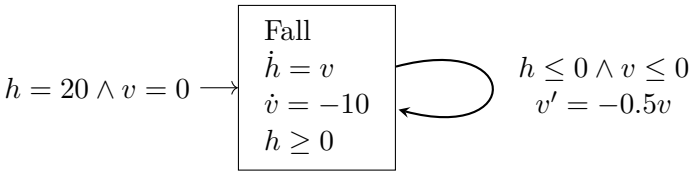
We can define two kinds of Zeno behaviors [88, 59]. The first one consists in an infinite number of transitions at the same instant: as soon as the jump is applied, a new guard allows to apply another jump, and so on. In that case, we have $\delta_i = 0$ for all i bigger than some natural number $Z \in \mathbb{N}$. This is typically the case with bang-bang controllers without hysteresis and no inertia in the continuous dynamics, as illustrated in Example 3. This first kind of Zeno behavior is sometimes called *chattering* [148]. The second kind of Zeno behaviors consists in a sequence of jumps that are closer and closer, resulting is a bounded time horizon. In that case, we have $\lim_{i\to\sup I} \delta_i = 0$, but without natural number Z such that $i \geq Z$ implies that $\delta_i = 0$. This is illustrated by Example 4.

Example 3 (Robot control with Zeno behavior of the first kind)
Consider a robot that can move at constant speed 1 on a unidimensional track. Its position is denoted x, initially set to -1, and we want it to follow a sine over time. A simple method is to move it toward positive (*resp.* negative) infinity if its position if below (*resp.* above) the value of the sine. It results in the following hybrid automaton:



In this case, as soon as the position x is equal to $\sin(t)$, an infinite number of jumps occur between the two modes. So, the time cannot progress after the smallest instant $t^* \geq t_0$ such that $x(t^*) = \sin(t^*)$, but the natural intuition is that $x(t) = \sin(t)$ for all $t \geq t^*$.

Example 4 (Bouncing ball with Zeno behavior of the second kind)
In order to illustrate the second kind of Zeno behaviors, we introduce a simple model of a bouncing ball without tangent velocity. Its state is then fully defined by a height h and a vertical velocity v.



Assuming that the initial time is $t_0 = 0$, the unique admissible infinite trajectory, *i.e.* with a set of indices $I = \mathbb{N}$, is $\tau = \Big((\delta_i, v_i, (h_i, v_i))\Big)_{i \in \mathbb{N}}$ such that

$$\forall i \in \mathbb{N},\ (\delta_i, v_i, (h_i, v_i)) = \begin{cases} (0, \text{Fall}, (20, 0)) & \text{if } i = 0 \\ (2, \text{Fall}, (0, -20)) & \text{if } i = 1 \\ (0, \text{Fall}, (0, 20 \cdot 2^{-n})) & \text{if } \exists n \in \mathbb{N}_{>0},\ i = 2n \\ (4 \cdot 2^{-n}, \text{Fall}, (0, -20 \cdot 2^{-n})) & \text{if } \exists n \in \mathbb{N}_{>0},\ i = 2n+1 \end{cases}$$

and we notice that

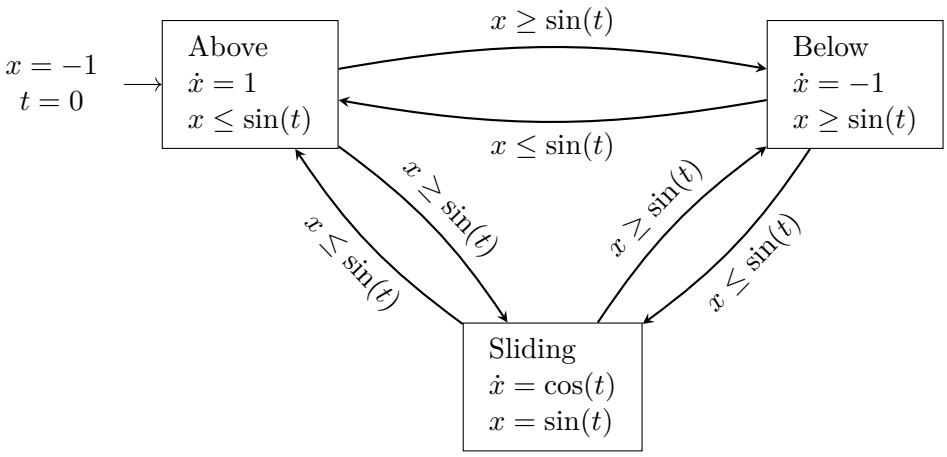
$$\sum_{i=0}^{\infty} \delta_i = 6$$

So, no admissible trajectories of the hybrid automaton can define a value of the state of the system at any instant $t^* \geq 6$. In reality, the ball stops bouncing at time $t = 6$ and stays on the ground, which is not handled by the model.

In both cases, the physical system has a state after the Zeno point. So, we want to

extend the set of admissible trajectories to allow admissible trajectories with infinite time horizon, *i.e.* $\sum_{i=0}^{\infty} \delta_i = +\infty$. In most cases, the intuitive evolution of the state is to stay on the set of states that satisfies the guards: the position of Example 3 stays equal to the sine function and the bouncing ball of Example 4 stays on the ground with a null velocity. On more complex hybrid automata, we can imagine Zeno behaviors that involve more than two transitions: a first transition e_1 is taken, resulting in the activation of a second transition e_2 that, if taken, results in the activation of a third transition e_3 whose jump results in the activation of the first transition e_1. A solution is to add an extra mode, called *sliding mode*[144], that allows extra admissible trajectories. This is illustrated in Example 5. Other methods consists in modifying the model to add some hysteresis, ensuring that the system stays in some modes for a non-null duration of time [88]. Those last methods are often used for simulations, because every deterministic automaton is transformed into a new deterministic automaton, but some admissible trajectories of the original automaton are not admissible trajectories of the resulting automaton.

Example 5 (Robot with sliding mode)
We can explicit a sliding mode of the hybrid automaton described in Example 3 of the robot control with constant input:



Every admissible trajectory of the automaton of Example 3 is an admissible trajectory of the automaton with sliding mode. Moreover, there exist admissible trajectories of the new automaton that do not have Zeno behaviors. For example, consider the trajectory $\tau = (\delta_i, v_i, x_i)_{i \in \mathbb{N}}$ defined by

$$\forall i \in \mathbb{N},\ (\delta_i, v_i, x_i) = \begin{cases} (0, \text{Below}, -1) & \text{if } i = 0 \\ (t^*, \text{Below}, \sin(t^*)) & \text{if } i = 1 \\ (0, \text{Sliding}, \sin(t^*)) & \text{if } i = 2 \\ (1, \text{Sliding}, \sin(t^* + i - 2)) & \text{if } i \geq 3 \end{cases}$$

with t^* the unique solution of the equation $t^* - 1 = \sin(t^*)$. τ is an admissible trajectory

of the hybrid automaton without Zeno behaviors:

$$\forall t \in \mathbb{R}_{\geq 0},\ \exists n \in \mathbb{N} :\ \sum_{i=0}^{n} \delta_i \geq t$$

2.1.3 Switched systems

In this work, we focus on hybrid automata with no jumps, *i.e.* $x' = x$ for all transitions, and in which the current mode does not depend on the past of the trajectories but only on the current states, times and inputs (the arguments of the right-hand side of the ordinary differential equations). We use the extension proposed by A. F. Filippov [63] in the case of ordinary differential equations with discontinuous right-hand sides. Such systems are called *switched systems*. We only consider in this work a subclass of the switched systems as defined by D. Liberzon [102]: we focus on state-dependent, time-dependent and autonomous switched systems. Example 3 is a typical case of switched system: if the position x is strictly lower (*resp.* greater) than $\sin(t)$, then the system is in the mode Below (*resp.* Above) that moves the robot in the right direction. Such *switched systems* can be rewritten as a simple system of ordinary differential equations with discontinuous right-hand side functions encoding the different modes. This is the property that we use to define switched systems. In that case, the differential equations have to be interpreted in the sense of Carathéodory (*cf.* Subsection 2.3.3 or [63, §1]), *i.e.* they should be satisfied for almost all time t in a given time interval $[t_0, t_1]$, denoted $\forall_{a.e.} t \in [t_0, t_1]$.

Definition 4 (Switched system)
Let I be a finite set of indices, $\{f_i\}_{i \in I}$ be a collection of locally Lipschitz functions with respect to their second argument (the state variables) from $\mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m$ to $\mathbb{R}^n$, and G be an upper hemicontinuous function from $\mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m$ to the powerset $\mathcal{P}(I)$. We assume that the set M of points whose image through G is not a singleton has measure zero.

A *switched system* is a dynamical system of the form

$$\forall_{a.e.} t \in [t_0, t_1],\ \begin{cases} \dot{x}(t) = f_i(t, x(t), u(t)) \\ i \in G(t, x(t), u(t)) \end{cases} \tag{2.2}$$

with t the time variable in any time interval $[t_0, t_1]$, x the vector of state variables of the system in $\mathbb{R}^n$ and u the vector of input variables in $\mathbb{R}^m$.

We call *dynamics* of the system the functions f_i and *switching signal* the function G.

When we say that a subset M of a set $\mathbb{R}^n$ ($M \subset \mathbb{R}^n$) has measure zero, it means intuitively that if we pick uniformly an element in $\mathbb{R}^n$, the probability that this element belongs to M is null. Another intuition is that the n-dimensional volume of M is equal to zero. For example, M could be a countable set of points in $\mathbb{R}$, a curve in $\mathbb{R}^2$, or a surface in $\mathbb{R}^3$.

A set-valued function G from a set A to a powerset $\mathcal{P}(B)$ of a set B is *upper hemi-continuous* [21, Definition 1] if for any neighborhood V of the image $G(a) \subset B$ of a point $a \in A$, there exists a neighborhood U of a such that for all $x \in U$, $G(u) \subset V$. In the case of a switched system, because B is the discrete set I of indices, the function G is constant over any connected subset of the complement of M in $\mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m$. Moreover, for any index i, the subset of the domain of G whose image contains i is a closed set.

We did not introduce the parameters in this definition, because they can be interpreted as constant input signals, *i.e.* a parameter p is equivalent to a constant input signal u. Moreover, a switched system is often combined with an initial condition, similar to hybrid automata but without the restriction to a mode, *i.e.* there exists a set $\mathit{Init} \subset \mathbb{R} \times \mathbb{R}^n$ defining the possible initial time and initial values of the state variables: $(t_0, x(t_0)) \in \mathit{Init}$. Some additional restrictions on the possible parameters and inputs can also be considered and they are specified when needed in the following.

Notice that we can consider some hybrid automata as switched systems if the invariants define a cover of the space (*i.e.* for all triples (x, p, u), there exists a mode $v \in V$ such that $\mathit{inv}(v)$ is satisfied), the interiors of the invariants are disjoints (*i.e.* the measure of the intersection of two different invariants is null), there exists a transition between all adjacent modes, the jumps are identities on the state (*i.e.* for all transition $e \in E$, $\mathit{jump}(e)(x, p, u) = x$), and the guards are satisfied if and only if the corresponding invariants are satisfied (*i.e.* for all transition $e_{i_1,i_2} \in E$ from a mode v_{i_1} to a mode v_{i_2}, $\mathit{inv}(v_{i_1})$ and $\mathit{inv}(v_{i_2})$ are satisfied if and only if $\mathit{guard}(e_{i_1,i_2})$ is satisfied). In that case, for all time t and state x, $G(t, x)$ is equal to the set of indices i such that the invariant $\mathit{inv}(v_i)$ associated to the mode v_i is satisfied on (t, x). Moreover, for any index i and the associated mode v_i, the dynamics f_i is equal to the flow condition $\mathit{flow}(v_i)$.

Conversely, a switched system can easily be considered as a hybrid automaton. Each index $i \in I$ defines a mode v_i, whose invariant $\mathit{inv}(v_i)$ is satisfied if and only if $i \in G(t, x)$ and whose dynamics $\mathit{flow}(v_i)$ is equal to f_i. Each ordered pair of indices $(i_1, i_2) \in I \times I$ defines a transition e_{i_1,i_2} without jumps ($\mathit{jump}(e_{i_1,i_2})(x, p, u) = x$), whose guard is activated if and only if the invariants of the two modes are satisfied ($\mathit{guard}(e_{i_1,i_2}) = \mathit{inv}(v_{i_1}) \wedge \mathit{inv}(v_{i_2})$). So, the definition of admissible trajectories (Definition 3) is still valid. With this definition, switched systems allow the existence of admissible trajectories with Zeno behaviors as soon as there exist admissible trajectories that activate different modes, because all the transitions are bidirectional and an admissible trajectory may infinitely switch between two adjacent modes.

A possibility to always allow trajectories beyond a Zeno point is to replace the equality constraint for all time on the derivative of the state by an inclusion constraint for almost all time. The right-hand side of the differential equation becomes a set-valued function, whose image is a singleton in the interior of a mode and a convex hull of multiple values when a guard is activated. Instead of a regularization using sliding modes, this method does not add extra constraints to the dynamics and it allows all possible behaviors [63].

Consider a switched system S defined by a collection of dynamics $(f_i)_{i \in I}$ and a switching signal G. We define a set-valued function F with the same domain of the

functions f_i (*i.e.* $\mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m$) to the set of subsets of $\mathbb{R}^n$ (*i.e.* $\mathcal{P}(\mathbb{R}^n)$) such that for any point $y = (t, x, u) \in \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m$, its value is equal to the convex hull of all values of derivatives $f_i(y)$ with $i \in G(y)$:

$$\forall (t, x, u) \in \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m,\ F(t, x, u) = \mathrm{CH}(\{f_i(t, x, u) \mid i \in G(t, x, u)\}) \tag{2.3}$$

We can now define the *switched system in the sense of Filippov* S', corresponding to the switched system S, as a differential inclusion for almost all time:

$$\forall_{a.e.} t \in [t_0, t_1],\ \dot{x}(t) \in F\big(t, x(t), u(t)\big) \tag{2.4}$$

This is also equivalent [63, 125] to consider that there exists a Lebesgue-measurable function α such that for all time $t \in [t_0, t_1]$, $\alpha(t)$ belongs to $F\big(t, x(t), u(t)\big)$ and

$$\forall_{a.e.} t \in [t_0, t_1],\ \dot{x}(t) = \alpha(t) \tag{2.5}$$

or equivalently (*cf.* Subsection 2.3.3)

$$\forall t \in [t_0, t_1],\ x(t) = x(t_0) + \int_{t_0}^{t} \alpha(s)\, ds \tag{2.6}$$

So, we add an extra input function α to the system in order to allow admissible trajectories to be defined beyond Zeno points. The definition of such new admissible trajectories is similar to Definition 3 replacing the modes by a collection of indices and the constraint on the derivative for all time by a constraint on its integral (equivalent to a constraint on the derivative for *almost* all time).

Definition 5 (Admissible trajectory of a switched system)
Let S' be a switched system in the sense of Filippov and let $\tau = \big((\delta_i, V_i, x_i)\big)_{i \in I}$ be a trajectory of it.

Let $\Delta_i = \sum_{k=0}^{i} \delta_k$ be the sum of all the previous elapsed times, *i.e.* the duration from the origin of time.

Given a set *Init* of possible initial states, τ is an *admissible trajectory* of S', or S' *admits* the trajectory τ, if and only if there exist a function $u : \mathbb{R}_{\geq 0} \to \mathbb{R}^m$ encoding the values of the inputs depending on the time and a Lebesgue-integrable function α such that the following conditions are satisfied:

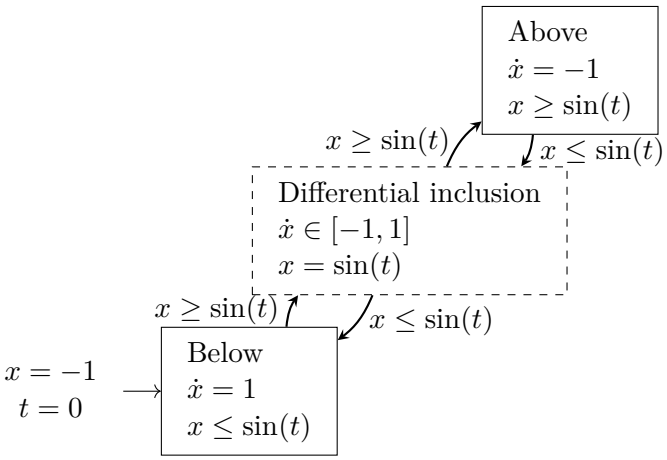
1. The first triple is a valid initial state, *i.e.* $x_i \in \mathit{Init}$ and $V_i \subset G\big(\delta_0, x_i, u(\delta_0)\big)$.
2. There exists a valid evolution between two successive states, *i.e.* for all $i \in I \backslash \{0\}$:
   - if $\delta_i = 0$, then $x_i = x_{i-1}$ and $V_i \subset G\big(\delta_0, x_i, u(\delta_0)\big)$;
   - otherwise $V_i = V_{i-1}$ and there exists a function f over $[0, \delta_i]$ such that $f(0) = x_{i-1}$, $f(\delta_i) = x_i$ and for all positive reals $\xi \in [0, \delta_i]$, $V_i \subset G\big(\Delta_{i-1} + \xi, f(\xi), u(\Delta_{i-1} + \xi)\big)$,

$$f(\xi) = f(0) + \int_0^{\xi} \alpha(\Delta_{i-1} + s)\, ds$$

and

$$\alpha(\Delta_{i-1}+\xi) \in F\big(\Delta_{i-1}+\xi, f(\xi), u(\Delta_{i-1}+\xi)\big)$$

Example 6 (Robot interpreted in the sense of Filippov)
Consider the model of a control of a robot presented in Example 3. If $x(t) = \sin(t)$, then the mode is uncertain: the system switches infinitely many times between the mode Below and the mode Above. We want to allow an admissible trajectory that stays in a virtual sliding mode as described by the following hybrid automaton:



We start by rewriting the model as a switched system:

$$\forall t \in [t_0, t_1],\ \begin{cases} \dot{x}(t) = 1 & \text{if } x(t) < \sin(t) \quad \text{(mode Below)} \\ \dot{x}(t) = -1 & \text{if } x(t) > \sin(t) \quad \text{(mode Above)} \\ \dot{x}(t) \in \{-1, 1\} & \text{if } x(t) = \sin(t) \quad \text{(on the boundary)} \end{cases} \tag{2.7}$$

with the initial state $x(t_0) = -1$ and initial time $t_0 = 0$. We can also explicitly define the dynamics f_i and the switching signal G:

$$\forall (t, x) \in [t_0, t_1] \times \mathbb{R},\ \begin{cases} f_0(t, x) = 1 \\ f_1(t, x) = -1 \\ x < \sin(t) \implies G(t, x) = \{0\} \\ x > \sin(t) \implies G(t, x) = \{1\} \\ x = \sin(t) \implies G(t, x) = \{0, 1\} \end{cases} \tag{2.8}$$

This definition guarantees the existence of Zeno behaviors in the neighborhood of $x(t) = \sin(t)$: the derivative has to switch from 1 to -1 as soon as $x(t) > \sin(t)$ and from -1 to 1 as soon as $x(t) < \sin(t)$. So, we define for almost all time t a convex set

of possible values of the derivative:

$$\forall_{a.e.} t \in [t_0, t_1],\ \dot{x}(t) \in F(t, x(t)) \quad \text{with} \quad F(t,x) = \begin{cases} \{1\} & \text{if } x < \sin(t) \\ \{-1\} & \text{if } x > \sin(t) \\ [-1,1] & \text{if } x = \sin(t) \end{cases} \tag{2.9}$$

Notice that in the interior of the modes ($x < \sin(t)$ or $x > \sin(t)$), the differential equation is equivalent to the original definition in equation 2.7. We can also interpret this differential inclusion as a differential equation depending on a constrained Lebesgue-measurable input α such that

$$\forall_{a.e.} t \in [t_0, t_1],\ \dot{x}(t) = \alpha(t) \quad \text{with} \quad \forall t \in [t_0, t_1],\ \alpha(t) \in F(t, x(t)) \tag{2.10}$$

In that case, let t^* be the unique solution of $t^* - 1 = \sin(t^*)$ and consider the input variable α defined by

$$\forall t \in \mathbb{R}_{\geq 0},\ \alpha(t) = \begin{cases} 1 & \text{if } t < t^* \\ \cos(t) & \text{if } t \geq t^* \end{cases} \tag{2.11}$$

Then, the function x defined by

$$\forall t \in \mathbb{R}_{\geq 0},\ x(t) = \begin{cases} t - 1 & \text{if } t < t^* \\ \sin(t) & \text{if } t \geq t^* \end{cases} \tag{2.12}$$

is solution of the switched system in the sense of Filippov (its derivative is defined for all time $t \neq t^*$ and it is equal to the value of α) and it is defined beyond a potential Zeno point at time t^*.

So, an associated admissible trajectory $\tau = \Big((\delta_i, v_i, x_i)\Big)_{i \in \mathbb{N}}$ could be defined as

$$\forall i \in \mathbb{N},\ (\delta_i, v_i, x_i) = \begin{cases} (0, \{\text{Below}\}, -1) & \text{if } i = 0 \\ (t^*, \{\text{Below}\}, \sin(t^*)) & \text{if } i = 1 \\ (1, \{\text{Below}, \text{Above}\}, \sin(t^* + i - 1)) & \text{if } i \geq 2 \end{cases} \tag{2.13}$$

The two first events are in the mode Below while the following events belong to the differential inclusion between the modes Below and Above.

2.2 Models in the Zélus language

While hybrid automata are convenient to study hybrid systems, they are not well adapted to design them. To design such systems, models have to exploit modularity: sub-systems such as electronic or mechanical components should be able to be modeled independently and combined in order to define more complex systems. Moreover, programs of controllers may require complex computations that would result in huge automata.

Software such as SIMULINK[1] or languages such as MODELICA[2] exploit modularity to allow users to define subsystems (*e.g.* blocks in SIMULINK) and to put them together (*e.g.* by connecting input and output signals in SIMULINK). However, their semantics are not well defined, which does not allow to compute any reachability analyses. Formal semantics can be defined for a subset of SIMULINK [39] to compensate for its absence on the whole language. Models written in SIMULINK can also be translated into networks of hybrid automata [91] to compute reachability analyses.

2.2.1 The Zélus language

In this work, we focus on the ZÉLUS[3] [41] language developed by the PARKAS team[4] at the École Normale Supérieure, which allows to implement a subclass of the base components of SIMULINK [40]. It is based on the synchronous dataflow programming language LUSTRE[5] [81], which has a well defined semantics. Programs written in LUSTRE [86] are composed of *nodes* that represents components with inputs and outputs. Their execution is assumed to be instantaneous and triggered by the inputs. A sequence of values for the inputs implies the computation of a sequence of values for the outputs. Such sequences are called *flows* or *signals* and allow to define a logical notion of time. The presence of values in the inputs defines an instant called *(logical) tick* and it allows to refer to the previous values at the previous ticks. The main restriction of such programs is the absence of syntactical causality loops to avoid any computation of fixed points. This restriction is illustrated in Example 8.

Example 7 (Counter in Lustre)
Consider the following Lustre program that defines a counter incrementing a value every time it receives a true value as input:

```
node counter(t : bool) returns (c : int);
let
  c = (0 -> pre c) + (if t then 1 else 0);
tel
```

The node `counter` takes as input a signal of Boolean values and returns a signal `c` of integers. The operator `->` defines a signal with an initial value (its left-hand side operand) following by other values (its right-hand side operand). So, at the initial tick, `0 -> pre c` is evaluated to `0`. For all following tick, it is evaluated to the previous computed value of the signal associated to the variable `c`.

In the following table, we present an example of execution of such a program with an arbitrary input signal `t`:

[1] https://www.mathworks.com/products/simulink.html
[2] https://modelica.org/
[3] https://zelus.di.ens.fr/
[4] https://parkas.di.ens.fr/
[5] https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/reactive-toolbox/

tick	0	1	2	3	4	5	6	7
t	false	false	true	false	true	true	true	false
c	0	0	1	1	2	3	4	4
pre c	-	0	0	1	1	2	3	4

Nodes can be composed to define more complex programs. For example, we present below a program that reuses the counter with a rising edge detector to count the number of rising edges:

```
node edge(t : bool) returns (e : bool);
let
  e = false -> t and not pre t;
tel

node edge_counter(t : bool) returns (ec : int);
  var e : bool;
let
  ec = counter(e);
  e = edge(t);
tel
```

With the same input signal, it results in the following execution of `edge_counter`:

tick	0	1	2	3	4	5	6	7
t	false	false	true	false	true	true	true	false
e	false	false	true	false	true	false	false	false
ec	0	0	1	1	2	2	2	2

Example 8 (Invalid syntactical loop)
Consider the following node:

```
node invalid_loop (c : bool) returns (x : int, y : int);
let
  x = if c then 1 else y;
  y = if c then x else 2;
tel
```

The signal associated to `x` may depend instantaneously on the one associated to `y` and reciprocally. However, if `c` is `true`, then only `y` depends on `x`. Reciprocally, if `c` is `false`, then only `x` depends on `y`. In both cases, there are no causality loops.

Such absences of causality loops are difficult to detecting statically. A sufficient condition to avoid possible causality loops is to reject any syntactical loop. Lustre exploits this sufficient condition. So, this example is not a valid program in Lustre because of the presence of the syntactical loop between `x` and `y`.

The Zélus language is based on the same structure to which it adds *continuous signals* in opposition to classical ones that are then called *discrete signals*. Depending on the context, the time can be logical (classical synchronous interpretation) or continuous.

The main mechanism to align the logical time on specific instants of the continuous one is the detection of zero-crossing events [148], which is classical in tools designed for simulation such as SIMULINK[6] or MODELICA[7]. A *zero-crossing event* is the instant at which a signal becomes non-negative. While its semantics in the case of the ZÉLUS language is defined as a non-standard one [31] and requires that the signal become positive (instead of non-negative) [28, Section 3.1], we provide a standard semantics of the operator `up` that triggers a rising zero-crossing of its operand signal:

$$\forall (t,n) \in \mathbb{R} \times \mathbb{N},\ up(x)_{(t,n)} = \Big(x_{(t,n)} \geq 0\Big) \wedge \Big(\forall \varepsilon > 0,\ \exists \delta \in [0,\varepsilon] :\ x_{(t-\delta,n)} < 0\Big) \quad (2.14)$$

where the indices are super-dense times [108, 110, 101, 103] at which the corresponding signals are evaluated: (t,n) is a super-dense time with t the real time and n the number of past discrete steps. With this definition, we consider that the operator `up` cannot trigger zero-crossing event in discrete contexts as currently implemented in ZÉLUS and illustrated by Example 10. This operator returns a special signal called *zero-crossing signal* whose values only exist at when a zero-crossing event occurs. A special structure allows to define a discrete context based on a zero-crossing signal:

```
present z -> do E done
```

where `z` is a zero-crossing signal (*e.g.* `up(x)`) and `E` is a set of expressions that will be evaluated in a discrete context as soon as `z` exists.

Continuous signals may be defined as combinatorial expressions of continuous signals. For example, if `x` and `y` are continuous signals, then `x + y` defines a continuous signal. Alternatively, continuous signals may be defined via ordinary differential equations:

```
der x = e1 init e0 reset z -> e2
```

defines a continuous signal associated to the variable `x` that is equal to the expression `e0` at the initial time and whose derivative is equal to the expression `e1` in continuous contexts and as soon as the zero-crossing signal `z` is true, the value of `x` is reset to the value of `e2`. If the signal is never reset in the given context, then the operator `reset` can be omitted:

```
der x = e1 init e0
```

Values of discrete signals are not defined in continuous contexts, but values of continuous signals can be manipulated in discrete contexts. The keyword `last` can be used to access the last defined value of a signal (continuous or discrete). It allows to use the value of a continuous signal in a discrete context, as illustrated in Example 9.

[6] https://www.mathworks.com/help/simulink/ug/zero-crossing-detection.html

[7] https://doc.modelica.org/Modelica%204.0.0/Resources/helpDymola/Modelica_Blocks_Logical.html#Modelica.Blocks.Logical.ZeroCrossing

Example 9 (Bouncing ball in Zélus)
Consider the following program modeling a bouncing ball with a counter of the bounces written in ZÉLUS:

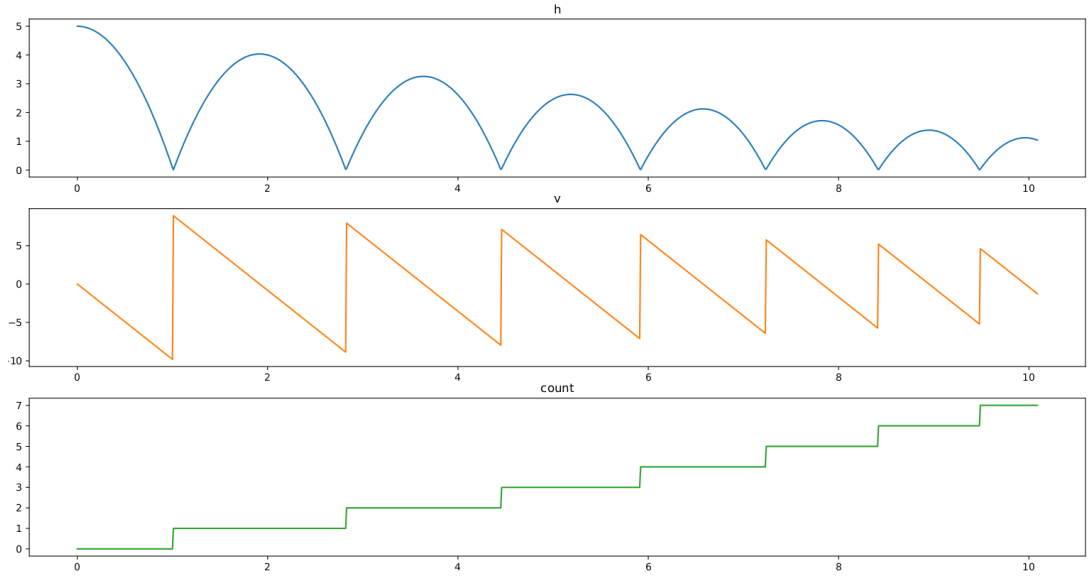
```
let g = 9.81

let hybrid bouncing_ball(h0) = (h,v,c) where
    rec der v = -. g init 0. reset z -> -. 0.9 *. last v
    and der h = v init h0
    and z = up(-.h)
    and init c = 0
    and present z -> do c = last c + 1 done
```

The gravity on Earth is defined as a global constant `g`. The bouncing ball and the counter are defined in a single node `bouncing_ball` that returns the height of the ball `h`, its vertical velocity `v` and the number `c` of past bounces. The variable `z` defines a zero-crossing signal that is activated at every instant the ball hits the ground, *i.e.* `x` (*resp.* `-.x`) becomes non-positive (*resp.* non-negative).

The (continuous) dynamics of `h` is given by a unique initial value problem: derivative equal to `v` and initial value equal to the initial value of `h0`. The dynamics of `v` is given by an initial value problem with its derivative equal to `-.g` and its initial value equal to zero, but its value is multiplied by `-.0.9` every time `z` is activated, *i.e.* every time the ball hits the ground. Finally, `c` defines an integer continuous signal (in contrary to `h` and `z` that are encoded using floating-point numbers) with initial value equal to zero. No definitions of its value in continuous contexts are provided, so it is assumed to remain constant. However, when `z` is activated, its value is incremented by one as defined in the last line of the definition of the node.

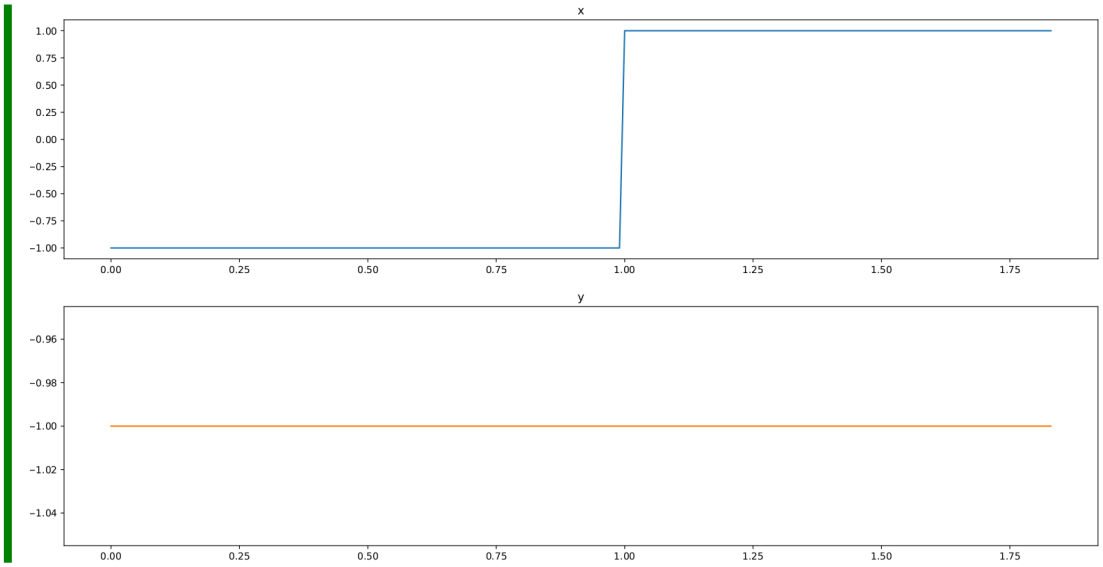
It can be simulated with a constant signal `h0` equal to `5.` (only its initial value really matters) and printing the value of its outputs every 10^{-2} seconds results in the following graph of each output signal with respect to time (from $t_0 = 0$):



Example 10 (No zero-crossing during discrete steps)
Consider the following example:

```
let hybrid test_zeroc(t) = (x,y) where
    rec init x = -1.
    and init y = -1.
    and present up(x) -> do y = 1. done
    and present up(t-.1.) -> do x = 1. done
```

The continuous signal `x` becomes equal to 1 as soon as the continuous input signal `t` becomes greater or equal to 1. The continuous signal `y` should become equal to 1 as soon as the continuous signal `x` becomes non-negative. However, with our definition of the operator `up`, the signal `y` should remain equal to -1. We indeed obtain this behavior by simulating this model with `t` equal to the time (*i.e.* `der t = 1. init 0.`):



Contrarily to hybrid automata, ZÉLUS models are deterministic. Then, given input signals, each ZÉLUS model admit a unique trajectory. The language has also been extended into PROBZELUS[8] [25] that introduces probabilistic signals and allows to infer the value of uncertain parameters based on observations [26].

Properties on models written in ZÉLUS can be verified thanks to its formal semantics. In particular, the tool ZLSCHECK[9] has been developed to perform falsification [61, 60]. Reachability analysis has also been computed for a subset of the language: first with only differential equations [43] and recently with automata [128]. In the two cases, the models written in ZÉLUS are first compiled into C++11 and guaranteed integration is computed using the DYNIBEX[10] [135, 119].

2.3 Theory of integration and ordinary differential equations

In the previous section, we defined hybrid systems relying on ordinary differential equations to model the continuous dynamics.

We consider ordinary differential equations of the form $\dot{x}(t) = f\big(t, x(t), p, u(t)\big)$ with $x(t)$ the vector of values of the state variables over the time, $\dot{x}(t)$ the vector of their derivatives over the time, p the vector of values of the parameters, and $u(t)$ the vector of values of the input variables over the time. In this section, we assume known p and u. So, we reduce the problem to ordinary differential equations of the form $\dot{x}(t) = f\big(t, x(t)\big)$.

We now present some results about existence and uniqueness of the solutions of ordinary differential equations. This section is composed of three parts: the first one focuses

[8] `https://github.com/IBM/probzelus`
[9] `https://github.com/ismailbennani/zlscheck`
[10] `https://perso.ensta-paris.fr/~chapoutot/dynibex/`

on continuous ordinary differential equation, the second one presents the theory of integration in the sense of Lebesgue, and the last one considers non-continuous right-hand sides of ordinary differential equations. We do not provide any proof of the theorems, but some can be found in [138].

2.3.1 Continuous ordinary differential equations

Consider the ordinary differential equation

$$\forall t \in [t_0, t_1],\ \dot{x}(t) = f\big(t, x(t)\big) \tag{2.15}$$

First, we want the initial value problem to have at least one solution for each possible initial state. It guarantees that an execution will not be blocked due to the flow equations. The Peano Existence Theorem proves that if the right-hand side of the ordinary differential equation is continuous, then there exists a solution in the neighborhood of the initial time and initial state.

Theorem 1 (Peano existence theorem [138, Theorem I-2-5])
Let $(t_0, x_0) \in \mathbb{R} \times \mathbb{R}^n$ be the initial conditions, f be a continuous function on a domain

$$D = \{(t, x) \mid \|t - t_0\| \leq a \wedge \|x - x_0\|_\infty \leq b\}$$

and $M = \max_D \|f(t, x)\|_\infty$.

Then, there exists a function $\varphi :]t_0 - \alpha, t_0 + \alpha[$ such that

- $\varphi(t_0) = x_0$
- $\forall t \in]t_0 - \alpha, t_0 + \alpha[,\ \dot{\varphi}(t) = f\big(t, \varphi(t)\big)$

with

$$\alpha = \begin{cases} a & \text{if } M = 0 \\ \min\left(a, \frac{b}{M}\right) & \text{if } M > 0 \end{cases}$$

Notice that Theorem 1 only guarantees the existence of solution in the neighborhood of the initial time, but not on the entire time interval.

Example 11
Consider the system

$$\begin{cases} \dot{x}(t) = x(t)^2 \\ x(0) = 1 \end{cases} \tag{2.16}$$

A solution of this system is

$$\forall t \in [0, 1],\ x(t) = \frac{1}{1 - t} \tag{2.17}$$

and we can prove later that this is the unique solution of the system.

However, no solutions exist at time $t = 1$.

We also want the uniqueness of the solution of ordinary differential equations for a given initial state. The Picard-Lindelöf Theorem proves that if the function of Theorem 1 satisfies a Lipschitz condition, then the solution is unique.

Theorem 2 (Picard-Lindelöf [138, Theorem I-1-4])
Let $(t_0, x_0) \in \mathbb{R} \times \mathbb{R}^n$ be the initial conditions, f be a continuous function on a domain

$$D = \{(t, x) \mid \|t - t_0\| \leq a \wedge \|x - x_0\|_\infty \leq b\}$$

and $M = \max_D \|f(t, x)\|_\infty$.

If there exists a constant $L > 0$ such that

$$\forall((t, x_1), (t, x_2)) \in D \times D,\ \|f(t, x_1) - f(t, x_2)\|_\infty \leq L\|x_1 - x_2\|_\infty \tag{2.18}$$

then, there exists a unique function $\varphi :]t_0 - \alpha, t_0 + \alpha[$ such that

- $\varphi(t_0) = x_0$
- $\forall t \in]t_0 - \alpha, t_0 + \alpha[,\ \dot{\varphi}(t) = f\big(t, \varphi(t)\big)$

with

$$\alpha = \begin{cases} a & \text{if } M = 0 \\ \min\left(a, \frac{b}{M}\right) & \text{if } M > 0 \end{cases}$$

We can show that if the Lipschitz condition 2.18 is not satisfied, multiple solutions may exist from the same initial state.

Example 12
Consider the system

$$\begin{cases} \dot{x}(t) = \sqrt{x(t)} \\ x(0) = 0 \end{cases} \tag{2.19}$$

The function $v \mapsto \sqrt{v}$ is not Lipschitz continuous in the neighborhood of $v = 0$. The two functions x_1 and x_2 defined by

$$\forall t \in \mathbb{R}_{\geq 0},\ x_1(t) = 0 \quad \text{and} \quad \forall t \in \mathbb{R}_{\geq 0},\ x_2(t) = \frac{t^2}{4} \tag{2.20}$$

are both solutions of the system.

While Theorem 2 does not explicitly provide an expression of the unique solution, a classical proof of it defines a recursive sequence of functions that converges to the unique solution. Consider the system

$$\begin{cases} \dot{x}(t) = f\big(t, x(t)\big) \\ x(t_0) = x_0 \end{cases} \tag{2.21}$$

with f a Lipschitz continuous function as in the Theorem 2, $t_0 \in \mathbb{R}$ an initial time and $x_0 \in \mathbb{R}^n$ an initial value. We define a sequence of functions $(\varphi_n)_{n \in \mathbb{N}}$ from $[t_0, t_0 + \alpha[$ to

$\mathbb{R}^n$ as

$$\begin{cases} & \forall t \in [t_0, t_1], \quad \varphi_0(t) = x_0 \\ \forall n \in \mathbb{N}, & \forall t \in [t_0, t_1], \quad \varphi_{n+1}(t) = x_0 + \int_{t_0}^{t} f(s, \varphi_n(s))\, ds \end{cases} \tag{2.22}$$

This sequence uniformly converges to a function φ that is solution of the system 2.21. For a detailed proof of this result, the reader can refer to [138, Proof of the Theorem I-1-4].

We will often use such an iteration in this document, so we define the *Picard operator* associated to such a system as the operator defining φ_{n+1} from φ_n. This operator is parameterized by the parameters of the system (in this example, x_0 and t_0).

Definition 6 (Picard operator)
Let $(t_0, x_0) \in \mathbb{R} \times \mathbb{R}^n$ be an initial state and let f be a Lipschitz continuous function as defined in Theorem 2.

The *Picard operator* associated to the system

$$\begin{cases} \dot{x}(t) = f(t, x(t)) \\ x(t_0) = x_0 \end{cases}$$

is an endomorphism of the set of continuous functions defined by

$$\mathbb{P}(\varphi) = t \mapsto x_0 + \int_{t_0}^{t} f(s, \varphi(s))\, ds$$

That ends the introduction about ordinary differential equations with continuous right-hand sides. In the next subsection, we introduce the theory of integration in the sense of Lebesgue, which is needed to handle ordinary differential equations whose right-hand sides are discontinuous.

2.3.2 Integral in the sense of Lebesgue

In the previous subsection, we presented some results about ordinary differential equations whose right-hand sides are continuous with respect to the time and to the state. However, the right-hand side of an ordinary differential equation depends often on an input function that is only partially known. Especially, a differential inclusion such as 2.4 may be interpreted as such an ordinary differential equation with input function. We call such a partially known function a *time-varying uncertainty*. A classical hypothesis on a time-varying uncertainty u is a given bounded range, *e.g.* a bounded set U is known such that for all time t, $u(t)$ belongs to U.

We want to use the weakest hypotheses that guarantee the existence and uniqueness of a solution from a given initial state. We decide to handle Lebesgue-measurable time-varying uncertainties, which can be considered as the weakest reasonable hypothesis, because it is impossible to exhibit a non-Lebesgue measurable set without the axiom of choice in the Zermelo-Frankel set theory [140]. This impossibility involves concepts well beyond the focus of this thesis and we do not give any details about it.

So, we present in this subsection the Lebesgue's theory of integration. Instead of properly defining every concept, we try to give an intuition with examples compared to

the Riemann integration. We only consider unidimensional functions, *i.e.* from a subset of $\mathbb{R}$ to a subset of $\mathbb{R}$, but that can be generalized to multidimensional functions, *i.e.* from a subset of $\mathbb{R}^n$ to a subset of $\mathbb{R}^m$.

