VERIFICATION OF LIVENESS PROPERTIES ON HYBRID DYNAMICAL SYSTEMS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2013

Rebekah Anne Carter
School of Computer Science

Contents

Word count 49634

List of Tables

List of Figures

List of Algorithms

Abstract

The University of Manchester
Rebekah Anne Carter
Doctor of Philosophy
Verification of Liveness Properties on Hybrid Dynamical Systems
June 17, 2013

A hybrid dynamical system is a mathematical model for a part of the real world where discrete and continuous parts interact with each other. Typically such systems are complex, and it is difficult to know how they will behave for general parameters and initial conditions. However, the method of formal verification gives us the ability to prove automatically that certain behaviour does or does not happen for a range of parameters in a system. The challenge is then to define suitable methods for proving properties on hybrid systems.

This thesis looks at using formal verification for proving liveness properties on hybrid systems: a liveness property says that something good eventually happens in the system. This work presents the theoretical background and practical application of various methods for proving and disproving inevitability properties (a type of liveness) in different classes of hybrid systems. The methods combine knowledge of dynamical behaviour of a system with the brute-force approach of model checking, in order to make the most of the benefits of both sides.

The work on proving liveness properties is based on abstraction of dynamical systems to timed automata. This thesis explores the limits of a pre-defined abstraction method, adds some dynamical knowledge to the method, and shows that this improvement makes liveness properties provable in certain continuous dynamical systems. The limits are then pushed further to see how this method can be used for piecewise-continuous dynamical systems. The resulting algorithms are implemented for both classes of systems.

In order to disprove liveness properties in hybrid systems a novel framework is proposed, using a new property called *deadness*. Deadness is a dynamically-aware property of the hybrid system which, if true, disproves the liveness property by means of a *finite* execution: we usually require an *infinite* execution to disprove a liveness property. An algorithm is proposed which uses dynamical properties of hybrid systems to derive deadness properties automatically, and the implementation of this algorithm is discussed and applied to a simplified model of an oilwell drillstring.

Declaration

Copyright statement

Acknowledgements

First I wish to thank my supervisor, Eva Navarro-López, who has encouraged me and pushed me all the way through my PhD. Eva, I am so grateful for the way you have questioned everything I have done, forcing me to really know what I am talking about! Also, I thank you for the hours you have put into reading all my writing, but particularly this thesis. This work only exists because of your passion for research.

I would also like to thank RCUK and EPSRC, who have supported me financially through a PhD studentship and through the framework of the EPSRC-funded project "DYVERSE: A New Kind of Control for Hybrid Systems" (EP/I001689/1).

I would also like to thank Mike and Giles who have talked with me in the office, improving the communication of my research and keeping me sane. And to all my new colleagues at CIS, thank you for supporting me as I started on my new adventure.

To my Mum and Dad, thank you for supporting me and encouraging me to keep going, through the good and the bad times of the last three and a half years. I know you are proud of what I have achieved, which means a lot to me.

To David and Chris, thanks for all your wisdom and encouragement, and for feeding us when we needed it most.

To Alan, you have been amazing and have been so patient with me, especially at the end where I have been rather in my own world. But throughout, you have tried your best to be a good husband, even when you were suffering so much yourself. Thank you for your gentle prodding to make me do things I did not want to do, especially when you did not want me to have to do them either. You could not have done more.

Lastly, I dedicate this thesis to God, without whom I would not be here. Thank you Lord for the intelligence and abilities you have given me, and for the people you have put around me. There have been ups and downs, but I know that you've led me all the way and will continue to do so.

Chapter 1

Introduction

A hybrid dynamical system is a mathematical model for a part of the real world where discrete and continuous parts interact with each other. Such systems can model all kinds of situations, from biological systems [Cook et al., 2011, Ghosh and Tomlin, 2004a] to a controller interacting with its environment [Stiver et al., 2001, Tomlin et al., 1998], from electronic circuits [Hejri and Mokhtari, 2008] to mechanical systems [Navarro-López and Carter, 2011].

Due to the vast number of situations which can be modelled as hybrid dynamical systems, having good ways of analysing their behaviour enables us to learn useful information about the parts of the real world they model. This analysis tells us about what happens as time evolves in a system, and we can then decide whether we are happy with the behaviour we see.

1.1 Desired behaviour in dynamical systems

When we analyse the behaviour of a dynamical system, we usually want to make sure that it satisfies two kinds of properties.

1. The first property is that the dynamical system never does a 'bad thing'. What a bad thing is depends on the system we are looking at, but if we can analyse the behaviour of the system and be sure it will never do something bad, we can be content to leave the system to its own devices.

2. The second property that we want to make sure of in the system is that it will

do some 'good thing' in the future. Again, the good thing will vary depending on the system we are looking at, but ensuring we can rely on the system to eventually do what we desire is important.

These types of properties have long been considered in the field of dynamical systems theory, with *stability* and *attractivity* being key concepts in dynamical systems. Stability is the idea that, if a system trajectory starts close to a point in space, then it will always remain close to that point in space — this is a property of the first type, where going far away from the point in space is a 'bad thing'. However, attractivity encompasses the idea that a system trajectory will keep getting closer and closer to a desired point in space, which is a property of the second type, where getting close to the desired point is a 'good thing'.

Stability and attractivity capture the intuition that we have about good behaviour of a dynamical system, but are not easy to prove automatically. However, it is desirable to use automatic methods of analysis on hybrid dynamical systems to get a lot of information about a system without much time and brain-power being expended. Performing automatic analysis on a system allows us to think about it in a way which makes the best use of our intelligence, intuition, and time.

1.2 Automatic analysis using formal verification

In the area of computer science, there is already a well developed field in the design and use of automatic methods to prove useful information about discrete space-time systems: this is the field of formal verification. Recognising that the ideas of this field are useful and well developed, the hybrid systems community has already put a lot of work into applying the theory and methods of formal verification to the complex field of hybrid dynamical systems. The majority of the work in formal verification of hybrid systems is to prove *safety* properties, which are those which say that a 'bad thing' never happens, the first type of property that we previously identified.

The second kind of property is that which says that some 'good thing' eventually happens — these properties are called *liveness* properties. In the field of formal verification of dynamical systems, these properties have mostly been considered in the context of practical examples, as the methods for proving them are not well developed. In fact, there are only a few cases where the formal verification of liveness in more general dynamical systems has been considered, and so there is a lot of room for new methods and concepts to increase our ability to prove liveness properties about dynamical systems.

The aims of this thesis are to investigate methods for proving and disproving liveness properties on hybrid dynamical systems, using dynamically-driven formal verification methods. We aim to propose theory and methods to prove or disprove liveness for classes of hybrid systems, and to prove that these methods are in some way useful for proving or disproving liveness. The idea is to take the first steps along the road with the methods we use, to provide a springboard for future work to develop theory and methods in this area.

1.3 Contributions of the thesis

This thesis adds to the body of knowledge in the relatively new area of formal verification of liveness properties in dynamical systems. Ultimately we are interested in proving liveness properties on hybrid dynamical systems, but as the field is so new there are no results for most methods in the underlying classes of continuous dynamical systems and piecewise-continuous dynamical systems. The contributions of this thesis therefore begin with results for a class of continuous dynamical systems, then extend the resulting method to a class of piecewise-continuous systems, with the final contribution considering liveness in the most general class, hybrid systems. In this way we build the foundations for new methods to prove liveness properties in hybrid dynamical systems.

All the methods proposed in this thesis can be classed as *dynamically-driven methods*, by which we mean that the way the methods work is guided by properties of the dynamics of the system. The dynamical properties used in the methods are those which are directly available from the description of the system, or are properties which can be derived automatically from the description of the system. We do not aim to build completely new proving systems, but will establish dynamically-driven methods which improve the usefulness of other algorithms and implementations that are already available. In using ready-made parts of theory and implementation we can improve the power of the dynamically-driven methods we create.

The specific contributions of this thesis are the following:

1. Definition of a dynamically-driven method to split the state space of a class of continuous dynamical systems, to improve the characteristics of the abstraction method of Maler and Batt [2008].

2. Proof that this improved abstraction method creates a timed-automaton abstraction which is equivalent to the original system with respect to proving inevitability (a type of liveness).

3. Extension of the splitting method for continuous dynamical systems to a class of piecewise-continuous dynamical systems, and analysis of the characteristics of the splitting.

4. Definition of the logical property of *deadness* and its relationship to the logical property of liveness.

5. Proposal of a practical method to find deadness properties for given liveness properties in hybrid systems.

1.4 Thesis overview

To aid the reader, we will now summarise the contents of each chapter. Note that Chapter 2 contains most of the background material for the rest of the thesis, so should be read first. Chapters 3 and 4 are closely linked, and should be read in order, but Chapter 5 stands alone and can be read independently from Chapters 3 and 4.

Chapter 2. Here we give background information for the rest of the thesis. The chapter is split into four parts. The first part summarises the key information about continuous dynamical systems that is relevant to the rest of the thesis, and the second part gives background information on hybrid dynamical systems and the related definitions for the thesis. We then consider the state of the art in formal verification of hybrid systems, and finally assess the current state in the field of dynamical systems in terms of proving liveness.

Chapter 3. In this chapter we look at the method of Maler and Batt [2008], proposing an improvement to it so that it creates timed-automaton abstractions which prove inevitability of a class of continuous dynamical systems. We prove that the method we present terminates and creates a system equivalent to the original with respect to proving inevitability. We also briefly discuss the implementation which has been made.

Chapter 4. We extend the method of Chapter 3 to be used for a class of piecewise-linear systems. We show that the resulting timed-automaton abstraction still includes all trajectories that existed in the original system, showing it is a sensible extension of the method of Maler and Batt [2008]. We also show that the resulting timed-automaton abstraction is equivalent with respect to inevitability for systems of maximum dimension two. We analyse the problems and suggest potential solutions for higher-dimensional piecewise-linear systems.

Chapter 5. First we formally define safety and liveness in hybrid dynamical systems. We then use these definitions to propose the idea of deadness, which is the concept 'if a system is dead it can never be live again'. We propose a practical method for finding such deadness properties on hybrid dynamical systems, discuss its implementation, and demonstrate its use on an example.

Chapter 6. Here, we conclude the thesis, and draw together all the future work that we foresee.

Chapter 2

Background and state of the art

This thesis is in the area of hybrid dynamical systems, which consist of interactions between continuous dynamics and discrete dynamics. We will firstly define the key concepts of continuous dynamical systems (Section 2.1) before moving on to consider hybrid dynamical systems in Section 2.2. We will then consider how we can prove properties of such systems computationally by introducing formal verification in Section 2.3, and to finish this chapter we will look at liveness properties, the particular class we consider in this thesis (Section 2.4).

2.1 Continuous dynamical systems

In this section we will introduce the relevant information about continuous dynamical systems, along with the notation we use. A continuous dynamical system is a mathematical model for a part of the real world, where the space is continuous over some dense set. We will be considering the continuous-time version, where the time variable t is continuous over the space $\mathbb{R}$. A continuous dynamical system in continuous-time is represented by differential equations, which involve parameters, variables, and the usual mathematical operators $(=, +, -, \times, /)$, but also derivatives of variables with respect to time, denoted by $\dot{x} = dx/dt$. Derivatives denote the rate of change of a variable.

An example of a continuous dynamical system is a simple model for a falling ball, given by the equations

$$\begin{aligned} \dot{x} &= v, \\ \dot{v} &= -g, \end{aligned}$$

where x is the height of the ball, v is its velocity, and $g \simeq 9.81\text{m/s}^2$ is the acceleration due to gravity. In this simple model the ball keeps accelerating forever, reaching infinite negative velocity at infinite time, but we know that in the real world the air resistance acts to slow down the ball's acceleration, resulting in a terminal velocity from some finite time onwards. This immediately leads to an important point: any analysis we perform on a dynamical system will only give us helpful information about this part of the real world if the mathematical model is a good representation of it.

In this thesis we will not be studying the process of finding a dynamical system which represents the real world, but will assume any given dynamical systems are 'good enough' to capture the relevant features of the real world system. We will be concerned with what properties of the dynamical system can be proven mathematically or computationally, and how such properties are proven.

2.1.1 General form of continuous dynamical systems

Let us consider a system with n real variables $x_1 \in \mathbb{R}, \dots, x_n \in \mathbb{R}$. We will write the vector of variables as $x = (x_1, \dots, x_n)^T \in \mathbb{R}^n$. In order to define the notion of a continuous function, we need the idea of a ball around a point in the n-dimensional space $\mathbb{R}^n$.

Definition 2.1 (Ball). A ball of radius $r > 0$ around a point $p \in \mathbb{R}^n$, is defined by

$$B(r, p) = \{x \in \mathbb{R}^n : \|x - p\| < r\},$$

where $\|\cdot\|$ denotes the 2-norm on the Euclidean space $\mathbb{R}^n$. ■

We can now use this notation to define the idea of a continuous function, which says that small changes in a variable only induce small changes in the function value.

Definition 2.2 (Continuous function [Hijab, 2011]). A function $f : \mathbb{R}^n \to \mathbb{R}^n$ is a continuous function at the point $p \in \mathbb{R}^n$ if for every $\epsilon > 0$ there exists a $\delta > 0$ such that $x \in B(\delta, p)$ implies $x \in B(\epsilon, p)$. The function will be called continuous in a set $U \in \mathbb{R}^n$ if it is continuous at every point $x \in U$. ■

In this work we will define that a continuous dynamical system has first-order differential equations of the form

$$\dot{x} = f(x), \qquad x \in \mathbb{R}^n, \tag{2.1}$$

where the function f is a continuous function of $x \in \mathbb{R}^n$. We will also define $x(t) \in \mathbb{R}^n$ as the solution or trajectory of such a set of equations, with $t \geq 0$ the time that has passed in the system. We will assume that $t = 0$ is the initial time of the system, without loss of generality.

Given this definition we will now consider what useful properties we can define and prove about the continuous dynamical system. In dynamical systems theory the main concept used is that of stability of a system. We introduce it since stability and the related concept attractivity will be properties we are interested in proving, and also because we will use the concepts in our method for disproving liveness (Chapter 5).

2.1.2 Stability of continuous dynamical systems

Stability of a dynamical system is classically defined about an equilibrium point, so we will firstly define this.

Definition 2.3 (Equilibrium point)**.** An equilibrium point of (2.1) is any $\overline{x} \in \mathbb{R}^n$ such that $f(\overline{x}) = 0$. ■

If we start the system at such an equilibrium point the dynamics will not move us away from it, due to the fact that the rate of change of x is zero ($\dot{x} = 0$) when $f(x) = 0$. We now consider what happens if we start a small distance away from the equilibrium point: if we stay close to the equilibrium point for the rest of time, then the equilibrium is called *stable in the sense of Lyapunov*, and if not then it is *unstable*. This is formalised in the following definition.

Definition 2.4 (Stable equilibrium point [Lyapunov, 1892])**.** $\overline{x}$ is said to be a stable equilibrium of (2.1), in the sense of Lyapunov, if for every $\epsilon > 0$ there exists a $\delta > 0$ such that $x(0) \in B(\delta, \overline{x})$ implies $x(t) \in B(\epsilon, \overline{x})$ for all $t \geq 0$. ■

Notice that this notion of stability does not guarantee that the equilibrium point is ever reached by the trajectories of the system, only that the trajectories stay close to it for all time. The idea of converging to the equilibrium is given by the notion of attractivity.

Definition 2.5 (Attractive equilibrium point [Lyapunov, 1892]). $\overline{x}$ is an attractive equilibrium if there exists a δ_1 such that $x(0) \in B(\delta_1, \overline{x})$ implies $\lim_{t\to\infty} x(t) = \overline{x}$. ■

We can also define more notions of stability in continuous dynamical systems.

- $\overline{x}$ is *asymptotically stable* if it is stable and attractive.
- $\overline{x}$ is *globally attractive* if δ_1 may be taken arbitrarily large in Def. 2.5.
- $\overline{x}$ is *globally asymptotically stable* if it is stable and globally attractive.
- $\overline{x}$ is *unstable* if it is not stable.