The integral in the sense of Lebesgue

First, we need to introduce the *Lebesgue measure* μ. It is a generalization of the notion of length (or surface and volume in higher dimensions). The Lebesgue measure can be interpreted as a function from a set of sets $\Sigma \subset \mathcal{P}(\mathbb{R})$ to the set of non-negative real numbers extended with infinity $\mathbb{R}_{\geq 0} \cup \{\infty\}$. A set is *measurable* if it belongs to the domain Σ of the *Lebesgue measure*. Because it generalizes the notion of length, for any interval $[a, b]$, we have $\mu([a, b]) = b - a$. The empty set is measurable and has null measure ($\mu(\emptyset) = 0$). Finally, for any countable collection of disjoint measurable sets $(E_i)_{i\in\mathbb{N}}$, the measure of the union is equal to the sum of the measures of the sets:

$$\Big(\forall (i,j) \in \mathbb{N}^2,\ i \neq j \implies E_i \cap E_j = \emptyset\Big) \implies \mu\left(\bigcup_{i\in\mathbb{N}} E_i\right) = \sum_{i\in\mathbb{N}} \mu(E_i) \tag{2.23}$$

We can then define the Lebesgue integral of *simple functions*, which are functions defined as a finite linear combination of indicator functions over measurable sets. Consider a simple function f_s. By definition, there exists a collection of measurable sets $(E_i)_{i\in[\![1,N]\!]}$ and a collection of real numbers $(a_i)_{i\in[\![1,N]\!]}$ such that

$$\forall x \in \mathbb{R},\ f_s(x) = \sum_{i=1}^{n} a_i \mathbb{1}_{E_i}(x) \tag{2.24}$$

Then, its integral over a bounded measurable set D is given by the formula

$$\int_D f_s \, d\mu = \sum_{i=1}^{n} a_i \mu(E_i \cap D) \tag{2.25}$$

For a non-negative function f, we define its Lebesgue-integral over a bounded measurable set D as the supremum of the integral over D of simple functions smaller or equal to f:

$$\int_D f \, d\mu = \sup\left\{ \int_D f_s \, d\mu \;\middle|\; f_s \text{ simple function and } f_s \leq f \right\} \tag{2.26}$$

Finally, for all real functions f with positive part f^+ and negative part f^- (f^+ and f^- are non-negative functions such that $f = f^+ - f^-$ and $|f| = f^+ + f^-$), we define its Lebesgue-integral over a bounded measurable set D as the difference of the integrals of f^+ and f^-:

$$\int_D f \, d\mu = \int_D f^+ \, d\mu - \int_D f^- \, d\mu \tag{2.27}$$

Comparison with the integral in the sense of Riemann

A classical definition of the integral is the one in the sense of Riemann. Consider a real interval $[a, b] \subset \mathbb{R}$. A function f is integrable in the sense of Riemann, with the integral of value I, if for all real $\varepsilon > 0$, there exists a real $\delta > 0$ such that for all sequences $(x_i)_{i\in[\![0,n]\!]}$ and $(t_i)_{i\in[\![0,n-1]\!]}$ such that for all $i \in [\![0, n-1]\!]$, $x_i \leq t_i \leq x_{i+1}$ and $x_{i+1} - x_i \leq \delta$, we have

$$\left|\sum_{i=0}^{n-1}(x_{i+1} - x_i)f(t_i) - I\right| \leq \varepsilon \tag{2.28}$$

and the integral in the sense of Riemann of f over $[a, b]$ is denoted

$$\int_a^b f(t)\,dt \tag{2.29}$$

where t can be any name of variable.

In particular, consider a sequence $(s_n)_{n\in\mathbb{N}^*}$ defined by

$$\forall n \in \mathbb{N}^*,\ s_n = \sum_{i=0}^{n-1}(x_{i+1} - x_i)f(t_i) \tag{2.30}$$

with for all $n \in \mathbb{N}^*$, $a = x_0$, $b = x_n$ and $i \in [\![0, n-1]\!]$, $x_i \leq t_i \leq x_{i+1}$ and $x_{i+1} - x_i \leq \frac{b-a}{n}$. The elements of s_n are called *sums of Riemann*. If the function f is indeed integrable in the sense of Riemann, also said *Riemann-integrable*, then the sequence $(s_n)_{n\in\mathbb{N}^*}$ converges to the integrable in the sense of Riemann of f over the interval $[a, b]$:

$$\int_a^b f(t)\,dt = \lim_{n\to\infty} s_n \tag{2.31}$$

Any Riemann-integrable function over an interval is also Lebesgue-integrable and the two definitions of integral return the same values on such functions. However, some functions are Lebesgue-integrable over an interval but non-integrable in the sense of Riemann. For example, the indicator function over the set of rational numbers is such a function and its Lebesgue-integral is null, because the measure of the set of rational numbers is null. We can exhibit two sequences of sums of Riemann over the interval $[0, 1]$ that do not converge to the same value:

$$\lim_{n\to\infty} \sum_{i=0}^{n-1}\left(\frac{i+1}{n} - \frac{i}{n}\right)\mathbb{1}_{\mathbb{Q}}\left(\frac{2i+1}{2n}\right) = 1 \tag{2.32}$$

and

$$\lim_{n\to\infty} \sum_{i=0}^{n-1}\left(\frac{i+1}{n} - \frac{i}{n}\right)\mathbb{1}_{\mathbb{Q}}\left(\frac{2i+\sqrt{2}}{2n}\right) = 0 \tag{2.33}$$

There even exist functions that are not integrable in the sense of Riemann, but that are integrable in the sense of Lebesgue and whose integral is positive, *e.g.* the indicator function of the Smith-Volterra-Cantor set [111, Section 4].

Some properties and notation

The integral in the sense of Lebesgue has the classical properties of integrals. In particular, it is linear and monotone. For all Lebesgue-integrable functions f and g over a set D and for all scalars $a \in \mathbb{R}$ and $b \in \mathbb{R}$, we have

$$\int_D (af + bg)\, d\mu = a \int_D f\, d\mu + b \int_D g\, d\mu \qquad \text{(linearity)}$$

And for all Lebesgue-integrable functions f and g over a set D, we have

$$\forall x \in D,\ f(x) \leq g(x) \implies \int_D f\, d\mu \leq \int_D g\, d\mu \qquad \text{(monotonicity)}$$

In the rest of the document, when there is no ambiguity, we use the notation of the Riemann-integrable for the Lebesgue's one over intervals: for any function f that is integrable over an interval $[a, b]$ in the sense of Lebesgue, we denoted its Lebesgue-integral

$$\int_a^b f(x)\, dx \tag{2.34}$$

where x can be any variable. In particular, we have

$$\int_a^b f(x)\, dx = \int_a^b f(y)\, dy = \int_{[a,b]} f\, d\mu \tag{2.35}$$

2.3.3 Discontinuous ordinary differential equation

As presented in [63, Chapter 1], an ordinary differential equation $\dot{x}(t) = f(t, x(t))$ is equivalent to the integral equation

$$x(t) = x(t_0) + \int_{t_0}^t f(s, x(s))\, ds \tag{2.36}$$

We say that a solution of the integral equation is a solution of the ordinary differential one. Using the integration in the sense of Lebesgue (*cf.* previous Subsection 2.3.2), we may use the results of the theory of differential equations in the sense of Carathéodory. This theory provides results on the existence and uniqueness of the solutions of such differential equations.

First, some conditions are identified to be sufficient in order to guarantee the existence of solutions of 2.36.

Definition 7 (Carathéodory conditions)
Let f be a function defined on a domain $D \subset \mathbb{R} \times \mathbb{R}^n$. The function f satisfies the *Carathéodory conditions* if

1. the function $f(t, x)$ is defined and continuous in x for almost all t;
2. the function $f(t, x)$ is measurable in t for all x;
3. $\|f(t, x)\| \leq m(t)$ and the function $m(t)$ is Lebesgue-integrable on every finite

interval.

Theorem 3 (Carathéodory existence theorem [63, Theorem 1])
For $(a, b) \in \mathbb{R}^2_{\geq 0}$, let be the integral equation

$$x(t) = x_0 + \int_{t_0}^{t} f(s, x(s))\, ds \tag{2.37}$$

with f a function that satisfies the Carathéodory conditions on a domain $\{(t, x) \mid t_0 \leq t \leq t_0 + a \wedge \|x - x_0\| \leq b\}$. Let d be a number such that

$$0 < d \leq a \qquad \text{and} \qquad \int_{t_0}^{t_0+d} m(s)\, ds \leq b \tag{2.38}$$

Then there exists a solution of the problem 2.37 on the closed interval $[t_0, t_0 + d]$.

Moreover, an extra condition of the right-hand sides of the ordinary differential equations, *i.e.* the functions f, is sufficient in order to guarantee the uniqueness of the solution for a given initial state.

Definition 8 (Carathéodory uniqueness conditions)
Let f be a function defined on a domain $D \subset \mathbb{R} \times \mathbb{R}^n$. The function f satisfies the *Carathéodory uniqueness conditions* if

1. the function f satisfies the *Carathéodory conditions* (*cf.* Definition 7)
2. there exists a Lebesgue-integrable function k such that for any points (t, x) and (t, y) in D,
$$\|f(t, x) - f(t, y)\|_\infty \leq k(t)\|x - y\|_\infty \tag{2.39}$$

Theorem 4 (Carathéodory uniqueness theorem)
Let f satisfying the Carathéodory uniqueness conditions of Definition 8, then there exists a unique solution to the problem 2.37 on the domain of definition.

In contrary to the case of ordinary differential equations with continuous right-hand side functions, the theorems in the case of non-continuous right-hand side functions do not provide any procedure to compute the solutions of the differential equations.

2.4 Reachability analysis

In Section 2.1, we defined models of hybrid systems. Those models define admissible trajectories, *i.e.* possible evolutions of the systems, from some constraints on the initial states, possible values of the parameters and possible inputs.

The study of such models allows to prove some properties. For example, given a model of a railway with some trains, a *safety property* could be: "the distance between

two trains on the same track is always bigger than 100 meters." This kind of property is often presented as "nothing bad will happen" or "the system is always safe".

Multiple methods exist to prove or to falsify such safety properties on hybrid models. First, we can try to find counter-examples, *i.e.* to exhibit an admissible trajectory of the model that violates the properties[123]. This could be efficient to identify a dangerous situation, but it is not able to prove the satisfaction of a property: the fact that we are not able to exhibit a counter-example is not a proof of the absence of a counter-example. So, we can try to prove the safety properties using deductive reasoning from a collection of axioms [129, 130]. The drawback of this method is the need of a proof for each property.

Finally, we can also try to compute the set of all states that the system can reach, *i.e.* the set of states reached by at least one admissible trajectory. These states are called *reachable states* and their union is called *reachable set*. However, computing such a reachable set is often undecidable [16]. We notice that if a set of states contains the reachable set and none of these states falsify the safety properties, then they are satisfied on the reachable set. So, we are interested in computing a set containing the reachable set and we call it an *over-approximation* of the reachable set. The drawback of this method is its incapability to falsify a safety property. If a state in the over-approximation falsifies a safety property, we cannot deduce that the safety property is falsified by the system: potentially, this state does not belong to the reachable set. In order to minimize the set of possibly reachable states, we want to compute over-approximations that are as small as possible. So, the methods to compute such over-approximations are designed in order to minimize it.

In what follows, we refer to such methods as *reachability analysis*.

We present in this section some representations of sets and some methods to compute over-approximations of the reachable sets of discrete, continuous or hybrid systems.

2.4.1 Representations of sets

Our goal is to construct an over-approximation of the reachable set. So, we need a representation of sets. Such a representation of sets should over-approximate the propagation of the uncertainties through all the operations. For example, for all binary operator $\oplus$ and two sets S_1 and S_2, we want a representation of the sets for which an operator $\tilde{\oplus}$ is defined such that:

$$\{x_1 \oplus x_2 \mid x_1 \in S_1 \wedge x_2 \in S_2\} \subset S_1 \tilde{\oplus} S_2 \tag{2.40}$$

In the rest of the document, when there are no ambiguities, we use the same notation for the operators on real numbers and the corresponding ones on the representations of sets.

Multiple representations of sets exist with different trade-offs between expressiveness, precision and computational efficiency [6, Table 1]. In particular, some representations are not closed under some operations and will necessarily introduce over-approximations. This consequence is called *wrapping effect* [118]. The simplest representation of sets is the *boxes* (multidimensional *intervals*) [118, 87], which is closed under intersection and

Minkowski sum, but not under union or linear mapping. It allows fast computation of over-approximations of expressions [13, 135, 119], subdivisions of spaces and fixed-point computations [44]. *Ellipsoids* [97, 98, 85, 20] are closed under linear mapping, which make them efficient for reachability analysis of linear dynamics, but they are not closed under union, Minkowski sum or intersection. *Zonotopes* [96, 69], which are bounded convex polyhedra with central symmetry (Minkowski sums of segments) and which can be encoded using a collection of vectors called *generators* evaluated via linear mapping with scalars in $[-1, 1]$, are closed under Minkowski sum and linear mapping, but not under union and intersection. They are efficient to compute and encode linear dependencies between variables both for hybrid systems [70, 14] and for static analysis [74]. They can be slightly adapted as *constrained zonotopes* to also be closed under intersection by introduction constraints on the scalars of the linear mapping applied to the generators [69, 137]. Alternatively, *zonotope bundles* were designed for similar purposes [11]. *Polytopes* (bounded convex polyhedra) [22, 38] encoded as the intersection of half-spaces ($\mathcal{H}$-polytope) or as the convex hull of its vertices ($\mathcal{V}$-polytope) are closed under the same operations, but the Minkowski sum is more expensive to compute while the intersection (*resp.* convex-hull) in the case of $\mathcal{H}$-polytopes (*resp.* $\mathcal{V}$-polytopes) are faster on large systems. All those representations of sets are convex and *support functions* generalize them [99, 5]. They encode an enclosure of a convex set based on the distance of the supporting hyperplanes in some directions. However, while they are closed under intersection, the exact value of the intersection of two support functions may not be computed and only an over-approximation of it is considered [66].

A representation of a set that is closed under nonlinear mapping has to be non-convex. Other representations have been proposed for the nonlinear case. *Taylor models* [34, 107] are defined as triples of a polynomial, its domain and an interval remainder. They are closed under polynomial mappings, but not under intersection and union. *Polynomial zonotopes* [4, 93] generalize zonotopes by introducing extra generators interpreted with a polynomial mapping instead of the linear one in the case of classical zonotopes. They are then closed under polynomial mappings and Minkowski sums but not under intersection and union. As constrained zonotopes, *constrained polynomial zonotopes* [92] add extra constraints on the scalars of the polynomial mapping of the generators of polynomial zonotopes and they are closed under all the present operations. *Generalized star sets* [58, 23, 143] have a similar definition as zonotopes but restricting the scalar of the linear mapping with a predicate instead of restricting them to the interval $[-1, 1]$. They generalize constrained zonotopes, but many operations does not have closed-form expressions due to the general definition of the predicates. Finally, *sublevel sets* [117, 89] encode sets of points whose images through a given function are non-positive. They are closed under all the presented operations, but they are difficult to encode due to their general structure.

In what follows, we focus on intervals for fast computation and Taylor models as over-approximations of sets of solutions of initial value problems.

2.4.1.1 Intervals and boxes

The simplest representation of sets is by intervals. Here, we present interval arithmetic introduced by R. E. Moore [118].

Definition 9
Let a and b be two real numbers such that $a \leq b$. An interval, denoted $[a, b]$, is a set of real numbers defined by

$$[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\} \tag{2.41}$$

An interval is a closed convex set of $\mathbb{R}$ and we can define operators on it to guarantee over-approximations of the results. The best over-approximation should be to take the minimum and maximum over each possible pair of elements:

$$I_1 \oplus I_2 = \left[\min_{x \in I_1,\ y \in I_2}(x \oplus y), \max_{x \in I_1,\ y \in I_2}(x \oplus y)\right] \tag{2.42}$$

which is the exact image of the Cartesian product $I_1 \times I_2$ through the operator $\oplus$. However, because we also want an efficient way to compute the results in order to perform automatic computations, each operator is usually defined using the bounds, based on the knowledge on their variations. For example, with $I_1 = [a, b]$ and $I_2 = [c, d]$, the classical arithmetic operators are defined by

$$[a, b] + [c, d] = [a + c, b + d] \tag{2.43}$$
$$[a, b] - [c, d] = [a - d, b - c] \tag{2.44}$$
$$[a, b] \times [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \tag{2.45}$$
$$\text{if } 0 \notin [c, d] \qquad [a, b]/[c, d] = [a, b] \times [1/d, 1/c] \tag{2.46}$$

For monotonic functions, such as the exponential, we can exploit the monotonicity to only compute the image of the boundaries of the interval. For example, if we consider an increasing function $f_\nearrow$ and a decreasing function $f_\searrow$, then we have for any interval $[a, b] \subset \mathbb{R}$,

$$f_\nearrow([a, b]) = [f_\nearrow(a), f_\nearrow(b)] \tag{2.47}$$
$$f_\searrow([a, b]) = [f_\searrow(b), f_\searrow(a)] \tag{2.48}$$

More details about interval arithmetic and specially about the handling of rounding are given in the book [118].

While multiplying an interval by a scalar whose absolute value is smaller than one reduces its width (*e.g.* with $\alpha \in [0, 1]$, $\alpha[a, b] = [\alpha a, \alpha b]$, whose initial width $b - a$ is reduced to $\alpha(b - a)$), the sum of two intervals results in an interval whose width is equal to the sum of the width of the operands (*e.g.* $[a, b] + [c, d] = [a + c, b + d]$ whose width is $(b + d) - (a + c) = (b - a) + (d - c)$). One of the main drawbacks of using intervals to compute over-approximations is the loss of dependencies between the variables. For example, the evaluation of the expression $x - x$ using interval arithmetic with $x \in [0, 1]$

results in the over-approximated interval $[-1, 1]$, which is twice larger than the original one, while the exact range is $\{0\}$.

Because we often manipulate vectors of some subsets of $\mathbb{R}^n$, we define a *box of* $\mathbb{R}^n$ as a Cartesian product of n intervals. When there is no ambiguity about the dimension, we simply use the term *box*. We say that a vector $x \in \mathbb{R}^n$ belongs to a box $[x] = \left[\underline{x_1}, \overline{x_1}\right] \times \cdots \times \left[\underline{x_n}, \overline{x_n}\right]$ if for all $i \in [\![1, n]\!]$, the component x_i of x belongs to the interval $\left[\underline{x_i}, \overline{x_i}\right]$.

2.4.1.2 Taylor models

Taylor models [34, 107, 106, 132, 47, 45] exploit the Taylor expansions of the results of initial value problems, *i.e.* polynomial approximations, and over-approximate the remainder terms (the errors of the approximation) using intervals. In a mathematical point of view, the Taylor models generalize the zonotopes, because they can be easily translated into Taylor models, but the associated functions differ preventing replacing zonotopes by Taylor models in the general case.

Definition 10 (Taylor model)
Let $D \in \mathbb{R}^n$ be a bounded domain. Let $k \in \mathbb{N}^*$ be a positive integer. Let $p : D \to \mathbb{R}$ be a polynomial of order at most k and let $I \subset \mathbb{R}$ be an interval.

Then the pair (p, I) is called a *Taylor model of order k on the domain D*, p is called its *polynomial part*, while I is called its *remainder*.

Remark 1 A multi-dimensional Taylor model can be defined as a vector of Taylor models. Let $n \in \mathbb{N}$. Consider a vector p of n polynomials on a common domain D of order at most k and a vector I of n interval. Then, for all $i \in [\![0, n-1]\!]$, (p_i, I_i) is a Taylor model of order k on the domain D. So, we say that (p, I) is a n-dimensional Taylor model of order k on the domain D.

Remark 2 If the domain (*resp.* the order) is already defined and if there is no ambiguity, the domain (*resp.* the order) may be omitted.

A Taylor model (p, I) on a domain D defines a set of values that are equal to an evaluation of the polynomial part plus a value in the interval remainder:

$$(p, I) \equiv \{p(x) + v \mid x \in D \wedge v \in I\} \tag{2.49}$$

but it can also be interpreted as a set-valued function over D that maps an element of the domain to the remainder interval translated by the evaluation of the polynomial part:

$$(p, I) \equiv x \mapsto \{p(x) + v \mid v \in I\} \tag{2.50}$$

Similarly to the case of intervals, all operators can be adapted for Taylor models in order to compute over-approximations of the results, as long as the Taylor models are defined on the same domain. For example, any linear application can be over-approximated computing the corresponding linear application on the polynomial parts

and on the remainders. Let (p_1, I_1) and (p_2, I_2) be two Taylor models of a given order k on a same given domain D, then any application $(x, y) \mapsto \alpha x + \beta y$ for $(\alpha, \beta) \in \mathbb{R}^2$ is over-approximated by the Taylor model (p, I) of order k defined on the domain D by

$$(p, I) = (\alpha p_1 + \beta p_2,\ \alpha I_1 + \beta I_2) \tag{2.51}$$

Notice that the over-approximation is only due to interval arithmetic for computing the remainder. It motivates to have the smallest as possible remainders.

The multiplication between (p_1, I_1) and (p_2, I_2) is based on the multiplication of polynomials, the multiplication of the remainders and an over-approximation of the interval range of the polynomials multiplied by the interval remainder of the other operand:

$$(p_1, I_1) \times (p_2, I_2) = (p_1 \times p_2 - p_e,\ I_1 \times I_2 + [p_1] \times I_2 + I_1 \times [p_2] + [p_e]) \tag{2.52}$$

with p_e the polynomial equal to the sum of all the monomials of $p_1 \times p_2$ whose degrees are bigger than k, and $[p_1]$, $[p_2]$ and $[p_e]$ interval over-approximations of the ranges of p_1, p_2 and p_e over the domain D.

Proof of the over-approximation Let $x \in D$, $v_1 \in I_1$ and $v_2 \in I_2$. We have

$$(p_1(x)+v_1)\times(p_2(x)+v_2) = p_1(x)\times p_2(x)+p_1(x)\times v_2+v_1\times p_2(x)+v_1\times v_2+p_e(x)-p_e(x)$$

By definition of p_e, $p_1 \times p_2 - p_e$ is a polynomial of degree at most k. Also by definition, $p_1(x)$ belongs to $[p_1]$, $p_2(x)$ belongs to $[p_2]$ and $p_e(x)$ belongs to $[p_e]$. So, because interval arithmetic over-approximates the results, we have

$$p_1(x) \times v_2 + v_1 \times p_2(x) + v_1 \times v_2 + p_e(x) \in [p_1] \times I_2 + I_1 \times [p_2] + I_1 \times I_2 + [p_e]$$

Because Taylor models are parameterized and dependent of a given domain D, given the maximal expected order k, we can define an over-approximation of the primitive operator with respect to a variable x_i whose range in D is $[\underline{x}_i, \overline{x}_i]$:

$$\int (p, I)\, dx_i = \left(\int_{\underline{x}_i}^{x_i} (p - p_e)\, dx_i,\ [0, (\overline{x}_i - \underline{x}_i)] \times ([p_e] + I) \right) \tag{2.53}$$

with p_e a polynomial equal to the sum of all the monomials of p whose degrees after primitivation would be bigger than k (*e.g.* if $p(x) = \sum_{i=0}^{k} \alpha_i x^i$, then $p_e(x) = \alpha_k x^k$), and $[p_e]$ an interval over-approximation of its range over D. It consists in computing a primitive of the polynomial part and scaling the remainder with respect to the width of the range of the primitive variable over the domain D. It is an over-approximation of the antiderivative of the functions that nullifies at the minimum value of x_i.

Proof of the over-approximation Consider a Taylor model (p, I) over a domain D and an integrable function f over the domain D that belongs to the Taylor model, *i.e.* there exists an integrable function $v : D \to I$ such that for all $x \in D$, $f(x) = p(x) + v(x)$.

Defining a polynomial p_e as above, the primitivation of f with respect to the

component x_i of a vector x becomes

$$\int_{\underline{x}_i}^{x_i} f(x)\,dx_i = \int_{\underline{x}_i}^{x_i} (p(x) - p_e(x))\,dx_i + \int_{\underline{x}_i}^{x_i} (p_e(x) + v(x))\,dx_i$$

We also notice that

$$\int_{\underline{x}_i}^{x_i} (p_e(x) + v(x))\,dx_i \in (x_i - \underline{x}_i) \times ([p_e] + I)$$

and the range of the expression $(x_i - \underline{x}_i)$ over the domain $[\underline{x}_i, \overline{x}_i]$ is equal to $[0, (\overline{x}_i - \underline{x}_i)]$, while the expression $([p_e] + I)$ is independent of x_i. So, for all $x_i \in [\underline{x}_i, \overline{x}_i]$, interval arithmetic guarantees

$$\int_{\underline{x}_i}^{x_i} (p_e(x) + v(x))\,dx_i \in [0, (\overline{x}_i - \underline{x}_i)] \times ([p_e] + I)$$

We can also define a Taylor model version of all smooth functions using Taylor expansions of their results, substituting the evaluation point by its corresponding Taylor model, truncating the polynomial up to a given order and over-approximating the error using the Lagrange remainder term [47, 45, 107, 8].

2.4.1.3 Bernstein decomposition

In order to apply compositions of operators on Taylor models, we have to compute over-approximations of the range of the polynomials over the domain. A simple evaluation using interval arithmetic may result in a huge over-approximation due to the loss of dependencies between the variables. However, we can use interval arithmetic over a subdivision of the domain to compute a tighter over-approximation [8, Section 2.3.2].

Another method to compute boundaries of a polynomial consists in computing its decomposition into Bernstein polynomials. Any interval that contains all the coefficients of such a decomposition is an over-approximation of the range of the polynomial [147, 68, 139]. For a polynomial p defined on a unit box domain $[0,1]^n$, with a collection of degrees with respect to each variable $d = (d_1, \ldots, d_n)$, called a *multi-index*, *i.e.* the degree of p with respect to a variable x_i is equal to d_i, for all multi-index $i = (i_1, \ldots, i_n)$ such that for all $k \in [\![1, n]\!]$, $0 \leq i_k \leq n$, the Bernstein coefficient b_i is defined as

$$b_i = \sum_{j_1=0}^{i_1} \cdots \sum_{j_n=0}^{i_n} \frac{\prod_{k=1}^{n} \binom{i_k}{j_k}}{\prod_{k=1}^{n} \binom{d_k}{j_k}} a_j \tag{2.54}$$

with the multi-indices $j = (j_1, \ldots, j_n)$ and for all integers α and β such that $\alpha \geq \beta \geq 0$, $\binom{\alpha}{\beta}$ is the binomial coefficient "α choose β" that can be defined as

$$\binom{\alpha}{\beta} = \frac{\alpha!}{\beta!(\alpha - \beta)!} \tag{2.55}$$

where for all natural number δ, $\delta!$ denotes the factorial of δ, *i.e.* $\delta! = \delta \times (\delta - 1) \times (\delta - 2) \times \cdots \times 2 \times 1$. With these notations, we have for all $x \in [0,1]^n$

$$\min_{0 \leq i \leq d} b_i \leq p(x) \leq \max_{0 \leq i \leq d} b_i \tag{2.56}$$

with $0 \leq i \leq d$ interpreted component wise, *i.e.* for all $k \in [\![1, n]\!]$, $0 \leq i_k \leq d_k$.

If the polynomial p is defined over a box $[\underline{x}, \overline{x}] = [\underline{x_1}, \overline{x_1}] \times \cdots \times [\underline{x_n}, \overline{x_n}]$ with a non-null volume, *i.e.* $x \in [\underline{x}, \overline{x}]$ is equivalent to for all $k \in [\![1, n]\!]$, $x_k \in [\underline{x_k}, \overline{x_k}]$ and $\underline{x_k} \neq \overline{x_k}$, then we can rescale the polynomial using an affine transformation

$$p_u(x) = p\left(\underline{x} + \frac{x}{\overline{x} - \underline{x}}\right) \tag{2.57}$$

where the operators are applied component wise. The polynomial p_u is then defined on the domain $[0,1]^n$ and it has the same range as p.

The number of coefficients of the Bernstein decomposition increases exponentially with respect to the number of variables. In this work, we implemented an exhaustive computation of them, but an algorithm exists that reduces the number of coefficients to compute in order to get the extremal ones [139].

2.4.2 Reachability analysis of initial value problems with uncertainties

We only consider dynamics defined using explicit ordinary differential equations of the first order (*cf.* Subsection 2.1.1 and Section 2.3). The problem we consider is an initial value problem with uncertainties:

$$\begin{cases} \forall_{ae.} t \in [t_0, t_1], & \dot{x}(t) = f(t, x(t), u(t)) \\ \forall t \in [t_0, t_1], & u(t) \in U \\ x(t_0) \in X_0 & \end{cases} \tag{2.58}$$

with U a subset of $\mathbb{R}^m$ and X_0 a subset of $\mathbb{R}^n$, where m is the number of input variables and n is the number of state variables.

We want to compute a set-valued function R from $[t_0, t_1]$ to the set of subsets of $\mathbb{R}^n$ such that for all solutions x of the problem 2.58, *i.e.* there exists a function u such that for almost all $t \in [t_0, t_1]$, $\dot{x}(t) = f(t, x(t), u(t))$ and $x(t_0)$ belongs to X_0, and for all time $t \in [t_0, t_1]$, we have $x(t) \in R(t)$. This set-valued function defines an over-approximation of the reachable set of the initial value problem with respect to the time.

Such a dynamics defines two kind of uncertainties: 1) an uncertain initial state $x(t_0)$ (constant uncertainties) and 2) an uncertain input signal u of which only the range U is known (time-varying uncertainties). As presented in Section 2.3, we have to make hypotheses about the right-hand side of the differential equation in order to guarantee the existence and the uniqueness of the solutions. In the following, we assume at least that the function f is continuous. Moreover, we always assume that for each $x(t_0)$ and u, the problem has a unique solution, *e.g.* with the hypotheses of the Theorem 2 or the one of the Theorem 4.

Multiple methods exist in the case of a linear dynamics, *i.e.* $f(t, x(t), u(t)) = A(t)x(t) + B(t)u(t)$: [97] using ellipsoid, [100] using support functions, [12] and [65] over-approximating Riemann sums of the state transition matrix. We focus on non-linear ones in the rest of this document.

A first method to handle such non-linear dynamics is to approximate the set of solutions and to enlarge it to over-approximate the error with respect to the exact set. The approximation can be computed using Runge-Kutta methods and automatic differentiation for the evaluation of the over-approximation of the error [135, 119] or using Taylor expansion and over-approximating the error via automatic differentiation or contraction of a set [34, 107, 132, 47, 45, 79]. We present the later method in Subsection 2.4.2.2. Instead of explicitly define an approximation, simulations of particular solutions can also be computed with an over-approximation of the error via expansion functions [57]. Both approaches can be mixed studying first a problem in which only a restricted class of uncertain inputs are considered and over-approximating the error made considering such a restriction: [90] only considers constant inputs before computing the error, while [72] proposes different restriction inputs such as affine or sinusoidal. This last method is implemented in the tool ARIADNE[24, 33].

Other methods consist in abstracting the dynamics by a simpler one. It consists in over-approximating the differential equation by a linear [14] or polynomial [4] differential equation with additive time-varying uncertainties. This methods are implemented in the tool CORA[9]. The abstractions may also be made replacing the differential equation by a hybrid automaton with simple continuous dynamics, which is called *hybridization* [18, 91]. Instead of defining simpler dynamics whose reachable sets contain the original ones, the dynamics may be abstracted using deterministic dynamics whose reachable sets are bounds of the original one [133]. Methods such as "Face-lifting approaches" [54, 19] study the dynamics locally: it consists in expanding a polyhedron of reachable states based on the maximal distance reached in the normal of each face for a given duration.

We present here two methods using the Picard operator (*cf.* Definition 6), that do not require the differentiability of the function f, but that require the continuity of the right-hand side with respect to the time, *i.e.* with continuous functions u, and (locally) Lipschitz with respect to the state x. This methods are implemented in the tool FLOW*[48, 46]. We use these methods as bases of our method with Lebesgue-integrable uncertainties (*cf.* Chapter 3) and in our prototypes (*cf.* chapters 4 and 5).

2.4.2.1 Rough over-approximation of continuous dynamics with time-varying uncertainties

Theorem 5 (Schauder fixed-point theorem [126, Theorem 1.2])
Let K be a convex subset of a normed vector space E. Each compact mapping $T : K \to K$, *i.e.* such that there exists a compact $C \subset K$ such that $T(K) \subset C$, has a fixed-point.

The first method to compute an over-approximation of the reachable set of an autonomous continuous dynamics exploits the Schauder fixed-point theorem (*cf.* Theorem 5) and interval arithmetic [122, 135]. It provides a rough over-approximation independent on time but that can be computed quickly. Given a differential equation $\dot{x} = f(x)$ and a set of possible initial states X_0, this method requires a box over-approximation $[x_0]$ of the set of possible initial states X_0, *i.e.* $X_0 \subset [x_0]$ and $[x_0]$ is defined as a Cartesian product of intervals, and an interval version of the function f, *i.e.* a function $[f]$ that takes boxes $[x]$ and that returns a box containing the images of the boxes $[x]$ through f, *i.e.* for all $x \in [x]$, $f(x) \in [f]([x])$. Such a function $[f]$ can easily be computed by replacing all the operators in the definition of f by their corresponding operators of interval arithmetic. The interval version $[\mathbb{P}]$ of the Picard operator $\mathbb{P}$ over the time interval $[t_0, t_1]$ can be defined as

$$[\mathbb{P}]([x]) = [x_0] + [0, t_1 - t_0] \times [f]([x]) \tag{2.59}$$

If a box $[x]$ that contains $[x_0]$ is contracted by $[\mathbb{P}]$, *i.e.* $[\mathbb{P}]([x]) \subset [x]$, then $[x]$ is an over-approximation of the reachable set over the time interval $[t_0, t_1]$ by the Schauder fixed-point theorem, *i.e.* for all solution x and time $t \in [t_0, t_1]$, $x(t) \in [x]$.

The algorithm of this first method is as follow:

1. Set $[x]$ to $[x_0]$
2. While $[\mathbb{P}]([x])$ is not included in $[x]$, enlarge $[x]$
3. While $[\mathbb{P}]([x])$ is strictly included in $[x]$, set $[x]$ to $[\mathbb{P}]([x])$
4. Return $[x]$

This algorithm requires a method to enlarge a box $[x]$. It can be done using two positive thresholds ε_a (for *absolute*) and ε_r (for *relative*): for each component $[\underline{x_i}, \overline{x_i}]$ of $[x]$, if $w_i = \overline{x_i} - \underline{x_i}$ (*i.e.* w_i is the width of $[\underline{x_i}, \overline{x_i}]$) is strictly smaller than ε_a, then we add to it the interval $[-\varepsilon_a, \varepsilon_a]$, otherwise, we add to it the interval $[-\varepsilon_r w_i, \varepsilon_r w_i]$. In order to speedup the termination of the algorithm, the condition of strict inclusion at the step 3 can be replaced by a condition of sufficient contraction, *e.g.* based on the width of the boxes (*cf.* Subsection 4.1.4).

This first method is often used to compute an *a priori* over-approximation that can be used to compute a tighter one. In particular, it can be used to compute an over-approximation of the Lagrange remainder of a Taylor expansion of the solutions [135].

Moreover, such a method is still valid in the case of continuous inputs u. In that case, the differential equation $\dot{x}(t) = f\big(t, x(t), u(t)\big)$ is replaced by the differential inclusion $\dot{x}(t) \in f\big(t, x(t), [U]\big)$, where $[U]$ is a box over-approximation of U. The only difference is the Picard operator that becomes

$$[\mathbb{P}]\big([x]\big) = [x_0] + [0, t_1 - t_0] \times [f]\big([t_0, t_1], [x], [U]\big) \tag{2.60}$$

Because interval arithmetic corresponds to Taylor model arithmetic with null polynomials, *i.e.* Taylor models of null order, a proof of the correctness of this method with

continuous inputs u in given in [45, Theorem 3.5.1], using Schauder fixed-point theorem (*cf.* Theorem 5).

We prove in Section 3.2 that it is still correct in the case of Lebesgue-integrable inputs u.

2.4.2.2 Tighter over-approximation using Taylor models

Now, we present a second method to compute an over-approximation of the reachable set with respect to time in the case of continuous function f (and locally Lipschitz in order to guarantee the existence and uniqueness of the solutions, *cf.* Theorem 2). This method [34] exploits the operations defined on Taylor models. However, it is only proven valid in the case of continuous time-varying uncertainties u. This method is (one of the methods) implemented in the tool Flow* [47, 45]. Alternatively, Ariadne [50] uses Lie derivatives to compute the different coefficients of the Taylor models and the remainder (which is also implemented in Flow* [45]). On the contrary, other tools such as VNODE-LP [121] rely on Taylor expansion using automatic differentiation (*e.g.* using libraries such as FADBAD++[11]) to evaluate each monomials, which in only accurate for small initial uncertainties (*cf.* [121, Section 1.4]). Similarly to VNODE-LP, MAGIC-CPS[12] [105, Section 4, Algorithm 2] uses automatic differentiation, but it exploit QR-factorization and mean-value of f [122] in order to reduce the size of the evaluations.

This method iteratively computes a polynomial approximation (a truncated Taylor expansion in the case of differentiable dynamics) of the solutions using a recursive sequence of polynomials up to a desired order. In the case of a differentiable function f, over-approximations of the coefficients of the Taylor expansions can be directly computed using automatic differentiation [120, 45, 76] and evaluated using the arithmetic associated to a representation of sets. Moreover, in that case, the remainder can be compute by evaluating its Lagrange form [76] on a rough over-approximation of the solution (*e.g.* computed as in Subsection 2.4.2.1).

First, set X_0 has to be over-approximated using Taylor models. In practice, X_0 is already defined as a (multi-dimensional) Taylor model or it is defined as a box. In this last case, we can define a multi-dimensional Taylor model using a variable for each dimension and the associated domain as the corresponding component of the box X_0. For example, if X_0 is a box $[\underline{x_1}, \overline{x_1}] \times \cdots \times [\underline{x_n}, \overline{x_n}]$, then we define a multi-dimensional Taylor model (p_0, I_0) with for each index $i \in [\![1, n]\!]$, $p_{0,i}(x) = x_i$, $I_{0,i} = [0, 0] = \{0\}$ and the domain D_0 equal to the box X_0.

Second, we compute a polynomial approximation of the solutions using the Picard operator adapted to Taylor models. The dimension of the domain is increased by one to handle a time variable t whose values are in $[t_0, t_1]$. The new domain is $D = [t_0, t_1] \times D_0$

[11] `http://www.fadbad.com/`

[12] `https://agora.bourges.univ-orleans.fr/ramdani/attachments/article/93/MAGIC-CPS-Install-EN.pdf`

(we arbitrarily set variable t as the variable of smallest index). Then, the Picard operator becomes

$$\mathbb{P}((p_n, I_n)) = (p_0, I_0) + \int f_{TM}\Big((t, [0,0]), (p_n, I_n), (0, [U])\Big)\, dt \tag{2.61}$$

with $(t, [0,0])$ a Taylor model whose polynomial part is equal to t and the remainder is null, $(0, [U])$ a Taylor model whose polynomial part is null and the remainder is a box over-approximating the range of the time-varying uncertainties U, and f_{TM} a function on Taylor models obtained replacing all the operators of the definition of f by their corresponding Taylor model version. The Picard operator is iterated on (p_n, I_n) (initially set to (p_0, I_0)) until the polynomial part p_n reaches a fixed-point, *i.e.* $p_{n+1} = p_n$.

Third, the remainder I_n of the Taylor model (p_n, I_n) is enlarged until it is contracted by the Picard operator, *i.e.* $I_{n+1} \subset I_n$. This enlargement can be performed as in the previous algorithm using interval arithmetic.

Finally, as in the previous algorithm, we iterate the Picard operator until the remainder reaches a fixed-point or until the relative contraction of the remainder becomes smaller than a given threshold (*cf.* Subsection 4.1.4).

We can summarize the algorithm as follow

1. Initialize a Taylor model (p, I) as over-approximation of X_0
2. While $p' \neq p$ with $(p', I') = \mathbb{P}((p, I))$, set (p, I) to $\mathbb{P}((p, I))$
3. While I' is not included in I with $(p', I') = \mathbb{P}((p, I))$, enlarge I
4. While I' is strictly included in I with $(p', I') = \mathbb{P}((p, I))$, set (p, I) to $\mathbb{P}((p, I))$
5. Return (p, I)

A proof of the autonomous case, *i.e.* with constant inputs u, is provided in [34], which uses the Arzelà-Ascoli theorem (*cf.* Theorem 6) in addition to the Schauder fixed-point theorem (*cf.* Theorem 5). One in the case of continuous time-varying uncertainties u is provided in [45, Theorem 3.5.1]. In the Section 3.2, we provide a proof similar to the one in [34] but with Lebesgue-integrable uncertainties.

Theorem 6 (Arzelà-Ascoli theorem)
Let $(f_n)_{n\in\mathbb{N}}$ be a sequence of continuous functions from a closed interval $[a, b]$ to $\mathbb{R}$. If the sequence is

1. *uniformly bounded*, *i.e.* there exists a real number M such that

$$\forall (x, n) \in [a, b] \times \mathbb{N},\ |f_n(x)| \leq M \tag{2.62}$$

and

2. *uniformly equicontinuous*, *i.e.* for all $\varepsilon > 0$, there exists $\delta > 0$ such that

$$\forall (x, y, n) \in [a, b] \times [a, b] \times \mathbb{N},\ |x - y| \leq \delta \implies |f_n(x) - f_n(y)| \leq \varepsilon \tag{2.63}$$

then there exists a subsequence $(f_{n_k})_{k\in\mathbb{N}}$ that converges uniformly.

2.4.3 Abstract interpretation in static program analysis

Static analysers, such as ASTRÉE [53] or FLUCTUAT [56], often exploit abstract interpretation to compute over-approximations of sets of reachable states of programs [51, 52]. It consists in replacing the (concrete) semantics of the programs by a sound approximation of it. In particular, to compute an over-approximation of the set of reachable states, the semantics of the operators on floating-point numbers can be replaced by a semantics on a representation of sets that guarantees the over-approximation of the results [77].

A classic constructive semantics of expressions defines a function that maps a context and an expression to a value. It is illustrated by Example 13. A sound abstract semantics with respect to the over-approximation can often be deduced from the arithmetic associated to the representation of sets used to define the abstract domain. Those possible representations of the abstract domain are similar to the ones presented in Subsection 2.4.1: boxes, zones [112, 113], octagons [113], template polyhedra [136], *etc.*

Example 13 (Constructive semantics of arithmetic operators)
Given a set X of variables, a set of possible values V and a context ρ that maps variables in X to values in V, we define a (concrete) semantics $[\![e]\!]^c_\rho$ of an expression e as a value in V:

$$[\![x]\!]^c_\rho = \rho(x) \tag{2.64}$$

$$[\![x \otimes y]\!]^c_\rho = \rho(x) \otimes \rho(y) \tag{2.65}$$

The semantics of a call to a variable is simply the value mapped to the variable in the context (*cf.* 2.64). The semantics of an expression with a binary operator $\otimes$ (*e.g.* $\otimes$ can be $+$, $\times$, $-$, *etc.*) is the application of the operator to the values mapped to the operands in the given context (*cf.* 2.65).

A possible abstract semantics using interval arithmetic consists in replacing the set of values V by a set of intervals and the binary operators by their corresponding ones in interval arithmetic.

A classic constructive semantics of instructions defines a function that maps a context and an instruction to a context. It is illustrated by Example 14. Abstract semantics sound with respect to the over-approximation have to manipulate abstract contexts, which often requires to compute unions and intersections of sets. For example, FLUCTUAT uses zonotopes to represent such contexts with an abstract semantics that keeps linear correlations between values [74].

Example 14 (Constructive concrete semantics of instructions and branches)
Given a set X of variables, a set of possible values V and a context ρ that maps variables in X to values in V, we define a (concrete) semantics $[\![i;]\!]^c_\rho$ of an instruction

i; as a context:

$$[\![x = e;]\!]^c_\rho = \{x' \mapsto \rho(x') \mid x' \in X \setminus \{x\}\} \cup \{x \mapsto [\![e]\!]^c_\rho\} \tag{2.66}$$

$$[\![i1; i2;]\!]^c_\rho = [\![i2;]\!]^c_{\rho'} \quad \text{with} \quad \rho' = [\![i1;]\!]^c_\rho \tag{2.67}$$

$$[\![\textbf{if } e \textbf{ then } i1; \textbf{ else } i2;]\!]^c_\rho = \begin{cases} [\![i1;]\!]^c_\rho & \text{if } [\![e]\!]^c_\rho = \textbf{true} \\ [\![i2;]\!]^c_\rho & \text{otherwise} \end{cases} \tag{2.68}$$

The affectation consists in mapping the variable to the value returned by the semantics of the expression while keeping all other mapping of the context (*cf.* 2.66). The semantics of a sequence of instructions is the chaining of their semantics (*cf.* 2.67). The semantics of a branch instruction is equal to the one of an instruction or another depending on the value of the semantics of an expression (*cf.* 2.68).

An abstract semantics that guarantees the over-approximations of the values has to possibly compute both branches of the branch instructions [52, Section 3.1]. Given an abstract context $\tilde{\rho}$ (*e.g.* mapping to interval values),

$$[\![\textbf{if } e \textbf{ then } i1; \textbf{ else } i2;]\!]^a_{\tilde{\rho}} = [\![i1;]\!]^a_{\tilde{\rho}_{e=\textbf{true}}} \cup [\![i2;]\!]^a_{\tilde{\rho}_{e=\textbf{false}}} \tag{2.69}$$

where $\tilde{\rho}_{e=\textbf{true}}$ (*resp.* $\tilde{\rho}_{e=\textbf{false}}$) is an abstract context whose concretization contains all concrete context of the concretization of $\tilde{\rho}$ in which the semantics of e is **true** (*resp.* **false**). In particular, it requires to compute the union of abstract context, *i.e.* a new abstract context whose concretization is a superset of the union of the concretization of the abstract context defined by the semantics of each branch. More details are provided in [37, Section 5.4].

In the case of hybrid systems, we want to exploit both concrete and discrete methods to compute over-approximations of sets of reachable states.

2.4.4 Reachability analysis of hybrid dynamics

In order to compute an over-approximation of the reachable set of a hybrid automaton, we have to compute an over-approximation of the continuous dynamics in each mode, but also an over-approximation of the set of states that activate some guards and an over-approximation of the images through the jumps. We focus on reachability analysis with bounded time horizon, *i.e.* on a given time interval $[0, T]$. We also need a set of possible inputs denoted U that are defined on the time interval $[0, T]$, *i.e.* a set of functions $u \in U$ whose domain is $[0, T]$.

In the previous subsection, we briefly presented some methods to compute an over-approximation of the reachable set with respect to the time of continuous dynamics [99, 45, 6]. However, those methods do not handle the invariants: it is possible that some of the computed states are not reachable due to the invariant conditions. To compute such a restriction, we have to compute an over-approximation of the intersection between the over-approximations computed following the continuous dynamics and the set of states that satisfy the invariant conditions.

2.4.4.1 Base abstract algorithm for reachability analysis of hybrid automata

Consider a hybrid automaton H as defined in the Definition 1, a time interval $[0, T]$ and a set U of possible input functions. We call *timed state* a pair (t, x) of a time t and a continuous state x of the hybrid automaton. We assume that we have a function $\text{Post}_{\text{cont}}$ that given a mode $v \in V$ and a set Y_0 of timed states, returns an over-approximation of the set of timed states that are reachable from a timed state in Y_0 following the flow condition $\mathit{flow}(v)$ with some (Lebesgue-integrable) input u of U. So, $\text{Post}_{\text{cont}}(v, Y_0)$ is an over-approximation of the set of timed states that are reachable from Y_0 in the current mode, *i.e.* without taking any transitions, with times in $[0, T]$. Such an over-approximation can be computed using the methods presented in the previous subsection.

Based on the function $\text{Post}_{\text{cont}}$, we define a function Post_{loc} that takes the same arguments as $\text{Post}_{\text{cont}}$ but that filters the set of timed states returned by $\text{Post}_{\text{cont}}$ to only keep the ones that are reachable without violating the invariant conditions. Because we want over-approximations, it can be computed as an over-approximation of the intersection of the set computed by $\text{Post}_{\text{cont}}$ and the set of timed states that satisfy the invariant conditions in the given mode. It can also be computed using contraction propagation [44].

In order to compute an over-approximation of the discrete behaviors, we assume that we have a function $\text{Post}_{\text{disc}}$ that given a edge $e \in E$, from mode v_1 to mode v_2, and a set Y_1 of timed states, returns an over-approximation of the set Y_2 of timed states that satisfy the invariant condition in the mode v_2 and that belong to the image of the jump condition $\mathit{jump}(e)$ associated to the transition e over the set of timed states in Y_1 that may satisfy the guard condition $\mathit{guard}(e)$ with some input $u \in U$. This function can be computed in two steps: 1) identify the subset $\tilde{Y}$ of Y_1 that may satisfy the guard condition $\mathit{guard}(e)$; 2) compute an over-approximation of all the possible images of $\tilde{Y}$ through the jump condition $\mathit{jump}(e)$ and 3) filter the set of computed timed states to only keep those that satisfy the invariant condition $\mathit{inv}(v_2)$. Such a function can be computed using an over-approximation of the intersection as in the case of the function Post_{loc} (over-approximation of $\mathit{guard}(e)$ and $\mathit{inv}(v_2)$) and using the arithmetic associated to the representation of sets (over-approximation of $\mathit{jump}(e)$).

A classic procedure [99, 6] to compute an over-approximation of the reachable set of an hybrid automaton uses a *worklist* (or *waiting list*) W whose elements are pairs (v, Y) of a mode and a set of timed states. It constructs a set R of pairs (v, y) of a mode and a timed state that is an over-approximation of the reachable set of the hybrid automaton. We define a function $\text{Visited}((v, Y), R)$ that returns true if and only if all the pairs (v, y) with $y \in Y$ belong to R. We also define a function $\text{InitState}(v)$ that returns a set of timed states that belong to the initial set *Init* in the mode v.

1. Set W to $\{(v, \text{InitState}(v)) \mid v \in V\}$ and R to an empty set
2. Pop a pair (v, Y) from W
3. Compute $\tilde{Y} = \text{Post}_{\text{loc}}(v, Y)$ and append $(v, \tilde{Y})$ to R
4. For each mode $v' \in V$ and for each edge $e_{v,v'} \in E$ from v to v':

(a) Compute $Y' = \text{Post}_{\text{disc}}(e_{v,v'}, \tilde{Y})$

(b) If Y' is not empty and Visited$((v', Y'), R)$ is false, then append (v', Y') to W

5. If W is empty, then return R. Otherwise, go to step 2

2.4.4.2 Multiple possible implementations

Different possible implementations of the operator $\text{Post}_{\text{cont}}$ were presented in Subsection 2.4.2. To obtain the operator Post_{loc}, tools have to detect the violation of the invariant conditions by the computed over-approximation of the set of continuous solutions. This problem is similar to the detection of the activation of guards that are part of the operator $\text{Post}_{\text{disc}}$.

If the predicates are defined as regions (*e.g.* $guard(e)(t, x, u) = (x \in G_e)$ with $G_e \subset \mathbb{R}^n$), the violation and the satisfaction of predicates only depends on the computation of an intersection between the result of $\text{Post}_{\text{cont}}$ and the regions associated to the predicates. This is the method with polyhedra intersections used in SpaceEx [64] via support functions [66] and CORA [2] via zonotopes bundles [11].

Alternatively, predicates can be encoded as constraints on the results of real multivariate functions (*e.g.* $guard(e)(t, x, u) = (g_e(t, x, u) \geq 0)$). In this case, the real multivariate functions are evaluated using the arithmetic associated to the chosen representation of sets. Notice that once the functions are evaluated, it amounts to the computation of intersections as in the previous case (*e.g.* $\{y \mid y = g_e(t, x, u)\} \cap \{y \mid y \geq 0\}$). It is for example implemented in Flow* [47] using Taylor models for the evaluation of functions and zonotopes to compute over-approximations of intersections. With Taylor models, it also defines *domain contraction* in order to obtain tight subsets of the Taylor models that activate the guards, using interval constraint propagation [27] or branch-and-prune algorithms [134]. Computing a tight over-approximation of the set of states that validate a given predicate can also be computed using *contractors* [44, 105] that exploit different evaluations to contract the initial set until it reaches a fixed-point. In Ariadne, such a computation is implemented using the interval Newton operator [50].

Finally, in the case of the $\text{Post}_{\text{disc}}$ operator, once such an (over-approximated) intersection is computed, its images through the corresponding jump can be computed using the arithmetic of the chosen representation of sets.

2.4.4.3 Reachability analysis of Simulink-like models

As mentioned in Section 2.2, models written in Simulink or Zélus have slightly different structures than hybrid automata and transitions are followed as soon as they are activated. To exploit classical methods on such models, they can first be translated into hybrid automata using the tools HyLink [109] or GreAT [1], but also into networks of hybrid automata using SL2SX [115, 114, 91] before computing reachability analyses.

In the case of Zélus, a compiler [43] has been developed to generate code in C++11 in order to use the DynIbex[13] to compute over-approximations of the sets of reachable

[13] `https://perso.ensta-paris.fr/~chapoutot/dynibex/`

states of continuous systems. Recently, this method has been improved to handle hybrid automata written in Zélus [128]. It only handles a subset of the Zélus language. In particular, it does not handle nested (or hierarchical) automata or resets on transitions.

2.4.4.4 Handling of Zeno behaviors

Zeno behaviors may result in endless computations of the reachable set. To avoid such behavior, some methods require that the considered hybrid automata do not allow Zeno behaviors [18, Assumption 1]. Some tools such as Flow* [48] require users to define a maximum number of transition that a trajectory can take during the computation, which excludes any Zeno behavior. Ariadne also allows to define such a maximum bound on the number of transitions[14], but it also allows to define instead a domain in which to compute the reachability analysis[14]: it allows to perform the reachability analysis on a finite discretization of the domain [33].

A method [95, 94] exists in order to handle Zeno behaviors and even computing sound over-approximations with respect to the interpretation in the sense of Filippov (*cf.* Subsection 2.1.3) using interval methods, but it terminates only when the trajectories at Zeno points converge toward finite stable orbits [94], which seems not compatible with time-varying guards.

[14] `https://www.ariadne-cps.org/tutorial/`

Chapter 3

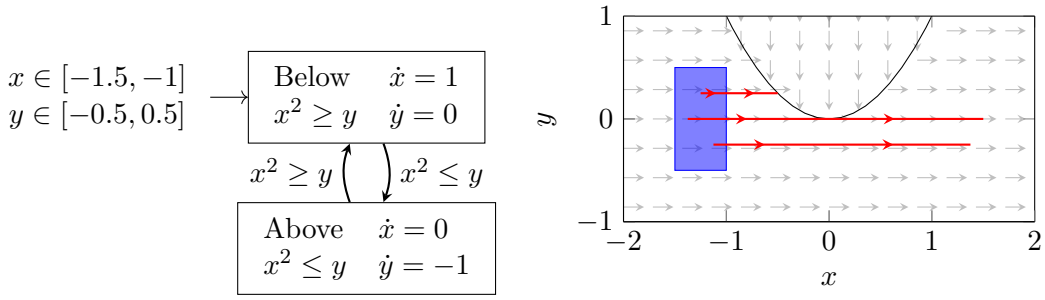
Reachability analysis with bounded time-varying uncertainties

We start this chapter by proving that a set-valued function that is contracted by a set-valued version of the Picard operator is an over-approximation of the set of solutions of a continuous dynamics in presence of Lebesgue-integrable uncertainties. This result demonstrates that some methods designed for continuous time-varying uncertainties are indeed correct also for Lebesgue-integrable ones. However, these existing approaches are generally imprecise in that case. Then, we present a method to compute an over-approximation of the reachable set of systems described as a generalization of input-affine systems with Lebesgue-integrable bounded uncertain inputs. We call such systems *separable systems with respect to time-varying uncertainties* [35]. Our method is based on a decomposition as a difference of positive functions of the right-hand side, reducing the stystem with measurable uncertainties to a system with constant uncertainties for which efficient techniques already exist. We then discuss an interpretation of the optimality of such decompositions. Finally, we apply this method to switched systems allowing Zeno behaviors by handling transitions with differential inclusions in the sense of Filippov.

3.1 Motivations

In order to obtain tight over-approximations, reachability tools make assumptions about the right-hand side of the ordinary differential equations. The stronger hypothesis is the differentiability of the right-hand side [135]. A more common one is the Lipschitz continuity of the right-hand side with respect to the state and continuity with respect to the time [14, 4], which is the case for most of autonomous systems, *i.e.* of the form $\dot{x}(t) = f(t, x(t), p)$. However, this is often a non-admissible hypothesis when it comes to handling noise. A more acceptable one is to assume that the right-hand side is piecewise continuous with respect to the state variables and time, or even Riemann-integrable [4].

But some input signals may also just be Lebesgue-integrable, *e.g.* in the super-twisting observers [55].

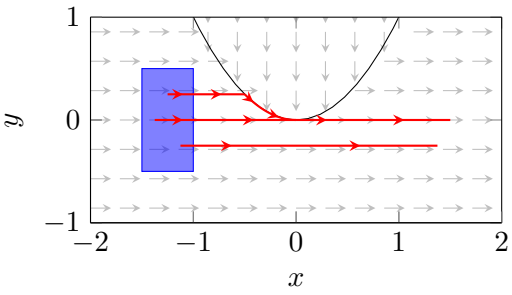


(a) Hybrid automaton

(b) Behavior with the classic interpretation (initial states in blue, admissible trajectories in red)

Figure 3.1 – Hybrid automaton with Zeno behaviors

$$\begin{cases} \forall_{a.e.} t \geq t_0,\ \dot{x}(t) = \alpha(t) \\ \forall_{a.e.} t \geq t_0,\ \dot{y}(t) = \alpha(t) - 1 \\ x(t_0) \in [-1.5, -1] \\ y(t_0) \in [-0.5, 0.5] \\ \forall t \geq t_0,\ x(t)^2 > y(t) \implies \alpha(t) = 1 \\ \forall t \geq t_0,\ x(t)^2 < y(t) \implies \alpha(t) = 0 \\ \forall t \geq t_0,\ x(t)^2 = y(t) \implies \alpha(t) \in [0, 1] \end{cases}$$



(a) Switched system in the sense of Filippov

(b) Behavior in the sense of Filippov (initial states in blue, admissible trajectories in red)

Figure 3.2 – Switched system in the sense of Filippov corresponding to the system presented in Figure 3.1

Now, consider the hybrid automaton represented in Figure 3.1. It is a typical example of a switched system with some Zeno behaviors (*cf.* Subsection 2.1.2): as soon as the system reaches a state such that $x^2 = y$, it cannot evolve anymore using the classic definition, because its trajectory should have either $\dot{x}(t) = 1$ or $\dot{y}(t) = -1$ and that should directly force the system to be in the opposite mode. We presented in the Subsection 2.1.2 a method proposed by A. F. Filippov [63] to allow admissible trajectories defined beyond any Zeno events using contraints on the derivatives as differential inclusions: from a Zeno event, the trajectories stay on the switching surface until some dynamics allow the trajectories to move away. On the example represented in Figure 3.1, if a trajectory starts by a state with a positive value of the y variable, then the trajectory eventually reaches

the surface defined by $x^2 = y$ and it stays on it until it reaches the state $x = 0 \wedge y = 0$. From that point, the trajectory can stay in the mode Below forever. This is represented in Figure 3.2 with an extra measurable input variable α, *i.e.* a time-varying uncertainty, whose range is constrained by the current state.

Due to the simplicity of this example, one may also compute a function $g(x, y) = x^2 - y$, whose sign only changes on the switching surface $x^2 = y$, and compute the explicit sliding mode using the Lie derivatives of g along the different dynamics. However, automatically detecting and computing such a sliding mode is not trivial and it typically needs a method of automatic differentiation. Moreover, in the general case, such a function g may not be differentiable and the Lie derivatives may not be computed.

To compute an over-approximation of the reachable set of such a system, we have to handle continuous dynamics with measurable time-varying uncertainty. As far as we can tell, it is not known whether considering bounded time-varying uncertainties as Riemann-integrable or Lebesgue-integrable yields the same reachable set. So, we have to consider only methods that are able to handle Lebesgue-integrable uncertainties.

The state-of-the-art tools that handle such nonlinear dynamical systems with uncertainties start by computing an over-approximation of the reachable set of the system considering a subclass of uncertainties (constant or continuous depending on some parameters) and add to it a set to over-approximate the error with respect to the whole class of possible measurable bounded uncertainties [90, 72]. For example, given a dynamical system $\dot{x}(t) = f\big(x(t), u(t)\big)$ where f is a continuous function and u is a measurable function with values in a bounded convex set U, they start by computing an over-approximation R_p of the reachable set of a parameterized dynamical system $\dot{x}_p(t) = f\big(x_p(t), y_p(t)\big)$ where y_p is an explicit continuous function depending on an uncertain parameter p, *i.e.* $\dot{x}_p(t) = f_p(t, x_p(t))$ with f_p a continuous function parameterized by p. Then, an upper bound M is computed such that $\min_p \|x - x_p\| \leq M$ for all solutions x of $\dot{x}(t) = f\big(x(t), u(t)\big)$. This upper bound M is computed based on a bound of the derivative of f with respect to x [90, 72] or a bound of the range of f [72]. Finally, the over-approximation of the reachable set of $\dot{x}(t) = f\big(x(t), u(t)\big)$ is defined as the Minkowsky sum of R_p and the ball of radius M centered on the origin.

Our method differs from the state-of-the-art by ensuring that the reachable set of the auxiliary continuous system is already an over-approximation of the reachable set of the original system interpreted in the sense of Filippov. So, we do not have to add an error *a posteriori*. However, it requires an *a priori* over-approximation of the reachable set, which we call *rough over-approximation*, in order to compute a valid auxiliary continuous system. So, we start to prove the correctness of methods based on the contraction of sets.

3.2 Criterion of over-approximation of non-linear dynamics with Lebesgue-integrable uncertainties

A first way to compute an over-approximation of non-linear dynamics with uncertain initial states and Lebesgue-integrable time-varying uncertainties consists in remplacing

the time-varying uncertainties by their range in the differential equation. It results in a differential inclusion. A set-valued function that is stable through the associated Picard operator is an over-approximation of the reachable set in the case of continuous time-varying uncertainties [45, 49].

Our proof of the correctness of this method in the case of Lebesgue-integrable uncertainties is inspired by the one by Martin Berz and Kyoko Makino [34], which uses Taylor models. It consists in constructing a parameterized compact and convex subset of the normed space of continuous functions with the supremum norm being stable through a Picard operator. Finally, the application of Schauder's fixed-point theorem provides the existence of the solution in the stable set, which proves that it is an over-approximation of the set of solutions.

Consider the initial value problem

$$\begin{cases} \forall_{ae.} t \in [t_0, t_1],\ \dot{x}(t) = f\big(t, x(t), p, u(t)\big) \\ \forall t \in [t_0, t_1],\ u(t) \in U \\ x(t_0) = x_0 \in X_0 \subset \mathbb{R}^n \\ p \in P \end{cases} \tag{3.1}$$

with X_0 a bounded set of possible initial states (subset of $\mathbb{R}^n$ with n a positive natural number), p an uncertain parameter (*i.e.* constant uncertainty) with values in P, u a bounded Lebesgue-integrable time-varying uncertainty with values in a bounded set U and f a continuous function. We assume that for any parameter p and all Lebesgue-integrable uncertainty u, $t \mapsto f\big(t, x(t), p, u(t)\big)$ satisfies the Carathéodory uniqueness conditions (*cf.* Definition 8). For every triple (x_0, p, u) of an initial state $x_0 \in X_0$, a parameter $p \in P$ and a bounded Lesbegue-integrable uncertainty u with values in U, there exists a unique solution to Problem 3.1.

Remark 3 Another interesting problem is obtained by replacing the differential equation by a differential inclusion:

$$\begin{cases} \forall_{ae.} t \in [t_0, t_1],\ \dot{x}(t) \in f\big(t, x(t), p, U\big) \\ x(t_0) = x_0 \in X_0 \subset \mathbb{R}^n \\ p \in P \end{cases} \tag{3.2}$$

Following [125, Definition 3], a solution x of such a problem is an absolutely continuous function that satisfies the constraints, *i.e.* there exists a parameter $p \in P$ such that for almost all $t \in [t_0, t_1]$, $\dot{x}(t) \in f\big(t, x(t), p, U\big)$. It implies that there exists a function u with values in U such that for almost all $t \in [t_0, t_1]$, $\dot{x}(t) = f\big(t, x(t), p, u(t)\big)$. So, any solution of 3.2 is a solution of 3.1 and any over-approximation of the reachable set of 3.2 is an over-approximation of the reachable set of 3.1.

However, [125, Theorem 7] only proves that the two problems admit the same set of solutions if the set $f\big(t, x(t), p, U\big)$ is convex for all $(t, x(t), p)$ and U a compact set.

Given an initial state $x_0 \in X_0$ and a parameter $p \in P$, we want to define a set-valued version of Picard operator $\mathbb{P}_{x_0,p}$ (parameterized by x_0 and p) that handles set-

valued functions from $[t_0, t_1]$ to $\mathbb{R}^n$ in order to apply Schauder's fixed-point theorem (*cf.* Theorem 5).

First, we define a version $\mathbb{P}_{x_0,p,u}$ that is also parameterized by a Lebesgue-integrable function u with values in U. It is the Picard operator associated to Problem 3.1.

$$\mathbb{P}_{x_0,p,u}(y) = t \mapsto x_0 + \int_{t_0}^{t} f\big(s, y(s), p, u(s)\big)\, ds \tag{3.3}$$

Then, we lift it to a set-valued version that takes a set-valued function φ from $[t_0, t_1]$ to $\mathbb{R}^n$ (*i.e.* a function from $[t_0, t_1]$ to the set of all subsets of $\mathbb{R}^n$, $\mathcal{P}(\mathbb{R}^n)$):

$$\mathbb{P}_{x_0,p,u}(\varphi) = \Big\{ \mathbb{P}_{x_0,p,u}(y) \;\Big|\; y \in \mathcal{C}([t_0, t_1], \mathbb{R}^n) \wedge \forall t \in [t_0, t_1],\ y(t) \in \varphi(t) \Big\} \tag{3.4}$$

Finally, we define the set-valued version $\mathbb{P}_{x_0,p}$ that gathers the results for all possible Lebesgue-integrable functions u whose range is included in U:

$$\mathbb{P}_{x_0,p}(\varphi) = t \mapsto \bigcup_{\substack{u \text{ Lebesgue-integrable} \\ \forall t \in [t_0, t_1],\ u(t) \in U}} \mathbb{P}_{x_0,p,u}(\varphi)(t) \tag{3.5}$$

For all pairs (x_0, p) of an initial state $x_0 \in X_0$ and a parameter $p \in P$, we assume that we have a set-valued function $\varphi_{x_0,p}$ such that for all $t \in [t_0, t_1]$, $\varphi_{x_0,p}$ is a closed bounded convex set and

$$\mathbb{P}_{x_0,p}(\varphi_{x_0,p})(t) \subset \varphi_{x_0,p}(t) \tag{3.6}$$

Remark 4 While for all $t \in [t_0, t_1]$, initial state $x_0 \in X_0$ and parameter $p \in P$, $\varphi_{x_0,p}(t)$ has to be a closed convex set, we do not need any hypothesis about their union. For example

$$\psi = t \mapsto \left\{ \begin{pmatrix} t \\ t^2 \end{pmatrix} \right\}$$

is such a set-valued function: for all t, $\psi(t)$ is a closed and convex set (it is in fact a singleton), but the union of its images is not convex in $\mathbb{R}^2$.

Remark 5 Such a set-valued function $\varphi_{x_0,p}$ always exists for small enough time interval $[t_0, t_1]$, because f is continuous and U is bounded.

For example, let $\varphi_{x_0,p}$ be defined by each of its components as

$$\forall i \in [\![1, n]\!],\ \forall t \in [t_0, t_1],\ \varphi_{x_0,p,i}(t) = [x_{0,i} - a_i, x_{0,i} + a_i] \tag{3.7}$$

Due to the continuity of f and due to the boundedness of U, there exists $b \in \mathbb{R}^n_+$ such that for all Lebesgue-integrable function u with values in U, we have

$$\forall i \in [\![1, n]\!],\ \forall t \in [t_0, t_1],\ f\big(t, \varphi_{x_0,p}(t), p, u(t)\big)_i \subset [-b_i, b_i] \tag{3.8}$$

Then, using interval arithmetic, we deduce

$$\forall i \in [\![1,n]\!],\ \forall t \in [t_0,t_1],\ \mathbb{P}_{x_0,p}(\varphi_{x_0,p})(t)_i \subset [x_{0,i} - (t_1-t_0)b_i, x_{0,i} + (t_1-t_0)b_i] \quad (3.9)$$

So, such a set-valued function $\varphi_{x_0,p}$ is contracted by $\mathbb{P}_{x_0,p}$ if

$$t_0 \leq t_1 \leq t_0 + \min_{i \in [\![1,n]\!]} \left(\frac{a_i}{b_i} \right) \quad (3.10)$$

We define φ as the union of all possible $\varphi_{x_0,p}$:

$$\forall t \in [t_0,t_1],\ \varphi(t) = \bigcup_{\substack{x_0 \in X_0 \\ p \in P}} \varphi_{x_0,p}(t) \quad (3.11)$$

Such a set-valued function φ corresponds to the one computed by reachability analysis tools applied to Problem 3.1. For example, it could be computed using Taylor models replacing the uncertainties by their interval ranges as in [45].

Now, we show that $\varphi(t)$ is actually an over-approximation of the set of reachable states of Problem 3.1 for all time $t \in [t_0,t_1]$. We first notice that, for all solution x of Problem 3.1, there exists a tuple (x_0,p,u) with an initial state $x_0 \in X_0$, a value of the parameters p and a Lebesgue-integrable function u, whose range is included in U, such that for all time $t \in [t_0,t_1]$, $x(t) = x_0 + \int_{t_0}^{t} f(s,x(s),p,u(s))\,ds$. Then, for a pair (x_0,p) of an initial state and a value of parameters, we define a convex subset $K_{x_0,p}$ of the normed vector space of k-Lipschitz functions such that $K_{x_0,p}$ is stable by the application of the operator $\mathbb{P}_{x_0,p}$. Finally, we apply Schauder's fixed-point theorem (*cf.* Theorem 5) to $K_{x_0,p}$ to deduce that the image of the solution of Problem 3.1 with initial state x_0 and parameter p belongs to φ. This final result means that for all time t, $\varphi(t)$ is an over-approximation of the reachable set of Problem 3.1 at time $t \in [t_0,t_1]$.

Lemma 1
Every solution of Problem 3.1 is a uniformly Lipschitz function.

Proof Consider the unique solution x of Problem 3.1. Let p the associated value of the parameters and u the input function such that

$$\forall t \in [t_0,t_1],\ x(t) = x(t_0) + \int_{t_0}^{t} f(s,x(s),p,u(s))\,ds \quad (3.12)$$

We call R the range of x over $[t_0,t_1]$, *i.e.*

$$R = \{x(t) \mid t \in [t_0,t_1]\} \quad (3.13)$$

Because x is a continuous function as integral of an integrable function on a compact subset of $\mathbb{R}$ (closed interval $[t_0,t_1]$), the set R is a compact set.

We define ρ as the function such that

$$\forall (t,x) \in [t_0,t_1] \times R,\ \rho(t,x) = f\big(t,x,p,u(t)\big) \tag{3.14}$$

By definition, f is continuous and u is bounded. So, ρ is bounded over $[t_0,t_1] \times R$. Let M be an upper bound of the norm of the image of ρ:

$$\forall (t,x) \in [t_0,t_1] \times R,\ \|\rho(t,x)\|_\infty \leq M \tag{3.15}$$

Let t and t' be two times in $[t_0,t_1]$. We have

$$\|x(t) - x(t')\|_\infty \leq \left\| \int_{t_0}^{t} \rho(s,x(s))\,ds - \int_{t_0}^{t'} \rho(s,x(s))\,ds \right\|_\infty \leq M|t-t'| \tag{3.16}$$

So x is a uniformly M-Lipschitz function.

This lemma shows that only uniformly Lipschitz functions can be considered to define a candidate set such that the corresponding Picard operator of Problem 3.1 is a contraction mapping. Given a pair (x_0,p) of initial state and parameter of the problem and an upper bound M as in the proof of Lemma 1, we define such a candidate set $K_{x_0,p}$ as the set of M-Lipschitz functions whose images at time t are included in $\varphi_{x_0,p}$ and such that the initial value is equal to x_0:

$$K_{x_0,p} = \left\{ y \in \mathcal{C}\big([t_0,t_1],\mathbb{R}^n\big) \;\middle|\; \begin{array}{c} \forall (t,t') \in [t_0,t_1]^2,\ \|y(t) - y(t')\|_\infty \leq M|t-t'| \\ \forall t \in [t_0,t_1],\ y(t) \in \varphi_{x_0,p}(t) \\ y(t_0) = x_0 \end{array} \right\} \tag{3.17}$$

In order to apply Schauder's fixed-point theorem to it, this set $K_{x_0,p}$ has to be a convex subset of a normed vector space and to be a compact set.

Lemma 2
For all pair (x_0,p) that satisfies the hypotheses of Problem 3.1, the set $K_{x_0,p}$ is a convex subset of the normed vector space of continuous functions $\mathcal{C}\big([t_0,t_1],\mathbb{R}^n\big)$ with the supremum norm $\|\cdot\|_\infty$.

Proof By definition, $K_{x_0,p}$ is a subset of $\mathcal{C}\big([t_0,t_1],\mathbb{R}^n\big)$. So, we only have to prove that $K_{x_0,p}$ is a convex set.

Let y_1 and y_2 be two elements of $K_{x_0,p}$ and α be an element of the interval $[0,1]$. Let y be the convex combination of y_1 and y_2 with coefficents α and $1-\alpha$, *i.e.* $y = \alpha y_1 + (1-\alpha) y_2$.

1. Let $t \in [t_0, t_1]$ and $t' \in [t_0, t_1]$:

$$\begin{aligned}\|y(t) - y(t')\| &= \|\alpha(y_1(t) - y_1(t')) + (1-\alpha)(y_2(t) - y_2(t'))\|_\infty \\ &\leq \alpha\|y_1(t) - y_1(t')\|_\infty + (1-\alpha)\|y_2(t) - y_2(t')\|_\infty \\ &\leq \alpha M|t - t'| + (1-\alpha)M|t - t'| \\ &\leq M|t - t'|\end{aligned}$$

   So y is a M-Lipschitz function.

2. Let $t \in [t_0, t_1]$. Because $\varphi_{x_0,p}(t)$ is convex and because, by definition, $y_1(t)$ and $y_2(t)$ belong to $\varphi_{x_0,p}(t)$, we have $y(t) \in \varphi_{x_0,p}(t)$.

3. The initial value of y is the image of t_0:

$$\begin{aligned}y(t_0) &= \alpha y_1(t_0) + (1-\alpha)y_2(t_0) \\ &= \alpha x_0 + (1-\alpha)x_0 \\ &= x_0\end{aligned}$$

   So, the initial value of y is equal to x_0.

Lemma 3
For all pair (x_0, p) that satisfies the hypotheses of the problem 3.1, the set $K_{x_0,p}$ is a compact set.

Proof To prove that $K_{x_0,p}$ is a compact set, we have to prove that every sequence in $K_{x_0,p}$ has a cluster point. Let $(y_n)_{n\in\mathbb{N}}$ be a sequence of functions in $K_{x_0,p}$. Because $\varphi_{x_0,p}$ is bounded and the image of every element of $(y_n)_{n\in\mathbb{N}}$ is in the image of $\varphi_{x_0,p}$, the sequence $(y_n)_{n\in\mathbb{N}}$ is *uniformly bounded*. Moreover, because every element of $(y_n)_{n\in\mathbb{N}}$ is M-Lipschitz, the sequence $(y_n)_{n\in\mathbb{N}}$ is uniformly equicontinuous: for all $\varepsilon > 0$ and for all $(v_1, v_2, n) \in [t_0, t_1] \times [t_0, t_1] \times \mathbb{N}$, we have

$$|v_1 - v_2| \leq \frac{\varepsilon}{M} \implies |y_n(v_1) - y_n(v_2)| \leq M|v_1 - v_2| \leq \varepsilon \tag{3.18}$$

Using Arzelà-Ascoli's theorem (*cf.* Theorem 6), we deduce that the sequence $(y_n)_{n\in\mathbb{N}}$ has a cluster point. Let y^* be such a cluster point of $(y_n)_{n\in\mathbb{N}}$ and $(y_{n_i})_{i\in\mathbb{N}}$ be a subsequence such that $\lim_{i\to\infty} y_{n_i} = y^*$. We will now prove $y^* \in K_{x_0,p}$:

1. Let $(t, t', i) \in [t_0, t_1]^2 \times \mathbb{N}$. We have

$$\|y_{n_i}(t) - y_{n_i}(t')\|_\infty \leq M|t - t'| \tag{3.19}$$

   By continuity of the norm and of the substraction, we deduce

$$\|y^*(t) - y^*(t')\|_\infty = \lim_{i\to\infty} \|y_{n_i}(t) - y_{n_i}(t')\|_\infty \leq M|t - t'| \tag{3.20}$$

So y^* is M-Lipschitz.

2. Because for all $t \in [t_0, t_1]$, $\varphi_{x_0,p}(t)$ is closed and for all $i \in \mathbb{N}$, $y_{n_i}(t) \in \varphi_{x_0,p}(t)$, $y^*(t) \in \varphi_{x_0,p}(t)$.

3. Because for every $i \in \mathbb{N}$, $y_{n_i}(t_0) = x_0$, $y^*(t_0) = x_0$.

So, the cluster point y^* belongs to $K_{x_0,p}$, which proves that $K_{x_0,p}$ is a compact set.

Now, we need to prove that the image of the candidate set $K_{x_0,p}$ through the Picard operator $\mathbb{P}_{x_0,p}$ is included in $K_{x_0,p}$, *i.e.* for all function $y \in K_{x_0,p}$ and for all Lebesgue-integrable function u with values in U, $\mathbb{P}_{x_0,p,u}(y) \in K_{x_0,p}$.

Lemma 4
For all pair (x_0, p) that satisfies the hypotheses of Problem 3.1, the image of the set $K_{x_0,p}$ through the Picard operator $\mathbb{P}_{x_0,p}$ is included in $K_{x_0,p}$.

Proof Consider $y \in K_{x_0,p}$ and u a Lebesgue-integrable function with values in U. Let z be the image of y through $\mathbb{P}_{x_0,p,u}$, *i.e.* $z = \mathbb{P}_{x_0,p,u}(y)$.

1. Let $(t, t') \in [t_0, t_1]^2$, using the upper bound M, we have

$$\|z(t) - z(t')\|_\infty \leq \left\| \int_{t_0}^{t} \rho(s, y(s))\, ds - \int_{t_0}^{t'} \rho(s, y(s))\, ds \right\|_\infty \leq M|t - t'| \tag{3.21}$$

   with $\forall (t, x)$, $\rho(t, x) = f(t, x, p, u(t))$. So z is M-Lipschitz.

2. Because for all $t \in [t_0, t_1]$, $\varphi_{x_0,p}(t)$ is closed and $y(t) \in \varphi_{x_0,p}(t)$, we have $z(t) \in \varphi_{x_0,p}(t)$.

3. By definition of $\mathbb{P}_{x_0,p,u}$, $z(t_0) = x_0$.

So $z \in K_{x_0,p}$.

Now, using the four previous lemmas, we can prove that the set-valued function φ defines an over-approximation of the reachable set.

Theorem 7
Let x be a solution of Problem 3.1 and let φ be the set-valued function defined in 3.11. We have

$$\forall t \in [t_0, t_1],\ x(t) \in \varphi(t) \tag{3.22}$$

and φ is an over-approximation of the reachable set of Problem 3.1.

Proof Let x be a solution of Problem 3.1 with a Lebesgue-integrable uncertainty u, an initial state x_0 and a parameter p. Then, x is the unique fixed-point of $\mathbb{P}_{x_0,p,u}$:

$$x = \mathbb{P}_{x_0,p,u}(x) \tag{3.23}$$

Let $K_{x_0,p}$ be a set defined as in 3.17 with M an upper bound as in the proof of

Lemma 1. Lemma 2 and Lemma 3 prove that the set $K_{x_0,p}$ is a compact convex subset of a normed vector space. Moreover, Lemma 4 proves that the image of $K_{x_0,p}$ through $\mathbb{P}_{x_0,p}$ is included in $K_{x_0,p}$. By definition of $\mathbb{P}_{x_0,p}$, we deduce that for all Lebesgue-integrable function u with values in U, $K_{x_0,p}$ is also stable by the application of the operator $\mathbb{P}_{x_0,p,u}$. Using Schauder's fixed-point theorem (*cf.* Theorem 5), we deduce that $\mathbb{P}_{x_0,p,u}$ has a fixed-point in $K_{x_0,p}$. Due to the definition of Problem 3.1 and due to Carathéodory's uniqueness theorem (*cf.* Theorem 4), we know that the fixed-point of $\mathbb{P}_{x_0,p,u}$ is unique. So, x belongs to $K_{x_0,p}$. Finally, by definition of $K_{x_0,p}$, for all $t \in [t_0, t_1]$, $x(t) \in \varphi_{x_0,p}(t)$. By definition of φ, we deduce that for all $t \in [t_0, t_1]$, $x(t) \in \varphi(t)$, which proves that φ is an over-approximation of the reachable set of the problem 3.1.

Remark 6 The uniqueness of the solution is mandatory as presented by the counter-example below.

Consider the integral system

$$\forall t \in [0,1],\ x(t) = \int_0^t \sqrt{x(s)}\, ds \tag{3.24}$$

This system has two solutions, $x(t) = 0$ and $x(t) = t^2/4$. Now, consider the set-valued function $\varphi(t) = \{t^2/4\}$. It is stable by the operator $\mathbb{P}(y) = t \mapsto \int_0^t \sqrt{y(s)}\, ds$ and for all $t \in [0,1]$, its image is compact and convex, but it does not contain all the possible solutions of the integral system.

Theorem 7 gives us a criterion to check whether a set-valued function is an over-approximation of the reachable set of Problem 3.1. We notice that the method proposed in [45] (*cf.* Subsection 2.4.2.2) computes such a function φ using Taylor models. So, its result is an over-approximation of the reachable set even in the case of Lebesgue-integrable uncertainties.

In the following, in order to compute a rough over-approximation of the reachable set of the systems, we use this method with intervals (*cf.* Subsection 2.4.2.1), which is a particular case of Taylor models with null polynomial parts, *i.e.* only the remainder part has to be computed.

3.3 Separable systems with respect to the time-varying uncertainties

Now, we focus on a particular class of systems that we called *separable with respect to the measurable uncertainties* and that can be described as a multiplication of a function only depending on the measurable uncertainties and a function depending on the state and the time. This second function can contain constant uncertainties, *i.e.* it can be a parameterized function. We leave such parameters implicit to simplify the notations.

Definition 11 (Separable system)
Let

- $[t_0, t_1] \subset \mathbb{R}$ be a time interval
- $X \subset \mathbb{R}^n$ be a set of possible states
- x_0 be a vector in a bounded set $X_0 \subset \mathbb{R}^n$
- u be a Lebesgue-integrable function from $[t_0, t_1]$ to $\mathbb{R}^m$
- g be a continuous function from $\mathbb{R}^m$ to $\mathbb{R}^{n \times k}$
- h be a continuous function from $[t_0, t_1] \times X$ to $\mathbb{R}^k$

A *separable system with respect to the measurable uncertainties*, or simply *separable system*, is a system of the form:

$$x(t) = x_0 + \int_{t_0}^{t} g(u(s)) \cdot h(s, x(s))\, ds \tag{3.25}$$

with t the time variable in $[t_0, t_1]$, x_0 the *initial state* and u the *time-varying uncertainties*.

We assume that $t \mapsto g(u(t))$ is bounded on $[t_0, t_1]$.

To guarantee the uniqueness of the solution (*cf.* Theorem 4), we also assume that the function $(t, x) \mapsto g(u(t)) \cdot h(t, x)$ satisfies Carathéodory's uniqueness conditions (*cf.* Definition 8), *i.e.* there exists a Lebesgue-integrable function $k : [t_0, t_1] \to \mathbb{R}^n_+$ such that

$$\forall (t, x, y) \in [t_0, t_1] \times X \times X,\ \left\| g(u(t)) \cdot h(t, x) - g(u(t)) \cdot h(t, y) \right\|_\infty \leq k(t) \|x - y\|_\infty \tag{3.26}$$

We recall that the notation of the integral $\int_{t_0}^{t} f(s)\, ds$ of a function f from an interval $[t_0, t]$ to $\mathbb{R}^n$, whose components are denoted f_i for $i \in [\![1, n]\!]$, is interpreted as the vector of the Lebesgue-integrals of each component f_i on the interval $[t_0, t]$:

$$\int_{t_0}^{t} f(s)\, ds = \begin{pmatrix} \int_{[t_0,t]} f_1\, d\mu \\ \vdots \\ \int_{[t_0,t]} f_n\, d\mu \end{pmatrix}$$

Remark 7 In practice, the continuous function h satisfies the Carathéodory uniqueness conditions (*cf.* Definition 8), *i.e.* there exists a Lebesgue-integrable function l such that

$$\forall (t, x, y) \in [t_0, t_1] \times X \times X, \|h(t, x) - h(t, y)\|_\infty \leq l(t) \|x - y\|_\infty \tag{3.27}$$

Because $g(u(t))$ is bounded, there exists a non-negative real M such that

$$\forall t \in [t_0, t_1],\ \|g(u(t))\|_\infty \leq M \tag{3.28}$$

Let $(t, x, y) \in [t_0, t_1] \times X \times X$, we have

$$\begin{aligned}\|g(u(t)) \cdot h(t,x) - g(u(t)) \cdot h(t,y)\|_\infty &= \|g(u(t)) \cdot (h(t,x) - h(t,y))\|_\infty \\ &\leq \|g(u(t))\|_\infty \|h(t,x) - h(t,y)\|_\infty \\ &\leq Ml(t)\|x - y\|_\infty\end{aligned}$$

which satisfies the equation 3.26 with $k(t) = Ml(t)$.

Definition 11 is more restrictive than the general initial value problem $x(t) = x_0 + \int_{t_0}^{t} f(s, x(s), u(s))\, ds$. However, it allows all linear dynamics, defining a slightly larger class than input-affine systems. Moreover, it is sufficient to encode differential inclusions in the sense of Filippov. For example, consider two dynamics $\dot{x}(t) = f_1(t, x)$ and $\dot{x}(t) = f_2(t, x)$ from either side of a guard. In the neighborhood of the guard, the derivative should take values in the convex hull of the two right-hand sides, defining a differential inclusion $\dot{x}(t) \in \mathrm{CH}(\{f_1(t, x), f_2(t, x)\})$ for almost all time. This differential inclusion in the sense of Filippov can be interpreted as the integral system $x(t) = x(0) + \int_0^t \alpha(s) f_1\big(s, x(s)\big) + \big(1 - \alpha(s)\big) f_2\big(s, x(s)\big)\, ds$ with $\alpha : \mathbb{R} \to [0, 1]$ being an uncertain Lebesgue-integrable function. This last integral system can be written as a separable system:

$$x(t) = x(0) + \int_0^t g\big(u(s)\big) \cdot h\big(s, x(s)\big)\, ds$$

$$\text{with} \begin{cases} u(t) &= \alpha(t) \\ g(u(t)) &= \begin{pmatrix} u(t) & 0 \\ 0 & 1 - u(t) \end{pmatrix} \\ h(t, x(t)) &= \begin{pmatrix} f_1(t, x(t)) \\ f_2(t, x(t)) \end{pmatrix} \end{cases} \tag{3.29}$$

In the rest of this chapter, we use the notation of the ordinary differential equations for almost all time instead of the integral form. This allows us to explicit the differential equation associated to each component of $x(t) = \big(x_1(t), x_2(t), \ldots, x_n(t)\big)^\top$:

$$\begin{pmatrix} \dot{x}_1(t) \\ \vdots \\ \dot{x}_n(t) \end{pmatrix} = \begin{pmatrix} g_{1,1}\big(u(t)\big) & \cdots & g_{1,k}\big(u(t)\big) \\ \vdots & \ddots & \vdots \\ g_{n,1}\big(u(t)\big) & \cdots & g_{n,k}\big(u(t)\big) \end{pmatrix} \cdot \begin{pmatrix} h_1(t, x(t)) \\ \vdots \\ h_k(t, x(t)) \end{pmatrix} \tag{3.30}$$

We focus here on the time-varying uncertainties $u(t)$, but constant uncertainties, *i.e.* uncertain parameters, can be handled in functions $h(t, x)$ and $g(u)$. For example, a simple dynamical system with a proportional controller with an uncertain constant gain

p and an uncertain friction coefficient α can be written as

$$\begin{cases} \dot{x}(t) = y(t) \\ \dot{y}(t) = u(t) - px(t) - \alpha y(t) \\ x(0) = x_0 \\ y(0) = y_0 \end{cases} \tag{3.31}$$

for all $t \in [0, 10]$ with $x_0 \in [2, 3]$, $y_0 \in [-1, 1]$, $\alpha \in [1, 2]$ and $p \in [4, 5]$. We can rewrite it to explicit $g(u)$ and $h(t, x)$:

$$\begin{pmatrix} \dot{x}(t) \\ \dot{y}(t) \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ u(t) & -p & -\alpha \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x(t) \\ y(t) \end{pmatrix} \quad \text{with} \quad \begin{pmatrix} x(0) \\ y(0) \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \tag{3.32}$$

This writing is not unique. For example, we can transfer all parameters to the function h and let g depend only on the time-varying uncertainty:

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ u(t) & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ y(t) \\ -px(t) - \alpha y(t) \end{pmatrix} \quad \text{with} \quad \begin{pmatrix} x(0) \\ y(0) \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \tag{3.33}$$

Now, for such systems, we want to compute an over-approximation of the reachable set over the time. In the next subsection, we show that for each separable system, there exists a continuous parameterized system, called *auxiliary system*, whose set of admissible trajectories contains all admissible trajectories of the separable system. So, computing an over-approximation of the reachable set of this auxiliary system results in an over-approximation of the reachable set of the separable one.