In general, it is not easy to prove the stability condition of Definition 2.4 directly in a system, since there are theoretically an infinite number of small parameters ϵ that we must check. However, we can prove Lyapunov stability of an equilibrium point or set of space by using Lyapunov functions, which give us stability for all values provided we can find a suitable function. We will firstly look at the definitions for these functions, and will then explain why they show stability.

Definition 2.6 (Lyapunov function candidate [Lyapunov, 1892]). A function $V : \mathbb{R}^n \to \mathbb{R}$ is a Lyapunov function candidate if it is positive definite in a set $U \in \mathbb{R}^n$ about the equilibrium $\overline{x}$. That is:

$$\begin{aligned} &V(x) > 0, \quad \forall x \in U \setminus \{\overline{x}\}, \\ &V(\overline{x}) = 0. \end{aligned}$$

■

Theorem 2.7 (Stability by a Lyapunov function [Lyapunov, 1892]). *Let $V : \mathbb{R}^n \to \mathbb{R}$ be a candidate Lyapunov function. If*

$$\dot{V}(x) = \frac{dV}{dx}\dot{x} \leq 0 \quad \forall x \in U \setminus \{\overline{x}\},$$

then the origin is stable (in the sense of Lyapunov). ■

These Lyapunov functions are defined in a set about the equilibrium point, are positive definite about this point, and are non-increasing along the trajectories of the system. Intuitively, this implies stability, because once the trajectory is inside a level set $V(x) = m$, for m positive, all the vector fields point into this set, and so the trajectories cannot leave. Hence, if we start close to the equilibrium we remain close to the equilibrium for all time.[1]

We can also refine these conditions to give asymptotic stability of the equilibrium point. For this, we need to have $\dot{V}(x) < 0 \; \forall x \in U \setminus \{\overline{x}\}$ in Theorem 2.7.

The explanation of this is the following. Firstly note that it is a specialisation of Theorem 2.7, and so we have Lyapunov stability. Hence, it only remains to show that we tend to the equilibrium point as time tends to infinity. We can see that the function V is always decreasing in the set $U \setminus \{\overline{x}\}$. Since $V \geq 0$, this means that we must tend to the point where $V = 0$ in infinite time, and hence we tend to the point where $x = \overline{x}$. Therefore, $|x(t)| \to \overline{x}$ as $t \to \infty$, and we have asymptotic stability.

It can be shown that when a system is stable then a Lyapunov function exists, so proving stability only requires finding this function. To calculate these functions by hand we have to guess a function and see if it works for the system under consideration, and if a function of the form we have guessed does not work then we must guess another one. For complex systems it can be very hard even to make an initial guess at a suitable function.

However, when we have a linear system, so that $\dot{x} = Ax + b$ with A an $n \times n$ real matrix and b a real vector of size n, we can always find a quadratic Lyapunov function $V(x)$ which has $\dot{V}(x) < 0$ everywhere in the space $\mathbb{R}^n \setminus \{\overline{x}\}$. The method is roughly this: we specify $V(x) = x^T P x$ where $P \in \mathbb{R}^{n \times n}$ is an arbitrary positive definite matrix,[2] and then we have $\dot{V}(x) = x^T(A^T P + PA)x$. We then wish to solve this equation to equal an always-negative function described by $-x^T Q x$ where Q is a positive definite matrix. This reduces to the *Lyapunov equation*

$$A^T P + PA = -Q. \tag{2.2}$$

To solve this equation we can choose any positive definite matrix Q, and then solve

[1]The formal proof that this defines Lyapunov stability in the sense of Definition 2.4 involves taking balls inside and outside of a level set, $V(x) = m$, and then provided we start in the inner ball we remain in the outer ball for all time. See for example Simmons [1972, pages 318-319] for more details.

[2]A positive definite matrix P is defined by the fact that $x^T P x > 0$ for all $x \in \mathbb{R}$.

the linear matrix equality that we have for the matrix P. This gives us a Lyapunov function $V(x) = x^T Px \geq 0$ for $x \in \mathbb{R}^n$. The solution of the Lyapunov equation (2.2) can be calculated in many ways, but the naive method (Gaussian elimination) is not the most efficient or stable algorithm for the task, and there have been many others proposed which improve on the performance (classic texts include those by Bartels and Stewart [1972] and Hammarling [1982]).

As the task of finding Lyapunov functions for linear systems is relatively easy and always possible, there have been various authors who attempt to extend these results to the nonlinear case. We note that a nonlinear system does not necessarily have global stability (unlike the linear case), so the methods for nonlinear systems not only need to find a Lyapunov function but also need to find the set in which it proves stability. One nonlinear Lyapunov function creator is due to Davison and Kurak [1971], where an optimisation method is used to extend the size and shape of a quadratic Lyapunov function calculated for the linearisation of the nonlinear system we are considering.

The set of all initial states from which the trajectories converge to the equilibrium point is called the *domain of attraction*: optimisation methods such as Davison and Kurak [1971] do not usually find the whole set, but find some subset of it. However, different methods can find different proportions of the domain of attraction, for instance (in general) the methods in Flashner and Guttalu [1988] and Grüne [2001] find a much larger subset of the domain of attraction than the method in Davison and Kurak [1971].

Other methods seek to use the nonlinear dynamics directly to calculate a Lyapunov function which proves stability of the system in the whole of $\mathbb{R}^n$. The most notable tool which employs a method of this type is SOSTOOLS by Prajna et al. [2005], which can find a Lyapunov function for systems with polynomial vector fields by means of sum of squares optimisation problem.

2.2 Hybrid systems

A very loose definition of hybrid systems is that they consist of interactions between continuous and discrete parts. By continuous we mean that the space of this part of the system is continuous: in general the time can evolve either in a continuous manner or in discrete steps and still be part of the continuous part of the dynamics. However, we only look at continuous-time systems, where the continuous part of the dynamics is described by continuous dynamical systems, like those discussed in Section 2.1. The discrete part of the dynamics does not evolve on its own, but affects how the differential equations of the continuous part are defined or what value in the state space we are at. In turn the continuous part affects how the discrete part behaves.

There are many ways of representing the behaviour of hybrid systems. We will firstly define the *hybrid automaton*, which is a commonly used formalism for describing hybrid systems and the one we use in this thesis. We will discuss other methods which can be used after this. Good introductions or overviews to hybrid systems and related concepts include Branicky [2005], Cortés [2008], Heemels et al. [2003], Khalil [2002], Kowalewski [2002], Matveev and Savkin [2000], and van der Schaft and Schumacher [2000].

2.2.1 Hybrid automata

Hybrid automata are a useful model of hybrid dynamical systems, since they explicitly show the interaction between the continuous and the discrete parts of the system. A variation on the classical automaton idea is used to model the discrete changes in the system, with differential equations used to model the continuous motion. We will assume the reader is familiar with the theory of finite state automata, but if not see the book by Hopcroft et al. [2007]. The notion of hybrid automata was first introduced by Alur et al. [1993b], but we will use notation more consistent with Johansson et al. [1999].

Definition 2.8 (Hybrid automaton [Johansson et al., 1999])**.** A hybrid automaton is a collection

$$H = (Q, E, \mathcal{X}, Dom, \mathcal{F}, Init, G, R)$$

that models a hybrid system, where

- Q is a finite set of locations.
- $E \subseteq Q \times Q$ is a finite set of edges called transitions or events.
- $\mathcal{X} \subseteq \mathbb{R}^n$ is the continuous state space.