3.3.1 Auxiliary parameterized continuous system

In this subsection, given a separable system, we want to compute a parameterized continuous system, *i.e.* a continuous system with constant uncertainties, such that every solution of the original separable system is a solution of the continuous one. This will imply that the reachable set of the continuous system is an over-approximation of the one of the separable system.

The first step is to over-approximate the result of the integral to construct the parameterized continuous system. To do so, we have to over-approximate the integral of each term of the matrix product $g_{i,j}(u(t)) \cdot h_j(t, x(t))$. We start with a lemma in the case of a non-negative unidimensional function:

Lemma 5
Let α and β be two Lebesgue-integrable functions from $[a, b]$ to $\mathbb{R}$. We assume that α

is bounded and β is non-negative. Then we have

$$\int_a^b \alpha(s)\beta(s)\,ds \in \left\{ c \int_a^b \beta(s)\,ds \;\middle|\; c \in \left[\inf_{s\in[a,b]} \alpha(s),\ \sup_{s\in[a,b]} \alpha(s) \right] \right\} \tag{3.34}$$

Proof Let $s \in [a,b]$ be a value in the domain of α and β. By definition of the infimum and the supremum, we have

$$\inf_{v\in[a,b]} \alpha(v) \leq \alpha(s) \leq \sup_{v\in[a,b]} \alpha(v)$$

Because $\beta(s)$ is non-negative, we obtain

$$\left(\inf_{v\in[a,b]} \alpha(v) \right) \beta(s) \leq \alpha(s)\beta(s) \leq \left(\sup_{v\in[a,b]} \alpha(v) \right) \beta(s)$$

Using the monotonicity of the integration, we deduce

$$\int_a^b \left(\inf_{v\in[a,b]} \alpha(v) \right) \beta(s)\,ds \leq \int_a^b \alpha(s)\beta(s)\,ds \leq \int_a^b \left(\sup_{v\in[a,b]} \alpha(v) \right) \beta(s)\,ds$$

Using the linearity of the integration, we finally have

$$\left(\inf_{v\in[a,b]} \alpha(v) \right) \int_a^b \beta(s)\,ds \leq \int_a^b \alpha(s)\beta(s)\,ds \leq \left(\sup_{v\in[a,b]} \alpha(v) \right) \int_a^b \beta(s)\,ds$$

So, because $\mathbb{R}$ is connected and complete, there exists $c \in \left[\inf_{s\in[a,b]} \alpha(s), \sup_{s\in[a,b]} \alpha(s)\right]$ such that

$$\int_a^b \alpha(s)\beta(s)\,ds = c \int_a^b \beta(s)\,ds$$

Remark 8 Only knowing the image of α, the inclusion is optimal. Let $[\underline{\alpha}, \overline{\alpha}]$ be an interval such that

$$\forall s \in [a,b],\ \alpha(s) \in [\underline{\alpha}, \overline{\alpha}] \tag{3.35}$$

For all $c \in [\underline{\alpha}, \overline{\alpha}]$, if α is such that

$$\forall s \in [a,b],\ \alpha(s) = \begin{cases} \underline{\alpha} & \text{if } s = a \\ \overline{\alpha} & \text{if } s = b \\ c & \text{otherwise} \end{cases} \tag{3.36}$$

then

$$\int_a^b \alpha(s)\beta(s)\,ds = c \int_a^b \beta(s)\,ds \tag{3.37}$$

It proves that every value of the set defined in Lemma 5 can be reached.

Corollary 1
If β is not a non-negative function, then given a decomposition of it as a difference of non-negative functions $\beta = \beta^+ - \beta^-$, we have

$$\int_a^b \alpha(s)\beta(s)\,ds \in \left\{ c_1 \int_a^b \beta^+(s)\,ds - c_2 \int_a^b \beta^-(s)\,ds \;\middle|\; (c_1, c_2) \in I \times I \right\} \tag{3.38}$$

with

$$I = \left[\inf_{s\in[a,b]} \alpha(s),\ \sup_{s\in[a,b]} \alpha(s) \right] \tag{3.39}$$

Proof Using the linearity of the integral, we have

$$\int_a^b \alpha(s)\beta(s)\,ds = \int_a^b \alpha(s)\beta^+(s)\,ds - \int_a^b \alpha(s)\beta^-(s)\,ds \tag{3.40}$$

Using Lemma 5, we deduce that there exist $c_1 \in I$ and $c_2 \in I$ such that

$$\int_a^b \alpha(s)\beta^+(s)\,ds = c_1 \int_a^b \beta^+(s)\,ds \quad \text{and} \quad \int_a^b \alpha(s)\beta^-(s)\,ds = c_2 \int_a^b \beta^-(s)\,ds \tag{3.41}$$

We deduce the result of the corollary by substitution in the equation 3.40:

$$\int_a^b \alpha(s)\beta(s)\,ds = c_1 \int_a^b \beta^+(s)\,ds - c_2 \int_a^b \beta^-(s)\,ds \tag{3.42}$$

Now, we can define a parameterized continuous system, which is simpler to study than the separable system of Definition 11. The new continuous system is computed from a decomposition as a difference of non-negative functions of h and by the introduction of extra uncertain parameters corresponding to the elements c_1 and c_2 of Corollary 1.

Definition 12 (Auxiliary parameterized continuous system)
Let

- $[t_0, t_1] \subset \mathbb{R}$ be a time interval
- $X \subset \mathbb{R}^n$ be a set of possible states
- x_0 be a vector in a bounded set $X_0 \subset \mathbb{R}^n$
- u be a Lebesgue-integrable function from $[t_0, t_1]$ to $\mathbb{R}^m$
- g be a continuous function from $\mathbb{R}^m$ to $\mathbb{R}^{n\times k}$
- h be a continuous function from $[t_0, t_1] \times X$ to $\mathbb{R}^k$

Let I_g be a matrix of intervals such that:

$$\forall (i,j) \in [\![1,n]\!] \times [\![1,k]\!],\ \forall s \in [t_0,t_1],\ g(u(s))_{i,j} \in I_{g_{i,j}} \tag{3.43}$$

Let A and B matrices in I_g.

Let $h^+ - h^- = h$ be a decomposition of h as difference of non-negative continuous functions:

$$\forall j \in [\![1,k]\!],\ h_j^+ \geq 0 \quad \text{and} \quad h_j^- \geq 0 \tag{3.44}$$

An *auxiliary parameterized continuous system*, or simply *auxiliary system*, associated to the separable system 3.25 is a system of the form

$$x(t) = x_0 + \int_{t_0}^{t} \left(Ah^+(s,x(s)) - Bh^-(s,x(s))\right) ds \tag{3.45}$$

with t the time variable in $[t_0,t_1]$, x_0 the *initial state* and A and B the parameters.

To guarantee the uniqueness of the solution (*cf.* Theorem 4), we also assume that the function $(t,x) \mapsto Ah^+(t,x) - Bh^-(t,x)$ satisfies the Carathéodory uniqueness conditions (*cf.* Definition 8), *i.e.* there exists a Lebesgue-integrable function $k : [t_0,t_1] \to \mathbb{R}_+^n$ such that

$$\begin{aligned} &\forall (t,x,y) \in [t_0,t_1] \times X \times X, \\ &\quad \left\| \left(Ah^+(t,x) - Bh^-(t,x)\right) - \left(Ah^+(t,y) - Bh^-(t,y)\right) \right\|_\infty \leq k(t) \|x-y\|_\infty \end{aligned} \tag{3.46}$$

Remark 9 A decomposition $h = h^+ - h^-$ always exists, but it is often non-unique. For example, if h is a function from $[t_0,t_1]$ to some closed interval $[a,b]$, then we can define

$$h^+ = \frac{1}{2}(h + \max(-a,b)) \qquad \text{and} \qquad h^- = \frac{1}{2}(\max(-a,b) - h) \tag{3.47}$$

Then h^+ and h^- are two non-negative continuous functions as the composition of an affine transformation and h, which is continuous by definition. In the case of a vector-valued function, the decomposition is applied to each component.

Multiple decompositions are compared in Subsection 3.3.2.

Remark 10 In practice, h^+ and h^- are Lipschitz compositions of h, *i.e.* there exist two Lipschitz functions σ_+ and σ_- such that

$$h^+ = \sigma_+ \circ h \qquad \text{and} \qquad h^- = \sigma_- \circ h \tag{3.48}$$

Moreover, as pointed in Remark 7, we can often assume that there exists a Lebesgue-integrable function l such that

$$\forall (t,x,y) \in [t_0,t_1] \times X \times X,\ \|h(t,x) - h(t,y)\|_\infty \leq l(t) \|x-y\|_\infty \tag{3.49}$$

Because h is continuous, so are h^+ and h^-. Let k_+ (*resp.* k_-) be a Lipschitz coefficient of σ_+ (*resp.* σ_-) on its domain. Let $(t, x, y) \in [t_0, t_1] \times X \times X$, we have

$$\begin{aligned}
&\Big\|(Ah^+(t,x) - Bh^-(t,x)) - (Ah^+(t,y) - Bh^-(t,y))\Big\|_\infty \\
&\qquad = \Big\|A(h^+(t,x) - h^+(t,y)) - B(h^-(t,x) - h^-(t,y))\Big\|_\infty \\
&\qquad \leq \|A\|\Big\|h^+(t,x) - h^+(t,y)\Big\|_\infty + \|B\|\|h^-(t,x) - h^-(t,y)\|_\infty \\
&\qquad \leq \|A\|k_+\|h(t,x) - h(t,y)\|_\infty + \|B\|k_-\|h(t,x) - h(t,y)\|_\infty \\
&\qquad \leq \|A\|k_+ l(t)\|x - y\|_\infty + \|B\|k_- l(t)\|x - y\|_\infty \\
&\qquad \leq (\|A\|k_+ + \|B\|k_-)l(t)\|x - y\|_\infty
\end{aligned}$$

so $(t, x) \mapsto Ah^+(t, x) - Bh^-(t, x)$ satisfies the Carathéodory uniqueness conditions.

Theorem 8
Let $h^+ - h^-$ be a decomposition of h as difference of non-negative functions. Let $\{\varphi_{\{x_0,A,B\}}\}$ be a collection of set-valued functions such that for all unique solution x of the auxiliary system with parameters $\{x_0, A, B\}$, we have

$$\forall t \in [t_0, t_1],\ x(t) \in \varphi_{\{x_0,A,B\}}(t) \tag{3.50}$$

The union $\varphi = t \mapsto \bigcup_{\{x_0,A,B\}} \varphi_{\{x_0,A,B\}}(t)$ over all possible values of the parameters is an over-approximation of the reachable set over the time of the associated separable system.

Proof Let x be a solution of the separable system for some initial state x_0 and some input u:

$$\forall t \in [t_0, t_1],\ x(t) = x_0 + \int_{t_0}^{t} g(u(s)) \cdot h(s, x(s))\, ds \tag{3.51}$$

Using Corollary 1, there exist matrices A and B in the matrix of intervals I_g, over-approximating the range of $g(u(t))$ over the time interval $[t_0, t_1]$, such that

$$\forall t \in [t_0, t_1],\ x(t) = x_0 + \int_{t_0}^{t} \Big(Ah^+\big(s, x(t)\big) - Bh^-\big(s, x(s)\big)\Big)\, ds \tag{3.52}$$

i.e. x is the unique solution of the auxiliary system with parameters $\{x_0, A, B\}$.

Because $\varphi_{\{x_0,A,B\}}$ defines an over-approximation of its reachable set over the time, we have for all time $t \in [t_0, t_1]$, $x(t) \in \varphi_{\{x_0,A,B\}}(t)$. So, by definition of φ, for all time $t \in [t_0, t_1]$, $x(t) \in \varphi(t)$, which proves the theorem.

This theorem proves that we just have to compute an over-approximation of the reachable set of the auxiliary system, with uncertain initial state x_0 and uncertain parameters A and B, to obtain an over-approximation of the reachable set of the associated separable system.

In order to compute such an over-approximation of the reachable set of the auxiliary system, we can use all method that is valid with the uncertain parameterized systems, *i.e.* of the form $\dot{x}(t) = f(x(t))$, because the auxiliary system is continuous with constant uncertainties, *i.e.* only uncertain parameters without time-varying uncertainties.

3.3.2 Optimal decomposition as a difference of non-negative functions

In the previous subsection, we showed that we need to decompose a function $h : \mathbb{R}\times\mathbb{R}^n \to \mathbb{R}^k$ as a difference of non-negative functions $h^+ - h^- = h$. In this subsection, we discuss the *best* decomposition to obtain the minimal over-approximation of the reachable set assuming everything is computing using Taylor models (*cf.* Subsection 2.4.1). To do so, we minimize the over-approximation of the Picard operator.

We here focus on the one dimensional case, *i.e.* $n = k = 1$, $g : U \to \mathbb{R}$ and $h : \mathbb{R}\times\mathbb{R} \to \mathbb{R}$. In that case, the images of h^+ and h^- are simple Taylor models (p^+, I^+) and (p^-, I^-) with p^+ and p^- two multivariate polynomials depending on multiple parameters including the initial state x_0 and the time t. Let $(\alpha, [0,0])$ and $(\beta, [0,0])$ be two other Taylor models encoding the parameters A and B in a domain $(\alpha, \beta) \in I_g \times I_g$, with I_g be the closed convex hul of the image of g over the set U of values of the uncertainties. Then, assuming that the initial time $t_0 = 0$, the image $\mathbb{P}_{x_0,\alpha,\beta}(x)$ of the Picard operator lifted with the Taylor model arithmetic, for a time step $\Delta t > 0$ (the domain of the time variable t is $[0, \Delta t]$), will be

$$\mathbb{P}_{x_0,\alpha,\beta}(x) = (x_0, [0,0]) + \int \Big((\alpha, [0,0]) \cdot \left(p^+, I^+\right) - (\beta, [0,0]) \cdot \left(p^-, I^-\right) \Big)\, dt \tag{3.53}$$

$$= (x_0, [0,0]) + \int \left(\alpha p^+ - \beta p^-, I_g \cdot I^+ - I_g \cdot I^- \right) dt \tag{3.54}$$

$$= \left(x_0 + \int (\alpha p^+ - \beta p^- - p_e)\, dt, (I_g \cdot I^+ - I_g \cdot I^- + [p_e]) \cdot [0, \Delta t] \right) \tag{3.55}$$

with $\int (p, I)\, dt$ be the primitive operator with respect to the variable t applied to the Taylor model (p, I) (*cf.* Subsection 2.4.1).

Our goal is to minimize the width of the interval $(I_g \cdot I^+ - I_g \cdot I^- + [p_e]) \cdot [0, \Delta t]$. The decomposition of h does not influence the value of Δt, but it influences I^+, I^- and p_e.

We want an easy automatic way to compute the decomposition of h, so we assume that h^+ and h^- are only functions of the image of h, not of the state or the time, *i.e.* there exist two functions λ^+ and λ^- such that $h^+ = \lambda^+ \circ h$ and $h^- = \lambda^- \circ h$. Those two functions will be over-approximated using the Taylor model arithmetic. So, in order to avoid extra remainders, we restrict our study to polynomial decompositions, *i.e.*

$$\lambda^+ : x \mapsto \sum_{i=0}^{n^+} a_i^+ x^i \qquad \text{and} \qquad \lambda^- : x \mapsto \sum_{i=0}^{n^-} a_i^- x^i \tag{3.56}$$

Because λ^+ and λ^- should be independent of the parameters α and β, the only way to limit p_e is to restrict λ^+ and λ^- to affine forms, *i.e.* $n^+ = n^- = 1$. In that case, because

$h = (\lambda^+ \circ h) - (\lambda^- \circ h)$, the functions become

$$\lambda^+ : x \mapsto ax + b \qquad \text{and} \qquad \lambda^- : x \mapsto (a-1)x + b \tag{3.57}$$

with $a \in \mathbb{R}$ and $b \in \mathbb{R}$. Because p_e cannot be affected by the affine decomposition, we assume that p_e is null and the remainder becomes $\big(aI_g \cdot I - (a-1)I_g \cdot I\big) \cdot [0, \Delta t]$. Notice that we cannot factorize by $I_g \cdot I$ due to the interval arithmetic (*cf.* Subsection 2.4.1). We also assume that the remainders are centered after every operation. Let $m \in \mathbb{R}_+$ such that $[-m, m] = I_g \cdot I$. Depending on the value of a, we have

$$\begin{array}{rcl} a \leq 0 & \implies & \big(aI_g \cdot I - (a-1)I_g \cdot I\big) \cdot [0, \Delta t] = [-m\Delta t, m\Delta t] + [2am\Delta t, -2am\Delta t] \\ a \in [0,1] & \implies & \big(aI_g \cdot I - (a-1)I_g \cdot I\big) \cdot [0, \Delta t] = [-m\Delta t, m\Delta t] \\ a \geq 1 & \implies & \big(aI_g \cdot I - (a-1)I_g \cdot I\big) \cdot [0, \Delta t] = [-m\Delta t, m\Delta t] + [-2am\Delta t, 2am\Delta t] \end{array} \tag{3.58}$$

So, for all $a \in \mathbb{R}$, $[-m\Delta t, m\Delta t] \subset \big(aI_g \cdot I - (a-1)I_g \cdot I\big) \cdot [0, \Delta t]$ and the remainder is minimal for $a \in [0, 1]$.

In practice, we do not know *a priori* the remainder of the Taylor model associated to h, because it depends on the Taylor model associated to the state that we try to compute. However, we can easily compute an interval over-approximation of h using interval arithmetic. This produces a rough over-approximation that is fast to compute. Let $[\underline{h}, \overline{h}]$ be such a rough over-approximation. If $\underline{h} \geq 0$, then h is non-negative and an obvious decomposition is

$$h^+ = h \qquad \text{and} \qquad h^- = 0 \tag{3.59}$$

It corresponds to an affine decomposition with $a = 1$ and $b = 0$, which is optimal. In the same way, if $\overline{h} \leq 0$, then h is non-positive and an obvious decomposition is

$$h^+ = 0 \qquad \text{and} \qquad h^- = -h \tag{3.60}$$

It corresponds to an affine decomposition with $a = 0$ and $b = 0$, which is also optimal. Otherwise, *i.e.* $\underline{h} < 0 < \overline{h}$, any values $a \in [0, 1]$ and $b \geq \max\big(-a\underline{h}, (1-a)\overline{h}\big)$ compute an optimal decompositions.

Due to (naive) implementations of the evaluation of polynomials in which we evaluate monomials with interval arithmetic, we also want to minimize the absolute value of the coefficients: the multiplication of an interval of width $w \geq 0$ by a real number $\alpha \in \mathbb{R}$ results in an interval of width $|\alpha|w$ and the sum of intervals results in an interval whose width is the sum of the widths of the operands. So, we would like to minimize the extra constant (non-negative) coefficient b of the decomposition. Because, when a increases, $-a\underline{h}$ increases and $(1-a)\overline{h}$ decreases, the lower bound of b is minimal when $-a\underline{h} = (1-a)\overline{h}$, *i.e.*

$$a = \frac{\overline{h}}{\overline{h} - \underline{h}} \tag{3.61}$$

and we can set the value of b to its lower bound

$$b = \frac{-\underline{h}\overline{h}}{\overline{h} - \underline{h}} \tag{3.62}$$

So, we defined an affine decomposition of the function h as difference of non-negative functions that is optimal for a computation using Taylor models, in the sense that it minimizes the remainder of the resulting Taylor models by the Picard operator.

3.3.3 Algorithm for reachability analysis of separable systems

In this subsection, we present an algorithm able to compute an over-approximation of the reachable set of a continuous system with measurable bounded uncertainties. To exploit our previous result, we assume that the continuous system is a separable system (*cf.* Definition 11). A detailed example of application of this algorithm is presented in the following Subsection 3.3.4.

Our algorithm takes as input a separable system (*i.e.* two functions g and h, a set of possible initial states X_0, a set U of values of the time-varying uncertainties and a time interval $[t_0, t_1]$) and a time step (*i.e.* $dt > 0$) and it returns a sequence of Taylor models $(\varphi_i)_{i \in [\![1, \lceil (t_1 - t_0)/dt \rceil]\!]}$, each one over-approximating the set of reachable states on the time interval $\big[t_0 + (i-1)dt, \min(t_1, i\,dt)\big]$: if x is solution of the system, then for all $i \in [\![1, \lceil (t_1 - t_0)/dt \rceil]\!]$ and $t \in \big[t_0 + (i-1)dt, \min(t_1, t_0 + i\,dt)\big]$, $x(t) \in \varphi_i(t)$. We split the algorithm in two parts: a subroutine in charge of a unique step of integration on a time interval $[t'_0, t'_1]$ and the main part iterating the subroutine until an over-approximation is computed on the whole time interval $[t_0, t_1]$.

First, we describe the subroutine in charge of computing the Taylor model over-approximating one step of integration, *i.e.* on a time interval $[t_0, t_1]$ (with t_0 and t_1 as local variables, potentially different than the ones used in the global algorithm). This subroutine takes as inputs a vector of Taylor models encoding the set of initial states (p_0, I_0), a matrix of intervals encoding the set of images of $\mathit{interval}(g(U))$ and the function h. It returns a vector of Taylor models encoding an over-approximation of the set of reachable states for every time $t \in [t_0, t_1]$, *i.e.* depending on the time variable t whose domain is $[t_0, t_1]$.

1. Compute a rough over-approximation of the reachable set over the time interval $[t_0, t_1]$ using the interval arithmetic, given the set of initial states at time t_0. Then, we compute a rough over-approximation of $h(t, x(t))$ over the considered time interval, replacing x be the rough over-approximation of the reachable set and t by the interval $[t_0, t_1]$.

2. Compute a decomposition of h as a difference of non-negative functions h^+ and h^-, given its rough over-approximation, following the definition of the optimal decomposition presented in the previous Subsection 3.3.2.

3. Compute the auxiliary parameterized continuous system using fresh variables encoding the uncertain parameters: $\alpha \in \mathit{interval}(g(U))$ and $\beta \in \mathit{interval}(g(U))$.

4. Compute an over-approximation of the auxiliary parameterized continuous system using classic guaranteed integration methods (*cf.* Subsection 2.4.2).

The main algorithm calls this subroutine to compute an over-approximation over a subdivision of the desired time interval $[t_0, t_1]$. It takes as inputs a time interval $[t_0, t_1]$, a time step $dt > 0$, a set of initial states X_0, a set of uncertainties U and a separable system defined by the function h and g (*cf.* Definition 11). Because we have to iterate over a subdivision of time intervals, we need local variables: an interval matrix $\mathit{interval}(g(U))$, a time interval $[t'_0, t'_1]$ on which to apply the subroutine (initialized to $[t_0, \min(t_1, t_0 + dt)]$), a Taylor model (p_0, I_0) encoding the set of local initial states (initialized to a Taylor models encoding X_0), a Taylor model (p, I) encoding the over-approximation over the current time interval (not initialized), and a sequence of time intervals and Taylor models $r = \Big(([t'_0, t'_1], (p_i, I_i))\Big)_i$ keeping track of the previous computed over-approximations (initialized to an empty list). A step of the algorithm consists in calling the subroutine with $[t'_0, t'_1]$, (p_0, I_0), h and $\mathit{interval}(g(U))$ and saving its output in (p, I). Then, we append (p, I) to r, we update (p_0, I_0) to the partial evaluation of (p, I) at time t'_1 and we update $[t'_0, t'_1]$ to its translation by dt intersected with $[t_0, t_1]$ (*i.e.* $[t'_1, \min(t_1, t'_1 + dt)]$). Finally, if t'_0 is strictly smaller than t_1 then we compute another step, otherwise we return the list of time intervals and Taylor models r.

3.3.4 Application of the algorithm on a simple example

Now, let us apply the algorithm on an example. We study a toy example with only one state variable and one time-varying uncertainty in order to be able to detail every step of the algorithm.

$$\begin{cases} \forall t \in [0, 0.2],\ \dot{x}(t) = (0.1 - t) \cdot u(t) \\ \forall t \in [0, 0.2],\ u(t) \in [-1, 1] \\ x(0) = 0 \end{cases} \tag{3.63}$$

This example has the particularity that the reachable set at time $t = 0.2$ is radically different depending on whether we consider u as a constant or as a time-varying function: if for all $t \in [0, 0.2]$, $u(t) = u(0) = p \in [-1, 1]$, then we have for all $t \in [0, 0.2]$, $x(t) = (0.1 - 0.5p)pt$ and for all $p \in [-1, 1]$, $x(0.2) = 0$. However, if for all $t \in [0, 0.1[$, $u(t) = 1$ and for all $t \in [0.1, 0.2]$, $u(t) = -1$, then we have for all $t \in [0, 0.1]$, $x(t) = (0.1 - 0.5t)t$ and for all $t \in [0.1, 0.2]$, $x(t) = 0.01 + (0.5t - 0.1)t$ and $x(0.2) = 0.01$.

Exact reachable set with respect to the time To compare the precision of our over-approximations, we compute the exact reachable set with respect to the time.

First, because $u(t) \in [-1, 1]$, we notice that for all $t \in [0, 0.1]$, we have

$$t - 0.1 \leq (0.1 - t) \cdot u(t) \leq 0.1 - t \tag{3.64}$$

Because $x(0) = 0$, we can integrate each expression over $[0, t]$ to obtain an enclosure of the solution

$$0.5t^2 - 0.1t \leq x(t) \leq 0.1t - 0.5t^2 \tag{3.65}$$

Moreover, the lower (*resp.* upper) bound is reached with the constant input $u(t) = -1$ (*resp.* $u(t) = 1$). So the exact reachable set at time $t \in [0, 0.1]$ is $[0.5t^2 - 0.1t, 0.1t - 0.5t^2]$.

Then, because $u(t) \in [-1, 1]$, we notice for all $t \in [0.1, 0.2]$, we have

$$0.1 - t \leq (0.1 - t) \cdot u(t) \leq t - 0.1 \tag{3.66}$$

Using the reachable set at time $t = 0.1$, we know that $-5 \cdot 10^{-3} \leq x(0.1) \leq 5 \cdot 10^{-3}$. We can integrate each expression over $[0.1, t]$ and add the initial value at $t = 0.1$ to obtain an enclosure of the solution

$$0.1t - 0.5t^2 - 0.01 \leq x(t) \leq 0.5t^2 - 0.1t + 0.01 \tag{3.67}$$

Moreover, the lower (*resp.* upper) bound is reached with the constant input $u(t) = 1$ (*resp.* $u(t) = -1$) and initial value $x(0.1) = -5 \cdot 10^{-3}$ (*resp.* $x(0.1) = 5 \cdot 10^{-3}$). So the exact reachable set at time $t \in [0.1, 0.2]$ is $\left[0.1t - 0.5t^2 - 0.01, 0.5t^2 - 0.1t + 0.01\right]$.

We computed the exact reachable set over the time interval $[0, 0.2]$ and we can deduce that $x(0.2) \in [-0.01, 0.01]$. This reachable set is compared to two over-approximations in Figure 3.3.

Computation with a single step The time interval $[0, 0.2]$ is small enough to be handled in only one step, *i.e.* we set $dt = 0.2$. We just have to apply the subroutine described in Subsection 3.3.3 with the inputs $[t_0, t_1] = [0, 0.2]$, the set of initial states $(0, [0])$, the convex hull of the image of g over the set of uncertainties $g(U) \subset [-1, 1]$ and the function $h : (t, x) \mapsto (0.1 - t)$:

1. We start computing a rough over-approximation of the solution with interval arithmetic. In this example, because the right-hand side of the differential equation does not depend on the state, we directly have a fixed-point:

$$\forall t \in [0, 0.2],\ x(t) \in [x] = 0 + \int_0^{0.2} (0.1 - [0, 0.2]) \cdot [-1, 1]\, ds = [-0.02, 0.02] \tag{3.68}$$

   N.B.: this over-approximation is useless for this example, because the function h does not depend on the state.

   Then, we compute an over-approximation of h using interval arithmetic:

$$\forall t \in [0, 0.2],\ h(t, x(t)) \in [\underline{h}, \overline{h}] = 0.1 - [0, 0.2] = [-0.1, 0.1] \tag{3.69}$$

2. Based on the rough over-approximation of h, we compute a decomposition as difference of non-negative functions following the optimal affine transformation presented in the equations 3.61 and 3.62 (*cf.* Subsection 3.3.2):

$$a = \frac{0.1}{0.1 - (-0.1)} = 0.5 \quad \text{and} \quad b = -\frac{(-0.1) \cdot 0.1}{0.1 - (-0.1)} = 0.05 \tag{3.70}$$

   so, we have

$$\begin{cases} h^+ = 0.5h + 0.05 = (t, x) \mapsto (0.1 - 0.5t) \\ h^- = -0.5h + 0.05 = (t, x) \mapsto 0.5t \end{cases} \tag{3.71}$$

3. We replace h by its decomposition and we introduce two parameters α and β in $[-1,1]$ to define the auxiliary parameterized continuous system:

$$\begin{cases} \forall t \in [0,0.2],\ \dot{x}(t) = \alpha(0.1 - 0.5t) - 0.5\beta t \\ x(0) = 0 \\ \alpha \in [-1,1] \\ \beta \in [-1,1] \end{cases} \tag{3.72}$$

4. Finally, we compute an over-approximation of the parameterized continuous system 3.72 using a classic method. We use here an iteration of the Picard operator from a constant approximation $\varphi_0 = (x_0, [0])$ with $x_0 = x(0) = 0$ using Taylor models up to order 2 (*cf.* Subsection 2.4.2):

$$\begin{array}{rcl} \varphi_1 & = & x_0 + \int \big((\alpha,[0]) \cdot (0.1 - 0.5(t,[0])) - 0.5(\beta,[0]) \cdot (t,[0])\big)\, dt \\ & = & \big(\int (0.1\alpha - 0.5(\alpha+\beta)t)\, dt, [0] \cdot [0,0.2]\big) \\ & = & \big(0.1\alpha t - 0.25(\alpha+\beta)t^2, [0]\big) \end{array} \tag{3.73}$$

The sequence reached a fixed-point:

$$\forall n \mathbb{N},\ n \geq 1 \implies \varphi_n = \varphi_1 \wedge \varphi_{n+1} \subset \varphi_n \tag{3.74}$$

So the resulting Taylor model is $\big(0.1\alpha t - 0.25(\alpha+\beta)t^2, [0]\big)$ with $t \in [0,0.2]$, $\alpha \in [-1,1]$ and $\beta \in [-1,1]$.

The Taylor model can be rewritten as $\big((0.1t - 0.25t^2)\alpha - 0.25t^2\beta, [0]\big)$ and because for all $t \in [0,0.2]$, $(0.1t - 0.25t^2) \geq 0$, we have

$$\forall t \in [0,0.2],\ x(t) \in \big[-0.1t, 0.1t\big] \tag{3.75}$$

This over-approximation implies $x(0.2) \in [-0.02, 0.02]$, which is twice larger than the exact reachable set. This over-approximation is represented in Figure 3.3.

Computation with two steps We can follow the same steps with a smaller time-step $dt = 0.1$. Over the time interval $[0, 0.1]$, the function h is non-negative, so its decomposition should be $h^+ = h$ and $h^- = 0$. Then we get a Taylor model $\big(\alpha_0(0.1t - 0.5t^2), [0]\big)$ with the parameter $\alpha_0 \in [-1,1]$ multiplying the non-negative function h^+ and a time variable $t \in [0, 0.1]$. During the computation, another parameter $\beta_0 \in [-1,1]$ should be introduced, but it is multiplied by the null function h^-.

To compute an over-approximation over the time interval $[0.1, 0.2]$, we substitute the time variable of the previous Taylor model by its maximal value, *i.e.* $t = 0.1$, resulting in a new Taylor model $(0.005\alpha_0, [0])$. Now, the function h is non-positive over the time interval $[0.1, 0.2]$, so its decomposition should be $h^+ = 0$ and $h^- = -h$. Then we get a Taylor model $\big(0.005(\alpha_0 + \beta_1) + 0.5\beta_1 t^2 - 0.1\beta_1 t, [0]\big)$ with the previous parameter $\alpha_0 \in [-1,1]$, a fresh parameter $\beta_1 \in [-1,1]$ multiplying the non-negative function h^- and a time variable $t \in [0.1, 0.2]$. During the computation, another parameter $\alpha_1 \in [-1,1]$ should be introduced, but it is multiplied by the null function h^+.

We notice that this over-approximation is equal to the exact reachable set. It is also drawn in Figure 3.3.

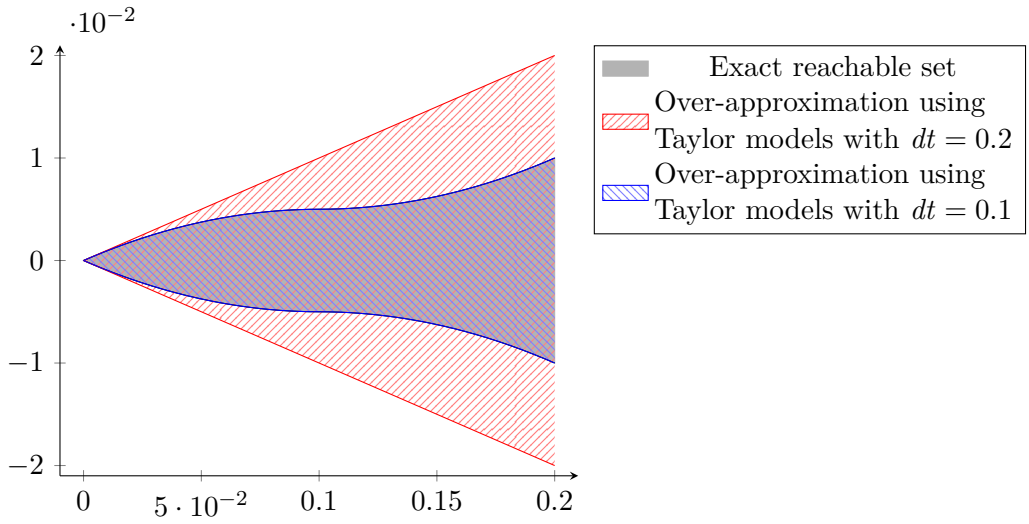


Figure 3.3 – Over-approximations computed following our algorithm with Taylor models

3.4 Over-approximation of the reachable set of switched systems interpreted with differential inclusions

We showed in the previous section how to compute an over-approximation of a separable system with Lebesgue-measurable time-varying uncertainties. Such systems arise when a switch is interpreted in the sense of Filippov in order to allow admissible trajectories beyond Zeno events (*cf.* Subsection 2.1.2).

In this section, we present an algorithm to compute over-approximations of the reachable set of a given switched system in presence of Zeno behaviors. First, we present the base of the algorithm with the definition of some useful functions. This base is a classic algorithm for hybrid automata without Zeno events and it is used in the rest of this chapter. Then, we adapt it to the case of switched systems without having to explicit the sliding modes interpreted in the sense of Filippov. Finally, we illustrate an application of the proposed algorithm to the example of Section 3.1.

For sake of simplicity, we consider only switched systems without time-varying uncertainties. However, the bounded time-varying uncertainties can be handled considering integration methods that are able to handle such uncertainties, *e.g.* the method presented in the previous Section 3.3. Similarly, the evaluations of the guards and of the invariants could be simply adapted to time-varying uncertainties replacing their occurrences by the sets representing their ranges (*cf.* Section 2.4).

All the sets of this section are assumed closed, bounded and convex.

3.4.1 Base of the algorithm

Consider a model of a hybrid automaton for which we want to compute an over-approximation of the reachable set with respect to the time over the interval $[t_0, T]$

with a set X_0 of possible initial states at time t_0. So, we want to compute a collection $\{([\underline{t_k}, \overline{t_k}], v_k, dt_k, \varphi_k)\}_{k \in [\![0,N]\!]}$ where $[\underline{t_k}, \overline{t_k}]$ is a time interval of an uncertain (local) initial time, v_k is the local activated mode, $dt_k \in \mathbb{R}_{\geq 0}$ is the duration of the local over-approximation φ_k, such that for all admissible trajectory $\Big((\delta_j, v_j, x_j)\Big)_{j \in J}$ of the system and for all index $j \in J$, there exist an index $k \in [\![0, N]\!]$, a time $t_k \in [\underline{t_k}, \overline{t_k}]$ and an elapsed time $\xi_k \in [0, dt_k]$ such that $v_j = v_k$, $\Delta_j = \sum_{l=0}^{j} \delta_l = t_k + \xi_k$ and $x_j \in \varphi_k(\xi_k)$.

We assume that we have a function *integrate* that, given a dynamics f_p (a continuous function parameterized by some parameters p in a given domain), a set of possible initial states X_k, an interval $[\underline{t_k}, \overline{t_k}]$ of possible initial times, and a duration $dt > 0$, returns a set-valued function φ over $[0, dt]$ such that φ is an over-approximation of the reachable set over the time from $[\underline{t_k}, \overline{t_k}]$ and X_k, *i.e.* for all $t_k \in [\underline{t_k}, \overline{t_k}]$, for all solution of the differential equation $\dot{x}(t) = f_p(t, x(t))$ with $x(t_k) \in X_k$ and for all time $\delta \in [0, dt]$, $x(t_k + \delta)$ belongs to $\varphi(\delta)$. Some possibilities for the implementation of such a function are described in Subsection 2.4.2.

We assume that we have a function *cover* that, given a pair $([\underline{t}, \overline{t}], X)$ of a time interval and a set of states, returns a collection of pairs $\{([\underline{t_i}, \overline{t_i}], X_i)\}$ such that for all time $t \in [\underline{t}, \overline{t}]$ and for all state $x \in X$, there exists an index i such that $t \in [\underline{t_i}, \overline{t_i}]$ and $x \in X_i$. Such a function is useful in order to tighten the over-approximations of the sets of states activated some guards. Notice that its implementation highly depends on the representation of the sets.

We also assume that we have a function *detect* that, given a triple $([\underline{t_k}, \overline{t_k}], dt_k, \varphi_k)$ as before and a boolean function g_b (typically a guard of a transition that returns true if it is activated and false otherwise), returns a collection of pairs $\Big(\Big[\underline{\delta_d}, \overline{\delta_d}\Big], X_d\Big)$ such that for all initial time $t_k \in [\underline{t_k}, \overline{t_k}]$ and for all absolutely continuous function x such that for all $\delta \in [0, dt_k]$, $x(t_k + \delta) \in \varphi_k(\delta)$, there exist an index d and an elapsed time $\delta_t \in \Big[\underline{\delta_d}, \overline{\delta_d}\Big]$ such that $x(t_k + \delta_t) \in X_d$, g_b evaluated at the state $(t_k + \delta_t, x(t_k + \delta_t))$ is true and for all $\delta_f < \delta_t$, g_b evaluated at the state $(t_k + \delta_f, x(t_k + \delta_f))$ is false. If the boolean function g_b is never true on the given domain, the returned collection is empty. Some possibilities for the implementation of such a function are described in Subsection 2.4.4, typically using functions similar to the function *cover*.

Similarly, we assume that we have a function *detectGuards* that, given a triple $([\underline{t_k}, \overline{t_k}], dt_k, \varphi_k)$ as defined before and a collection of activated modes V_k, returns a collection of pairs $\Big(\Big[\underline{\delta_i}, \overline{\delta_i}\Big], X_i\Big)$ such that for all edge e from a mode $v \in V_k$ and for all pair $\Big(\Big[\underline{\delta_d}, \overline{\delta_d}\Big], X_d\Big)$ returned by the function *detect* applied on the triple $([\underline{t_k}, \overline{t_k}], dt_k, \varphi_k)$ and a boolean function encoding the predicate *guard*(e), there exists an index i such that $\Big[\underline{\delta_d}, \overline{\delta_d}\Big] \subset \Big[\underline{\delta_i}, \overline{\delta_i}\Big]$ and $X_d \subset X_i$. This function could be implemented as the union of the results of *detect* over all guards from a mode in V_k.

Finally, we assume that we have a function *refine* that, given a triple $([\underline{t_k}, \overline{t_k}], dt, \varphi)$ as defined before and a collection of activated modes V_k, returns a collection of pairs (dt_l, φ_l) such that their union is an over-approximation of the trajectories that stay in the current modes, *i.e.* for all time $t_k \in [\underline{t_k}, \overline{t_k}]$ and for all absolutely continuous function

x solution of the differential inclusion associated to the modes in V_k interpreted in the sense of Filippov such that $x(t_k) \in \varphi(0)$, if for all $\delta_x \in [0, dt]$ such that for all $\delta \in [0, \delta_x]$, there exists a mode $v \in V_k$ such that $inv(v)$ is satisfied by $\big(t_k + \delta, x(t_k + \delta)\big)$, then there exists an index l such that $\delta_x \in [0, dt_l]$ and for all $\delta \in [0, \delta_x]$, $x(t_k + \delta) \in \varphi_l(\delta)$. Typically, this function could be implemented using the function *detect* and some dichotomy on the parameters of φ.

Now, we have all the functions needed to define the base of the algorithm. The proposed algorithm takes as input an interval of possible initial times $[\underline{t_0}, \overline{t_0}]$ (usually reduced to the unique initial time t_0), a collection of possible initial modes $\{v_i\}_{i \in I_0}$, a set of possible initial states X_0 and a time-step $dt > 0$ and it returns a collection $R = \big\{\big([\underline{t_k}, \overline{t_k}], v_k, dt_k, \varphi_k\big)\big\}_{k \in [\![0, N]\!]}$:

1. Define a collection $C = \big\{\big([\underline{t_0}, \overline{t_0}], v_i, X_0\big) \mid i \in I_0\big\}$
2. Define an empty collection R to store the over-approximations
3. Until C is empty:
   (a) Remove a triple $\big([\underline{t_k}, \overline{t_k}], v_k, X_k\big)$ from C
   (b) Compute an over-approximation of the continuous dynamics, using the function *integrate*
   (c) Refine the over-approximation using the function *refine*
   (d) Append to R the local over-approximations
   (e) Detect all activated guards from the refined over-approximations using the function *detectGuards*
   (f) Compute the set-valued images for each activated transition
   (g) Append to the collection C all the pairs computed in the previous steps that do not reached the final time
4. Return the final collection R

This algorithm iterates until a collection C of local initial states is empty, *i.e.* it is a worklist algorithm. An iteration of the loop consists in a sequence of steps. First, we pick an element from the collection C. This element defines the possible intermediate initial times $[\underline{t_k}, \overline{t_k}]$, the intermediate mode v_k and the possible intermediate initial states X_k. Then, at step 3b, we apply the function *integrate* to it in order to compute an over-approximation of the continuous dynamics. At that point, we have a set-valued function φ defined on an interval $[0, dt]$ that over-approximates the local reachable set with respect to the time. However, its domain can be too large: the invariant associated to the current dynamics can be false at some instants, indicating that the over-approximated admissible trajectories are no more in the current mode. At step 3c, we refine this over-approximation using the function *refine* in order to remove of it most of the states that cannot be reached without a change of mode. This results in a collection of pairs (dt_i, φ_i) that over-approximates the set of trajectories in the current mode from X_k at

a local initial elapsed time $\delta = 0$. We can now update the output at step 3d appending all the tuples $(\lfloor \underline{t_k}, \overline{t_k} \rceil, v_k, dt_i, \varphi_i)$ to the collection R. At step 3e, we apply the function *detectGuards* in order to get a collection of pairs $\left(\left\lfloor \underline{\delta_i}, \overline{\delta_i} \right\rceil, X_i\right)$ of elapsed times and set of states that activate some guards on the refined over-approximations. At step 3f, we compute the images of all the sets X_i and time intervals $\lfloor \underline{t_k}, \overline{t_k} \rceil + \left\lfloor \underline{\delta_i}, \overline{\delta_i} \right\rceil$, obtained at the previous step, through the corresponding transitions, *i.e.* if X_i corresponds to the states that activate the guard of the transition e_i, from the mode v_k to the mode v_i, on an elapsed time interval $\left\lfloor \underline{\delta_i}, \overline{\delta_i} \right\rceil$, then we compute the set X_i' that contains all the images $jump(e_i)(t_k + \delta_i, x_i)$ with $t_k \in \lfloor \underline{t_k}, \overline{t_k} \rceil$, $\delta_i \in \left\lfloor \underline{\delta_i}, \overline{\delta_i} \right\rceil$ and $x_i \in X_i$. This is computed using guaranteed set-valued arithmetics, *e.g.* Taylor model arithmetic. Finally, at step 3g, for all pair (dt_i, φ_i) computed in step 3c such that $dt_i = dt$ and $\underline{t_k} + dt_i \leq T$, we append the triple $(\lfloor \underline{t_k}, \overline{t_k} \rceil + dt_i, v_k, \varphi_i(dt_i))$ to the collection C. This allows to compute the evolution of trajectories that stay in the current mode. Moreover, for all pair $\left(\left\lfloor \underline{\delta_i}, \overline{\delta_i} \right\rceil, X_i'\right)$ computed in the step 3e such that $\underline{t_k} + \underline{\delta_i} \leq T$, we append the triple $\left(\lfloor \underline{t_k}, \overline{t_k} \rceil + \left\lfloor \underline{\delta_i}, \overline{\delta_i} \right\rceil, v_i, X_i'\right)$ to the collection C for the next iterations of the loop. This allows to compute the evolution of trajectories that take a transition.