- $Dom : Q \to 2^{\mathcal{X}}$ is the location domain (sometimes called an invariant). It assigns a set of continuous states to each discrete location $q \in Q$, thus, $Dom(q) \subseteq \mathcal{X}$.
- $\mathcal{F} = \{f_q(x) : \ q \in Q\}$ is a finite set of vector fields describing the continuous dynamics in each location, such that $f_q : \mathcal{X} \to \mathcal{X}$. Each $f_q(x)$ is assumed to be Lipschitz continuous on the location domain for q in order to ensure that the solution exists and is unique.
- $Init \subseteq \bigcup_{q \in Q} q \times Dom(q) \subseteq Q \times \mathcal{X}$ is a set of initial states.
- $G : E \to 2^{\mathcal{X}}$ is a guard map. G assigns to each edge a set of continuous states; this set contains the states which enable the edge to be taken.
- $R : E \times \mathcal{X} \to 2^{\mathcal{X}}$ is a reset map for the continuous states for each edge. It is assumed to be non-empty, so that the dynamics can only be changed, not destroyed. ∎

Definition 2.9 (Hybrid state space [Johansson et al., 1999])**.** The hybrid state space is the set defined by

$$\mathcal{Z} \equiv \bigcup_{q \in Q} q \times Dom(q) \subseteq Q \times \mathcal{X}.$$

That is, the set of all pairs (q, x) which the hybrid automaton allows to exist. A *hybrid state* z is a member of the hybrid state space, or $z = (q, x) \in \mathcal{Z}$. ∎

We will defer the definition of other related concepts until they are used in Chapter 5. The main idea to note is that the automaton is assumed to *accept* an allowed behaviour (as is the case in finite-state automata), which is different from the notion that is used in continuous dynamical systems that the system *defines* a behaviour.

It is also possible to add more concepts to this automaton to encompass more modelling requirements. Control theorists add inputs and outputs to this automaton to allow for the design and study of controls [Navarro-López, 2009]. In this work we are only interested in what we can prove about a given model, and will not consider how to control the model to fit certain requirements, so we do not require these control aspects.

We will only consider deterministic hybrid systems, which are those where the trajectories have no choice in where they go. Contrasted with this are non-deterministic hybrid systems, where at some or all points in the system the trajectories have a choice in how they evolve. In a hybrid automaton there are two ways that non-determinism can be introduced:

1. Non-determinism in the continuous flow. This can happen if the continuous-time dynamics are described by differential inclusions instead of differential equations within each location [Casagrande et al., 2008], that is the derivatives in each discrete location can take multiple values $\dot{x} \in F(x)$ where F is a multivalued function.

2. Non-determinism in the discrete transitions. One possibility for this is that a discrete transition can be taken along a range of values of a trajectory, with no particular value having precedence, which enable choice in when the discrete transition is taken. The second possibility is that more than one guard condition is defined at the same point in the domain, which enables a trajectory to choose which location to transition to when it reaches this point.

In many non-deterministic systems the uncertainties present are in some way quantifiable, so there is a strong case for using *probabilistic hybrid systems*. In these models, the probabilities of certain events happening are encoded directly in the model, allowing us to look explicitly at the likelihood of bad or good things happening, rather than simply considering whether they do happen or not. There is a large body of work in this interesting area, from different modelling frameworks [Mitra, 2007, Segala, 1995] to methods and tools for proving properties [Kwiatkowska et al., 2007, 2011]. In some systems the non-determinism may be in the form of noise, and then the systems may be better described by a stochastic hybrid system, with proofs made by use of various methods [Abate et al., 2008, 2010, Bujorianu, 2012].

In our case we will enforce determinism by (1) only allowing single-valued differential equations, (2) putting a rule in place to enforce transition as soon as a guard becomes enabled, and (3) ensuring that only one guard is defined at each point of the domain. We also assume there is no noise present, so that we do not have a stochastic hybrid automaton.

On the other hand, to make analysis of these automata more tractable, simpler versions can be defined to model certain systems. These simpler versions include timed automata [Alur and Dill, 1994], rectangular hybrid automata [Henzinger et al., 1998, Puri and Varaiya, 1994], and linear hybrid automata [Alur et al., 1995]. All these simpler hybrid automata make restrictions on the type of continuous behaviour that is allowed, and also can restrict the type of functions allowed to specify the location domains, as well as the type of functions allowed to specify the transition guards and resets.

- Timed automata only allow continuous variables with rate of change $\dot{x}_i = 1$ for every i in every location (the x_i are usually called *clocks*). The domains are only allowed to be non-negative sets of the x_i's, the guards are only allowed to be conjunctions of clocks compared to constants, and the resets can only set clocks to zero. We will look at timed automata in closer detail in Section 2.2.3, as these type of automata will form a major part of this thesis.
- Rectangular hybrid automata only restrict the continuous dynamics, and not the domains, guards, or resets. The restriction on the dynamics is of the form $\dot{x}_i \in [\lambda, \upsilon]$, where λ and υ are constants, and these constants can be different for each variable x_i and each location of the automaton. Such limits on the dynamics are easy to work with when analysing the system, but can lose a lot of useful information about the real world if used inappropriately.
- Linear hybrid automata restrict the dynamics to be a constant value $\dot{x}_i = c$ that can be different for each variable in each location: when these type of dynamics are explicitly solved we get solutions that are linear functions of time. These type of automata also require the domains, guards, and resets to be formed of conjunctions of linear functions of the x_i, although the resets are allowed to have multiple possible values. These automata are relatively easy to analyse, but are restricted in the type of dynamics they portray.

2.2.2 Comparison with other modelling frameworks

There are various other modelling methods available to us when looking at hybrid dynamical systems, which have each got their pros and cons. Most modelling methods which are not automata-based are equation-based, where the continuous dynamics and the discrete transitions are both represented as equations. An equation-based representation for a subclass of hybrid systems is the class of piecewise-smooth systems [Filippov, 1988], which have been investigated for many years. Piecewise-smooth systems usually are defined by equations of the form

$$\dot{x} = \begin{cases} f_1(x), & \text{when } x \in D_1, \\ \vdots \\ f_m(x), & \text{when } x \in D_m, \end{cases}$$

where the D_i form a partition of the space $\mathbb{R}^n$, that is $\cup_{i=1,\dots,m} D_i = \mathbb{R}^n$ and $D_i \cap D_j = \emptyset$ for each pair $1 \le i, j \le m$, In this model, no resets are considered. We will look at we can prove about this class of piecewise-smooth systems in Chapter 4.

A more general equation-based method has continuous dynamics of the form $\dot{x} = f(x)$ and an equation is given for the discrete transitions of the form $x' = g(x)$ [Antsaklis et al., 1993, Branicky et al., 1998]. Here x' indicates the value of the vector of variables x after a discrete transition has taken place. These two equations will have certain defined sets where they are allowed to progress the dynamics. And even more general than this is the model of Goebel et al. [2009] which defines inclusions for both the continuous and discrete dynamics, so that $\dot{x} \in F(x)$ for $x \in C$ and $x' \in G(x)$ for $x \in D$ (for some sets C and D).

A common equation-based representation is the *switched system model* [Liberzon, 2003, Liberzon and Morse, 1999], which consists of multiple subsystems with differing dynamics and a switching signal dependent on time. There can be multiple solutions for a switched system model, one for each switching signal. This model has been considered particularly for consideration of stability properties, and also for creation of controllers for hybrid systems.

Other hybrid systems modelling methods include mixed logic dynamical systems [Bemporad and Morari, 1999], which directly represent the system by inequalities containing both continuous and binary variables, and Petri-net–based models [Koutsoukos et al., 1998], where a Petri-net is used to represent the discrete dynamics. There is also a lot of work in the related class of discrete-event systems [Lafortune and Cassandras, 2008], which model systems with both discrete-time and discrete-space properties — we do not use these as we use both continuous space and time.

2.2.3 Timed automata

We have already mentioned timed automata, from the point of view of how they relate to hybrid automata. We will now define them formally, as we will make use of them to prove liveness properties about continuous-time systems. Here we define a slightly generalised version of the timed-automaton, taken from Maler and Batt [2008], as we will make use of the method in that paper in this thesis. The generalisation consists of allowing clocks to be switched on and off, with a clock always being reset to zero whenever it is restarted.