Notice that appending a triple to the collection C should be computed in a smart way in order to avoid an explosion of the number of pairs. Typically, we do not append any triple $(\lfloor \underline{t}, \overline{t} \rceil, v, X)$ such that for all pair (t, x) with $t \in \lfloor \underline{t}, \overline{t} \rceil$ and $x \in X$, there exists a triple $(\lfloor \underline{t_i}, \overline{t_i} \rceil, v_i, X_i)$ in C such that $t \in \lfloor \underline{t_i}, \overline{t_i} \rceil$, $v = v_i$ and $x \in X_i$. In other words, appending a triple that is already covered by the ones in C should result in no modifications on the collection C.

In the following subsection, we adapt this algorithm to the case of switched systems, in which no jumps can occur, interpreted in the sense of Filippov in order to allow admissible trajectories beyond Zeno events.

3.4.2 Switched systems without input

In the case of switched systems, all the information of the modes are contained in the states. So, we do not have to specify the modes associated to the intermediate times and states $(\lfloor \underline{t_k}, \overline{t_k} \rceil, X_k)$ in the collection C. Moreover, we notice that the step 3f is caducous, because the jumps are identity functions (*cf.* Subsection 2.1.2).

Consider a switched system as in Definition 4, *i.e.* a collection of dynamics $\{f_i\}_{i \in I}$ and a switching signal G, for which we want to compute an over-approximation of the reachable set with respect to the time over the interval $[t_0, T]$ with a set X_0 of possible initial states at time t_0.

The main algorithm described in the previous subsection is initialized with the pair (t_0, X_0). Then, the iterations of the loop are slightly adapted in order to detect and to handle sliding modes in a conservative way: given a pair $(\lfloor \underline{t_k}, \overline{t_k} \rceil, X_k)$ picked from the collection C,

1. We evaluated the function G over the pair of sets $(\lfloor \underline{t_k}, \overline{t_k} \rceil, X_k)$, using for example the Taylor model arithmetic (*cf.* Subsection 2.4.1), in order to obtain an over-

approximation of the union of the images through G of all the pairs (t, x) such that $t \in [\underline{t_k}, \overline{t_k}]$ and $x \in X_k$. Let $J \subset I$ be an over-approximation of the image of G. We deduce a collection of dynamics $\{f_j\}_{j\in J}$ that can be followed by the trajectories from a state of X_k at a time $t_k \in [\underline{t_k}, \overline{t_k}]$.

2. Now, we can compute the step 3b. Two cases can occur depending on the number of activated modes:

   (a) The collection of dynamics $\{f_j\}_{j\in J}$ is a singleton, *i.e.* $J = \{j_0\}$. Then, the set X_k is included in the interior of a mode and its evolution can be computed using classic methods as presented in the previous subsection. So, we call the function *integrate* on $([\underline{t_k}, \overline{t_k}], dt, f_{j_0})$ in order to obtain a set-valued function φ defined on $[0, dt]$.

   (b) At least two different dynamics belong to $\{f_j\}_{j\in J}$. Then, the set X_k contains states in different modes and a Zeno event may occur. Let j_0 be an index in J and let J_1 be the set of indices in J different of j_0, *i.e.* $J_1 = J \setminus \{j_0\}$. Let $\{\alpha_j\}_{j\in J_1}$ be a collection of Lebesgue-measurable uncertain functions that take values in $[0, 1]$. We define the local sliding mode in the sense of Filippov as the convex combination with coefficients $\{\alpha_j\}_{j\in J_1}$, *i.e.* $\dot{x}(t) = f^*(t, x(t))$ with

   $$f^* = (t, x) \mapsto \left(1 - \sum_{j\in J_1} \alpha_j(t)\right) f_{j_0}(t, x) + \sum_{j\in J_1} \alpha_j(t) f_j(t, x) \tag{3.76}$$

   We then apply the procedure described in the previous section (*cf.* Subsection 3.3.3) in order to obtain a set-valued function φ on $[0, dt]$.

   In both cases, we obtained a set-valued function φ on the elapsed time interval $[0, dt]$.

3. The rest of the body of the loop is mostly as defined in the previous subsection:

   step 3c: We apply the function *refine* to the computed over-approximation φ in order to get a collection of pairs (dt_i, φ_i) that defines a tighter over-approximation of the trajectories that do not take any transitions.

   step 3d: For all pair (dt_i, φ_i) computed in the previous step, we append the tuple $([\underline{t_k}, \overline{t_k}], J, dt_i, \varphi_i)$ to the collection R.

   step 3e: For all pair (dt_i, φ_i) computed in the step 3c, we compute a collection of pairs $\left(\left[\underline{\delta_l}, \overline{\delta_l}\right], X_l\right)$ using the function *detectGuards*.

   step 3g: For all pair $\left(\left[\underline{\delta_l}, \overline{\delta_l}\right], X_l\right)$ computed in the previous step, we append to the worklist C the pair $\left([\underline{t_k}, \overline{t_k}] + \left[\underline{\delta_l}, \overline{\delta_l}\right], X_l\right)$ such that $\underline{t_k} + \underline{\delta_l} \leq T$ and such that the set of activated modes is not included in the set of the ones composing the sliding mode, *i.e.* $G\left([\underline{t_k}, \overline{t_k}] + \left[\underline{\delta_l}, \overline{\delta_l}\right], X_l\right) \not\subset G([\underline{t_k}, \overline{t_k}], X_k)$.

Moreover and contrary to the base algorithm, for all pair (dt_i, φ_i) computed in the step 3c such that $\underline{t_k} + dt_i \leq T$ and $dt_i = dt$, we append to the worklist C all the pairs $\left(\left[\underline{t_j}, \overline{t_j}\right], X_j\right)$ returned by the function *cover* applied on $\left(\left[\underline{t_k}, \overline{t_k}\right] + dt_i, \varphi_i(dt_i)\right)$.

The main part of the algorithm that differs from the base (*cf.* Subsection 3.4.1) is the selection of the dynamics to handle during an iteration of the loop: the considered dynamics is the convex combination of the ones that are activated by some states of the current set X_k. If a unique dynamics is activated, then the convex combination is reduced to the activated dynamics without extra uncertainties (case 2a). Otherwise, a Zeno event may occur and the convex combination with extra time-varying uncertainties α_j allows to compute an over-approximation of all admissible trajectories, even considering sliding modes in the sense of Filippov (case 2b). The second difference is the way we update the worklist C during the step 3g. In order to limit the emergence of infinite loops due to instantaneous chattering effects, *i.e.* infinite many transitions taken in zero time, we do not append to C all the pairs returned by *detectGuard*, because this would always return some pairs containing a null elapsed time during a sliding mode, *i.e.* $\underline{\delta_i} = 0$. We only append the pairs that activate modes that do not belong to the sliding mode. Moreover, because our algorithm to handle sliding modes introduces extra uncertainties, we call the function *cover* during the step 3g in order to reduce the size of the over-approximations activating each mode after an iteration. Specifically, it allows us to limit the growth of the over-approximations in a sliding mode. We illustrate this point in the following Subsection 3.4.3.

Correctness of the result Now, assuming that the algorithm terminates, we prove by induction that this procedure computes an over-approximation of the reachable set of a given switched system. Consider an admissible trajectory $((\delta_\nu, I_\nu, x_\nu))_{\nu \in V}$ of the switched system: V denotes here a set of indices ν ($V = [\![0, N]\!]$, with $N \in \mathbb{N}$, or $V = \mathbb{N}$), δ_ν denotes the elapsed time since the previous event (triple with index $\nu - 1$ or the origin of the time if $\nu = 0$), I_ν denotes the set of activated modes (multiple indices in case of sliding modes, *cf.* Subsection 2.1.2), and x_ν denotes the value of the state. We want to prove that for all event $(\delta_\nu, I_\nu, x_\nu)$ of the admissible trajectory, there exists a tuple $\left(\left[\underline{t_k}, \overline{t_k}\right], I_k, dt_k, \varphi_k\right)$ in the collection R such that $I_\nu \subset I_k$ and there exists a time $t_k \in \left[\underline{t_k}, \overline{t_k}\right]$, a state $y_\nu \in \varphi_k(0)$, an elapsed time $\xi_\nu \in [0, dt_k]$ and an absolutely continuous function ρ_k such that $\Delta_\nu = t_k + \xi_\nu$, $\rho_k(0) = y_\nu$, $\rho_k(\xi_\nu) = x_\nu$, for all elapsed time $\delta \in [0, \xi_\nu]$, $\rho_k(\delta) \in \varphi_k(\delta)$, and for almost all elapsed time $\delta \in [0, \xi_\nu]$, the derivative $\dot{\rho}_k(\delta)$ of ρ_k at the elapsed time δ belongs to the convex hull of the dynamics of modes in I, *i.e.* $\dot{\rho}_k(\delta) \in \mathrm{CH}(\{f_i(t_k + \delta, \rho_k(\delta)) \mid i \in I\})$. We say that the event $(\delta_\nu, I_\nu, x_\nu)$ *belongs to the over-approximation* R and that the tuple $\left(\left[\underline{t_k}, \overline{t_k}\right], I_k, dt_k, \varphi_k\right)$ over-approximates the event $(\delta_\nu, I_\nu, x_\nu)$. Notice that ξ_ν may be equal to zero and, in that case, we have $\Delta_\nu = t_k$ and $x_\nu = y_\nu$.

By definition of the admissible trajectory, we have $\delta_0 \in \left[\underline{t_0}, \overline{t_0}\right]$ and $x_0 \in X_0$. The evaluation of G over the pair $\left(\left[\underline{t_0}, \overline{t_0}\right], X_0\right)$ using a guaranteed arithmetic on the rep-

resentation of the sets (*e.g.* Taylor models arithmetics) returns an over-approximation of the modes that are activated by the admissible trajectory at time $\Delta_0 = \delta_0$, *i.e.* $I_0 \subset G(\lfloor \underline{t_0}, \overline{t_0} \rceil, X_0)$. Moreover, by definition of the functions *integrate* and *refine*, the union of the set-valued functions φ_i evaluated at the initial elapsed time 0 covers the set X_0, *i.e.* $X_0 \subset \bigcup_i \varphi_i(0)$. So, there exists a tuple $(\lfloor \underline{t_k}, \overline{t_k} \rceil, I_k, dt_k, \varphi_k)$ in R that contains the initial event of the admissible trajectory, *i.e.* $\Delta_0 \in \lfloor \underline{t_k}, \overline{t_k} \rceil$, $I_0 \subset I_k$ and $x_0 \in \varphi_k(0)$.

Let $\nu \in V$ be a non-maximal index of the admissible trajectory, *i.e.* $\nu + 1$ belongs to V. We assume that the event $(\delta_\nu, I_\nu, x_\nu)$ belongs to the over-approximation R. We want to prove that the event $(\delta_{\nu+1}, I_{\nu+1}, x_{\nu+1})$ also belongs to R.

First, we assume that $\delta_{\nu+1} = 0$. Because of the definitions of admissible trajectories and switched systems, we have $x_{\nu+1} = x_\nu$. If $I_{\nu+1} \subset I_\nu$, then the event $(\delta_{\nu+1}, I_{\nu+1}, x_{\nu+1})$ belongs to R. Otherwise, if $I_{\nu+1}$ is not a subset of I_ν, then we notice that the function *detectGuards* ensures that a pair $(\left\lfloor \underline{\delta_i}, \overline{\delta_i} \right\rceil, X_i)$ is computed at the step 3e such that $x_{\nu+1} \in X_i$ and there exists an elapsed time $\xi \in \left\lfloor \underline{\delta_i}, \overline{\delta_i} \right\rceil$ such that $\Delta_{\nu+1} = t_k + \xi$. Because such a pair is appended to C during the step 3g, we are in a same configuration as the one with the initial event. So, if $\delta_{\nu+1} = 0$, then $(\delta_{\nu+1}, I_{\nu+1}, x_{\nu+1})$ belongs to R.

Now, we assume that $\delta_{\nu+1} > 0$. Let F_ν be the convex hull of the activated dynamics, *i.e.* for all pair $(t, x) \in \mathbb{R}_{\geq 0} \times \mathbb{R}^n$, $F_\nu(t, x) = \mathrm{CH}(\{f_i(t, x) \mid i \in I_\nu\})$. By definition of the admissible trajectory, $I_{\nu+1} = I_\nu$ and there exists an absolutely continuous function $\rho_{\nu+1}$ such that $x_\nu = \rho_{\nu+1}(0)$, $x_{\nu+1} = \rho_{\nu+1}(\delta_{\nu+1})$ and for almost all elapsed time $\delta \in [0, \delta_{\nu+1}]$, $\dot{\rho}_{\nu+1}(\delta) \in F_\nu(\Delta_\nu + \delta, \rho_{\nu+1}(\delta))$. The function *integrate* and the algorithm of the previous section (*cf.* Subsection 3.3.3) guarantee that, for all elapsed time $\delta \in [0, \delta_{\nu+1}]$, $\rho_{\nu+1}(\delta)$ belongs to $\varphi(\delta)$.

Consider a tuple $(\lfloor \underline{t_k}, \overline{t_k} \rceil, I_k, dt_k, \varphi_k)$ in the collection R that over-approximates the event $(\delta_\nu, I_\nu, x_\nu)$. So, there exists a function ρ_k and an elapsed time ξ_ν such that $\xi_\nu \in [0, \delta_\nu]$, $\rho_k(\xi_\nu) = x_\nu$ and for all $\delta \in [0, \xi_\nu]$, $\rho_k(\delta) \in \varphi_k(\delta)$. We define the function $\rho_{k,\nu+1}$ on the elapsed time interval $[0, \xi_\nu + \delta_{\nu+1}]$ as

$$\begin{cases} \forall \delta \in [0, \xi_\nu], & \rho_{k,\nu+1}(\delta) = \rho_k(\delta) \\ \forall \delta \in [\xi_\nu, \xi_\nu + \delta_{\nu+1}], & \rho_{k,\nu+1}(\delta) = \rho_{\nu+1}(\delta) \end{cases} \tag{3.77}$$

We have $\rho_{k,\nu+1}(0) \in \varphi_k(0)$, $\rho_{k,\nu+1}(\xi_\nu + \delta_{\nu+1}) = x_{\nu+1}$, $\rho_{k,\nu+1}$ is an absolutely continuous function and for almost all elapsed time $\delta \in [0, \xi_\nu + \delta_{\nu+1}]$, there exists an index $i \in I_\nu$ such that $\dot{\rho}_{k,\nu+1}(\delta) = f_i(t_k + \delta, \rho_{k,\nu+1}(\delta))$. If $\xi_\nu + \delta_{\nu+1} \leq dt$, by definition of the function *integrate*, of the algorithm in Subsection 3.3.3 and of the function *refine*, then there exist an index k_2 and a tuple $\left(\left\lfloor \underline{t_{k_2}}, \overline{t_{k_2}} \right\rceil, I_{k_2}, dt_{k_2}, \varphi_{k_2}\right)$ in R such that $I_{\nu+1} \subset I_{k_2}$ and for all elapsed time $\delta \in [0, \xi_\nu + \delta_{\nu+1}]$, $\rho_{k,\nu+1}(\delta) \in \varphi_{k_2}(\delta)$. Otherwise, if $\xi_\nu + \delta_{\nu+1} > dt$, then there exists a tuple $\left(\left\lfloor \underline{t_{k_2}}, \overline{t_{k_2}} \right\rceil, I_{k_2}, dt_{k_2}, \varphi_{k_2}\right)$ in R such that $I_{\nu+1} \subset I_{k_2}$, $dt_{k_2} = dt$ and for all elapsed time $\delta \in [0, dt_{k_2}]$, $\rho_{k,\nu+1}(\delta)$ belongs to $\varphi_{k_2}(\delta)$. So, the pair $\left(\left\lfloor \underline{t_{k_2}}, \overline{t_{k_2}} \right\rceil + dt, \varphi_{k_2}(dt)\right)$ was appended to C during an execution of the step 3g and there exists a tuple $\left(\left\lfloor \underline{t_{k_3}}, \overline{t_{k_3}} \right\rceil, I_{k_3}, dt_{k_3}, \varphi_{k_3}\right)$ in R such that $I_{\nu+1} \subset I_{k_3}$ and for all elapsed time $\delta \in [0, \min(\xi_\nu + \delta_{\nu+1} - dt, dt_{k_3})]$, $\rho_{k,\nu+1}(\delta + dt)$ belongs to $\varphi_{k_3}(\delta)$. Because there exists a

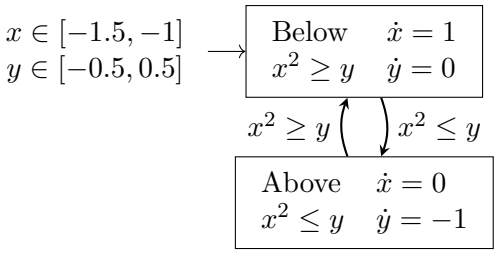
minimal integer $n \in \mathbb{N}$ such that $\xi_\nu + \delta_{\nu+1} - n\,dt \leq dt$, we deduce by induction that there exists a tuple $\left(\left[\underline{t_{k_n}}, \overline{t_{k_n}}\right], I_{k_n}, dt_{k_n}, \varphi_{k_n}\right)$ in R such that $I_{\nu+1} \subset I_{k_n}$ and for all elapsed time $\delta \in [0, \xi_\nu + \delta_{\nu+1} - n\,dt]$, $\rho_{k,\nu+1}(\delta + n\,dt) \in \varphi_{k_n}(\delta)$. So, the event $(\delta_\nu, I_\nu, x_\nu)$ belongs to R.

Because for all admissible trajectory, the event (δ_0, I_0, x_0) belongs to R and if an event $(\delta_\nu, I_\nu, x_\nu)$ belongs to R, then the event $(\delta_{\nu+1}, I_{\nu+1}, x_{\nu+1})$ also belongs to R, we proved by induction that, if the algorithm terminates, then the collection R is an over-approximation of the set of admissible trajectories.

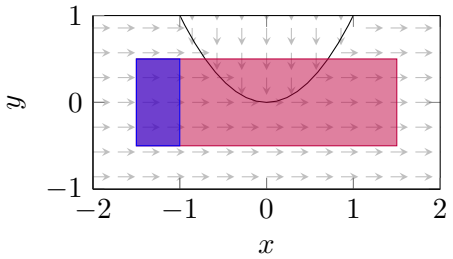
Possible improvement We use a constant time-step in the proposed algorithm. However, such a time-step may be simultaneously too small for the integration in a unique mode (case 2a) and too big for the integration in a sliding mode in the sense of Filippov (case 2b). So, we may consider different time-steps depending on the case: a relatively big time-step for simple dynamics in a unique activated mode and a small one for integration in a sliding mode. Those two time-steps may be adaptive in order to reach some tradoff between computation time and tightness of the returned over-approximation.

3.4.3 Application on the motivating example

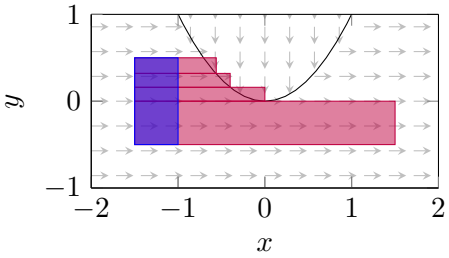
Consider the switched system presented in Figure 3.1:



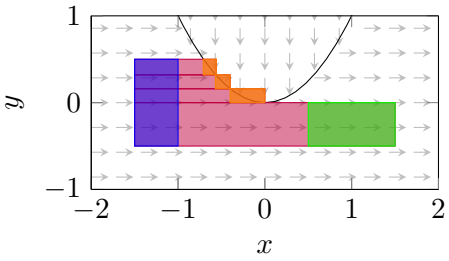
We illustrate some steps of the algorithm to compute an over-approximation. We only draw the computed sets in the plan representing y with respect to x, but the associated times should also be computed while they are not represented here.



From the initial set of states $X_0 = \{x \in [-1.5, -0.5] \wedge y \in [-0.5, 0.5]\}$ (in blue), we compute an *a priori* over-approximation (in purple) using the function *integrate*. This corresponds to the step 3b. At this step, all switches are ignored.

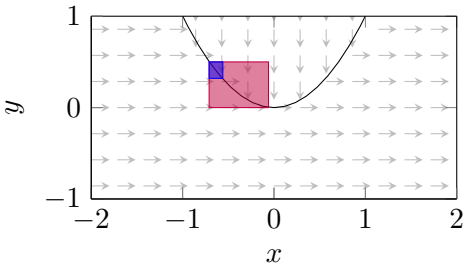


The step 3c consists in calling the function *refine* that returns a collection of set-valued functions (in purple) whose domains are reduced from the previous one in order to remove states that cannot be reached without switches.



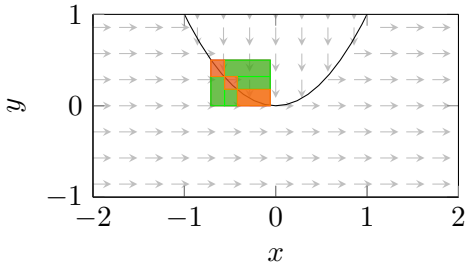
Then, we call the function *detectGuards* that returns over-approximations (in orange) of the sets that activate some guards. We also notice that the domain of one of the set-valued functions is equal to the whole time interval (the image of the last elapsed time is drawn in green).

We assume that the time associated to the last set (in green) is already bigger than the final time T of the analysis. So, it is not appended to the set C. However, we append all the orange sets with their associated time interval to the set C in order to be handled in the next iterations of the loop.



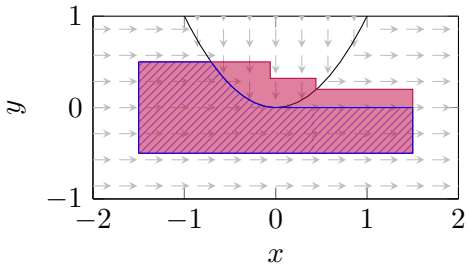
At the next iteration of the loop, we pick for example the top left orange set of states drawn in the previous picture. The set gathers states in different modes. So, the dynamics is interpreted as a differential inclusion in the sense of Filippov.

During the step 3b, we compute its evolution following the algorithm presented in the Section 3.3, Subsection 3.3.3. The resulting set, *i.e.* the range of the set-valued function φ, is drawn in purple. Due to the extra uncertainties introduced by the algorithm, the call to the function *refine* during the step 3c returns doubtless the same set-valued function φ on the same elapsed time interval. Moreover, the only guards that can be activated are the ones between modes composing the sliding mode. So, all the pairs computed using the function *detectGuards* during the step 3e are ignored. If they were not ignored, then a pair $\left([\underline{t_k}, \overline{t_k}] + \left[0, \overline{\delta_i}\right], X_i\right)$ would be appended to C after each iteration of the loop. So, there would always be some pair in C whose minimal time would be strictly smaller than the final time T. In that case, the algorithm would never terminate.

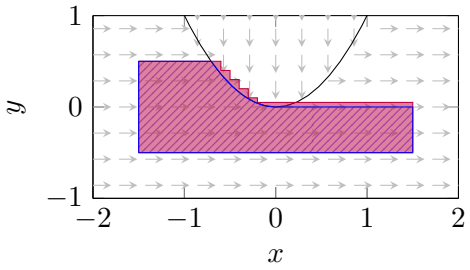


During the step 3g, the function *cover* returns a collection of pairs of time intervals and sets of states that cover $\varphi(dt)$. All those pairs are appended to the collection C. They are drawn in the figure depending on the number of modes that are activated: the sets in green activate only one mode each, while the sets in orange activate the two modes. Notice that without the function *cover*, only one set would be appended to C (the set $\varphi(dt)$) and it would always stay in the sliding mode while its size would growth until the associated minimal time becomes bigger than the final time T.

The following iterations of the loop until the collection C is empty are similar to the two presented here. Due to the uncertainties during the integration in the sliding mode, some sets can reach the interior of the mode Above, *i.e.* $y > x^2$, but none of the admissible trajectories can.



Finally, when the collection C is empty, the collection R defines an over-approximation that is represented in purple in the figure on the left. We also drawn in hashed blue the exact reachable set over the considered time interval $[0, T]$.



Notice that reducing the time-step dt reduces the growth of the over-approximation in the sliding mode. This results in a tighter over-approximation, but the execution of the algorithm needs more iterations of the loop in order to terminate.

In this chapter, we presented algorithms to compute over-approximations of the reachable sets of dynamical systems, systems that we called *separable systems with respect to time-varying uncertainties*, and of switched systems interpreted in the sense of Filippov in order to allow admissible trajectories beyond any Zeno events.

In the following chapter, we present an implementation of the algorithm presented in Subsection 3.3.3. We also compare the resulting over-approximations to the ones obtained using state-of-the-art tools.

Chapter 4

Experimental comparison of our method with state-of-the-art tools

In this chapter, we present a prototype implementing the algorithm to compute an over-approximation of the reachable set of a given separable system with respect to the time-varying uncertainties defined in the Subsection 3.3.3. We then present some comparisons on small examples of the results of our prototype with the state-of-the-art tools Flow*, CORA and Ariadne to them in order to evaluate the suitability of our prototype with respect to existing methods.

4.1 Implementation details

We start presenting some high-level implementation details of our prototype. It is written in C++11 and the real numbers are replaced by variables with the double precision floating-point type `double`. We do not handle the errors due to floating-point arithmetic and we assume that they are negligible compared to the errors introduced by the Taylor model arithmetic and the integration methods. So, our prototype is only a demonstrator and it should not be used as an industrial tool in order to prove safety properties of real models.

We do not present the source code, because the prototype is composed of over 2500 lines of codes (over 3000 counting the headers).

4.1.1 Interval arithmetic

In order to compute rough over-approximations, we use interval arithmetic which is also needed for implementing the Taylor model arithmetic (*cf.* Subsection 2.4.1). We implement our own library for the prototype to be able to modify any part of the code if needed. An interval is simply defined by its lower and upper bounds. So, we represent intervals as objects with two private fields `min` and `max` of type `double`.

Because we do not handle the errors due to floating-point arithmetic, all the elementary arithmetic operators (addition, substraction, multiplication and division) are

overloaded using the classic formula presented in the Subsection 2.4.1. In the case of the operators `sin` and `cos` (respectively sine and cosine), we follow the implementation proposed by the authors of CORA in [7]. We decide to implement `cos` following this method and to implement `sin` as a function $x \mapsto \cos(\frac{\pi}{2} - x)$.

The first step to compute the image of an interval $[\underline{x}, \overline{x}]$ through function cos is to check whether $\overline{x} - \underline{x} \geq 2\pi$. In that case, the image is equal to $[-1, 1]$, because of the periodicity of the cosine function. If $\overline{x} - \underline{x} < 2\pi$, then we compute the remainder of the Euclidean division of each bound of the interval by 2π, *i.e.* $\underline{y} \geq 0$ and $\overline{y} \geq 0$ such that

$$\underline{y} \equiv \underline{x} \mod 2\pi \qquad \text{and} \qquad \overline{y} \equiv \overline{x} \mod 2\pi$$

Then, we determine the domain to which belong $\underline{y}$ and $\overline{y}$ among $D_1 = [0, \pi[$, $D_2 = [\pi, 2\pi[$. Notice that the cosine function is monotonic on each of these domains. Finally, we return a new interval depending on different conditions:

$\underline{y} \in D_1$**:**
- $\overline{y} \in D_2$**:** return $\Big[-1, \max\Big(\cos(\underline{y}, \overline{y})\Big)\Big]$
- $\overline{y} \in D_1$**:**
  - $\underline{y} \leq \overline{y}$**:** return $\Big[\cos(\overline{y}), \cos(\underline{y})\Big]$
  - $\underline{y} > \overline{y}$**:** return $[-1, 1]$

$\underline{y} \in D_2$**:**
- $\overline{y} \in D_1$**:** return $\Big[\min\Big(\cos(\underline{y}, \overline{y})\Big), 1\Big]$
- $\overline{y} \in D_2$**:**
  - $\underline{y} \leq \overline{y}$**:** return $\Big[\cos(\underline{y}), \cos(\overline{y})\Big]$
  - $\underline{y} > \overline{y}$**:** return $[-1, 1]$

Notice that we could define the operator `sin` is a similar way, but it requires three different domains instead of two in the case of the operator `cos`. So, we expect our implementation of `sin` calling the one of `cos` to be slightly faster than computing a more complex disjunction of cases.

4.1.2 Taylor model arithmetic

Because the intervals do not allow to define dependencies between variables, we prefer to use Taylor models that are essentially a triple (p, I, D) where D is a tuple of intervals, *i.e.* if D is a tuple of n intervals, we have $D \subset \mathbb{R}^n$, p is a polynomial depending on variables in D, *i.e.* p defines a function from D to $\mathbb{R}$, and I is an interval. More details are given in Subsection 2.4.1.

In order to reduce the size of the data, we enforce all the domains D to be equal. With only this constraint, we still have to store the range of each variable, *i.e.* the components of D. Because the algorithm presented in Subsection 3.3.3 creates extra variables at each iteration and because each extra variable induces an extra dimension

of the domain, this could quickly result in a huge amount of data in order to store all the dimensions of the domain. So, we decide fix the range of all variables equal to the interval $[0, 1]$. A classic choice of such an interval is $[-1, 1]$ (*e.g.* [8, Section 2.2]) to exploit the interval arithmetic in order to compute enclosures of the Taylor models, but we use a Bernstein decomposition for that purpose, which requires to transform the ranges of the variables into $[0, 1]$. This can be easily done using an affine transformation: instead of introducing a new variable x with range $[\underline{x}, \overline{x}]$, we introduce a variable x_0 with range $[0, 1]$ and we replace the occurrences of x by the polynomial $\underline{x} + (\overline{x} - \underline{x})x_0$.

So, because we already implemented the interval arithmetic as presented in the previous subsection, we just have to implement a representation of polynomials in order to define a representation of Taylor models. Because a polynomial is a sum of monomials, we start implementing monomials. A monomial is defined by a coefficient multiplied by a finite product of variables. For example, if for all $k \in [\![1, n]\!]$, x_k is a variable of the considered polynomial, then for all monomial m, there exist a coefficient $\alpha \in \mathbb{R}$ and a collection of natural numbers $\{v_k\}_{k \in [\![1,n]\!]}$ such that

$$m = \alpha \prod_{k=1}^{n} x_k^{v_k}$$

If the maximal degree of the polynomials is much smaller than the number of variables, *e.g.* smaller than $\frac{n}{2}$ with n the number of variables, then for all monomial, most of the exponents i_k are null. Indeed, the degree of a monomial is equal to the sum of its exponents and the degree of a polynomial is equal to the maximal degree of its monomials. So, we decide to encode such exponents as a collection of pairs (k, v_k) where k is the index of a variable and v_k the non-null values of its exponent. If such a v_k is null, then it does not belong to the collection.

Now, we just need a representation of the variables that allows to create new ones during the computation. We decide to use natural numbers as indices of the variables and we use a static counter of the number of existing variables in order to generate a new index: if n variables are already defined and the first one has the index 0, then the index of an extra variable should be n and the static counter is incremented by one. We also consider that the variable with index 0 is the time variable used only during a step of integration.

To summarize, the different objects needed to implement Taylor models are defined as follow:

- an `Interval` is implemented according to the previous subsection
- an `Index` is a variable of type `int`
- a `Monomial` is a list of pairs of an `Index` and an exponent, which is represented by a variable of type `int`, *i.e.* a list of pairs `(Index,int)`
- a `Polynomial` is a list of `Monomial`
- a `TaylorModel` is a pair of a `Polynomial` and an `Interval`

Moreover, the order of the Taylor models, *i.e.* the maximal degree of the polynomials, is encoded as a static variable `order` of type `int`.

The operations on Taylor models are applied in a similar way as presented in Subsection 2.4.1, except for taking primitives with respect to the time that requires a slight modification in order to be valid. Indeed, because we scaled the domain of the time variable to $[0, 1]$, the primitive has also to be scaled. Consider a Taylor model (p, I) depending on a variable t on a domain $[\underline{t}, \overline{t}]$. Then, the primitive with respect to t (ignoring the maximal allowed degree) should be

$$\int (p, I)\, dt = \left(\int p\, dt, I \cdot (\overline{t} - \underline{t}) \right) \tag{4.1}$$

However, because we enforce the domain to be equal to $[0, 1]$, there exists a variable t_0 on domain $[0, 1]$ such that $t = \underline{t} + (\overline{t} - \underline{t})t_0$. The actual Taylor model is then (p_0, I) with $p_0(t_0) = p(\underline{t} + (\overline{t} - \underline{t})t_0) = p(t)$. We deduce that $dt = (\overline{t} - \underline{t})\, dt_0$ and

$$\int (p, I)\, dt = (\overline{t} - \underline{t}) \cdot \int (p_0, I)\, dt_0 \tag{4.2}$$

So, our `primitive` method on Taylor models takes as arguments the index of the variable with respect to which we want to compute the primitive and the associated domain. Because this method is only called on the time variable, the domain is typically $[0, dt]$ with dt the fixed time-step of the algorithm (parameter of our implementation).

In the previous paragraph, we ignored the limit on the order of the Taylor models. In practice, this constraint implies the existence of a polynomial p_e, that contains all the monomials of p whose degree is equal or bigger than the order of the Taylor models, and an over-approximation of its range has to be computed in order to update the remainder of the Taylor model during the execution of the method `primitive`. The classical method consists in evaluating the polynomial p_e using the interval arithmetic replacing each variable by its domain $[0, 1]$. To obtain a tighter enclosure, we may subdivide the domains of the variables [45, 8]. Instead, we decide to exploit the Bernstein decomposition of multivariate polynomials [68, 147].

The Bernstein coefficients can be computed based on the binomial coefficients: given a polynomial p, a multi-index $d = (d_0, \ldots, d_n)$ where d_k is the degree of the polynomial p with respect to the variable of index k, and a multi-index $i = (i_0, \ldots, i_n)$ such that $0 \leq i_k \leq d_k$ for all $k \in [\![0, n]\!]$.

$$b_i = \sum_{j_0=0}^{i_0} \cdots \sum_{j_n=0}^{i_n} \frac{\prod_{k=0}^{n} \binom{i_k}{j_k}}{\prod_{k=0}^{n} \binom{d_k}{j_k}} a_j \tag{4.3}$$

with for all integers a and b such that $a \geq b \geq 0$, $\binom{a}{b}$ is the binomial coefficients "a choose b" and a_j is the coefficient of the monomial whose degree is equal to j, *i.e.* the exponent of the variable of index k is j_k.

We compute such Bernstein coefficients naively following the formula 4.3 for each valid multi-index j. However, in order to speedup the computation, we implemented

some memoization of the binomial coefficients: the Pascal's triangle is stored in a static matrix and we compute it recursively until the desired coefficient is reached. Moreover, because we notice that the computation of polynomial enclosures is time-consuming and because we evaluate multiple times the enclosure of the same polynomials during the fixed-point computations (*cf.* algorithm in Subsection 3.3.3), we implemented memoization as a mapping of a polynomial to its enclosure over the domain equal to the unit box $[0,1]^{n+1}$, where $n+1$ is the number of variables.

Regularly during the execution, we have to reduce the number of variables of a Taylor model. The first method consists in only keeping the monomials with highest absolute value of coefficients. This is computed by the method `reduce` that sorts the list of monomials with respect to the absolute value of their coefficients, truncates the resulting list at some index and computes an enclosure of the polynomials defined by the remaining monomials. The second method consists in evaluating some of the variable over their entire domain. Consider a Taylor model (p, I) with p a polynomial of (multi-index) degree l such that

$$p = \sum_{i=0}^{l} a_i x^i \tag{4.4}$$

with i a multi-index from 0 to l, a_i the coefficient of the monomial of index i, and x^i the variables raised to the power i, *i.e.* the product of all variables x_k raised to the power i_k. Because for all variable, the domain is equal to $[0,1]$, as soon as the associated exponent is positive, its evaluation is equal to $[0,1]$. So, a partial evaluation of p consists in replacing each monomial by a pair of a monomial and a remainder such that the resulting monomial does not contain any occurrences of the variable that has to be evaluated. Consider a multi-index i as in 4.4 and an index k and let $\tilde{i}$ be a multi-index equal to i expect for the k-th component which is null, *i.e.* for all $j \neq k$, $\tilde{i}_j = i_j$ and $\tilde{i}_k = 0$. So, the partial evalution of the monomial $a_i x^i$ results in the inclusion

$$a_i x^i \in \frac{a_i}{2} x^{\tilde{i}} + \left[-\frac{a_i}{2}, \frac{a_i}{2}\right] \tag{4.5}$$

We can then obtain a reduced Taylor model $\left(\tilde{p}, \tilde{I}\right)$ with $\tilde{p}$ the polynomial defined as the sum of the monomials $\frac{a_i}{2} x^{\tilde{i}}$ and $\tilde{I}$ the remainder defined as the sum of the remainder I and an over-approximation of the range of the polynomial $p - \tilde{p}$, *i.e.* the sum of the intervals $\left[-\frac{a_i}{2}, \frac{a_i}{2}\right]$ such that $\tilde{i} \neq i$.

Finally, we consider that a Taylor model (p_1, I_1) is included into a Taylor model (p_2, I_2) only if $p_1 = p_2$ and $I_1 \subset I_2$. This does not respect the classic definition of the inclusion, *i.e.* for all $x_1 \in [0,1]^n$ and $v_1 \in I_1$, there exist $x_2 \in [0,1]^n$ and $v_2 \in I_2$ such that $p_1(x_1) + v_1 = p_2(x_2) + v_2$, but it is correct during the fixed-point computation, when we already checked that $p_1 = p_2$.

4.1.3 Separable system with respect to the measurable uncertainties

One of the inputs of our algorithm is a *separable system with respect to the measurable uncertainties*, or simply *separable system*. As presented in Definition 11, a separable

system is fully specified by

- a time interval $[t_0, t_1] \subset \mathbb{R}$
- a set of possible states $X \subset \mathbb{R}^n$
- a vector x_0 in a bounded set $X_0 \subset \mathbb{R}^n$
- a Lebesgue-integrable function u from $[t_0, t_1]$ to $\mathbb{R}^m$
- a continuous function g from $\mathbb{R}^m$ to $\mathbb{R}^{n \times k}$
- a continuous function h from $[t_0, t_1] \times X$ to $\mathbb{R}^k$

We assume without loss of generality that $t_0 = 0$, because the origin of the time can be translated: for all function f from $[t_0, t_1]$ to a set V, we can use the translated function f_0 from $[0, t_1 - t_0]$ to V defined for all time $t \in [0, t_1 - t_0]$ as $f_0(t) = f(t + t_0)$. Moreover, the time interval $[0, T]$ on which to compute the reachability analysis is another input of our algorithm, so, we do not specify it for each separable system.

The set of possible states is assumed to be equal to the whole space, *i.e.* $X = \mathbb{R}^n$, and the range of the Lebesgue-integrable uncertain function u is assumed to be bounded, *i.e.* $u : [0, T] \to U$ with $U \subset \mathbb{R}^m$ bounded. This last hypothesis ensures that the range of $g \circ u$ is bounded.

So, because we want to compute an over-approximation of the reachable set for all possible initial states and Lebesgue-integrable functions, we just have to specify X_0, U, g and h. While we would have been able to define X_0 and U as arrays (`std::vector`) of Taylor models, *i.e.* one Taylor model encoding the domain of each variable, it is much more convenient to define them as boxes, *i.e.* arrays of intervals. Indeed, it enforces no dependencies between the variables and we can easily use those domains in our implementation of Taylor model arithmetic. The function g and h are implemented using the header `functional` of the C++11 standard library: the function g is a matrix of functions that take an array of Taylor models (the uncertainties) and return a Taylor model (a component of the matrix), and the function f is an array of functions that take a pair of a Taylor model (the time) and an array of Taylor models (the states) and return a Taylor model (a component of the vector).

Finally, we associate to each separable system a name (`std::string`) and an array of names of the variables (`std::vector<std::string>`) such that the name of the k-th variable is the k-th name in the array. Morevover, because we expect some modularity in our prototype, all the separable systems are implemented as classes derived from an abstract one. So, the solver can use polymorphism in order to handle any separable system.

4.1.4 Main solver

The solver that implements the algorithm presented in the Subsection 3.3.3 is an object parameterized by a pointer to a separable system as presented in the previous subsection, a time-step $dt > 0$, a final time T_{end} and a maximal order *order* of the Taylor

models. It starts initializing submodules such as integrators implementing the Picard operator or contractors implementing the fixed-point computation using the integrators. Both exist in two versions: one for the computation of the rough over-approximation (classic Picard operator over the whole time interval) and another for the computation using the decomposition of h as difference of non-negative functions. It allows for example to precompute the image of U through the function g used in the first version. The contractors are also initialised with some threshold to limit the duration of the computation of fixed-points: we consider that a fixed-point is almost reached and that the contraction should stop if the polynomial part of a Taylor model and the one of its image through the Picard operator are equal, the remainder of its image is included in its remainder and the relative difference of their widths is smaller than the given threshold, *i.e.* $\frac{w_1 - w_2}{w_1} \leq \mathit{threshold}$ where w_1 is the width of the remainder of the first Taylor model and w_2 the width of its image. If the width of the remainder of the first Taylor model is null, then the relative difference is defined as equal to zero. The contractors also enlarge the input, *i.e.* they enlarge the interval remainders, until it is contracted by the Picard operator before computing the contraction until reaching an almost fixed-point.

Then, the solver iterates a main loop on a state x_i and a time t_i until t_i becomes bigger than the final time T_{end}.

First, it computes a rough over-approximation over the time interval $[t_i, t_i + dt]$ using Taylor models setting the maximal order to 0, thus the Taylor models are reduced to translated intervals, *i.e.* a constant polynomial part and an interval remainder. It starts reducing the Taylor models x_i with a maximal order equal to 0, *i.e.* as a box over-approximation. Then, it simply calls the first contractor and returns its result. It also restore the value of the maximal order of Taylor models for the next computations.

Second, using the rough over-approximation, it computes a valid decomposition of the function h into a difference of non-negative functions. We use again the header `functional` of the C++11 standard library in order to define an object that, given a time and a state, computes only once the image of h and returns the images of $\lambda^+ \circ h$ and $\lambda^- \circ h$ as presented in Subsection 3.3.2.

Third, using the decomposition and after updating the second integrator with it, it computes a polynomial expansion of the solutions of the system over the time interval $[t_i, t_i + dt]$. This is done calling the second integrator (Picard operator of the auxiliary system with the given decomposition) until the polynomial part of the Taylor models reaches a fixed-point. This terminates in a finite number of iterations as presented in [45]. To speedup this process, we limit the number of monomials using a truncation of polynomials based on a fixed threshold after each application of the Picard operator.

Fourth, it computes a valid remainder, *i.e.* it enlarges the interval remainders of the Taylor models until they are contracted. At that point, the polynomial parts are fixed due to the previous step, and when a contraction is detected, it is contracted until it reaches an almost fixed-point, defined by the threshold of the second contractor.