Definition 2.10 (Timed-automaton [Maler and Batt, 2008])**.** A timed-automaton (TA) is a tuple $\mathcal{A} = (Q, Q_0, \mathcal{C}, I, \Delta)$ where:

- Q is a finite set of discrete states,
- $Q_0 \subseteq Q$ is a set of initial states,
- $\mathcal{C}$ is a set of clock variables ranging over $\mathbb{R}_{\geq 0} \cup \{\perp\}$, where $\perp$ is a special symbol indicating that the clock is inactive,
- I is the invariant (domain) which assigns to every state q a conjunction I_q of conditions of the form $c < d$ for clock c and integer d.
- Δ is the transition relation which consists of tuples of the form (q, g, ρ, q'), where q and q' are discrete states, the guard g is a positive combination of conditions of the form $c \geq d \vee c = \perp$, and ρ is a clock transformation defined by one or more assignments of the form $c := 0 \vee c := \perp$. ■

A timed state of the timed-automaton is a pair (q, z), where $q \in Q$ is a timed-automaton state, and z is a valuation of the clocks. A *run* of the automaton starting from a state (q_0, z_0) is a finite or infinite sequence of alternating time steps and discrete steps of the form

$$\xi : (q_0, z_0) \xrightarrow{t_1} (q_0, z_0 + t_1) \xrightarrow{\delta_1} (q_1, z_1) \xrightarrow{t_2} \cdots$$

with $q_0 \in Q_0$ and run duration $\sum t_i$. We can also consider this ξ to be a function of time, $\xi : \mathbb{R}_{\geq 0} \to Q$, where $\xi(t) = q$ if after a duration of t the run is in state q.

Timed automata have many important properties, most of which relate to how easy it is to prove properties about them. The first important property which we will discuss is that these timed automata do not distinguish between the set of reals $\mathbb{R}$ and the set of rationals $\mathbb{Q}$, because the only factor that matters is the *denseness* of the underlying domain [Alur and Dill, 1994]. Practically this means that we can always enumerate the state space in some way, in order to prove properties about the runs of timed automata.

2.2.4 Other work related to timed automata

There have been many works which use the concept of timed automata. There are those who study timed automata as a representation of some real-time problems, mainly for the purpose of verification of trajectories of the system [Alur, 1999, Bengtsson and Yi, 2004]. Because of this interest in verification, a method was developed which reduces the infinite nature of the trajectories of a timed-automaton to a more useful countable representation. The method creates a *region automaton*, which considers the relative values of clocks: more details can be found in [Alur, 1999, Alur et al., 1993a].

There are many special classes or extensions to timed automata that have been posed over the years. One of these is event clock automata, which are a subclass of timed automata which have the advantage of being determinisable, that is every nondeterministic event clock automaton can be transformed into a deterministic event clock automaton which describes the same timed regular language [Alur et al., 1999]. These automata only have two types of clock — *event-recording* clocks which record the time elapsed since the event they measure occurred, and *event-predicting* clocks which give the time until the next occurrence of the event.

An extension to timed automata is the *pushdown timed-automaton*, which is a timed-automaton augmented with a stack [Dang, 2003, Dang et al., 2004]. The use of this stack makes it possible to describe and verify certain properties that cannot be described with classic timed automata. Another extension to timed automata are the class of *alternating timed automata* [Lasota and Walukiewicz, 2008, Ouaknine and Worrell, 2005], which are nondeterministic timed automata with two types of transition conditions — an `OR` type transition, where a certain input takes the current state to one of a choice of states, and an `AND` type transition, where a certain input takes the current state to both of two new states. These alternating timed automata, even with only one clock, can accept more languages than normal timed automata, and with only one clock have the advantage of a decidable emptiness problem.

Another type of timed automata are *priced timed automata*, where each transition or location we visit uses a certain amount of "energy", and we wish to know how the energy in the system changes with different paths through the timed-automaton [Bouyer et al., 2010]. These type of automata are useful for considering how we can achieve tasks with certain time constraints, which also have energy requirements.

There is also work on how we implement timed automata in practice. In [Altisen and Tripakis, 2005], the authors consider the transformation of a timed-automaton into a program, which can then be executed and checked for whether it satisfies a desired property. The concept of *perturbed timed automata* has been considered by Alur et al. [2005], which assumes that the clocks of the timed-automaton do not necessarily change at a rate of 1, but could change at a rate in the range $[1 - \epsilon, 1 + \epsilon]$. This reflects what could happen with practical clocks. Similarly, robust safety of timed automata is considered by De Wulf et al. [2008], by allowing the clocks to drift by an amount ϵ, and also allowing the guards on transitions to be widened by a positive amount Δ.

There are a large number of different timed automata available. In Chapters 3 and 4 we will use the type of timed automata defined in Section 2.2.3, as they are suitable for our purpose.

2.2.5 An overview of stability in hybrid systems

The concepts involved in Lyapunov stability can be extended from smooth systems to hybrid systems in an intuitive fashion, as they are simply the ideas of 'staying close to a point' or 'tending towards a point'. We will not give definitions here, as we do not require them, but we will discuss how to prove such properties in hybrid systems, as the usual Lyapunov function methods are only valid for smooth systems.

When proving stability of hybrid systems via Lyapunov functions, there are two main ways that we can extend the continuous methods to hybrid systems:

1. The use of multiple Lyapunov functions, one for each subsystem [Branicky, 1994, 1995, 1998]. The idea is to find Lyapunov functions for each subsystem, and impose some conditions on when the trajectories are allowed to switch between

the subsystems, so that the combined effect of the Lyapunov functions ensures that all trajectories of the hybrid system converge to a stable equilibrium if they start within the domain of attraction of this equilibrium.

2. The use of a common Lyapunov function for all the subsystems [Liberzon, 2003, Vu and Liberzon, 2005]. Here the idea is to find a single function that is a Lyapunov function for each of the subsystems, but also can ensure stability across transitions between subsystems.

It is possible to show that for switched systems, a subclass of hybrid systems, asymptotic stability holds if and only if a common Lyapunov function exists [Liberzon, 2003]. However, this does not mean that we can find such a function — in practice the multiple Lyapunov function method may be easier to use to prove stability.

These results on common and multiple Lyapunov functions are mainly useful for the class of switched systems, and this is one of the few classes of hybrid systems with practical stability results. There have also been a lot of results on stability in reset control systems [Baños and Barreiro, 2012]. However, there have been very few stability results for general hybrid systems, or at least, very few practically applicable results, but there are some ideas worth mentioning, which may be useful for specifications of systems more general than switched systems.

Goebel et al. [2009] have introduced the idea of pre-asymptotic stability of compact sets in hybrid systems, which is asymptotic stability without the condition that maximal solutions are complete. This means that solutions do not need to be defined for all time, so they may stop at a certain point in time. This sort of relaxation is important for hybrid systems, as we can easily define systems for which solutions are not defined after a certain point in time — the bouncing ball is a good example of this, since it tends to a limit point in time where it will have stopped bouncing.

Some other references for Lyapunov-type stability for hybrid systems can be found in Cai et al. [2007, 2008], Decarlo et al. [2000], Goebel et al. [2009], He and Lemmon [1998], Ye et al. [1998]. Some other stability-like properties which have been investigated in hybrid systems are stabilisation [Amato et al., 2011a] where we want to control a system to a stable state, robustness [Pettersson and Lennartson, 1996] which means that perturbed Lyapunov functions can still prove stability, and dissipativity

[Haddad and Chellaboina, 2001, Zhao and Hill, 2008] which is the study of energy in a system to show stability.

To calculate Lyapunov functions for hybrid systems, various methods have been defined. These include works by Johansson and Rantzer [1998], Mojica-Nava et al. [2010], Oehlerking and Theel [2009]. We can also calculate the domain of attraction by works such as those by Amato et al. [2011b], Ratschan and She [2010] — the domain of attraction is usually a lot larger than any set created by a Lyapunov function. Another definition of stability-type properties is the notion of set invariance [Blanchini, 1999], with calculations of invariant sets through methods such as Haimovich and Seron [2010], Sankaranarayanan [2010].

2.3 Formal verification

The basic idea of formal verification is to prove in an automated way that the behaviour of a system is "correct" according to some specification. This correctness can be formed in terms of a *safety* condition, for instance by specifying an unsafe set of states that we do not want to reach, or could be a *liveness* condition, for instance where we specify that we would like to reach a particular set of states in the end [Guéguen and Zaytoon, 2004]. Formal verification originated within computer science to prove that software or hardware did what it was designed to. It has since been applied to prove properties about continuous-time systems, and also hybrid discrete-continuous systems.