Finally, it appends the obtained Taylor models, the time t_i and the time-step dt to a collection (implemented as a `std::vector` of C++11 structures) that will be returned at the end of the computation. It also updates the time t_i to the value $t_i + dt$ and the

set of state x_i to the value of the Taylor models evaluated at time $t_i + dt$, *i.e.* partially evaluating the 0-th variable t to the value 1 (end of its domain). In order to avoid an explosion of the number of variables and so, of monomials, we reduce the obtained Taylor models x_i partially evaluating all the uncertain parameters, that are introduced for the Picard operator of the auxiliary system, over their whole domain. It allows us to reuse the same variables for the uncertainties during the next iterations of the loop. We also reduce the Taylor models with respect to a threshold on the maximum number of monomials using a simple truncation of the polynomial as presented in the previous subsection. Finally, to keep a maximum of the dependencies between the state variables during the next iterations, we create an extra variable to replace each remainder. For example, given a reduced Taylor model $(p, [a, b])$, we replace it by the Taylor model $(p + a + (b - a)x_{\nu+1}, [0, 0])$ where $x_{\nu+1}$ is a fresh new variable. So, at each iteration of the loop, the number of defined variables is increased by the dimension of the state of the system. However, the number of monomials used to define the over-approximation x_i is bounded by $(M + 1)n$ where M is the threshold used during the reduction of the number of monomials and n is the dimension of the state.

4.1.5 Experimentations using our prototype

As described in the previous subsections, our solver is parameterized by a separable system, a time-step `dt`, a final time `Tend`, a maximal order `order` of the Taylor models and some thresholds. In addition to these parameters, we fix to 50 the maximum number of monomials after the application of the Picard operator of the auxiliary system, and to 3 the one at the end of each iteration of the loop of the main solver (*cf.* Subsection 4.1.4). This last threshold is very low and implies a potentially huge loss of dependencies between the state variables in case of high-dimensional systems. However, this is a reasonable number for the examples with a small number of state variables that we consider in the rest of this chapter.

The experiments are defined as functions that construct the object corresponding to the desired separable system, set the parameters `dt`, `Tend` and `order`, execute the solver, and print the boundaries of the over-approximation of each state variable on each time interval in a CSV file naming after the name of the separable system. All the experiments are hard-coded, *i.e.* written inside a C++11 source file and compiled with the prototype, and the main function executes one of them depending on the arguments of the program. So, the user can define multiple experiments and execute them independently with only compiling the prototype once.

4.2 Experiments

In this section, we give details about the examples on which we will evaluate the suitability of our method to compute tight over-approximations in presence of Lebesgue-measurable uncertainties.

Because some tools, to which we compare the results of our prototype, are only

able to handle respectively continuous and Riemann-integrable uncertainties , we choose simple examples for which we are able to compute the exact reachable set with respect to the time or, at least, its exact projection for each state variable. So, we can check that even if our prototype returns a tigher over-approximation than the other tools, it is still a valid over-approximation and not an error of the implementation.

4.2.1 Experiment: Simple

The first experiment, which we call "Simple", corresponds to the example presented in Subsection 3.3.4:

$$\begin{cases} \forall t \in [0, 0.2],\ \dot{x}(t) = (0.1 - t) \cdot u(t) \\ \forall t \in [0, 0.2],\ u(t) \in [-1, 1] \\ x(0) = 0 \end{cases} \tag{4.6}$$

This example illustrates the need to handle time-varying uncertainties as such and not as constant ones (*cf.* Subsection 3.3.4).

The experiment is performed using a single step of integration over the time interval $[0, 0.2]$, *i.e.* with parameters $T_{end} = dt = 0.2$.

The exact reachable set $R_x(t)$ of x with respect to the time is

$$\begin{cases} \forall t \in [0, 0.1],\ R_x(t) = [0.5t^2 - 0.1t, 0.1t - 0.5t^2] \\ \forall t \in [0.1, 0.2],\ R_x(t) = [0.1t - 0.5t^2 - 0.01, 0.5t^2 - 0.1t + 0.01] \end{cases} \tag{4.7}$$

4.2.2 Experiment: Exponential

This experiment, which we call "Exponential", corresponds to a classic linear example of decreasing exponential dynamics with an uncertain time-varying coefficient:

$$\begin{cases} \forall t \in [0, 5],\ \dot{x}(t) = -u(t)x(t) \\ \forall t \in [0, 5],\ u(t) \in [1, 2] \\ x(0) \in [1, 1.1] \end{cases} \tag{4.8}$$

The experiment is performed using a constant time-step $dt = 0.05$.

The minimal (*resp.* maximal) value of x is reached when its derivative is always minimal (*resp.* maximal). So, the exact reachable set $R_x(t)$ of x with respect to the time is

$$\forall t \in [0, 5],\ R_x(t) = \left[\mathrm{e}^{-2t}, 1.1\mathrm{e}^{-t}\right] \tag{4.9}$$

4.2.3 Experiment: NonLinear

This experiment, which we call "NonLinear", is a slight variation of the decreasing exponential dynamics with an uncertain time-varying coefficient and an artificial nonlinearity

via an extra state variable:

$$\begin{cases} \forall t \in [0,5],\ \dot{x}(t) = -x(t) - x(t)y(t)u(t) \\ \forall t \in [0,5],\ \dot{y}(t) = -y(t) \\ \forall t \in [0,5],\ u(t) \in [-1,1] \\ x(0) = 1 \\ y(0) = 2 \end{cases} \tag{4.10}$$

The experiment is performed using a constant time-step $dt = 0.05$.

The dynamics of y is independent of any uncertainty, so, for all $t \in [0,5]$, $y(t) = 2\mathrm{e}^{-t}$. The dynamics of x becomes $\dot{x}(t) = -x(t) - 2\mathrm{e}^{-t}u(t)x(t)$. The minimal (*resp.* maximal) value of x is reached when its derivative is always minimal (*resp.* maximal). So, the exact reachable sets $R_x(t)$ and $R_y(t)$ of x and y with respect to the time are

$$\begin{cases} \forall t \in [0,5],\ R_x(t) = \left[\mathrm{e}^{2(\mathrm{e}^{-t}-1)-t}, \mathrm{e}^{2(1-\mathrm{e}^{-t})-t}\right] \\ \forall t \in [0,5],\ R_y(t) = \left\{2\mathrm{e}^{-t}\right\} \end{cases} \tag{4.11}$$

4.2.4 Experiment: SimpleSwitching

This experiment, which we call "SimpleSwitching", is a slight variation of the decreasing exponential dynamics with an uncertain time-varying setpoint:

$$\begin{cases} \forall t \in [0,20],\ \dot{x}(t) = u(t) - x(t) \\ \forall t \in [0,20],\ u(t) \in [0,1] \\ x(0) = 3 \end{cases} \tag{4.12}$$

This example is linear, but it allows multiple equilibrium points.

The experiment is performed using a constant time-step $dt = 0.1$.

The minimal (*resp.* maximal) value of x is reached when its derivative is always minimal (*resp.* maximal). So, the exact reachable set $R_x(t)$ of x with respect to the time is

$$\forall t \in [0,20],\ R_x(t) = \left[4\mathrm{e}^{-t} - 1, 2\mathrm{e}^{-t} + 1\right] \tag{4.13}$$

4.2.5 Experiment: DubinsCar

This experiment, which we call "DubinsCar", corresponds to a slight variation of the Dubins' car model that is a classic kinematic model of a car [127, 146, 62]. The steering velocity and the velocity of the car are controlled by two independent inputs:

$$\begin{cases} \forall t \in [0,1],\ \dot{x}(t) = u_1(t)\cos(\theta(t)) \\ \forall t \in [0,1],\ \dot{y}(t) = u_1(t)\sin(\theta(t)) \\ \forall t \in [0,1],\ \dot{\theta}(t) = u_2(t) \\ \forall t \in [0,1],\ u_1(t) \in [0.9,1] \\ \forall t \in [0,1],\ u_2(t) \in [0,1] \\ x(0) = 0 \\ y(0) = 0 \\ \theta(0) = 0 \end{cases} \tag{4.14}$$

The state variables x and y are the Cartesian coordinates of the car in the plan, while the state variable θ is the steering of the car. The time-varying uncertainty u_1 is the velocity of the car, while the time-varying uncertainty u_2 is its steering velocity.

The experiment is performed using a constant time-step $dt = 0.01$.

The minimal (*resp.* maximal) value of θ is reached when its derivative is always minimal (*resp.* maximal). So, for all time $t \in [0, 1]$, $\theta(t) \in [0, t]$. Similarly, using the monotonicity of sine and cosine, we deduce the reachable sets of the state variables with respect to the time:

$$\begin{cases} \forall t \in [0,1],\ R_x(t) = [0.9\cos(t), 1] \\ \forall t \in [0,1],\ R_y(t) = [0, \sin(t)] \\ \forall t \in [0,1],\ R_\theta(t) = [0, t] \end{cases} \tag{4.15}$$

4.3 State-of-the-art tools and parameters

In this section, we briefly present the specificities of the different classic state-of-the-art tools and the parameters that we selected to perform the experimentations. We compare our results to the classic tools which are able to compute reachable sets of nonlinear hybrid systems.

We tried to use similar parameters when those are available. All the tools allow to set a fixed time-step, but, as far as we know, the reachability function in CORA does not use Taylor models, so it is impossible to fix a constant maximal order of them.

4.3.1 Flow*

Flow*[1] [47] is a well-known tool able to compute over-approximations of the reachable sets of hybrid automata. Here, we are only interested in its ability of computing reachable sets of initial value problems with uncertainties.

The method of integration [46, 48] used by this tool is similar to ours, *i.e.* using contraction of Taylor models, but time-varying uncertainties are handled differently. In Flow*, the time-varying uncertainties are defined replacing their occurrences in the differential equations by their range. In that case, they are interpreted as interval coefficients of the computed polynomials [45]. Because we introduce extra variables to handle such uncertainties, we expect that our method produces tighter over-approximations than the ones computed by Flow*.

The parameters are set following the advice given in the manual [46]. For all the examples, we set the precondition method to the QR method (`QR precondition`), the drawn representation of the over-approximations as octagons via the utility Gnuplot[2] (`gnuplot octagon`), a cutoff threshold to 10^{-10} (*i.e.* all monomial whose range is included in $[-10^{-10}, 10^{-10}]$ is appended to the remainder in order to reduce the complexity of Taylor models) and a precision for the MPFR library to 53.

[1] `https://flowstar.org/`

[2] `http://www.gnuplot.info/`

Experiment	TM order	remainder estimation	integration scheme
Simple	5	0.076	`poly ode 1`
Exponential	5	0.033	`poly ode 1`
NonLinear	5	0.1444	`poly ode 1`
SimpleSwitching	5	0.056	`poly ode 1`
DubinsCar	5	0.007	`nonpoly ode`

Table 4.1 – Parameters for Flow*

We fix the maximal order of Taylor models for each example to 5, as for the other tools. Moreover, we use different remainder estimations for each example, *i.e.* different interval remainders that have to be contracted after every step of integration. Finally, we use different ODE specifications depending on the operators used in the differential equations and the order of Taylor models: for polynomial differential equations, we use `poly ode 1` with an order of Taylor models smaller or equal to 5, and for non-polynomial differential equations, we use `nonpoly ode`. This last parameter fixes how the polynomial expansion of the solution is computed [45, Section 3.3.2]. We summarize these parameters in Table 4.1. We tried to set the remainder estimation at the lowest possible level with the considered time-step. The estimation of the lowest admissible value was computed by dichotomy. We also tried to increase the order of Taylor models without detecting improvement of the computed over-approximation.

Notice that we had to add an extra state variable t to each example in order to return the over-approximation of the x variable with respect to the time. The differential equation satisfied by t is $\dot{t}(t) = 1$ and the initial condition is $t(0) = 0$. So, the systems used in Flow* have a dimension higher by one than the systems used in the other tools.

4.3.2 CORA

CORA[3] [2, 9] is a tool-box for reachability analysis of hybrid systems implemented in Matlab. In this work, we used the version Release 2020 while the last version (Release 2021) was released on December 10, 2021. Regarding our use of the tool-box, the main difference between the two versions is the auto-tuning of the reachability analysis following the paper [145] or [10, Section 1.1]. However, because we want to enforce similar parameters for all the tools, in particular the same time-steps, we could not use such a method.

Because it is a tool-box, a lot of choices are available for computing the reachability analysis. First, it seems impossible to use Taylor models to represent reachable sets [9, Section 2.2.3.1], while they are implemented for bounding functions over a given domain [9, Section 2.2.1]. Following the manual providing with the tool about the reachability analysis function, we choose to use Zonotopes [9, Section 2.2.1.1] to represent sets. We also tried to use Polynomial Zonotopes [9, Section 2.2.1.5], which are similar to Taylor models [93] and supported by the reachability function, but we did not obtain any

[3] `https://tumcps.github.io/CORA/`

Experiment	.alg	.tens0	.taylT	.zono0	.error0	.inter0
Simple	'poly'	3	4	50	20	50
Exponential	'lin'	2	10	50	10	50
NonLinear	'poly'	3	4	50	20	50
SimpleSwitching	'poly'	3	4	50	20	50
DubinsCar	'poly'	3	5	50	1	50

Table 4.2 – Parameters for CORA

improvement of the computed over-approximation with our parameters.

Except for the parameters corresponding to the definitions of the experiments as presented in the previous section, the selected parameters are summarized in Table 4.2. We tried to select the ones that return the tightest over-approximation without exceeding a few minutes of computation following the recommendations in [9, Section 4.2.5.1]. First, we decide to use a conservative polynomialization algorithm [4], except for the "Exponential" experiment, for which we use a conservative linearization [14]. This is selected by setting the option `.alg` to `'poly'` or `'lin'`. The two algorithms compute a differential inclusion with additive uncertainties based on a Taylor expansion of the right-hand side of the ordinary differential equations. The number of terms of this expansion is set by the option `.tensorOrder` (`.tens0` in Table 4.2), whose recommended values are 2 or 3. Similarly, the option `.taylorTerms` (`.taylT` in Table 4.2) specifies the number of terms of the Taylor expansion of the matrix exponential of the linearized system [3, Eq. (3.2)]. Note that increasing this value may result in a wider over-approximation due to set-valued computations. The options `.zonotopeOrder`, `.errorOrder` and `.intermediateOrder` (respectively `.zono0`, `.error0` and `.inter0` in Table 4.2), define respectively the upper bound of the order of the zonotope (*i.e.* the ratio between their degree and the dimension of the states), the order to which zonotopes are reduced before linearization or polynomialization and the upper bound of the order during the execution of the algorithms. The other parameters are left to their default values.

4.3.3 Ariadne

Ariadne is the only of the three tools that is designed to handle uncertainties that are not integrable in the sense of Riemann. It implements the methods presented in [72]. However, this functionality is still experimental, in development, and the results are not guaranteed. As suggested by the developpers of the tool, we use the implementation of Ariadne available on the branch `dynamics-di#198` of the Git repository[4].

The used method consists in computing an over-approximation of the system considering only a subset of the time-varying uncertainties depending on parameters and then enlarging the remainder of the obtained Taylor model by an upper bound of the error between the two sets of solutions[72, Section 4].

[4] `https://github.com/ariadne-cps/ariadne/tree/dynamics-di%23198` (December 2021)

Experiment	sweep threshold	maximal error
Simple	0.02	0.1
Exponential	10^{-6}	10^{-4}
NonLinear	$5 \cdot 10^{-6}$	10^{-3}
SimpleSwitching	10^{-20}	10^{-3}
DubinsCar	0.01	10^{-3}

Table 4.3 – Parameters for Ariadne

The first parameter is the subset of considered uncertainties. Five of them are implemented in the tool: the singleton of null functions (`ZeroApproximation`), the set of constant uncertainties of the form $u(t) = a$ (`ConstantApproximation`), the one of affine uncertainties of the form $u(t) = at + b$ (`AffineApproximation`), the one of sinusoidal uncertainties of the form $u(t) = a + b\sin(\gamma t)$ with $\gamma = 4.1632$ (`SinusoidalApproximation`) and, finally, the one of uncertainties that are piecewise constant on half of the time-step (`PiecewiseApproximation`). The tool failed to compute an over-approximation on our examples, throwing an exception during computation if we allow the use of `AffineApproximation` and `SinusoidalApproximation`. So, we only allow the use of the constant null approximations `ZeroApproximation`, the constant ones `ConstantApproximation` and the piecewise constant ones `PiecewiseApproximation`.

As for Flow* and for our prototype, we set the order of the Taylor models to 5. This is specified in Ariadne setting the arguments `minimum_temporal_order` and `maximum_temporal_order` of the integrator `TaylorPicardIntegrator`. Notice that this order is the order with respect to the time variable, whereas Flow* and our prototype use a "global order", *i.e.* the maximal degree of the monomials.

In order to limit the number of monomials of the Taylor models, we have to set the "sweep threshold" [72, Section 4] *sweep*: if the range of a monomial is included in $[-sweep, sweep]$, then this monomial is removed from the polynomial part and its range is added to the remainder. While a smaller value of this parameter implies a higher precision, it also implies a higher computation time. So, we tried to select values of it by dichotomy in order the enforce the computation to terminates in only a few minutes.

Because the method in Ariadne implements an algorithm with adaptive temporal order of Taylor models, we also have to define a threshold on the width of the computed error at each step. While this should not change the result with the same minimal and maximal orders, we tried to set it to its lowest admissible value. This minimal values were deduced by dichotomy.

Finally, the other parameters are left to their default values.

4.4 Comparison of the results

4.4.1 Method of comparison

In order to compare the tightness of the computed over-approximations and in addition to the visual representations, we compare the areas of the over-approximations of the state variable x with respect to time. The computation of these areas is performed by a script in PYTHON3 using the returned over-approximations: a script GNUPLOT for FLOW* and ARIADNE or a CSV file for CORA and our prototype. In the case of FLOW* and ARIADNE, the scripts already define the polygons as a collection of their vertices, while the CSV files for CORA and our prototype define the boundaries of each variable over each time interval and we use a script written in PYTHON3 to convert them into a collection of vertices that define boxes over-approximating the values of the variable x with respect to time.

We chose the over-approximation of the state variable x with respect to time as the criterium of comparison for four reasons: 1) all the considered systems have a state variable x; 2) the area of the exact reachable set of x with respect to the time can easily be computed by hand for each of the examples; 3) the tightest over-approximation, *i.e.* the exact reachable set, is the one that has the smallest area; and 4) the over-approximations can easily be plotted in order to compare them visually. However, this criterium does not catch any information about the dependencies between state variables. For example, consider the two over-approximations R_1 and R_2 of the reachable set of a model composed of two variables x and y such that, for all time $t \in [0, 1]$,

$$R_1(t) = \left\{ \begin{pmatrix} x(t) = a \\ y(t) = a \end{pmatrix} \;\middle|\; a \in [0, t] \right\} \quad \text{and} \quad R_2(t) = \left\{ \begin{pmatrix} x(t) = a \\ y(t) = b \end{pmatrix} \;\middle|\; a \in [0, t] \wedge b \in [0, t] \right\}$$

In both cases, $x(t)$ belongs to $[0, t]$ and the criterium of comparison would return the same value for R_1 and R_2 over the time interval $[0, 1]$ but the over-approximation R_1 is indeed tighter than R_2 as illustrated in Figure 4.1 at time $t = 1$.

4.4.2 Results and discussion

The areas of the over-approximations of the state variable x with respect to time for each example and for each tool are gathered in Table 4.4. Some of the approximations computed using ARIADNE are not over-approximations as illustrated in Figure 4.2 where no ranges are defined for some times, probably due to a bug in the implementation. In these cases, the computed areas are colored in red in Table 4.4 and they cannot properly be compared with the others.

We notice that for all the experiments, our prototype returns a tighter over-approximation than the other tools, even though it considers a broader class of uncertainties than CORA (Lebesgue-integrable uncertainties instead of continuous or Riemann-integrable ones). These experiments illustrate that our proposed algorithm is suitable to compute over-approximations of continuous systems with Lebesgue-measurable time-varying uncertainties.

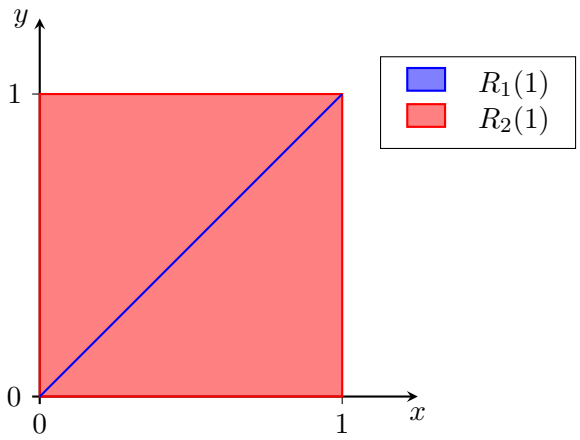
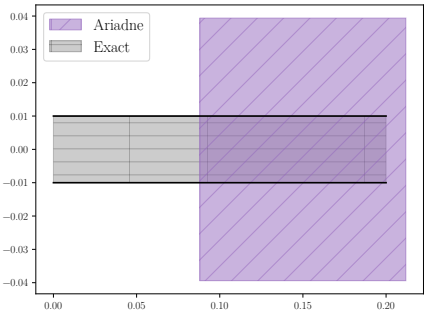


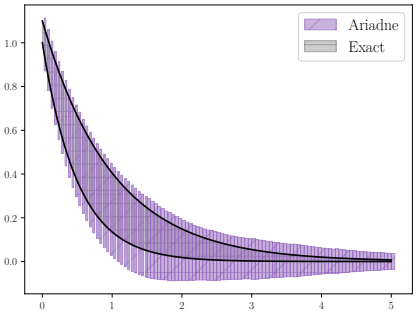
Figure 4.1 – Comparison of $R_1(1)$ and $R_2(1)$

	exact	**prototype**	**Flow***	**CORA**	**Ariadne**
Simple	0.004000	**0.008000**	0.024000	0.024000	0.009754
Exponential	0.592611	**0.840463**	1.296397	1.056670	0.924948
NonLinear	3.576489	**4.865639**	7.734936	5.955817	11.150959
SimpleSwitching	19.000000	**19.249388**	26.794954	23.325168	21.270921
DubinsCar	0.086272	**0.098562**	0.114053	0.114263	0.060890

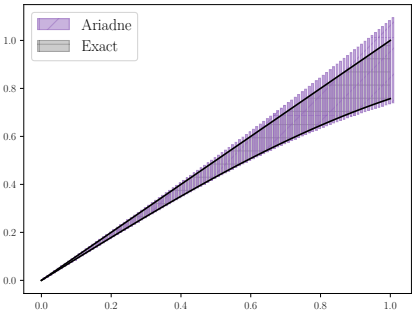
Table 4.4 – Areas of the over-approximations of the state variable x with respect to the time for each tool and experiment



(a) Simple



(b) Exponential



(c) DubinsCar

Figure 4.2 – x with respect to the time using ARIADNE

The over-approximations of x with respect to time are drawn in Figure 4.4. We notice that the over-approximations computed by our prototype have similar shapes as the ones computed by the other tools. Moreover, our over-approximations are always included in the others. In particular, our prototype returns a much tighter over-approximation than the other tools in the case of the experiment "SimpleSwitching", which is very close to the exact reachable set.

The experiments that we performed have a small number of state variables and a relatively wide range of the time-varying uncertainties. However, the state-of-the-art tools mainly focus on systems with small ranges of time-varying uncertainties or disturbances (*e.g.* [72, Section 5.3] or [14, Section VII]). So, it is likely that our method could be more suitable for reachability analysis of systems with large time-varying uncertainties, in particular when these uncertainties are not additive as in the case of an emerging sliding mode presented in the Subsection 3.1.

The computation times of each tool are summarized in Table 4.5. The times associated to the tool CORA are missing, because they were not recorded during our experiments. We notice that the tool Flow* is much faster than the others. It suggests that it could be relevant to use this tool with a much smaller time-step for those experiments in order to obtain a tighter over-approximation. Indeed, we tried to increase the order of Taylor models without detecting improvement of the results. We also notice that our prototype is faster than the tool Ariadne (which is the only one of the three tools that handles Lebesgue-integrable uncertainties) while returning a tighter over-approximation for the same fixed time-step. However, we did not investigate the comparison of the computation times further, because all the tools use lots of different parameters that are difficult to tune and our focus was on the tightness of the computed over-approximations using the same time-steps.

	prototype	**Flow***	**Ariadne**
Simple	0.001	0.001	0.036
Exponential	0.730	0.054	13.176
NonLinear	62.555	0.088	74.034
SimpleSwitching	0.494	0.081	2.411
DubinsCar	22.734	0.229	147.958

Table 4.5 – Computation times in seconds for each tool and each experiment

We focused on minimizing the over-approximation of each variable with respect to the time, but we did not try to keep a maximum of the dependencies between the state variables. As illustrated on Figure 4.3, while our prototype computed a tighter over-approximation of y with respect to x over the time interval $[0.99, 1]$ than Flow*, the tool CORA kept more dependencies between the sets over-approximating x and y, *i.e.* while the over-approximation computed by our prototype is tighter, a non-negligible part of it is not included in the one computed by CORA. While this loss of dependencies between the state variables could be a problem for reachability analysis on a wide time interval due to the wrapping effect (*cf.* Subsection 2.4.1 or [124, 106]),

it should not become an issue in the algorithm proposed in Subsection 3.4.2, because the resulting over-approximation would be subdivided into smaller sets. So, it should mostly yield an increase in the computation time (more sets to consider during the next iterations) instead of an explosion of the width of the over-approximation. However, the implementation of this last algorithm is left for future work.

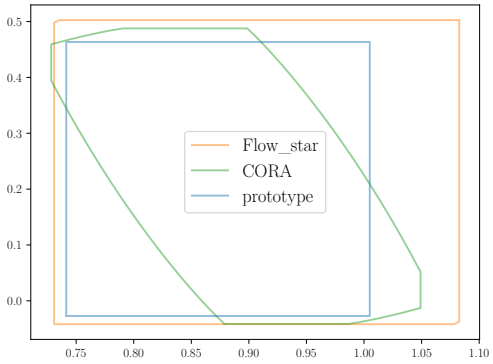
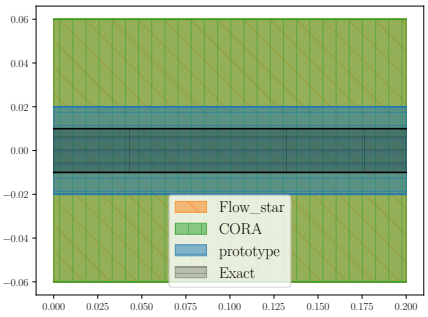
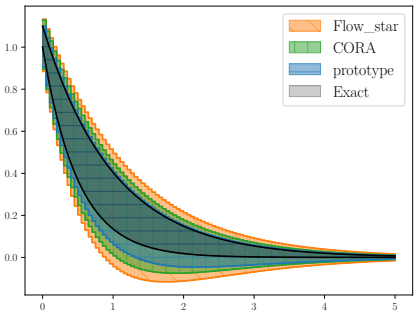


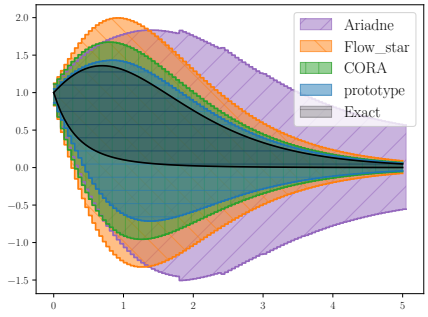
Figure 4.3 – Over-approximation of y with respect to x over the last time interval $[0.99, 1]$



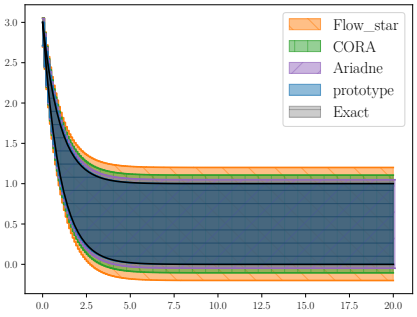
(a) Simple



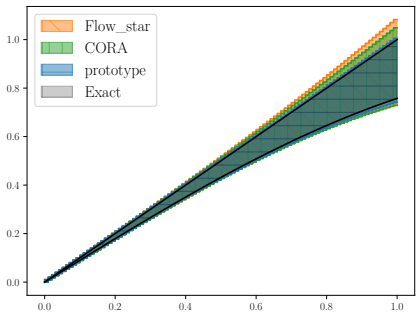
(b) Exponential



(c) NonLinear



(d) SimpleSwitching



(e) DubinsCar

Figure 4.4 – x with respect to the time using the different tools

Chapter 5

From concrete to set-valued simulations of hybrid models

In this chapter, we discuss the conditions to allow set-valued computation (*e.g.* reachability analysis) on models of hybrid systems used by tools for Model-Based Design. We start our reflection from a representation adapted to simulations of hybrid systems and we slightly modify it in order to allow set-valued computations. We also apply one possible approach, based on a parametrization by a representation of sets, to a restricted case of models written in the Zélus language.

This work is a collaboration with Marc Pouzet from the PARKAS[1] team at the Département d'informatique at the École normale supérieure and it is still in progress. Contrary to [43] and [128] (*cf.* Subsection 2.4.4.3), we do not want to create a compiler dedicated to the reachability analysis of models written in Zélus, but rather to slightly adapt the existing one in order to allow both computations of simulation (as currently) and reachability analysis.

In Section 5.1, we derive an interface between a simulation engine and external solvers into an interface allowing reachability analysis of hybrid models. In Section 5.2, we define an interface for the hybrid models that is an intermediate representation of models compatible with the ones written in Zélus. Finally, in Section 5.3, we present our implementation of set-valued intermediate models and its application on an example.

5.1 High level representation and interfaces

In this section, we present a possible interface for an engine that simulates a given model. We start with the case of concrete simulation, *i.e.* a classic simulation with at most one value associated to each variable at every instant. Then, we define a set-valued version of it with only slight modifications in order to solve difficulties induced by the handling of sets.

[1] `https://parkas.di.ens.fr/`

5.1.1 Concrete simulation

A tool that simulates the behavior of a model M actually defines a function *Simulate* that takes as arguments the model M and some input signal u and returns an output signal o:

$$Simulate(M)(u) = o \tag{5.1}$$

Such a function typically corresponds to an interpretation of a program: given a source code and an input of the program, it computes its output. In this case, the input signal u and the output signal o can be interpreted as sequences of values, *i.e.* $u = (u_n)_{n\in\mathbb{N}}$ and $o = (o_n)_{n\in\mathbb{N}}$, as streams in a synchronous data-flow language (*e.g.* LUSTRE[2] [81]). The indices $n \in \mathbb{N}$ can be interpreted as values of a logical time with a total order.

If M is a model of a hybrid system, then the signals u and o cannot be just discrete sequences of values at specific instants. Indeed, the signals may exist over real time as solutions of differential equations over some time intervals. Moreover, the discrete behaviors are assumed to be instantaneous. So, discrete events may be chained at the same real instant: we call this behavior a *cascade* [31]. It requires a way to encode the ordering of events at the same real instant. The paper [31] presents a non-standard semantics for such streams, but we can also use superdense time ones [108, 42] that gathers a real time t in $\mathbb{R}$ (real duration since the initial time) and a microstep n in $\mathbb{N}$ (the number of past transitions). Thus, a stream can be interpreted as a sequence $\left([\underline{t_n}, \overline{t_n}], f_n\right)_{n\in\mathbb{N}}$ such that for all $n \in \mathbb{N}$, $[\underline{t_n}, \overline{t_n}]$ is the domain of f_n and $\overline{t_n} \leq \underline{t_{n+1}}$. When a stream is not defined for some value n, f_n can defined as equal to a special value $\perp$ [29, Definition 1]. We propose the following definition for concrete streams:

Definition 13 (Stream)
Let V be a set of possible values. A stream u with values in V is a sequence $\left([\underline{t_n}, \overline{t_n}], f_n\right)_{n\in\mathbb{N}}$ such that for all $n \in \mathbb{N}$,

$$\overline{t_n} \leq \underline{t_{n+1}} \quad \text{and} \quad f_n : [\underline{t_n}, \overline{t_n}] \to V \cup \{\perp\}$$

Definition 14 (Evaluation of a concrete stream)
Given a stream u with values in V and a superdense time $(t, n) \in \mathbb{R} \times \mathbb{N}$, the value of u at time (t, n) is given by

$$u((t,n)) = \begin{cases} f_n(t) & \text{if } t \in [\underline{t_n}, \overline{t_n}] \\ \perp & \text{otherwise} \end{cases}$$

In the following, we denote $\mathbb{S}_V$ the set of streams with values in V.

Independently of the representation of the time, we often have to solve differential equations in order to obtain the value of the state with respect to the real time. Because solving differential equations is a complex task, the simulation engine often delegates it to an external tool. We consider such a tool as a parameter *csolve* (for *continuous solver*) of the *Simulate* function. Similarly, the discrete behaviors of the hybrid models

[2] `https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/reactive-toolbox/`

are triggered by conditions depending on some signals. The detection of the change of value of these conditions, typically zero-crossing events, is delegated to another external tool that is another parameter *zsolve* (for *zero-crossing solver*) of the simulation engine. Finally, the simulation defines a function

$$Simulate(csolve, zsolve)(M)(u) = o \tag{5.2}$$

Now, we present the interface that we can expect for the external tools *csolve* and *zsolve*. The continuous solver should return an interpolation function that takes as input a time and returns a mapping of values of the continuous variables. The continuous solver used by ZÉLUS is SUNDIALS CVODE[3] [41]. The returned value is typically an array of the type representing real numbers, *e.g.* a double-precision floating-point format. In order to compute such an interpolation function, the continuous integrator needs an initial value of those variables and the corresponding initial time. It also needs a duration for the interpolation function: the domain of the returned interpolation function is then guaranteed to be an interval starting at the initial time and whose width is not bigger than the given duration. Finally, it needs the flow condition in order to estimate the interpolation function. The flow condition *Flow* is a finitary relation over the possible values of the time, the continuous variables and the derivatives. In the next subsections, the flow condition *Flow* is assumed to be the right-hand side of the explicit ordinary differential equations, *i.e. Flow* is a function that takes as input the time and the values of the continuous variables and returns the corresponding derivatives. So, the continuous solver defines a function

$$csolve(Flow)(t_0, dt, x_0) = (dt', Approx) \tag{5.3}$$

with *Flow* the flow condition, t_0 the initial time, dt the expected duration, x_0 the initial state, dt' the actually computed duration, and *Approx* the interpolation function defined for all time in the closed interval $[0, dt']$. dt' may be strictly smaller than dt if no solutions exist on the whole interval $[0, dt]$ or if the solver subdivides the domain for precision concerns.

Similarly, the detector of events needs an indexed collection of predicates *Zout*, *i.e.* a collection of functions of the time and of the values of the continuous variables that returns boolean values, the approximation computed by the continuous solver and the boundaries of its time domain:

$$zsolve(Zout)(t_0, dt')(Approx) = (t_1, Zin) \tag{5.4}$$

with *Zout* an indexed collection of predicates, t_0 the initial time, dt' the duration, *Approx* a function that associates values of the continuous variables of the system to each time in the time interval $[0, dt']$ (*e.g.* the interpolation function computed by *csolve*), t_1 the time at which the first event has been detected in the time interval, and *Zin* an indexed collection of boolean values indicating for each index of *Zout* if the associated value

[3] `https://sundials.readthedocs.io/en/latest/cvode/index.html`

at time t_1 changes. The indexed collection *Zin* may be needed, because an evaluation of the model at time t_1 may not be sufficient to identify events, especially in case of zero-crossing events as illustrated in Example 15. If no changes of values occur, then t_1 should be equal to $t_0 + dt'$ and *Zin* only contains false values. Typically, *Zout* and *Zin* are implemented using arrays of the same size.

Example 15 (Need of *Zin* to trigger zero-crossing events)
Consider the following hybrid node written in ZÉLUS:

```
let hybrid f() = (zx,zy) where
    rec der x = 1. init -1.
    and der y = 0. init 0.
    and zx = up(x)
    and zy = up(y)
```

The node is composed of two ordinary differential equations, defining `x` and `y`, and two zero-crossing signals, `zx` and `zy`. The detector of events should return the time $t_1 = 1$. At that time, both `x` and `y` are equal to 0, but only `zx` should be activated (*cf.* 2.14 in Section 2.2). So, *Zin* is needed to explicit that `zx` is activated at $t_1 = 1$, while `zy` is not.

Finally, the simulation procedure consists in two phases [30, 41]: the simulation iterates discrete steps until all events at the current time are handled (*e.g.* cascades of events), then it integrates via *csolve* until *zsolve* detects an event (*cf.* [41, Section 3.2] and Figure 5.1). The simulation starts by computing a discrete step that initializes the state of the model at the first instant. Then, the simulation engine initializes the integrator *csolve* and this integrator is called until an event is detected by the detector *zsolve*. As soon as an event is detected, the simulation engine computes the images of the state through the jumps (*i.e.* discrete behaviors) associated to the triggered cascade of events, without increasing the real time. When no more events have to be handled, the simulation engine reinitializes the integrator (needed for multistep solvers) and the integrator is called again. This procedure ends if the target final time is reached or if a step failed, *e.g.* the integrator cannot compute an accurate enough interpolation function (based on some thresholds as parameters) or the detector of events detects events too close to the initial time during multiple iterations, which can be a Zeno behavior (*cf.* Example 4 of a bouncing ball).

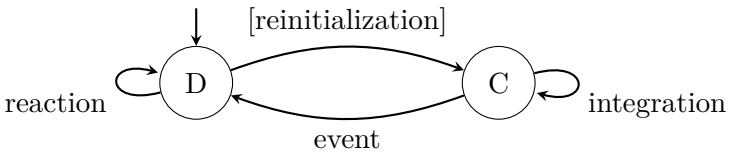


Figure 5.1 – Base of hybrid simulation ([41, Figure 4])

5.1.2 Set-valued streams and implications

In order to handle reachability analysis or set-valued simulation, we start by replacing the representation of real numbers by a representation of sets (*cf.* Subsection 2.4.1). Then, we may define set-valued streams the same way as concrete ones with values in a powerset. However, replacing concrete values by sets does not catch all possible trajectories, because set-valued computations may introduce uncertainty on time, as illustrated by Example 16. Because a time interval can be represented as an initial time and a duration, we decide to lift only the initial time to an uncertain quantity and to leave the duration exact. So, we define set-valued streams as sequences of triples, each one containing an uncertain initial time, a duration and a function that maps elapsed time to sets:

Definition 15 (Set-valued stream)
Let V be a set of possible values for a concrete stream (*cf.* Definition 13). A set-valued stream $\tilde{u}$ with values in $\mathcal{P}(V)$ (set of all subsets of V) is a sequence $\left([\underline{t_n}, \overline{t_n}], dt_n, \tilde{f}_n\right)_{n\in\mathbb{N}}$ such that for all $n \in \mathbb{N}$,

$$\underline{t_n} \leq \underline{t_{n+1}} \quad \text{and} \quad \tilde{f}_n : [0, dt_n] \to \mathcal{P}(V) \cup \{\bot\}$$

We assume that for all $n \in \mathbb{N}$, either $\tilde{f}_n$ is always equal or always different than $\{\bot\}$:

$$(\forall \delta \in [0, dt_n],\ \tilde{f}_n(\delta) = \{\bot\}) \vee (\forall \delta \in [0, dt_n],\ \tilde{f}_n(\delta) \neq \{\bot\})$$

Because our definition of set-valued streams is slightly different to the one of concrete streams, we also have to modify the definition of their evaluation. The evaluation of a set-valued stream at a given superdense time has to handle the uncertainty on the initial time of each domain. We define the evaluation as the union of all possible values over compatible initial real times. It is illustrated in Example 16.

Definition 16 (Evaluation of a set-valued stream)
Given a set-valued stream $\tilde{u}$ with values in $\mathcal{P}(V)$ and a superdense time $(t, n) \in \mathbb{R}\times\mathbb{N}$, the value of $\tilde{u}$ at time (t, n) is given by

$$u((t,n)) = \begin{cases} \{v \mid \exists t_0 \in [\underline{t_n}, \min(t, \overline{t_n})] : v \in \tilde{f}_n(t - t_0)\} & \text{if } t \in [\underline{t_n}, \overline{t_n} + dt_n] \\ \{\bot\} & \text{otherwise} \end{cases}$$

A set-valued stream encodes a set of streams whose values are included in one of the set-valued streams at the same times.

Definition 17 (Concrete streams in set-valued ones)
Consider a set-valued stream $\tilde{u}$ with values in $\mathcal{P}(V)$.

A (concrete) stream u *belongs* to $\tilde{u}$ if any evaluation of u belongs to the one of $\tilde{u}$ at the same superdense time, *i.e.*

$$\forall (t, n) \in \mathbb{R} \times \mathbb{N},\ u((t, n)) \in \tilde{u}((t, n))$$

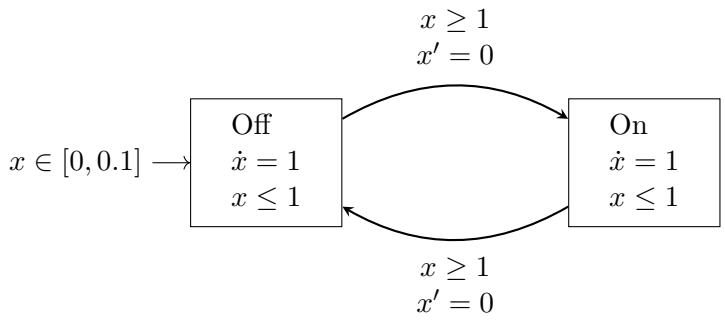
Using this definition, we can interpret a set-valued stream as a collection of concrete ones. Consider a set-valued stream $\tilde{u}$ with values in $\mathcal{P}(U)$. $\tilde{u}$ defines a set of streams u with values in U that belong to it, *i.e.*

$$\tilde{u} = \left\{ u \in \mathbb{S}_V \;\middle|\; \forall (t,n) \in \mathbb{R} \times \mathbb{N},\ u\big((t,n)\big) \in \tilde{u}\big((t,n)\big) \right\} \tag{5.5}$$

Such set-valued streams can be encoded as an indexed collection of parameterized sets, *e.g.* a list of Taylor models (*cf.* Subsection 2.4.1) with the real time as one of the parameters.

Using set-valued streams may result in the existence of multiple set-valued streams at the same real time non-trivial intervals, contrarily to concrete streams. They may be split when some concrete streams that are encoded take different branches. The first case of splitting set-valued stream occurs at the activation of a guard, as illustrated by Example 16 (more splits can occur if multiple guards are activated in a common time interval). The second case of splitting occurs during the evaluation of a discrete evolution, as illustrated by Example 17.

Example 16 (Uncertain guard)
Consider the following hybrid automaton that models a flashing lamp with a fixed frequency equal to a half hertz and uncertain initial phase:



For all real time $t \in [0.9, 1]$, a concrete stream associated to dt can still be in the mode Off (at superdense time $(t, 0)$) or it can be in the mode On (at superdense time $(t, 1)$).

A set-valued stream $\tilde{x}$ that over-approximates the set of admissible concrete streams associated to x could be:

$$\begin{aligned} &\left(\left[\underline{t_0}, \overline{t_0}\right], dt_0, \tilde{f}_0\right) = \left([0,0], 1, \delta \mapsto [\delta, \delta + 0.1]\right) \\ &\forall n \in \mathbb{N}^*,\ \left(\left[\underline{t_n}, \overline{t_n}\right], dt_n, \tilde{f}_n\right) = \left([n - 0.1, n], 1, \delta \mapsto \delta\right) \end{aligned}$$

We notice that such a set-valued stream is a strict over-approximation of the set of admissible concrete streams, because it contains streams that reach values strictly bigger than 1 at time $t = 1$. Moreover, the set-valued stream $\tilde{x}$ defines two disjoint

sets over the real time interval $[0.9, 1]$:

$$\tilde{x}([0.9, 1], 0) = [0.9, 1.1]$$
$$\tilde{x}([0.9, 1], 1) = [0, 0.1]$$

Example 17 (Discrete evolution with multiple possible branches)
Consider the following node in ZÉLUS:

```
let node f(x) = y where
    y = if x < 0. then x -. 1. else x +. 1.
```

If a set-valued stream $\tilde{x}$ evaluated at the superdense time (t, n) is equal to $[-1, 1]$, then a set-valued stream $\tilde{y}$ associated to y may be equal to $[-2, -1[\cup[1, 2]$. However, the value of $\tilde{y}$ is neither closed nor contiguous, contrary to the classical representations of sets (*cf.* Subsection 2.4.1). Using an interval representation, we can either compute $\tilde{y}((t, n)) = [-2, 2]$, which is a huge over-approximation, or we can decide to produce two set-valued streams $\tilde{y}_1$ and $\tilde{y}_2$ that have the same past, but such that $\tilde{y}_1((t, n)) = [-2, -1]$ and $\tilde{y}_2((t, n)) = [1, 2]$.

5.1.3 Set-valued simulation

In the previous subsections, we presented set-valued streams and their implications on the different steps of the simulation. In particular, we noticed that a set-valued input stream may result in multiple set-valued output streams due to the effect of uncertainties on the transitions and branches (conditional structures like `if/then/else`).

First, we define a set-valued version of the interface of the continuous solvers. They need necessarily a set-valued version of the flow condition $\tilde{Flow}$ and a set-valued initial state $\tilde{x}_0$. They also need a time domain over which to compute the continuous evolution of the state. Because the detection of the activation of the transitions may introduce uncertainties on the time, the continuous solvers should be able to handle them. So, the initial time has to be a set $\tilde{t}_0$. However, we decide to specify a concrete expected duration dt of the evolution and we expect that the returned set-valued interpolation function $\tilde{Approx}$ is packed with an actually computed concrete duration dt' (*e.g.* a value of type `float`). So, $\tilde{Approx}$ will be a set-valued function from $[0, dt']$ to the powerset of continuous states and the interface of the continuous solvers is

$$\tilde{csolve}\Big(\tilde{Flow}\Big)(\tilde{t}_0, dt, \tilde{x}_0) = \Big(dt', \tilde{Approx}\Big) \tag{5.6}$$

We notice that this set-valued interface is compatible with the concrete one.

Then, we define a set-valued version of the interface of the zero-crossing solvers. As for the continuous solvers, the inputs have to be set-valued ones: the collection of predicates $\tilde{Zout}$ should be evaluated on sets, the initial time $\tilde{t}_0$ is uncertain, the interpolation function is the set-valued one $\tilde{Approx}$ that is returned by the continuous solver, as is the duration dt' defining the maximal value of the time domain. Because

the zero-crossing events may be highly uncertain and because we want to be able to compute an over-appromation of the reachable set, the zero-crossing solvers should be able to return a collection of detected events. We also expect the zero-crossing solvers to return a subset of the set-valued continuous state that activates each detected zero-crossing event. So, the set-valued interface of the zero-crossing solvers becomes

$$\tilde{zsolve}\Big(\tilde{Zout}\Big)(\tilde{t}_0, dt')\Big(\tilde{Approx}\Big) = \{(\tilde{t}_i, Zin_i, \tilde{x}_i)\}_{i\in I} \tag{5.7}$$

where i is an index in a finite discrete set I (typically, $I = [\![0, n]\!]$), $\tilde{t}_i$ is the uncertain time at which an event is detected, Zin_i is an indexed collection of boolean values indicating for each index of $\tilde{Zout}$ if the associated value changes at the detected event, and $\tilde{x}_i$ is the subset of $\tilde{Approx}([0, dt'])$ that activates the event at a time in $\tilde{t}_i$ ($\tilde{x}_i$ is potentially an over-approximation). We notice that this set-valued interface is compatible with the concrete one returning a singleton and replacing $\tilde{x}_i$ by $Approx(t_1 - t_0)$.

Finally, we define the set-valued version of the interface of the simulation engine. It should be parameterized by a set-valued continuous solver $\tilde{csolve}$ and a set-valued zero-crossing solver $\tilde{zsolve}$. Similarly, the model should be handled in a set-valued version $\tilde{M}$, *i.e.* in which all expressions can be evaluated over sets. Then, the set-valued simulation takes a collection of set-valued input streams $\{\tilde{u}_i\}_{i\in I}$ and returns a collection of set-valued output streams $\{\tilde{o}_j\}_{j\in J}$, where I and J are two finite sets of indexes.

$$\tilde{Simulate}\Big(\tilde{csolve}, \tilde{zsolve}\Big)\Big(\tilde{M}\Big)(\{\tilde{u}_i\}_{i\in I}) = \{\tilde{o}_j\}_{j\in J} \tag{5.8}$$

Contrary to the concrete simulation engine, the set-valued one has to handle the collection of set-valued inputs and to construct the one of outputs. In practice, such a set-valued simulation engine computes the union of the results for each set-valued input stream $\tilde{u}_i$.

We did not implement such a generic set-valued simulation engine, but we propose in the next section an implementation of one that is able to handle models without inputs as a first step for implementing a solver that handles time-varying inputs.

5.2 Intermediate representation of the models

We presented a set-valued interface for a simulation engine using external tools. However, we did not provide any interface of the models. A model could be interpreted directly from the source code, *e.g.* from the abstract syntax tree, as a classic interpretation of a program. Instead we consider that the model is first translated from the source code into an intermediate representation similar to the S-functions of Simulink[4] or to the FMU for Model Exchange of Modelica [67, Section 3]. The creation of such an intermediate representation of the model is the task of a compiler, for example from a source code written in Zélus into a collection of functions in OCaml [41].

In this section, we focus on models written in Zélus, but it should be relevant for other tools such as Simulink.

[4] `https://www.mathworks.com/help/simulink/sfg/what-is-an-s-function.html`

We start by presenting the intermediate representation of the models that we consider in this work. Then, we define the representations of the states. Finally, we present a set-valued version of the intermediate models.

5.2.1 Functional intermediate model

We provide a purely functional interpretation of hybrid models, *i.e.* without side effects, in order to simplify reasoning about them. We focus on models written in Zélus (*cf.* Section 2.2), so the flow conditions are explicit first-order ordinary differential equations, *i.e.* of the form $\dot{x}(t) = f_s(x(t), u(t))$ with f_s a continuous function depending on the current state s, and the transitions are triggered by rising zero-crossing events, *i.e.* g_s a continuous function depending on the current state s such that

$$\forall \varepsilon > 0,\ \exists \delta \in [0, \varepsilon] : g_s(x(t-\delta), u(t-\delta)) < 0 \quad \text{and} \quad g_s(x(t), u(t)) \geq 0$$

These functions are required by the external solvers [41]. Finally, we need a function that given the state and the input returns the value of the output signal. We did not introduce the time as parameter of these functions, because it is not accessible in Zélus and it can be considered as an input, *i.e.* a component of u.

This representation of hybrid models is similar to hybrid automata (*cf.* Definition 1) except that the graph is implicit (the modes and transitions are determined by the states and the zero-crossing events) and the guards and the invariants are defined by the zero-crossing events (an invariant is true in a mode unless a zero-crossing event is detected, which activates the corresponding guard). This representation is convenient in the sense that it allows an easy parallel composition of models: we do not have to compute the strong product (or normal product) of the graphs, because it emerges from the concatenation of the different sets (states variables, right-hand sides of ordinary differential equations and zero-crossing functions). It allows a compiler to translate modular models into this intermediate representation [32, 41].

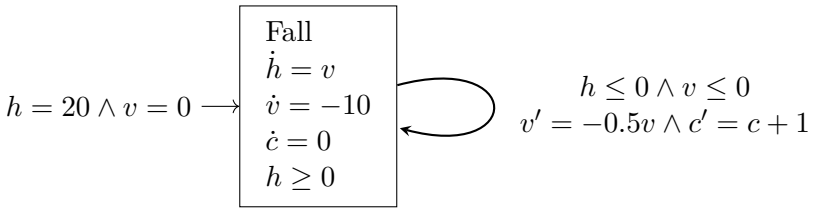
Besides the functions related to the implicit hybrid automata, we need functions to manipulate the states s. The state of a model can be a complex object, but the continuous solvers only manipulate part of it that is determined by the ordinary differential equations. So, we use the following interface (written in OCaml):

```
type model = {
    init : state;
    getx : state -> cont;
    deriv : state * cont * input -> der;
    zeroc : state * cont * input -> zout;
    update : state * cont * input * zin -> state;
    output : state * cont * input -> output;
  }
```

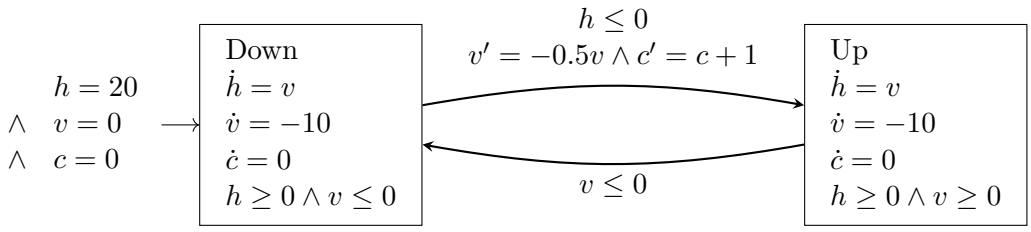
with `init` the initial state (type `state`) of the model, `getx` the function that returns the continuous variables (type `cont`) from a state, *i.e.* the variables whose evolution is defined by their derivatives, `deriv` the function that returns the values of the derivatives (type

`der`), `zeroc` returns the values of zero-crossing functions (type `zout`), `update` returns the state after a transition and `output` returns the values of the output signal (type `output`). All these functions depend on the current state of the model, but also on the value of the continuous variables and the value of the input signal (type `input`). Moreover, the function `update` takes the set of activated transitions (type `zin`) as an extra argument. In practice, `cont` and `der` are equal, `cont`, `der`, `zout` are arrays of elements of the same type (*e.g.* floating-point number, interval, Taylor model, *etc.*), and `zin` is an array of elements of booleans.

Example 18 (Intermediate model of a bouncing ball)
We consider a bouncing ball (*cf.* Example 4). We adapt the previous automaton in order to add to it a counter of bounces:



In order to exhibit the zero-crossing functions, we can decompose the mode into two:



So, the state s of the model is defined by the value of $s.h$, $s.v$, $s.c$ and $s.mode$, where $s.h$ and $s.v$ are real values, while $s.c$ is an integer and $s.mode$ is either Up or Down (sum type).

We present a possible implementation using the OCaml programming language. The initial state is a record:

```
let init = {
    h = 20.;
    v = 0.;
    c = 0;
    mode = Down;
  }
```

`mode` has a sum type `type mode = Up | Down`, while `h` and `v` have type `float`, and `c` has type `int`.

While the derivative of $s.c$ is explicitly defined in the hybrid automaton, it is in fact piecewise constant and it can only take integer values. So, it does not belong to the set of continuous variables. It is the same for the variable $s.mode$. Finally, the function *getx* returns an array with the values of $s.h$ and $s.v$ as elements:

```
let getx s = Array.of_list [s.h;s.v]
```

The function *deriv* should return an array with two elements corresponding to the derivatives of the continuous variables:

```
let deriv (s,x,u) = Array.of_list [x.(1);-10.]
```

where `x.(1)` corresponds to the value of v returned by the approximation.

The zero-crossing functions trigger the discrete events, *i.e.* the activation of the guards of the transitions. If the current mode is Down, then the unique transition is triggered by h, while if the current mode is Up, then the unique transition is triggered by v. So, *zeroc* could return a single value. However, in order to easily associate an index to each zero-crossing signal, *zeroc* returns an array whose elements are the values of the zero-crossing signal. If a signal is not defined in a mode, its value is set to a negative one:

```
let zeroc (s,x,u) =
  match s.mode with
  | Up   -> Array.of_list [-42.; -. x.(1)]
  | Down -> Array.of_list [-. x.(0); -42.]
```

The function update modifies the state only if the current mode is Down and if the transition is activated. In this case, only part of the state is modified: the velocity $s.v$ becomes the opposite of half of its last computed value (stored in `x.(1)`) and the counter $s.c$ is incremented by one. In all the other cases, the state is returned unchanged:

```
let update (s,x,u,zin) =
  if s.mode = Down && zin.(0) = true then
    { s with v = -0.5 *. x.(1); c = s.c + 1 }
  else s
```

Finally, the output is not specified by the hybrid automaton. We may consider that the outputs are the signals associated to the variables h, v and c. If so, we have

```
let output (s,x,u) =
  (x.(0),x.(1),s.c)
```

Because h and v are continuous variables, we return the values in x, while we return $s.c$ for the value of the counter c.

We notice on this example that the mode $s.mode$ of the automaton is an internal variable of the state s, *i.e.* it is not exposed by the interface.

While such a functional representation of the intermediate models is convenient for reasoning and prototyping, it is unlikely to be implemented as such in tools such as Simulink or Zélus. Indeed, with our representation, the functions have to compute copies of the states that can be huge structures in the case of large models. So, we use this interface for reasoning, but we implement an iterative version with side-effects when it is possible: only a pointer to the state is passed to the functions, which modify it inplace (*cf.* Subsection 5.3.2).

5.2.2 Set-valued representation of states

Before defining a set-valued version of the functional intermediate model, we have to define the set-valued version of the states.

The states are tuples most likely implemented as records. The elements can have different types: continuous types (*e.g.* a real number implemented as floating-point number), discrete types (*e.g.* integer or sum type), or the type of the state of a nested model. Our idea is to replace the continuous types by the type of a representation of sets, *e.g.* intervals of Taylor models (*cf.* Subsection 2.4.1), and the states of nested models by their set-valued version. It is illustrated by Example 19.

Example 19 (Lifting of a state to its set-valued version)
Consider the following state (written in OCaml) defined by two continuous variables `x0` and `x1` of type `float`, a discrete variable `d0` of type `int`, another discrete variable `d1` of a user-defined sum type `mode`, and an instance `i0` of a nested model:

```
type mode = Mode1 | Mode2
type state_nested
type state = {
    x0 : float;
    x1 : float;
    d0 : int;
    d1 : mode;
    i0 : state_nested;
  }
```

Following the presented procedure and using Taylor models as the representation of sets (type `taylor_model`), the set-valued version of this state is obtained by replacing the type `float` by `taylor_model` and replacing the type of the instance of the nested model by its set-valued version (defined in a similar way):

```
type taylor_model
type mode = Mode1 | Mode2
type setvalued_state_nested
type setvalued_state = {
    x0 : taylor_model;
    x1 : taylor_model;
    d0 : int;
    d1 : mode;
    i0 : setvalued_state_nested;
  }
```

This set-valued representation of states is convenient for the continuous flowpipes, *i.e.* sets of continuous evolutions without jumps. This representation is also suitable for the case of hybrid systems with only continuous jumps, *i.e.* discrete behaviors that are continuous with respect to the values of the continuous variables. However, as presented in Subsection 5.1.2, due to the uncertain location of the states, the evaluation of non-continuous jumps may result in multiple possible values of the discrete variables. Such a set-valued function is presented in Example 20.

Example 20 (Function that may produce multiple set-valued states) Consider the following `update` function (written in OCaml):

```
let update (s,x,u) =
  if x.(0) <= 0. then {s with c = 1}
  else {s with c = 2}
```

where `s.c` is an integer variable.

If `x.(0)` defines an interval such that 0 belongs to its interior, *e.g.* a Taylor model whose range is equal to $[-0.1, 0.1]$, then both branches can be activated and, in the case of a reachability analysis, the function `update` should return two states that cannot be gathered as a unique set-valued state.

We may have a similar behavior with the function `output` whose result can contain discrete values.

In order to handle such functions that may return simultaneously multiple set-valued states, we could have defined collections of set-valued states as the lifting of concrete states. However, it yields a much more difficult implementation and we prefer to restrict our study to functions that, given a set-valued input, can only return a unique set-valued state.

5.2.3 Set-valued functional intermediate model

Based on our definition of a set-valued state of an intermediate model (*cf.* Subsection 5.2.2), we define a set-valued functional intermediate model. To avoid handling functions that may return simultaneously multiple set-valued states (*cf.* Example 20), we only consider models whose discrete dynamics are continuous with respect to the continuous state variables or inputs. A sufficient condition to avoid such difficulties is

to reject any model that contains non-continuous operators (*e.g.* rounding function) or branches (*e.g.* `if/then/else` constructions) depending on the values of continuous variables.

With such a restriction on the models, we can define an interface for set-valued functional intermediate models as follow:

```
type model = {
    init : set_state;
    getx : set_state -> set_cont;
    deriv : set_state * set_cont * set_input -> set_der;
    zeroc : set_state * set_cont * set_input -> set_zout;
    update : set_state * set_cont * set_input * zin -> set_state;
    output : set_state * set_cont * set_input -> set_output;
  }
```

where `set_type` corresponds to the type of objects of type `type` (equals to `state`, `cont`, `input`, `output`, `der` or `zout` of the original functional intermediate model) in which every type representing real numbers is replaced by the type of the chosen representation of sets (*e.g.* the type `float` is replaced by the type `taylor_model`).

Despite its inability to encode all possible results in case of non-continuous functions, this set-valued interface of intermediate models is compatible with the concrete one defined in Section 5.2.1.

5.3 Details of implementation

In this section, we describe a possible implementation of intermediate models that allows to easily define a set-valued version as presented in Subsection 5.2.3.

5.3.1 Implementation of set-valued functional intermediate models

We defined the intermediate models (*cf.* Subsection 5.2.1) as collections of functions. These functions are defined in a programming language, *e.g.* OCaml in the case of models written in Zélus. So, we may deduce set-valued intermediate models from the source codes, *e.g.* using a dedicated compiler as in [128] or an interpreter as computed by static analysers (*cf.* Subsection 2.4.3).

Because concrete and set-valued intermediate models have similar interfaces (*cf.* Subsections 5.2.1 and 5.2.3), we want to define intermediate models that are independent of the chosen representation of sets. In particular, we could define a representation of singletons (*i.e.* sets with only one element) to obtain the concrete intermediate model of Subsection 5.2.1.

We parameterize the intermediate models by a representation of sets using functors in OCaml. We need to define a common interface for all the representations of sets (modules with a type to store the sets and functions to manipulate them) and we can

define the intermediate models for any representation that satisfies the interface. The method is illustrated in Example 21.

Example 21 (Parametric intermediate model of a bouncing ball)
Consider the model of a bouncing ball presented in Example 18. We can define its intermediate model as follows, parameterized by an abstract representation of sets here called `MySet` that implements the interface `Representation`. So, we start defining the interface `Representation`:

```
module type Representation = sig
  type t

  val of_float : float -> t
  val ( ~-. ) : t -> t
  val ( *. ) : t -> t -> t
end
```

Such an interface could be redefined for each model, but it can also be defined once with an exhaustive collection of functions independent of any model.

To define the parametric intermediate model, we simply wrap the concrete intermediate model presented in Example 18 into an OCaml functor parameterized by a module `MySet` encoding a representation of sets, we replace every operator on `float` by its corresponding version in the module `MySet` and we replace all `float` constant values by the sets encoding them:

```
module BouncingBall(MySet : Representation) = struct
  type mode = Up | Down

  type state = {
      h : MySet.t;
      v : MySet.t;
      c : int;
      mode : mode;
    }

  let init = {
      h = MySet.of_float 20.;
      v = MySet.of_float 0.;
      c = 0;
      mode = Down;
    }

  let getx s = Array.of_list [s.h;s.v]

  let deriv (s,x,u) = Array.of_list [x.(1); MySet.of_float (-1.)]

  let zeroc (s,x,u) =
    match s.mode with
    | Up   -> Array.of_list [MySet.(-. x.(1))]
    | Down -> Array.of_list [MySet.(-. x.(0))]

  let update (s,x,u,zin) =
    if s.mode = Down && zin.(0) = true then
      { s with v = MySet.((of_float (-0.5)) *. x.(1)); c = s.c + 1 }
    else s

  let output (s,x,u) =
    (x.(0),x.(1),s.c)

  let zeroc (s,x,u) =
    match s.mode with
    | Up   -> Array.of_list [MySet.of_float (-42.); MySet.(-. x.(1))]
    | Down -> Array.of_list [MySet.(-. x.(0)); MySet.of_float (-42.)]
end
```

As presented in Example 20, the conditional structures may be difficult to lift into a set-valued version. We avoid the difficulty by assuming that the conditions do not depend on the continuous variables (*cf.* Subsection 5.2.3).

However, we could handle conditional structures by considering them as operators. For example, we could write in OCaml:

```
let if_then_else
      (cond : bool)
      (f_true : 'a Lazy.t)
      (f_false : 'a Lazy.t)
    : 'a
  =
  if cond then Lazy.force f_true
  else Lazy.force f_false
```

In that case, we could lift this operator into one that is compatible with set-valued computations. Because a condition evaluated on sets may be neither `true` nor `false` (*cf.* Example 17), we have to introduce a type for encoding the results of conditions that allows a third possible value:

```
type trilean = True | False | Unknown
```

Depending on the value of the condition, the set-valued conditional operator should return either a unique evaluation of a branch or the union of the results of the two branches:

```
let if_then_else
      (union : 'a -> 'a -> 'a)
      (cond : trilean)
      (f_true : 'a Lazy.t)
      (f_false : 'a Lazy.t)
    : 'a
  =
  match cond with
  | True -> Lazy.force f_true
  | False -> Lazy.force f_false
  | Unknown -> union (Lazy.force f_true) (Lazy.force f_false)
```

The difficulty of such an implementation of the conditional structures is the implementation of the `union` that should be able to handle numerous cases: sets, set-valued states, tuples with different types, *etc.* Such an implementation is left for a future work.

Remark 11 (Abstract interpretation of branches) Static analysers are able to handle such conditional structures, but their interpretation affects the whole abstract contexts and not only individual signals (*cf.* Example 14), contrarily to our approach that delegates the evaluation to the model itself.

5.3.2 Implementation of set-valued intermediate models with side effects

The implementation proposed in the previous subsection implies lots of array creations. In particular, the continuous solver may call the function `deriv` many times, which creates a new array each time.

To avoid lots of copies, we can define global arrays and structures that are modified in place by the different functions. Such a method is implemented in the current version of Zélus, but we describe our own interface in this subsection. The interface of intermediate models becomes in OCaml:

```
type ('state,'input,'output,'set) model = {
    csize : int;
    zsize : int;
    alloc : int -> 'set array -> 'set array
            -> int -> 'set array -> bool array
            -> 'state;
    init : 'state -> unit;
    deriv : 'state * 'input-> unit;
    zeroc : 'state * 'input -> unit;
    update : 'state * 'input -> unit;
    output : 'state * 'input -> 'output;
    copy : 'state -> 'state;
    get_carray : 'state -> 'set array;
    get_darray : 'state -> 'set array;
    get_zout : 'state -> 'set array;
    get_zin : 'state -> bool array;
  }
```

with `'state` an abstract type of the state of the model, `'input` and `'output` abstract types of inputs and outputs of the model, and `'set` an abstract type that will be instantiated with the type of a representation of unidimensional sets.

Contrarily to the previous subsection, the `init` function does not return a state, but it initializes the values of an already allocated state. The allocation of a new state is done by the function `alloc` such that

```
alloc coffset carray darray zoffset zout zin
```

returns a new state with `carray` an array for storing the values of the continuous signals, `darray` an array for their derivatives, `zout` an array for the values of the zero-crossing functions and `zin` the activation of each zero-crossing. `coffset` and `zoffset` are indices offsets for the arrays allowing to share the arrays for multiple models. The arrays `carray` and `darray` should contain at least `coffset+csize` elements and the arrays `zout` and `zin` should contain at least `zoffset+zsize` elements in order to avoid any buffer overflows.

A function `copy` must be provided to compute the evolution of a given set-valued state through different activated transitions (*cf.* Subsection 5.1.2). The copied state has to point to copied versions of the arrays `carray`, `darray`, `zout` and `zin`. To access to those copies, the models have to provide accessor functions `get_carray`, `get_darray`, `get_zout` and `get_zin`.

While the interface is defined using parametric polymorphism, the actual implementation of the models is done as presented in the previous subsection using functors in OCaml.

5.3.3 Handling of zero-crossing events

As presented in Subsection 5.1.2 , a detection of zero-crossing events with set-valued streams may result in multiple evolutions due to uncertainties. It can be handled with copies of the state as presented in Subsection 5.3.4.

However, if all the concrete streams encoded by a set-valued one have activated a zero-crossing before a given time, then the set-valued stream should not exist beyond that point. It is illustrated in Example 22.

Example 22 (Simple clock in Zélus)
Consider the following model of a clock written in ZÉLUS:

```
let hybrid clock(x0) = x where
    rec der x = 1. init x0 reset z -> 0.
    and z = up(x -. 1.)
```

It is a similar model to the one presented in Example 16 but with a single mode and zero-crossings as guards of the transitions.

If `x0` belongs to $[0, 0.1]$, then a set-valued stream $\tilde{x}$ that over-approximates the set of admissible concrete streams associated to `x` could be as in Example 16:

$$\begin{array}{l}([\underline{t_0}, \overline{t_0}], dt_0, \tilde{f_0}) = ([0,0], 1, \delta \mapsto [\delta, \delta + 0.1]) \\ \forall n \in \mathbb{N}^*,\ ([\underline{t_n}, \overline{t_n}], dt_n, \tilde{f_n}) = ([n - 0.1, n], 1, \delta \mapsto \delta)\end{array}$$

We notice that at the superdense time $(t, n) = (1, 0)$, $\tilde{x}((1, 0))$ is equal to $[1, 1.1]$. Because at the initial time, $\tilde{x}((0, 0))$ is equal to $[0, 0.1]$, we deduce that any concrete stream contained in $\tilde{x}$ triggers a zero-crossing event at some superdense time $(t, 0)$ with $t \in [0, 1]$. So, the set-valued stream $\tilde{x}$ should not be defined for any superdense time $(t, 0)$ with $t > 1$, because no concrete stream can be defined at such a superdense time.

The difficulty is to detect zero-crossing events that are activated by all the represented concrete streams. So, we want to distinguish these zero-crossing events from the ones that are only activated by only some of the concrete streams.

Definition 18 (Set-valued zero-crossing event)
Consider a continuous set-valued function $\tilde{f}$ from $[t_0, t_1]$ to $\mathcal{P}(\mathbb{R})$.

The set-valued function $\tilde{f}$ triggers a *set-valued zero-crossing event* if there exists a time different to the lower bound such that zero belongs to the image of $\tilde{f}$, *i.e.*

$$\exists t \in]t_0, t_1],\ 0 \in \tilde{f}(t)$$

Definition 19 (Total zero-crossing event)
A continuous set-valued function $\tilde{f}$ from $[t_0, t_1]$ to $\mathcal{P}(\mathbb{R})$ triggers a *total zero-crossing event* if for all continuous function f such that

$$\forall t \in]t_0, t_1],\ f(t) \in \tilde{f}(t)$$

f triggers a zero-crossing event on $]t_0, t_1]$, *i.e.*

$$\exists t^* \in]t_0, t_1] : \Big(f(t^*) = 0\Big) \wedge \Big(\forall \varepsilon > 0,\ \exists \delta \in [0, \varepsilon] : t^* - \delta \in [t_0, t_1] \wedge f(t^* - \delta) < 0\Big)$$

Definition 20 (Partial zero-crossing event)
A continuous set-valued function $\tilde{f}$ from $[t_0, t_1]$ to $\mathcal{P}(\mathbb{R})$ triggers a *partial zero-crossing event* if it triggers a set-valued zero-crossing but not a total zero-crossing event.

Remark 12 We considered all the concrete streams that are represented by a set-valued one for simplicity, but we could also only consider concrete streams that are admissible by the model. In that case, a set-valued zero-crossing is total as soon as the set of possible derivatives is positive, which could be checked using automatic differentiation (*e.g.* using FADBADML[a]).

[a] `https://fadbadml-dev.github.io/FADBADml/`

The identification of total and partial zero-crossing events is crucial in order to reduce the computation of non-admissible trajectories. So, we want to define a simple criterion to detect total zero-crossing events. First, we notice a property that directly results from the Bolzano's theorem.

Lemma 6
Consider a continuous set-valued function $\tilde{f}$ from $[t_0, t_1]$ to $\mathcal{P}(\mathbb{R})$.
If there exist $t_{\min}$ and $t_{\max}$ in $[t_0, t_1]$ such that

$$t_{\min} \leq t_{\max} \wedge \max \tilde{f}(t_{\min}) < 0 \wedge \min \tilde{f}(t_{\max}) \geq 0$$

then, $\tilde{f}$ triggers a total zero-crossing event.

Proof Consider $\tilde{f}$, $t_{\min}$ and $t_{\max}$ as defined in Lemma 6.
Consider a continuous function f from $[t_0, t_1]$ to $\mathbb{R}$ such that

$$\forall t \in [t_0, t_1],\ f(t) \in \tilde{f}(t)$$

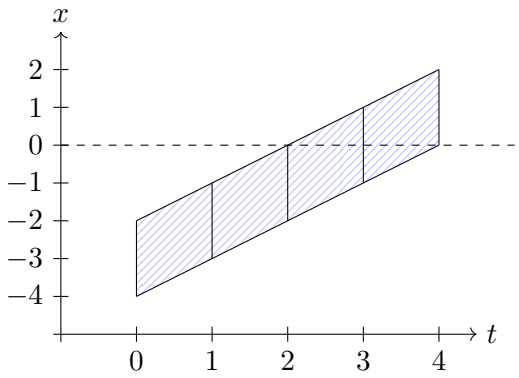
By hypothesis, we have

$$f(t_{\min}) < 0 \wedge f(t_{\max}) \geq 0$$

Due to the Bolzano's theorem, there exists a time t_z in $[t_{\min}, t_{\max}]$ such that $f(t_z) = 0$.
So, $\tilde{f}$ triggers a total zero-crossing event.

It allows to detect total zero-crossing events if the domains of the continuous set-valued functions are entirely known. However, the continuous set-valued functions are often computed on bounded domains. So, a microstep (from (t, n) to $(t, n+1)$) of a set-valued stream does not necessarily imply a jump of a value: the microstep may simply be enforced by the continuous solver. In that case, considering the continuous set-valued functions independently is not sufficient to detect total zero-crossing events. It is illustrated in Example 23.

Example 23 (Set-valued stream with fixed time-steps)
Consider a set-valued stream $\tilde{x} = \left(\left[\underline{t_n}, \overline{t_n}\right], dt_n, \tilde{f}_n\right)_{n\in\mathbb{N}}$ such that for all $n \in [\![0, 3]\!]$, $\underline{t_n} = \overline{t_n} = n$, $dt_n = 1$ and for all $\delta \in [0, dt_n]$, $\tilde{f}_n(\delta) = [\delta - 4, \delta - 2]$.



We notice that $\tilde{f}_2$ and $\tilde{f}_3$ trigger *partial* zero-crossing events on $[0, 1]$, which corresponds to the time interval $[2, 3]$ for $\tilde{f}_2$ and to the time interval $[3, 4]$ for $\tilde{f}_3$. However, if the model does not introduce any jumps at time $t = 3$, then all the concrete streams that belong to $\tilde{x}$ are continuous at time $t = 3$. So, $\tilde{f}_3$ should trigger a *total* zero-crossing event, because all concrete signals are continuous, they are negative at some time in the past and they are all non-negative at time $t = 4$.

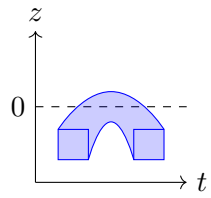
To detect total zero-crossing events even in the case of continuous set-valued functions whose image of the initial time is not entirely negative, we add to each zero-crossing function a boolean value that is true only if the associated set-valued stream was entirely negative since the last jump. Because this information is only used by the simulation engine and not by the models, the handling of such boolean values is left to the simulation engine: it creates an indexed collection similar to `zin` (*cf.* Subsection 5.3.2), *i.e.* an array of boolean values.

We define such an array `a` of boolean values with the same size as `zin` and `zout`. For all indices `i`, `a.(i)` (`i`-th value in the array `a`) is set to `true` if the maximum value of `zout.(i)` is negative. We only check it at the first instant of each continuous set-valued function of the set-valued streams that encode the zero-crossing functions, because it is impossible to check at every time and the last instant is either the first instant of the following continuous set-valued function or it is not relevant because of a total zero-crossing event. At every discrete behavior, the array `a` is filled with `false` values, because a jump may introduce discontinuities in the signals.

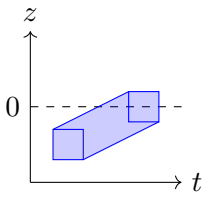
Remark 13 The values of `a` are reset after every discrete transition, because we assume that we only know the interface of the models. Knowing the structure of the models (*e.g.* the tree of nodes) may allow to only reset the values of `a` that correspond to zero-crossing signals defined in subtrees of the nodes triggering zero-crossing events.

Example 24
Consider the Example 23. We assume that x is the unique signal of the model that can trigger a zero-crossing event. So, the arrays `zout`, `zin` and `a` have the size equal to one with a unique possible index equal to zero.

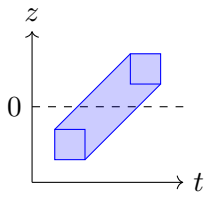
At time $t = 0$, the values in the set-valued stream $\tilde{x}$ are all negative. So, `a.(0)` is set to `true`. At time $t = 1$, all the values are still negative, but `a.(0)` is already equal to `true`. At time $t = 2$, the maximum value in $\tilde{x}$ is equal to zero, so `a.(0)` keeps its previous value (`true` in this example). A partial zero-crossing event is detected over the time interval $[2, 3]$, because there exist no instants at which the values are all non-negative. At time $t = 3$, `a.(0)` keeps its previous value, because the maximum value of $\tilde{x}$ is positive. Finally, a total zero-crossing event is detected over the time interval $[3, 4]$, because the minimum value at $t = 4$ is equal to zero (all the values are then non-negative) and because `a.(0)` is `true`, which denotes that all the concrete signals were negative at some instants in the past.



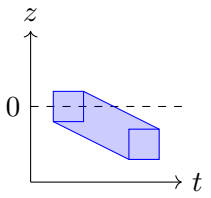
(a) simple zero-crossing with `a.(i)` true at the beginning



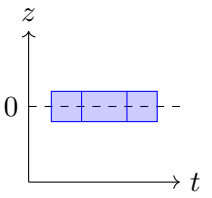
(b) simple zero-crossing with `a.(i)` true at the beginning



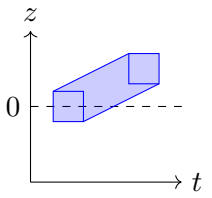
(c) complete zero-crossing



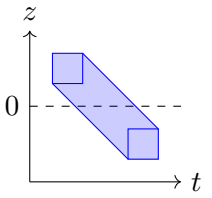
(d) simple zero-crossing with `a.(i)` true at the end



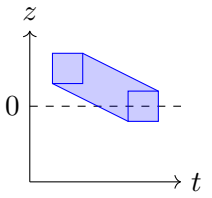
(e) simple zero-crossing



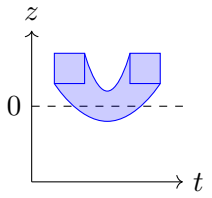
(f) complete zero-crossing if `a.(i)` true



(g) simple zero-crossing with `a.(i)` true at the end



(h) simple zero-crossing



(i) simple zero-crossing

Figure 5.2 – All possible cases of set-valued zero-crossing of a signal z during a single integration step around a zero-crossing event

All the different cases of set-valued zero-crossing events are represented in Figure 5.2.

Notice that non-negative sets should be considered as positive to match the cases presented in the figure.

5.3.4 Algorithm for reachability analysis

We implemented an algorithm inspired by the classic reachability analysis algorithm presented in Subsection 2.4.4.1. It iterates over a collection of states of the simulation (the time, the state of the model and an array `a` of boolean values, *cf.* Subsection 5.3.3) until it becomes empty. However, it does not check whether the future of a state has already been explored or not: if it reaches a state twice, then it computes its evolution twice. Moreover, our implementation only computes reachability analysis over a bounded time interval whose expected final time is given as parameter of the simulation engine.

First, we have to instantiate the model with the chosen representation of sets. In our case, we decided to use Taylor models as representation of sets, which has been implemented as a module `TM` with a compatible signature. The instantiated module is named `NodeSet`.

Then, we have to instantiate the initial state of the simulation. We start creating three arrays of sets (each of type `TM.t array`), two of size `NodeSet.csize` for the continuous variables and their derivatives and one of size `NodeSet.zsize` for the values of the zero-crossing signals. We also create an array of size `NodeSet.zsize` with boolean elements for the zero-crossing events. With these four arrays, we can call to function `alloc`:

```
let alloc_state () =
  let carray = Array.make NodeSet.csize TM.zero in
  let darray = Array.make NodeSet.csize TM.zero in
  let zoarray = Array.make NodeSet.zsize TM.zero in
  let ziarray = Array.make NodeSet.zsize false in
  NodeSet.alloc 0 carray darray 0 zoarray ziarray
```

We thereby obtain an object of type `NodeSet.state` that can be initialized calling the function `NodeSet.init` and we can create the initial state of the simulation (`ts0` for *timed-state*) as a record. We define by convention the initial time of the simulation to be equal to zero. We also create an array of boolean values, all set to `false`, in order to encode the array `a`.

```
let s0 = alloc_state () in
let () = NodeSet.init s0 in
let ts0 = {
    time = TM.zero;
    state = s0;
    az = Array.make NodeSet.zsize false;
  } in
```

Finally, the program iterates over a collection of states of the simulation until it becomes empty in order to compute and to return a list of set-valued functions that return over-approximations of the reachable states depending on the time (lists of records

of type `dense` in the code that gather an initial set-valued time, a concrete maximal elapsed time `dt` of type `float` and a function from `[0,dt]` to the set of set-valued states). Such a collection is called *worklist* and it is implemented in a module `WK`. If the worklist is empty, then the program simply returns the list of set-valued functions. Otherwise, it removes an element from the worklist. If its time is bigger than the expected final one, the element is dropped. Otherwise, the program computes its continuous evolution, calling the continuous solver `csolve`, and computes all possible discrete evolutions, depending on the zero-crossing activations returned by the detector of events `zsolve`. The continuous evolution is appended to the resulting list of set-valued functions and all the new states after discrete evolutions are appended to the worklist for the following iterations.

Instead of printing the resulting collection of set-valued functions, we print a box over-approximation of the continuous evolution at each iteration into the standard output that we convert to the CSV format and plot the result using a script written in Python3.

5.3.5 Continuous evolutions

The solver of continuous dynamics is implemented using Taylor models (*cf.* Subsection 2.4.2). First, it computes a polynomial expansion of the dynamics calling the Picard operator associated to the system of ordinary differential equations until it reaches a fixed point. At that point, it manipulates an array of Taylor models whose polynomial parts define a fixed-point of the Picard operator. Then, it enlarges the remainders until they are contracted by the Picard operator.

This solver requires a function that returns the derivatives from a value of the continuous variables and the time (*cf.* Section 5.1). Due to the interface of the intermediate model, we have to introduce an extra function that updates the state with the values of the continuous variables, computes the derivatives and returns the array of their values:

```
let deriv s (c,t) =
  let () = NodeSet.set_carray s c in
  let () = NodeSet.deriv s in
  NodeSet.get_darray s
```

Reciprocally, the continuous solver returns a function that maps (set-valued) times to arrays of (set-valued) continuous variables and we introduce another function that converts the result as a copy of the state in which the values of the continuous variables have been updated with the new values:

```
let csolve s c h =
  CSolver.csolve (deriv s) c TM.zero h
```

So, instead of calling the continuous solver, the top-level algorithm calls a function `integrate` that takes as input the state of the model and a duration and returns a function that maps a set-valued time to a set-valued state:

```
let integrate s h =
  let c = NodeSet.get_carray s in
  let denseC = csolve s c h in
  let sRef = NodeSet.copy s in
  fun t ->
  let s = NodeSet.copy sRef in
  let carray = denseC t in
  let () = NodeSet.set_carray s carray in
  s
```

5.3.6 Discrete evolutions

The detector of events is implemented using dichotomy on the time interval in order to detect the first time and the last one whose image contains zero, for each zero-crossing signal. It returns a list of pairs, whose first element is a list of the indices of the activated zero-crossings and the second one a time interval on which those zero-crossings are activated.

This implementation does not match the interface of $\tilde{zsolve}$ presented in Section 5.1.3. So, another function computes an over-approximation of the values of the continuous variables over the time intervals returned by the detector of events, checks whether at least one of the zero-crossing is complete (*cf.* Subsection 5.3.3) and, before which the array `zin` is set to appropriate values, computes the image of the state after a discrete step (function `update` of the model). If only one zero-crossing event is activated, then the corresponding component of the array `zin` is set to true. If multiple zero-crossing events are activated, then a discrete step is computed for each possible subset of the activated events.

Moreover, this function updates the values of the array `a` before any call to the detector of events as presented in Subsection 5.3.3. After every discrete step, the array `a` of boolean values is reset to an array of false values, because a discrete step may *a priori* introduce discontinuities in a zero-crossing signal. Finally, if no zero-crossing event is complete, then the set-valued state at the last instant of the approximation is appended to the following possible states with its current array `a` of boolean values.

Such a function is called `handle_zeroc` in the code and it takes as argument the current array `a` of boolean values, the initial set-valued time `t0`, the set-valued approximation of the continuous dynamics `dense` and the duration `h` (defining the domain of `dense` as $[0, h]$). It returns a list of approximations with their associated domains (in our current implementation, it is reduced to a singleton of the function `dense` on the interval $[0, h]$ with initial uncertain time `t0`) and a list of all possible states of the simulation after the discrete behaviors, possibly containing ones that did not activate any zero-crossing events (at the last instant of the continuous dynamics).

5.3.7 Application to an example

We only test our implementation of such a set-valued computation on a small model of a counter in parallel with a bouncing ball, because we have not yet modified the Zélus

compiler to produce intermediate models parameterized by a representation of sets. The intermediate model is implemented following the structure of the initial model, *i.e.* a main node contains an instance of the counter node and an instance of the bouncing ball node. We implement every node of ZÉLUS as a functor in OCAML as presented in Subsection 5.3.2.

The first node of our model is the bouncing ball. While it can be implemented in ZÉLUS as a hybrid automaton with a unique mode and a unique transition activated when the height of the ball crosses zero, this results in a chattering effect during the set-valued computation: if the transition is activated, then it is possibly still activated after the discrete step, because the height is not modified and stays in the neighborhood of zero. So, we decide to implement the bouncing ball as an automaton with two modes: a mode `Up` in which the velocity is non-negative and a mode `Down` in which it is non-positive. The mode is registered in a field `mode` of the state. In both modes, the derivative of the height `h` is equal to the velocity `v` and the derivative of the velocity is equal to `-9.81`, *i.e.* an approximation of the acceleration due to gravity on the surface of the Earth. In order to trigger the switching between the two modes, we need two zero-crossing signals: one equal to `-.h` to detect bounces from `Down` to `Up` and another equal to `-.v` to detect switching from `Up` to `Down`. To reduce the number of false detections of zero-crossing events, those signals are set to (arbitrary) strictly negative values in the modes in which they are not relevant: the first signal is equal to `-.h` in the mode `Down` and to `-42.` in the mode `Up`, while the second is equal to `-.v` in the mode `Up` and to `-42.` in the mode `Down`. The discrete steps switch the value of `mode` and, in case of a switching from `Down` to `Up`, the value of `v` is updated to `0.8 *. v`. It could be written in ZÉLUS as follow:

```
let hybrid ball(h0,v0) = (h,v) where
    rec zh = up(-.h)
    and zv = up(-.v)
    and automaton
        | Down(h0,v0) -> do
              der h = v init h0
          and der v = -1. init v0
          until zh then Up(h, -0.8 *. v)
        | Up(h0,v0) -> do
              der h = v init h0
          and der v = -1. init v0
          until zv then Down(h, v)
        init Down(h0,v0)
```

The second node is a counter incremented every second, *i.e.* a timer. It is composed of two continuous variables (`t` for the time and `dt` for the elapsed time since the previous update) and a discrete one (`count` of type `int` for counting the number of elapsed seconds). All the derivatives are equal to one. A unique zero-crossing signal is equal to `dt-.1.` and the associated discrete step resets `dt` to zero and increments the value of `count`. I could be written in ZÉLUS as follow:

```
let hybrid timer() = (t,dt,count) where
    rec der t = 1. init 0.
    and der dt = 1. init 0. reset tic -> 0.
    and tic = up(dt -. 1.)
    and init count = 0
    and present(tic) -> do count = (last count) + 1 done
```

A third node simply gathers the two others to define the parallel composition. Its state is only composed of the instantiations of the states of the two other nodes. Its functions call the corresponding functions of the two other nodes with correct offsets for the modifications of the global arrays `carray`, `darray`, `zout` and `zin`. It could be written in ZÉLUS as follow:

```
let hybrid simu() = (t,dt,count,h,v) where
    rec h,v = ball(20.,0.)
    and t,dt,count = timer()
```

Executing our prototype on this example with a final time `tEnd = 10.` and a time-step `dt = 0.1` for the integration, and after plotting the result using our PYTHON3 script, we obtain the Figure 5.3. In this model, no uncertainties are introduced in the initial state, they only occur during the integration of differential equations and the detection of zero-crossing events. We notice that multiple set-valued states coexist simultaneously, for example at the time 9 for which `count` is equal to 8 or 9 during a small duration. We also notice that our algorithm is able to effectively eliminate non-relevant dynamics via the array `a` of boolean values, because `dt` does not reach the value 2 and `h` does not reach the value -2 on the time interval $[0, 10]$. However, multiple set-valued trajectories diverge from a time approximatively equal to 10, as illustrated by the value of `h` that reaches strictly negative values.