Broadly speaking, there are two approaches to formal verification [Kowalewski, 2002]. The first is *model checking*, where the model of the system is checked using algorithms to automatically see if it satisfies the specification. The second approach is *deductive verification*, or *theorem proving*, where we use known axioms and rules to logically prove the correctness of a system.

In this section we discuss the theory behind model checking, and then specialise to the discussion of model checking of hybrid systems. We will then discuss theorem proving in general and the progress that has been made towards hybrid system theorem proving. We will finish with a broad overview of the logics that can be used to specify properties in continuous and hybrid systems. In this thesis, we will use mainly model checking of properties, so we will spend the majority of this section discussing this.

2.3.1 Model checking

Model checking is the process of testing automatically whether a system meets the desired specification. The system is described by a formal model (for instance a finite state machine or a hybrid automaton), temporal logic is used to specify desired properties of the system, and an efficient search procedure is used to check the specification against the model [Emerson, 2008]. The algorithms used to compare the specification and the model make use of the structure of the model to be able to reduce the infinite nature of the system to something finite, which can be checked in a finite length of time.

Model checking should always terminate with a “yes” or a “no” for any given input, since either the whole state space is searched and found to be good, or the search terminates with a counterexample that does not solve the specification. However, if we have a very large system, the model checking process could incorrectly terminate due to a lack of memory.

Some advantages of model checking over other formal verification methods (basically, theorem proving) are considered to be [Clarke, 2008]:

- The process is automatic — there is no need for the user to be involved beyond the specification of the system and the constraints.
- It is fast in comparison to other formal verification methods in practice.
- When a property is falsified, a counterexample is produced, which can help to correct the behaviour of complex systems.
- Model checking can be performed on a partial specification of a system, which can help in the specification of the rest of the system.

The main drawback of model checking is the state-explosion problem [Clarke and Grumberg, 1987]. This is where complex systems create very large numbers of states in the course of model checking, which become hard to check on any finite machine in a reasonable time. In software and hardware model checking there have been attempts to get around this problem through the use of partial order reduction (exploiting concurrency) [Godefroid, 1994], abstraction (only modelling relevant parts of the system) [Clarke et al., 1994], and symmetry reduction (utilising regularities in the system)

[Ajami et al., 1998, Emerson and Sistla, 1996]. Perhaps the most effective method, however, has been the use of bounded model checking (BMC) [Biere et al., 1999], where all finite paths of length k steps are considered at each step k of the algorithm. Bounded model checking is very effective at decreasing the number of states produced during model checking, as the length of the paths is minimal at any point in the algorithm.

2.3.2 Model checking of hybrid systems

In hybrid systems, model checking approaches are generally based on the same three elements as in classical model checking. These are [Guéguen and Zaytoon, 2004]:

1. A modelling formalism to model the system under study.
2. The specification of the desired properties.
3. A verification algorithm.

There are three types of specified properties that have usually been considered for hybrid systems, these are *safety*, *liveness* and *timeliness* properties. Safety properties specify “bad” configurations that must never happen, whereas liveness properties specify that something “good” must eventually happen. Timeliness properties give constraints on the times between occurrence of some particular events or solutions — this could be a maximum or a minimum time between two particular points in the system.

In hybrid systems, since we have a continuous state space in addition to discrete behaviour, we must deal with this continuous behaviour in some way. There are two approaches to this — either the given model of the system is used, or the model is abstracted so that we can use algorithms designed for simpler systems. These two approaches suit different types of specifications. For a recent overview of tools for verification of hybrid systems, see Carloni et al. [2006].

Directly checking the model of the hybrid system

If we want to check some property of the model of the hybrid system directly, there are many different problems encountered, and so there are many different solutions that have been proposed. Some of the proposed methods work directly on the exact dynamics, but only for specific classes of hybrid systems, and some methods rely on approximations, but work for more general systems.

Most of the current work is to prove safety properties of hybrid systems, with the safety properties considered being specified in terms of sets of the hybrid state space that the system should avoid or remain in.

Such safety properties are rewritten in terms of reachability of the initial set, or backwards reachability of a final set. That is, a safety property given by an initial set *Init* and a safe set *Safe* could be proved by forward reachability as:

1. Calculate the set reachable from the initial states, $\mathit{Reach}(\mathit{Init})$.
2. If this set is included in the safe set, $\mathit{Reach}(\mathit{Init}) \subseteq \mathit{Safe}$, then the system is safe.

Alternatively, by backwards reachability we obtain:

1. Calculate the backwards reachable set from the safe states, $\mathit{BackReach}(\mathit{Safe})$.
2. If this set contains the initial set, $\mathit{BackReach}(\mathit{Safe}) \supseteq \mathit{Init}$, then the system is safe.

It is also possible to use the unsafe set as the final set, and then we want to show that the trajectories of the system never enter it. Using forward reachability we want to show that $\mathit{Reach}(\mathit{Init}) \cap \mathit{Unsafe} = \emptyset$, where *Unsafe* is the unsafe set, and $\emptyset$ is the empty set. Similarly, for backward reachability, we want $\mathit{BackReach}(\mathit{Unsafe}) \cap \mathit{Init} = \emptyset$ [Dang, 2000].

There are broadly two ways of finding reachable sets: through exact calculation of the reachable space, and through approximated calculation of the reachable space. The exact methods are called symbolic analysis methods [Alur et al., 1997, Damm et al., 2012, Henzinger, 1994], and they attempt to extend the discrete systems idea of enumerating the state space. That is, we use logical constraints or inequalities to represent exact sets that can be reached from the initial set of a hybrid system. Because we represent exact reachable sets we can only consider certain kinds of simple dynamics. The original paper by Alur et al. [1997] considered only linear hybrid automata so that everything can be written in terms of linear inequalities.

The second class of methods which use approximated reachability calculations can consider more complex dynamics, but do require a property-preserving approximation of the system. That is, when we prove the equivalent property on the approximated

Final set	Direction	Condition for safety	Safety preserving under...
Safe	Forward	$\mathit{Reach}(\mathit{Init}) \subseteq \mathit{Safe}$	Over-approximation
Safe	Backward	$\mathit{BackReach}(\mathit{Safe}) \supseteq \mathit{Init}$	Under-approximation
Unsafe	Forward	$\mathit{Reach}(\mathit{Init}) \cap \mathit{Unsafe} = \emptyset$	Over-approximation
Unsafe	Backward	$\mathit{BackReach}(\mathit{Unsafe}) \cap \mathit{Init} = \emptyset$	Over-approximation

Table 2.1: Conditions for the safety property to hold for the different pairs of final set and reachability direction. We also show the type of reachability approximation that preserves the condition.

system, we will effectively prove the desired property on the original system [Halbwachs et al., 1994, Tomlin et al., 2003]. To be able to prove safety results we must deliberately either over-approximate or under-approximate the sets so that we can prove the actual sets will satisfy the condition [Mitchell, 2007]. For instance, if we are doing forward reachability with a safe final set, we will check that the approximated forward reachable set from *Init* is contained in *Safe*, or $\mathit{ApprReach}(\mathit{Init}) \subseteq \mathit{Safe}$. But this will only prove that $\mathit{Reach}(\mathit{Init}) \subseteq \mathit{Safe}$ if $\mathit{Reach}(\mathit{Init}) \subseteq \mathit{ApprReach}(\mathit{Init}) \subseteq \mathit{Safe}$. Hence, for this property to be proved, we must use over-approximation of the reachable set, since then $\mathit{Reach}(\mathit{Init}) \subseteq \mathit{ApprReach}(\mathit{Init})$ holds. Table 2.1 summarises the important information for the four types of safety-related reachability calculations.

There are many methods proposed for finding reachable sets, for example through the use of zonotopes [Althoff et al., 2007] or polytopes [Amato et al., 2011b] for representing the reachable space. Polyhedra are also used to represent the reachable space, and there are many tools based on this idea [Asarin et al., 2002, Chutinan and Krogh, 2003, Frehse, 2008, Henzinger and Ho, 1998, Henzinger et al., 1997]. Interval constraint propagation [Henzinger et al., 2000, Ramdani and Nedialkov, 2011] is also used, which effectively uses hyper-rectangles to represent the reachable space. Another type of methods are level set methods [Cross and Mitchell, 2008, Mitchell and Tomlin, 2000], where certain limits on a special function are used to limit the space that the dynamics can exist in, and related to this is the concept of barrier certificates [Prajna and Jadbabaie, 2004, Prajna et al., 2007, Sloth et al., 2012], where a boundary which the dynamics can only cross in one direction (a 'barrier') is found in the space to separate the reachable space from the unsafe set.