While our prototype could be highly improved, it already illustrates that we can simply overload the operators on the continuous variables (*i.e.* of type `float`) in a ZÉLUS model that does not contain non-continuous functions in order to compute reachability analysis on a bounded time interval. It also illustrates that the handling of uncertainties on the zero-crossing events can be achieve without introducing extra constructions to the ZÉLUS language but with an external array of boolean values (*cf.* Subsection 5.3.3).

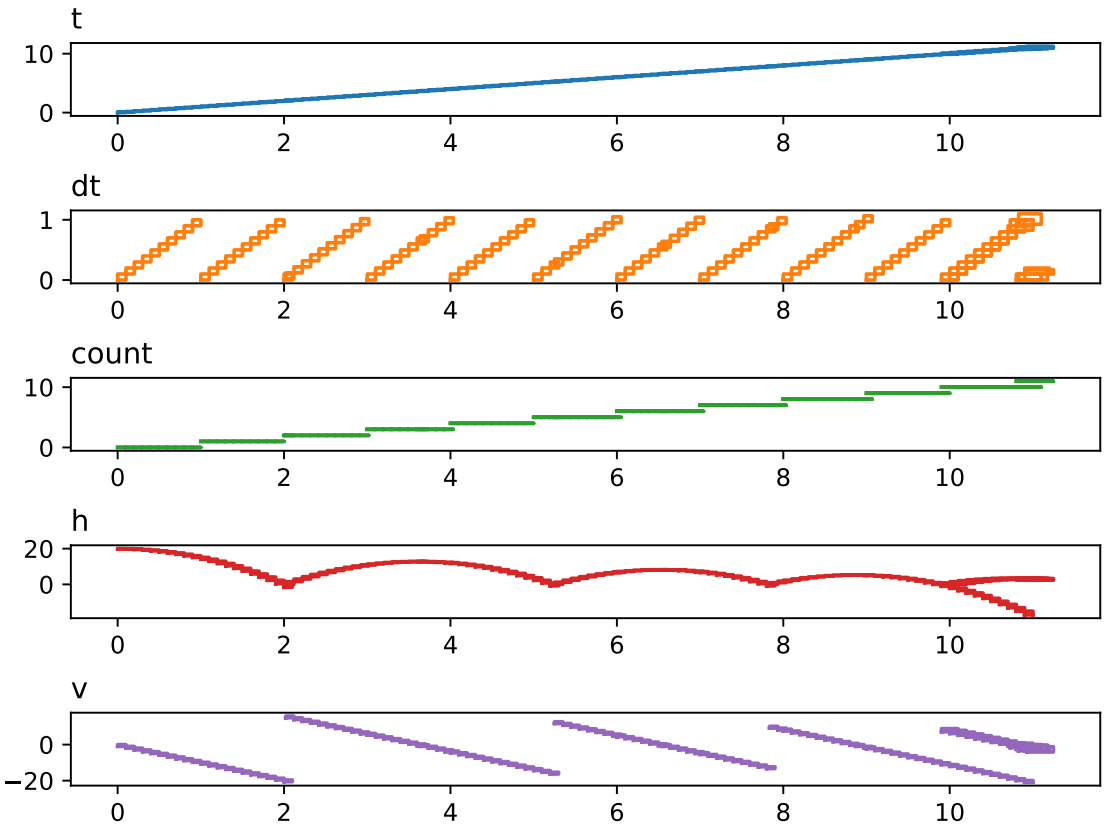


Figure 5.3 – Over-approximation with respect to the time computed by our prototype on an intermediate model similar to the ones produced by the Zélus compiler

Chapter 6

Conclusion

6.1 Summary

In the case of initial value problems in the presence of Lebesgue-integrable uncertainties, we proved that a set-valued function that is contracted by a set-valued Picard operator is an over-approximation of the sets of reachable states (*cf.* Section 3.2). We also proposed an algorithm based on decomposition into a difference of non-negative functions of the right-hand sides of ordinary differential equations (*cf.* Section 3.3) in order to compute tighter over-approximations of the set of reachable states in the case of separable systems, a generalization of the input-affine systems. We implemented a prototype and we showed that our prototype indeed produces tighter over-approximation than some state-of-the-art tools on some examples (*cf.* Chapter 4).

In the case of switched systems, we proposed an algorithm to compute an over-approximation of the set of reachable states in the presence of possible Zeno behaviors (*cf.* Section 3.4). We illustrated it on an example, but we did not provide any implementation of it. So, we cannot currently evaluate its efficiency with respect to state-of-the-art tools. Its direct application to hybrid automata failed as presented in the following section.

Finally, we presented our current reflection with Marc Pouzet about the integration of set-valued reachability analysis into tools designed to simulate hybrid models. We presented the difficulties introduced by the set-valued computations. We proposed interfaces between the models and the different solvers with only slight modifications of the concrete interfaces, *i.e.* the ones designed for classic simulations. We also proposed a method based on parametrization of intermediate models to easily convert concrete models into set-valued ones. We implemented a prototype to illustrate this method on an example using functors in OCaml. We want to apply it to models written in Zélus that are compiled into intermediate models in OCaml, but it requires to adapt the generated code to allow parametrization: the intermediate models should be parameterized by a representation of continuous values (*e.g.* intervals or Taylor models) in order to redefine the associated operators. Such an implementation is left for future work.

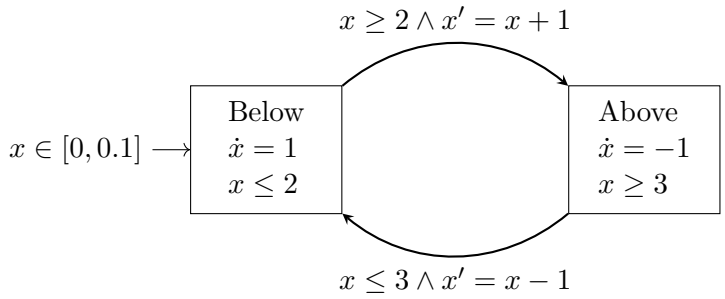
6.2 Difficulties about hybrid automata

In Chapter 3, we presented how to compute over-approximations of sets of reachable states of switched systems in the presence of possible Zeno behaviors (*cf.* Subsection 2.1.2).

In the case of hybrid automata, we no large assume that the jumps are identity functions and that the interiors of the sets defined by the invariants are disjoint. Moreover, even if a set of states activates a unique guard, the successive jumps at the same real time may result in a Zeno behavior.

So, we have to detect the set of states that is a fixed-point of the compositions of jumps of the successively activated transitions. The sets of states that do not contain such a fixed-point produce no Zeno behaviors of the first kind, *i.e.* an infinite number of transitions in zero elapsed time (*cf.* Subsection 2.1.2). On the contrary, if a set of states activates an infinite sequence of transitions in zero elapsed time, then we want to interpret its differential equations in the sense of Filippov. It is illustrated in Example 25.

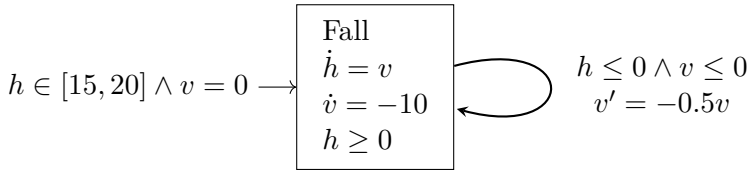
Example 25 (Simple jump with bang-bang controller)
Consider a bang-bang controller similar to the one presented in Example 3 but with an artificial jump of the position:



Considering the initial time $t = 0$, the guard is activated by the state $x = 2$ at some time $t \in [1.9, 2]$ in the mode Below. Its image through the activated transition is $x = 3$ with the same set of time $t \in [1.9, 2]$ in the mode Above. At that point, the other guard is activated and results in the state $x = 2$ in the mode Below, which is the first state activating a guard. So, we detected a fixed-point set at time $t \in [1.9, 2]$ composed of $x = 2$ in the mode Below and $x = 3$ in the mode Above. We could at that point apply on both triples $([1.9, 2], \text{Below}, [2])$ and $([1.9, 2], \text{Above}, [3])$ the same method as in the case of switched systems using a differential inclusion as the convex hull of the dynamics associated to the modes Below and Above. This would introduce some chattering, but the resulting set-valued functions would contain the admissible trajectories in the sense of Filippov, *i.e.* for all time, $x = 2$ in Below or $x = 3$ in Above.

However, the second kind of Zeno behaviors, *i.e.* an infinite number of transitions in a bounded time interval that are not reduced to a single instant (*cf.* Subsection 2.1.2), cannot be handled by simply detecting fixed-points, as illustrated in Example 26.

Example 26 (Reachability analysis of a bouncing ball)
Consider the bouncing ball model presented in Example 4. We introduce some uncertainties on the initial condition:



Considering the initial time $t = 0$, the guard is activated by a state $x = 0 \wedge v \in \left[-20, -10\sqrt{3}\right]$ at some time $t \in \left[\sqrt{3}, 2\right]$. Its image through the activated transition is then $x = 0 \wedge v \in \left[5\sqrt{3}, 10\right]$, which does not activate any guards. In that case, the base algorithm presented in Subsection 3.4.1 is able to compute a valid over-approximation of the set of admissible trajectories.

However, after some iterations and due to the over-approximation of all the operators, especially the function *integrate*, we may obtain some set $x = 0 \wedge v \in [\underline{v_k}, \overline{v_k}]$ that activates the guard and with $0 \in [\underline{v_k}, \overline{v_k}]$. Its image through the activated transition is then $x = 0 \wedge v \in [-0.5\overline{v_k}, -0.5\underline{v_k}]$, which also activates the guard. So, we detect a potential Zeno point. We can try to compute the smallest set of velocities $[\underline{v_\infty}, \overline{v_\infty}]$ such that $[\underline{v_\infty}, \overline{v_\infty}] \subset [\underline{v_k}, \overline{v_k}]$ and that is stable through the transition, *i.e.* $[-0.5\overline{v_\infty}, -0.5\underline{v_\infty}] \subset [\underline{v_\infty}, \overline{v_\infty}]$. Such a set is $[\underline{v_\infty}, \overline{v_\infty}] = \{0\}$. So, we identify a Zeno point. Unfortunately, the convex hull of the relevant dynamics is equal to the single dynamics in the mode Fall. So, an interpretation in the sense of Filippov from the state $x = 0 \wedge v = 0$ would still result in a set-valued function that becomes strictly negative. This proves that the method presented in the previous Section 3.4 is not adapted to such a hybrid automaton.

A method to compute over-approximations of the reachable set of such systems exists [95, 94], but it cannot terminate if the time is part of the system or if the system depends on time-varying inputs, because the time cannot reach a fixed-point: its derivative is always equal to one.

At that point of our work, we do not have any satisfactory solution to handle hybrid automata without restricting our study to a subclass of models, *e.g.* assuming that all jumps are identity functions as in the case of switched systems. The study of over-approximations of hybrid automata in the presence of possible Zeno behaviors with time-varying uncertainties in the guard conditions is left for a future work.

6.3 Possible extensions

The work presented in this thesis may be extended in different ways.

First, the reflection about the inclusion of reachability analysis into tools for simulations of models of hybrid systems can be continued in order to integrate such a functionality into the ZÉLUS language. It requires a modification of the compiler to gen-

erate parameterized intermediate models and a definition of uncertainties in the Zélus language, *e.g.* introducing new keywords or directives. Moreover, such reachability analyses may be used to check dynamically some assertions written in the models.

Second, we could improve the proposed algorithm to compute over-approximations of the sets of reachable states of separable systems in order to keep more dependencies between the variables. Indeed, our current method handles every component of the states independently. In case of huge uncertainties, *i.e.* wide range of the time-varying uncertainties, it seems to result in over-approximations as boxes (*cf.* Figure 4.3). In order to test on larger systems, the prototype has to be improved, possibly using existing tools for the computation of the dynamics with only constant uncertainties and experimenting different representations of sets.

Finally, as presented in the Section 6.2, extra work is required in order to adapt our method for hybrid automata with Lebesgue-integrable uncertainties and guards depending on the time.

Bibliography

[1] Aditya Agrawal, Gyula Simon, and Gabor Karsai. "Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations". In: *Electronic Notes in Theoretical Computer Science* 109 (2004), pp. 43–56.

[2] Matthias Althoff. "An introduction to CORA 2015". In: *Proc. of the workshop on applied verification for continuous and hybrid systems*. 2015, pp. 120–151.

[3] Matthias Althoff. "Reachability analysis and its application to the safety assessment of autonomous cars". PhD thesis. Technische Universität München, 2010.

[4] Matthias Althoff. "Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets". In: *Proceedings of the 16th international conference on Hybrid systems: computation and control*. 2013, pp. 173–182.

[5] Matthias Althoff and Goran Frehse. "Combining zonotopes and support functions for efficient reachability analysis of linear systems". In: *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE. 2016, pp. 7439–7446.

[6] Matthias Althoff, Goran Frehse, and Antoine Girard. "Set propagation techniques for reachability analysis". In: *Annual Review of Control, Robotics, and Autonomous Systems* 4 (2021), pp. 369–395.

[7] Matthias Althoff and Dmitry Grebenyuk. "Implementation of interval arithmetic in {CORA} 2016". In: *Proc. of the 3rd International Workshop on Applied Verification for Continuous and Hybrid Systems*. 2016.

[8] Matthias Althoff, Dmitry Grebenyuk, and Niklas Kochdumper. "Implementation of Taylor models in CORA 2018". In: *Proc. of the 5th International Workshop on Applied Verification for Continuous and Hybrid Systems*. 2018.

[9] Matthias Althoff, Niklas Kochdumper, and Mark Wetzlinger. *CORA 2020 Manual*. URL: `https://tumcps.github.io/CORA/data/Cora2020Manual.pdf`.

[10] Matthias Althoff, Niklas Kochdumper, and Mark Wetzlinger. *CORA 2021 Manual*. URL: `https://tumcps.github.io/CORA/data/Cora2021Manual.pdf`.

[11] Matthias Althoff and Bruce H Krogh. "Zonotope bundles for the efficient computation of reachable sets". In: *2011 50th IEEE conference on decision and control and European control conference*. IEEE. 2011, pp. 6814–6821.

[12] Matthias Althoff, Colas Le Guernic, and Bruce H Krogh. “Reachable set computation for uncertain time-varying linear systems”. In: *Proceedings of the 14th international conference on Hybrid systems: computation and control*. 2011, pp. 93–102.

[13] Matthias Althoff, Olaf Stursberg, and Martin Buss. “Reachability analysis of linear systems with uncertain parameters and inputs”. In: *2007 46th IEEE Conference on Decision and Control*. IEEE. 2007, pp. 726–732.

[14] Matthias Althoff, Olaf Stursberg, and Martin Buss. “Reachability analysis of nonlinear systems with uncertain parameters using conservative linearization”. In: *2008 47th IEEE Conference on Decision and Control*. IEEE. 2008, pp. 4042–4048.

[15] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, P-H Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. “The algorithmic analysis of hybrid systems”. In: *Theoretical computer science* 138.1 (1995), pp. 3–34.

[16] Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. “Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems”. In: *Hybrid systems*. Springer, 1992, pp. 209–229.

[17] Eugene Asarin, Olivier Bournez, Thao Dang, and Oded Maler. “Approximate reachability analysis of piecewise-linear dynamical systems”. In: *International workshop on hybrid systems: Computation and control*. Springer. 2000, pp. 20–31.

[18] Eugene Asarin, Thao Dang, and Antoine Girard. “Hybridization methods for the analysis of nonlinear systems”. In: *Acta Informatica* 43.7 (2007), pp. 451–476.

[19] Eugene Asarin, Thao Dang, and Oded Maler. “d/dt: A tool for reachability analysis of continuous and hybrid systems”. In: *IFAC Proceedings Volumes* 34.6 (2001), pp. 741–746.

[20] Leonhard Asselborn, Dominic Gross, and Olaf Stursberg. “Control of uncertain nonlinear systems using ellipsoidal reachability calculus”. In: *IFAC Proceedings Volumes* 46.23 (2013), pp. 50–55.

[21] J-P Aubin and Arrigo Cellina. *Differential inclusions: set-valued maps and viability theory*. Vol. 264. Springer Science & Business Media, 2012.

[22] David Avis, David Bremner, and Raimund Seidel. “How good are convex hull algorithms?” In: *Computational Geometry* 7.5-6 (1997), pp. 265–301.

[23] Stanley Bak and Parasara Sridhar Duggirala. “Simulation-equivalent reachability of large linear systems with inputs”. In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 401–420.

[24] Andrea Balluchi, Alberto Casagrande, Pieter Collins, Alberto Ferrari, Tiziano Villa, and Alberto L Sangiovanni-Vincentelli. “Ariadne: a framework for reachability analysis of hybrid automata”. In: *In: Proceedings of the International Syposium on Mathematical Theory of Networks and Systems.(2006*. Citeseer. 2006.

[25] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. “Reactive probabilistic programming”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 898–912.

[26] Guillaume Baudart, Louis Mandel, Marc Pouzet, and Reyyan Tekin. “Inférence parallèle pour un langage réactif probabiliste”. In: *33èmes Journées Francophones des Langages Applicatifs*. 2022.

[27] Frédéric Benhamou and Laurent Granvilliers. “Continuous and interval constraints”. In: *Foundations of Artificial Intelligence* 2 (2006), pp. 571–603.

[28] Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet. “A Type-Based Analysis of Causality Loops in Hybrid Modelers”. In: *17th International Conference on Hybrid Systems: Computation and Control (HSCC'14)*. Berlin, Germany, Apr. 2014, pp. 71–82. URL: `http://zelus.di.ens.fr/hscc2014/fullpaper.pdf`.

[29] Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet. “A type-based analysis of causality loops in hybrid systems modelers”. In: *Nonlinear Analysis: Hybrid Systems* 26 (2017), pp. 168–189.

[30] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. “Divide and recycle: types and compilation for a hybrid synchronous language”. In: *ACM SIGPLAN Notices* 46.5 (2011), pp. 61–70.

[31] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. “Non-standard semantics of hybrid systems modelers”. In: *Journal of Computer and System Sciences* 78.3 (2012), pp. 877–910.

[32] Albert Benveniste, Timothy Bourkey, Benoît Caillaud, and Marc Pouzet. “A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code”. In: *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*. IEEE. 2011, pp. 137–148.

[33] Luca Benvenuti, Davide Bresolin, Alberto Casagrande, Pieter Collins, Alberto Ferrari, Emanuele Mazzi, Alberto Sangiovanni-Vincentelli, and Tiziano Villa. “Reachability computation for hybrid systems with Ariadne”. In: *IFAC Proceedings Volumes* 41.2 (2008), pp. 8960–8965.

[34] Martin Berz and Kyoko Makino. “Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models”. In: *Reliable computing* 4.4 (1998), pp. 361–369.

[35] François Bidet, Éric Goubault, and Sylvie Putot. “Reachability Analysis of Generalized Input-Affine Systems with Bounded Measurable Time-varying Uncertainties”. In: *IEEE Control Systems Letters* (2021).

[36] François Bidet, Éric Goubault, and Sylvie Putot. “Work in Progress: Reachability Analysis for Time-triggered Hybrid Systems, The Platoon Benchmark (short paper).” In: *CICM Workshops*. 2018.

[37] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A static analyzer for large safety-critical software". In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 2003, pp. 196–207.

[38] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Frédéric Viry, Andreas Podelski, and Christian Schilling. "Reach set approximation through decomposition with low-dimensional sets and high-dimensional matrices". In: *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*. 2018, pp. 41–50.

[39] Olivier Bouissou and Alexandre Chapoutot. "An operational semantics for Simulink's simulation engine". In: *ACM SIGPLAN Notices* 47.5 (2012), pp. 129–138.

[40] Timothy Bourke, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. "A synchronous look at the Simulink standard library". In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (2017), pp. 1–24.

[41] Timothy Bourke and Marc Pouzet. "Zélus: A synchronous language with ODEs". In: *Proceedings of the 16th international conference on Hybrid systems: computation and control*. 2013, pp. 113–118.

[42] David Broman, Lev Greenberg, Edward A Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. "Requirements for hybrid cosimulation standards". In: *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. 2015, pp. 179–188.

[43] Jason Brown and François Pessaux. "Interval-Based Simulation of Zélus IVPs Using DynIbex". In: *Acta Cybernetica* (2020), pp. 1–16.

[44] Gilles Chabert and Luc Jaulin. "Contractor programming". In: *Artificial Intelligence* 173.11 (2009), pp. 1079–1100.

[45] Xin Chen. "Reachability analysis of non-linear hybrid systems using taylor models". PhD thesis. Fachgruppe Informatik, RWTH Aachen University, 2015.

[46] Xin Chen. *Users Manual of Flow**. URL: `https://www.cs.colorado.edu/~xich8622/manual/manual-2.0.0.pdf`.

[47] Xin Chen, Erika Abraham, and Sriram Sankaranarayanan. "Taylor model flowpipe construction for non-linear hybrid systems". In: *2012 IEEE 33rd Real-Time Systems Symposium*. IEEE. 2012, pp. 183–192.

[48] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. "Flow*: An analyzer for non-linear hybrid systems". In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 258–263.

[49] Xin Chen and Sriram Sankaranarayanan. "Decomposed reachability analysis for nonlinear systems". In: *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2016, pp. 13–24.

[50] Pieter Collins, Davide Bresolin, Luca Geretti, and Tiziano Villa. “Computing the evolution of hybrid systems using rigorous function calculus”. In: *IFAC Proceedings Volumes* 45.9 (2012), pp. 284–290.

[51] Patrick Cousot and Radhia Cousot. “Abstract interpretation frameworks”. In: *Journal of logic and computation* 2.4 (1992), pp. 511–547.

[52] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252.

[53] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “The ASTRÉE analyzer”. In: *European Symposium on Programming*. Springer. 2005, pp. 21–30.

[54] Thao Dang and Oded Maler. “Reachability analysis via face lifting”. In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 1998, pp. 96–109.

[55] Jorge Davila, Leonid Fridman, and Arie Levant. “Second-order sliding-mode observer for mechanical systems”. In: *IEEE transactions on automatic control* 50.11 (2005), pp. 1785–1789.

[56] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. “Towards an industrial use of FLUCTUAT on safety-critical avionics software”. In: *International Workshop on Formal Methods for Industrial Critical Systems*. Springer. 2009, pp. 53–69.

[57] Alexandre Donzé and Oded Maler. “Systematic simulation using sensitivity analysis”. In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2007, pp. 174–189.

[58] Parasara Sridhar Duggirala and Mahesh Viswanathan. “Parsimonious, simulation based verification of linear systems”. In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 477–494.

[59] Magnus Egerstedt. “Behavior based robotics using hybrid automata”. In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2000, pp. 103–116.

[60] Gidon Ernst, Paolo Arcaini, Ismail Bennani, Aniruddh Chandratre, Alexandre Donze, Georgios Fainekos, Goran Frehse, Khouloud Gaaloul, Jun Inoue, Tanmay Khandait, et al. “ARCH-COMP 2021 Category Report: Falsification with Validation of Results”. In: *EPiC Series in Computing* 80 (2021), pp. 133–152.

[61] Gidon Ernst, Paolo Arcaini, Ismail Bennani, Alexandre Donze, Georgios Fainekos, Goran Frehse, Logan Mathesen, Claudio Menghi, Giulia Pedrinelli, Marc Pouzet, et al. “Arch-comp 2020 category report: Falsification”. In: *EPiC Series in Computing* (2020).

[62] Andrey A Fedotov and Valerii S Patsko. “Investigation of reachable set at instant for the Dubins car”. In: *58th Israel annual conference on aerospace sciences, IACAS 2018*. 2018, pp. 1655–1669.

[63] A. F. Filippov. *Differential Equations with Discontinuous Righthand Sides*. en. Ed. by F. M. Arscott and M. Hazewinkel. Vol. 18. Mathematics and Its Applications. Dordrecht: Springer Netherlands, 1988. ISBN: 978-90-481-8449-1 978-94-015-7793-9. DOI: 10.1007/978-94-015-7793-9. URL: http://link.springer.com/10.1007/978-94-015-7793-9 (visited on 12/20/2020).

[64] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. “SpaceEx: Scalable verification of hybrid systems”. In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 379–395.

[65] Goran Frehse, Rajat Kateja, and Colas Le Guernic. “Flowpipe approximation and clustering in space-time”. In: *Proceedings of the 16th international conference on Hybrid systems: computation and control*. 2013, pp. 203–212.

[66] Goran Frehse and Rajarshi Ray. “Flowpipe-guard intersection for reachability computations with support functions”. In: *IFAC Proceedings Volumes* 45.9 (2012), pp. 94–101.

[67] *Functional Mock-up Interface for Model Exchange and Co-Simulation*. 2021.

[68] Jürgen Garloff, Antek Schabert, and Andrew P Smith. “Bounds on the range of multivariate rational functions”. In: *PAMM* 12.1 (2012), pp. 649–650.

[69] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. “A logical product approach to zonotope intersection”. In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 212–226.

[70] Antoine Girard. “Reachability of uncertain linear systems using zonotopes”. In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2005, pp. 291–305.

[71] Rafal Goebel, Ricardo G. Sanfelice, and Andrew R. Teel. “Hybrid dynamical systems”. In: *IEEE Control Systems Magazine* 29.2 (2009), pp. 28–93. DOI: 10.1109/MCS.2008.931718.

[72] Sanja Zivanovic Gonzalez, Pieter Collins, Luca Geretti, Davide Bresolin, and Tiziano Villa. “Higher Order Method for Differential Inclusions”. In: *arXiv preprint arXiv:2001.11330* (2020).

[73] Eric Goubault, Olivier Mullier, Sylvie Putot, and Michel Kieffer. “Inner approximated reachability analysis”. In: *Proceedings of the 17th international conference on Hybrid systems: computation and control*. 2014, pp. 163–172.

[74] Eric Goubault and Sylvie Putot. “A zonotopic framework for functional abstractions”. In: *arXiv preprint arXiv:0910.1763* (2009).

[75] Eric Goubault and Sylvie Putot. “Forward inner-approximated reachability of non-linear continuous systems”. In: *Proceedings of the 20th international conference on hybrid systems: computation and control*. 2017, pp. 1–10.

[76] Eric Goubault and Sylvie Putot. “Inner and outer reachability for the verification of control systems”. In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 2019, pp. 11–22.

[77] Eric Goubault and Sylvie Putot. “Static analysis of numerical algorithms”. In: *International Static Analysis Symposium*. Springer. 2006, pp. 18–34.

[78] Eric Goubault and Sylvie Putot. “Under-approximations of computations in real numbers based on generalized affine arithmetic”. In: *International Static Analysis Symposium*. Springer. 2007, pp. 137–152.

[79] Eric Goubault, Sylvie Putot, and Lorenz Sahlmann. “Inner and outer approximating flowpipes for delay differential equations”. In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 523–541.

[80] Vineet Gupta, Thomas A Henzinger, and Radha Jagadeesan. “Robust timed automata”. In: *International Workshop on Hybrid and Real-Time Systems*. Springer. 1997, pp. 331–345.

[81] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. “The synchronous data flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[82] Thomas A Henzinger. “The theory of hybrid automata”. In: *Verification of digital and hybrid systems*. Springer, 2000, pp. 265–292.

[83] Thomas A Henzinger, Benjamin Horowitz, Rupak Majumdar, and Howard Wong-Toi. “Beyond HYTECH: Hybrid systems analysis using interval numerical methods”. In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2000, pp. 130–144.

[84] Thomas A Henzinger, Peter W Kopke, Anuj Puri, and Pravin Varaiya. “What’s decidable about hybrid automata?” In: *Journal of computer and system sciences* 57.1 (1998), pp. 94–124.

[85] Marcus J Holzinger and Daniel J Scheeres. “Reachability results for nonlinear systems with ellipsoidal initial sets”. In: *IEEE transactions on aerospace and electronic systems* 48.2 (2012), pp. 1583–1600.

[86] Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. “The Lustre V6 reference manual”. In: *Verimag, Grenoble, Dec* (2016).

[87] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. “Interval analysis”. In: *Applied interval analysis*. Springer, 2001, pp. 11–43.

[88] Karl Henrik Johansson, Magnus Egerstedt, John Lygeros, and Shankar Sastry. “On the regularization of Zeno hybrid automata”. In: *Systems & control letters* 38.3 (1999), pp. 141–150.

[89] Morgan Jones and Matthew M Peet. “Using SOS and sublevel set volume minimization for estimation of forward reachable sets”. In: *IFAC-PapersOnLine* 52.16 (2019), pp. 484–489.

[90] Tomasz Kapela and Piotr Zgliczyski. “A Lohner-type algorithm for control systems and ordinary differential inclusions”. In: *arXiv preprint arXiv:0712.0910* (2007).

[91] Nikolaos Kekatos, Marcelo Forets, and Goran Frehse. “Constructing verification models of nonlinear Simulink systems via syntactic hybridization”. In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE. 2017, pp. 1788–1795.

[92] Niklas Kochdumper and Matthias Althoff. “Constrained polynomial zonotopes”. In: *arXiv preprint arXiv:2005.08849* (2020).

[93] Niklas Kochdumper and Matthias Althoff. “Sparse polynomial zonotopes: A novel set representation for reachability analysis”. In: *IEEE Transactions on Automatic Control* (2020).

[94] Michal Konen, Walid Taha, Ferenc A Bartha, Jan Duracz, Adam Duracz, and Aaron D Ames. “Enclosing the behavior of a hybrid automaton up to and beyond a Zeno point”. In: *Nonlinear Analysis: Hybrid Systems* 20 (2016), pp. 1–20.

[95] Michal Konen, Walid Taha, Jan Duracz, Adam Duracz, and Aaron Ames. “Enclosing the behavior of a hybrid system up to and beyond a zeno point”. In: *2013 IEEE 1st international conference on cyber-physical systems, networks, and applications (CPSNA)*. IEEE. 2013, pp. 120–125.

[96] Wolfgang Kühn. “Rigorously computed orbits of dynamical systems without the wrapping effect”. In: *Computing* 61.1 (1998), pp. 47–67.

[97] Alexander B Kurzhanski and Pravin Varaiya. “Ellipsoidal techniques for reachability analysis”. In: *International workshop on hybrid systems: Computation and control*. Springer. 2000, pp. 202–214.

[98] Alex A Kurzhanskiy and Pravin Varaiya. “Ellipsoidal techniques for reachability analysis of discrete-time linear systems”. In: *IEEE Transactions on Automatic Control* 52.1 (2007), pp. 26–38.

[99] Colas Le Guernic and Antoine Girard. “Reachability analysis of hybrid systems using support functions”. In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 540–554.

[100] Colas Le Guernic and Antoine Girard. “Reachability analysis of linear systems using support functions”. In: *Nonlinear Analysis: Hybrid Systems* 4.2 (2010), pp. 250–262.

[101] Edward A Lee and Haiyang Zheng. “Operational semantics of hybrid systems”. In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2005, pp. 25–53.

[102] Daniel Liberzon. *Switching in systems and control*. Springer Science & Business Media, 2003.

[103] Xiaojun Liu, Eleftherios Matsikoudis, and Edward A Lee. "Modeling timed concurrent systems". In: *International Conference on Concurrency Theory*. Springer. 2006, pp. 1–15.

[104] John Lygeros. *Hierarchical, hybrid control of large-scale systems*. University of California, Berkeley, 1996.

[105] Moussa Maiga, Nacim Ramdani, Louise Travé-Massuyès, and Christophe Combastel. "A comprehensive method for reachability analysis of uncertain nonlinear hybrid systems". In: *IEEE Transactions on Automatic Control* 61.9 (2015), pp. 2341–2356.

[106] Kyoko Makino and Martin Berz. "Suppression of the wrapping effect by Taylor model-based verified integrators: The single step". In: *International Journal of Pure and Applied Mathematics* 36.2 (2007), p. 175.

[107] Kyoko Makino and Martin Berz. "Taylor models and other validated functional inclusion methods". In: *International Journal of Pure and Applied Mathematics* 6 (2003), pp. 239–316.

[108] Oded Maler, Zohar Manna, and Amir Pnueli. "From timed to hybrid systems". In: *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer. 1991, pp. 447–484.

[109] Karthik Manamcheri, Sayan Mitra, Stanley Bak, and Marco Caccamo. "A step towards verification and synthesis from Simulink/Stateflow models". In: *Proceedings of the 14th international conference on Hybrid systems: computation and control*. 2011, pp. 317–318.

[110] Zohar Manna and Amir Pnueli. "Verifying hybrid systems". In: *Hybrid Systems*. Springer, 1992, pp. 4–35.

[111] James R Meyer. "On Smith-Volterra-Cantor sets and their measure". In: (). URL: `https://www.jamesrmeyer.com/pdfs/svc-sets-and-measure.pdf`.

[112] Antoine Miné. "A new numerical abstract domain based on difference-bound matrices". In: *Symposium on Program as Data Objects*. Springer. 2001, pp. 155–172.

[113] Antoine Miné. "Weakly relational numerical abstract domains". PhD thesis. Ecole Polytechnique X, 2004.

[114] Stefano Minopoli and Goran Frehse. "From simulation models to hybrid automata using urgency and relaxation". In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. 2016, pp. 287–296.

[115] Stefano Minopoli and Goran Frehse. "SL2SX translator: from Simulink to SpaceEx models". In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. 2016, pp. 93–98.

[116] Ian M Mitchell. “Comparing forward and backward reachability as tools for safety analysis”. In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2007, pp. 428–443.

[117] Ian M Mitchell, Alexandre M Bayen, and Claire J Tomlin. “A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games”. In: *IEEE Transactions on automatic control* 50.7 (2005), pp. 947–957.

[118] Ramon E Moore. *Interval analysis*. Vol. 4. Prentice-Hall Englewood Cliffs, 1966.

[119] Olivier Mullier, Alexandre Chapoutot, and Julien Alexandre dit Sandretto. “Validated computation of the local truncation error of Runge-Kutta methods with automatic differentiation”. In: *Optimization Methods and Software* 33.4-6 (2018), pp. 718–728.

[120] Nedialko S Nedialkov. “Implementing a rigorous ODE solver through literate programming”. In: *Modeling, Design, and Simulation of Systems with Uncertainties*. Springer, 2011, pp. 3–19.

[121] Nedialko S Nedialkov. “VNODE-LP”. In: *Dept. of Computing and Software, McMaster Univ. TR CAS-06-06-NN, Hamilton, ON, Canada* (2006).

[122] Nedialko S Nedialkov, Kenneth R Jackson, and George F Corliss. “Validated solutions of initial value problems for ordinary differential equations”. In: *Applied Mathematics and Computation* 105.1 (1999), pp. 21–68.

[123] Truong Nghiem, Sriram Sankaranarayanan, Georgios Fainekos, Franjo Ivanci, Aarti Gupta, and George J Pappas. “Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems”. In: *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*. 2010, pp. 211–220.

[124] Karl Nickel. “How to fight the wrapping effect”. In: *International Symposium on Interval Mathematics*. Springer. 1985, pp. 121–132.

[125] JW Nieuwenhuis. “Some remarks on set-valued dynamical systems”. In: *The ANZIAM Journal* 22.3 (1981), pp. 308–313.

[126] Donal O’Regan. *Existence Theory for Nonlinear Ordinary Differential Equations*. en. Dordrecht: Springer Netherlands, 1997. ISBN: 978-90-481-4835-6 978-94-017-1517-1. DOI: 10.1007/978-94-017-1517-1. URL: http://link.springer.com/10.1007/978-94-017-1517-1 (visited on 06/17/2021).

[127] Valerii S Patsko and Varvara L Turova. “From Dubins car to Reeds and Shepps mobile robot”. In: *Computing and visualization in science* 12.7 (2009), pp. 345–364.

[128] François Pessaux, Julien Alexandre Dit Sandretto, and Alexandre Chapoutot. “Hybrid Systems and Contracts with Zélus and DynIbex Zeldyn: a Compilation and Verification Toolchain”. In: (2022).

[129] André Platzer. “A complete uniform substitution calculus for differential dynamic logic”. In: *Journal of Automated Reasoning* 59.2 (2017), pp. 219–265.

[130] André Platzer and Yong Kiam Tan. “Differential equation invariance axiomatization”. In: *Journal of the ACM (JACM)* 67.1 (2020), pp. 1–66.

[131] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Distribution available at: `www.lri.fr/~pouzet/lucid-synchrone`. Université Paris-Sud, LRI. Apr. 2006.

[132] Tarek Raïssi, Nacim Ramdani, and Yves Candau. “Robust nonlinear continuous-time state estimation using interval taylor models”. In: *IFAC Proceedings Volumes* 39.9 (2006), pp. 691–696.

[133] Nacim Ramdani, Nacim Meslem, and Yves Candau. “A hybrid bounding method for computing an over-approximation for the reachable set of uncertain nonlinear systems”. In: *IEEE Transactions on automatic control* 54.10 (2009), pp. 2352–2364.

[134] Nacim Ramdani and Nedialko S Nedialkov. “Computing reachable sets for uncertain nonlinear hybrid systems using interval constraint-propagation techniques”. In: *Nonlinear Analysis: Hybrid Systems* 5.2 (2011), pp. 149–162.

[135] Julien Alexandre Dit Sandretto and Alexandre Chapoutot. “Validated explicit and implicit Runge-Kutta methods”. In: *Reliable Computing electronic edition* 22 (2016).

[136] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. “Scalable analysis of linear systems using mathematical programming”. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2005, pp. 25–41.

[137] Joseph K Scott, Davide M Raimondo, Giuseppe Roberto Marseglia, and Richard D Braatz. “Constrained zonotopes: A new tool for set-based estimation and fault detection”. In: *Automatica* 69 (2016), pp. 126–136.

[138] Yasutaka Sibuya, Po-Fang Hsieh, and Yasutaka Sibuya. *Basic theory of ordinary differential equations*. Springer Science & Business Media, 1999.

[139] Andrew Paul Smith. “Fast construction of constant bound functions for sparse polynomials”. In: *Journal of Global Optimization* 43.2 (2009), pp. 445–458.

[140] Robert M Solovay. “A model of set-theory in which every set of reals is Lebesgue measurable”. In: *Annals of Mathematics* (1970), pp. 1–56.

[141] Duan M Stipanovi, Inseok Hwang, and Claire J Tomlin. “Computation of an over-approximation of the backward reachable set using subsystem level set functions”. In: *2003 European Control Conference (ECC)*. IEEE. 2003, pp. 300–305.

[142] Claire J Tomlin, Ian Mitchell, Alexandre M Bayen, and Meeko Oishi. “Computational techniques for the verification of hybrid systems”. In: *Proceedings of the IEEE* 91.7 (2003), pp. 986–1001.

[143] Hoang-Dung Tran, Feiyang Cai, Manzanas Lopez Diego, Patrick Musau, Taylor T Johnson, and Xenofon Koutsoukos. "Safety verification of cyber-physical systems with reinforcement learning control". In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.5s (2019), pp. 1–22.

[144] Vadim I Utkin. *Sliding modes in control and optimization*. Springer Science & Business Media, 2013.

[145] Mark Wetzlinger, Adrian Kulmburg, and Matthias Althoff. "Adaptive parameter tuning for reachability analysis of nonlinear systems". In: *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*. 2021, pp. 1–11.

[146] Albert Wu. *Guaranteed avoidance of unpredictable, dynamically constrained obstacles using velocity obstacle sets*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE DEPT OF AERONAUTICS and ASTRONAUTICS, 2011.

[147] Michael Zettler and Jürgen Garloff. "Robustness analysis of polynomials with polynomial parameter dependency using Bernstein expansion". In: *IEEE Transactions on Automatic Control* 43.3 (1998), pp. 425–431.

[148] Fu Zhang, Murali Yeddanapudi, and Pieter J Mosterman. "Zero-crossing location and detection algorithms for hybrid system simulation". In: *IFAC Proceedings Volumes* 41.2 (2008), pp. 7967–7972.

ECOLE DOCTORALE

Titre : Analyse d'atteignabilité avec incertitudes variables dans le temps et intégrables au sens de Lebesgue

Mots clés : systèmes cyberphysiques, analyse d'atteignabilité, modèles de Taylor, Lebesgue, Zélus

Résumé : L'analyse d'atteignabilité, technique calculant l'ensemble des états atteignables d'un système, permet de prouver des propriétés de sûreté. Ce calcul exact étant impossible dans le cas général, nous calculons des sur-approximations, i.e. ensembles garantis de contenir tous les états atteignables. Ces calculs sont plus complexes à cause d'incertitudes variant dans le temps qui permettent de modéliser l'environnement dans lequel évolue le système étudié. Différentes hypothèses sont faites sur la nature de ces incertitudes. Dans ce travail, nous nous intéressons au cas général d'incertitudes intégrables au sens de Lebesgue. Certaines méthodes les prennent en compte mais elles produisent généralement de larges sur-approximations lorsque les ensembles de valeurs des incertitudes sont grands.

Nous prouvons que, sous certaines hypothèses, des fonctions ensemblistes contractées par une version ensembliste de l'opérateur de Picard associé à un problème de Cauchy sont des sur-approximations des ensembles d'états atteignables en fonction du temps. Cela prouve en particulier que les intervalles contractés par un tel opérateur sont des sur-approximations valides dans le cas général d'incertitudes intégrables au sens de Lebesgue.

Nous proposons ensuite un nouvel algorithme basé sur une décomposition des équations différentielles sous forme de différences de fonctions positives dans le cas des systèmes que nous nommons séparables par rapport aux incertitudes variant dans le temps, une généralisation des systèmes affines par rapport aux contrôles. Un prototype illustre la précision de cet algorithme par rapport à ceux implémentés dans les outils classiques de l'état de l'art.

Enfin, nous présentons les implications de l'adaptation des outils de simulations de systèmes hybrides pour effectuer des analyses d'atteignabilité. Nous nous concentrons sur le langage Zélus pour proposer une paramétrisation des modèles intermédiaires générés par son compilateur. La faisabilité de cette méthode est illustrée par un prototype exécuté sur un exemple.

Title : Reachability analysis with Lebesgue-integrable time-varying uncertainties

Keywords : cyberphysical systems, reachability analysis, Taylor models, Lebesgue, Zélus

Abstract : Reachability analysis, technique computing the set of reachable states of a system, allows to prove safety properties. The exact computation being impossible in the general case, we compute over-approximations, i.e. sets guaranteed to contain all the reachable states. These computations are more complex due to time-varying uncertainties that allow to model the environment in which the studied system evolves. Different hypotheses are made about these uncertainties. In this work, we focus on the general case of uncertainties that are integrable in the sense of Lebesgue. Some methods handle such uncertainties, but they often produce huge over-approximations when the sets of values of the uncertainties are large.

We prove, under some assumptions, that set-valued functions contracted by a set-valued version of the Picard's operator associated to an initial value problem are over-approximations of the set of reachable states over time. In particular, it proves that intervals contracted by such an operator are valid over-approximations in the general case of Lebesgue-integrable uncertainties.

Then, we propose an algorithm based on a decomposition of the differential equations as differences of non-negative functions in the case of systems that we called separable with respect to uncertainties, a generalization of control-affine systems. A prototype illustrates the accuracy of the algorithm compared to the ones implemented in the classical state-of-the-art tools.

Finally, we present the implications of the adaptations of tools for the simulation of hybrid systems to allow the computation of reachability analyses. We focus on the Zélus language to propose a parametrization of the intermediate models that are generated by the compiler. The feasibility of this method is illustrated by a prototype on an example.

Institut Polytechnique de Paris
91120 Palaiseau, France