Possibly the most promising method that has been proposed recently for calculating reachable sets is called *hybridization* [Dang et al., 2011]. The basic idea is to find over- and under-approximating linearised dynamics for the nonlinear hybrid system and use these to calculate an over-approximation of the reachable space using these very simple dynamics. Each limiting approximation will only be valid in a small set, and so when we go outside the set where the linearisation is valid, we create a new linearisation of the space for the next part of the trajectory. Repeating this along the course of the reachable space allows us to quickly build up a picture of the reachable space by using reachability on linear dynamics, but we achieve this by only performing linearisation along the parts of the trajectory which are part of the reachable space.

All of the above work on directly checking the model of the hybrid system is focussed on reachability for proving safety. We will defer discussion of the work on liveness until Section 2.4.

Checking the system using an abstracted model for the hybrid system

The idea of abstracting a discrete model from the system is that we want to capture all of the useful features of the system, but leave out details that are not necessary. For instance, if we can create a finite-state automaton which captures relevant aspects of a hybrid system, then we may be able to use a model checking algorithm for finite-state automata to prove properties on the hybrid system — such algorithms are well-developed.

The idea is to create an abstracted model that is *similar* to the original model, in the sense that every evolution of the hybrid model has a corresponding evolution in the abstraction. This leads to the idea of *simulation relations*, which tell us if two models relate. Ideally we would like every evolution of the abstraction to relate to an evolution of the hybrid model as well, so that the two models are *bisimilar*. The most recent and helpful book on this subject is by Tabuada [2009], which discusses bisimulations for hybrid systems at length.

Some work uses these simulation relations to relate timed automata to untimed automata, so that classical model checking methods can be used [Tripakis and Yovine, 2001], whilst Alur et al. [2000] demonstrated some classes of hybrid systems can be related to purely discrete systems. Henzinger et al. [1998] translated rectangular hybrid automata into timed automata, and in another work Henzinger and Ho [1998] abstracted hybrid automata by two different types of linear hybrid automata (for particular classes of hybrid automata). These latter abstractions have been implemented in the HyTech model checker [Henzinger et al., 1997]. The properties of bisimulations have been discussed particularly in Henzinger [1995, 1996].

There is also the idea of using simulation relations to relate smooth and hybrid dynamical systems to timed automata, so that timed model checking methods can be used. One such method by Olivero et al. [1994] finds classes of linear hybrid systems which have bisimilar timed automata for proving any properties in timed computational tree logic (TCTL) — we will discuss this logic in Section 2.3.4. In Chapters 3 and 4 of this thesis we will effectively be finding bisimulations for classes of smooth and hybrid systems with linear derivatives — this is different to the work by Olivero et al. [1994], as they consider linear hybrid systems which have constant derivatives (and linear dynamics).

Other abstractions to timed automata include two methods by Stursberg et al. [2000], which both consider what kind of abstraction is created by interpreting a rectangular splitting of the state space as a timed automata. One of the methods defines the discrete transitions between rectangles by considering the flow between the rectangles, and the other method defines the transitions by considering numerical integration of the dynamics in the rectangles. The first method was picked up by Maler and Batt [2008], who added the idea of *slices* to the abstraction to attempt to make a tighter abstraction of the system, as the first method of Stursberg et al. [2000] loses a lot of information in the abstraction. The idea of slices on their own was picked up by Sloth and Wisniewski [2011, 2013], who related them to Lyapunov functions to prove timing properties on polynomial smooth systems.

Tools for proving properties on timed automata include PRISM [Kwiatkowska et al., 2011], IF [Bozga et al., 2002], KRONOS [Daws et al., 1996], and UPPAAL [Behrmann et al., 2004]. Of these, UPPAAL and PRISM are the most developed and most efficient tools, offering support for many different types of TA, with UPPAAL being suitable for analysing non-probabilistic systems, and PRISM being suitable for probabilistic systems. In this thesis, we do not consider probabilistic properties, so we will use UPPAAL when we make abstractions of hybrid systems to TA in Chapters 3 and 4.

More recent work using simulation relations includes extensions to *approximate simulation relations*, where the distance between the actual and abstracted systems is bounded by a certain precision [Girard et al., 2008]. This extension to approximate similarity means that these simulation relations can be used for a much wider class of hybrid systems. Simulation relations have also been used to relate models of systems to their specifications, to show that the models behave as wanted [Kerber and van der Schaft, 2010].

We should also make a note here about other ways of using discrete abstraction for hybrid systems. A basic principle is to make an abstraction which simulates the real system, but refine this abstraction if it is not good enough. Some examples of this are by Gentilini et al. [2007], Henzinger et al. [2002] One of these is called *counter-example guided abstraction refinement*, or CEGAR, which is where we start with a coarse grained abstraction for the hybrid system, and use counter-examples to improve the discrete model of the hybrid system, so that we capture all the useful information [Alur et al., 2006, Clarke et al., 2003a,b]. Some interesting work in this context is by Klaedtke et al. [2007] and Ratschan and She [2007], who define practical algorithms for implementing CEGAR model checking for hybrid systems. This work has led to a basic implementation for formal verification of hybrid systems: HSolver. Another method for abstraction of hybrid systems was defined by Tiwari and Khanna [2002, 2004], and implemented in the tool HybridSAL[3].

2.3.3 Deductive verification for hybrid systems

Also known as theorem proving, deductive verification involves using logical inference to prove that a formal specification holds. The idea comes from mathematical proof techniques, where a list of axioms and proof rules are known and are used to show that a certain conclusion holds (or does not hold). There are many implemented theorem provers in classical computer science, for many different logical specifications. As we do not use deductive verification in this thesis, we will not consider the classical computer science techniques, but will simply give an overview of the methods for proving properties about hybrid systems.

Theorem proving has been rather neglected in the formal verification of hybrid systems and dynamical systems in general, because model checking advanced much

[3]The website for HybridSAL is `sal.csl.sri.com/hybridsal`.

quicker and became much more useful. However, the state-explosion problem has hindered model checking, and so there have been efforts to improve deductive verification of hybrid systems. Most of these efforts have focussed on the theoretical side of hybrid systems verification, but a few examples of practical deductive verification have been attempted.

One interesting theoretical contribution is [Kapur et al., 1994]. This work proposes a methodology for the specification, verification and design of hybrid systems, and uses hybrid temporal logic (HTL), hybrid automata, and concrete phase transition systems. The authors of this work have established useful deductive rules for continuous-time logics.

Another interesting work uses inductive assertion to make statements about hybrid systems (in form of automata), in both individual and composed forms [Ábrahám-Mumm et al., 2001]. This is a proof methodology, and is implemented using an already established interactive theorem prover, PVS [Owre et al., 1992]. Also called a proof assistant, PVS performs logical proof of properties about systems, with some automated strategies of proof alongside some other rules which are applied by the user. For some interesting applications of PVS to prove the correctness of real-arithmetic see the work by Lester and Gowland [2003], Lester [2012].

A well developed tool for the logical verification of hybrid systems is KeYmaera [Platzer, 2010, Platzer and Quesel, 2008]. It makes use of symbolic mathematics for solving differential equations, real algebra rules for manipulating real numbers, along with logical proof rules to actually prove the propositions made. There was also been an effort to make a mixed theorem prover and model checker, called STeP [Bjørner et al., 1996, Manna and Sipma, 1997]. However, this was never developed enough to be useful. The most likely developments in this area will come from either the team behind KeYmaera, or from interactions between current real-number theorem provers with the hybrid systems community. Some potential provers include the proof assistant PVS [Ábrahám-Mumm et al., 2001, Archer and Heitmeyer, 1997, Graf and Saidi, 1997], which can be extended to use various different underlying theories, or the real-arithmetic theorem prover MetiTarski [Akbarpour and Paulson, 2009, 2010, Denman et al., 2009], where support for hybrid systems is underway.

2.3.4 Temporal logics

Temporal logics add the idea of time into logical statements. In timed systems this is essential, because a predicate formula that is not true at the moment might be expected to hold eventually. There are broadly two approaches in temporal logics — linear time logics and branching time logics. The two commonly used specifications within these categories are called linear temporal logic (LTL), and computation tree logic (CTL), respectively [Emerson, 2008]. These logics are related, but can each describe some properties that the other cannot, and are both contained in a higher specification called CTL*.

In this section we will give a brief overview of these two types of logics as they relate to hybrid systems, and will then discuss other kinds of logics that can be used for specifying real-time hybrid dynamical systems. Good overviews of logics for real-time systems include those by Alur and Henzinger [1991] and Davoren and Nerode [2000].

Linear time logics

A linear time logic is based on the idea of time being one time line, with the future fixed, although usually unknown. We can reason about what actually happens in a system, without considering all the other possibilities that do not actually occur. The most used linear time logic is linear temporal logic (LTL)[4]. It consists of the predicate logic with temporal operators that can act on these formulae to tell us about when they occur. It was first introduced by Amir Pnueli [Pnueli, 1977] — here we introduce LTL for real-time systems, rather than discrete-time ones.[5]

The available temporal operators are given below.

- $\Box$ means "always" (classically denoted G, for "globally"). The formula $\Box f$ is true if f is true from now onwards.
- $\Diamond$ means "eventually" (classically denoted F, for "in the future"). The formula $\Diamond f$ is true if f becomes true at some point in the future (or f is true now).

[4]LTL is sometimes referred to as propositional temporal logic (PTL).

[5]This only means that we lose the usually defined *next state* operator, $\bigcirc$.

- $\mathcal{U}$ means "until", and it is a binary operator. The formula $f\mathcal{U}g$ is true if at some point in the future g holds, and until then f must hold.

- $\mathcal{R}$ means "release", and it is another binary operator. The formula $f\mathcal{R}g$ should be read "f releases g", so that the formula is true if g holds at least until it sees f become true once.

In LTL these operators can be applied to each other to form more complicated expressions. For instance, $\Box\Diamond f$ means that f becomes true infinitely often, whereas $\Diamond\Box f$ means that at some point in the future f becomes true and then remains true for all time. Note that this second example is not the same as "when f becomes true it remains so for all time", as $\Diamond\Box f$ allows f to become true and then go false again as many times as it likes before it then must become true forever.

There are some relationships between the operators — in fact the whole set of temporal properties can be written in terms of the operator $\mathcal{U}$: we have that $\Box f \equiv \neg\Diamond(\neg f)$ with $\Diamond f \equiv \top\mathcal{U}f$, and that $f\mathcal{R}g \equiv \neg(\neg f\mathcal{U}\neg g)$.

The main extensions to LTL for hybrid systems are *metric temporal logic* (MTL) [Koymans, 1990, Ouaknine and Worrell, 2005] and *metric interval temporal logic* (MITL) [Alur et al., 1996a]. MTL constrains the temporal operators by intervals, so that for instance $\Diamond_{[1,2]}f$ means that f holds at least once between 1 and 2 time units from now. The intervals that can be considered as constraints are either closed intervals, half open, or open intervals, and also intervals which go to positive infinity. However, the possibility of requiring exact timing for formula satisfaction makes MTL undecidable [Alur and Henzinger, 1994], and so Alur et al. [1996a] defined the new logic MITL where these equalities are not allowed.

Branching time logics

Branching time logics consider the possibility of an event occurring on some computation path, or the guarantee that it definitely occurs on all computation paths. This logic reasons about whether some event could occur or definitely will occur in our system, but does not actually relate to the specific path that we will see in the system. Computation tree logic (CTL) is the most widely used branching time logic, and it is based on a logic introduced by Emerson and Clarke [1980].

A computation tree is a representation for the computation steps in a state transition system, where each computation step is effectively a time step. Progressing through the tree gives us a path which is a possible run for the system. The computation tree logic has two path quantifiers:

- A — for all computation paths.
- E — for some computation path.

It also has the same temporal operators as LTL: $\Box$, $\Diamond$, $\mathcal{U}$, and $\mathcal{R}$, but these are only acted upon by the path quantifiers, not by each other. The basic CTL formula $\mathrm{E}\Box f$ is true in state s if f is true in all states along some possible path of the system starting at s, whereas $\mathrm{A}\Box f$ is true in s if f is true in all states along all possible paths of the system that start in s. The minimum set needed to express any formula in CTL is E, and $\mathcal{U}$, along with the propositional logic operators.

Extensions of the branching time logics have mainly been directed towards timed logics for timed automata systems. One example of this is *timed computation tree logic* (TCTL) [Alur et al., 1993a], which puts clocks in the system that constrain the temporal operators with time. For example, we could have the operator $\Diamond_{\leq 5}$, which requires that its operand becomes true at least once within 5 time units. The full set of TCTL formulae can be defined by the propositional logic operators, and the operators E, A, and $\mathcal{U}$, with time constraints on the $\mathcal{U}$ operator.

Another CTL extension is *integrator computation tree logic* (ICTL) [Alur et al., 1996b], which uses stopwatches, rather than clocks, in the system. The logic is called *integrator* CTL because these stopwatches are referred to as integrators. An integrator increases with time only when we are in one of a predefined set of states, and is stopped from increasing at other times. These sort of time counters are particularly useful for posing constraints on the comparison of the lengths of time we are in particular sets of states.

Other kinds of logics for dynamical and hybrid systems

There has been a lot of research over the years into how these logics (and others) could be extended to dynamical or hybrid systems. A good summary paper for these logics is [Davoren and Nerode, 2000], which contains most of the logics used today. We will just mention these logics for completeness, as we will not make use of them in this thesis.

Some hybrid logics which use intervals are *hybrid temporal logic* [Henzinger et al., 1993] and the *extended duration calculus* [Chaochen et al., 1993]. These are part of a larger class of logics, outside of CTL* and its parent, the μ-calculus. The temporal logic of actions (TLA) [Lamport, 1994] is also another formalism outside of the typical CTL* logics.

The last logic we will consider is *differential-algebraic dynamic logic* (DAL) [Platzer, 2008, 2010]. This logic combines discrete jump constraints with first-order differential-algebraic constraints. It has similar temporal operators to the other temporal logics, except that the temporal operators depend on the "program" we are running: $[\alpha]\phi$ means every run of program α leads to states satisfying ϕ, and $\langle\alpha\rangle\phi$ means at least one run of program α leads to a state satisfying ϕ. In practice a "program" is the system specification, along with an initial condition specification.

2.4 Liveness properties

A liveness property says that something good will eventually happen in a system, and is the type of property considered in this thesis. The concept was initially proposed by Lamport [1977] for the purpose of proving correctness of a program.

To avoid confusion, we should distinguish the concept of liveness from the concept of livelock, which comes from the theory of parallel processes. Livelock and deadlock are both notions of two (or more) processes interfering with each other to stop useful motion occurring. Deadlock is where the two systems block each other from any further motion, whereas livelock is when the states of the systems keep changing, but the desired thing never happens. In this sense, both deadlock and livelock are like counter-examples to liveness, as they prove that the desired thing does not happen. In the world of parallel programs, tight definitions of deadlock and livelock are given by Tai [1994], and in hybrid control systems definitions are given by Abate et al. [2006, 2009]. A hybrid control system consists of interacting hybrid systems with controllers, so the ideas of systems blocking each other (deadlock) or being stuck in an infinite cycle where they make no progress (livelock) carry over in a suitable fashion.

2.4.1 Liveness in discrete systems

In discrete–space-time systems, a generally accepted formal definition of liveness was given by Alpern and Schneider [1985, 1987]. They defined properties as ω-regular languages: sets of infinite-length sequences of symbols formed by using certain operations on the set of symbols. Alpern and Schneider then used language inclusion to say whether a particular execution of the system satisfied the property. We now present their definitions for liveness in discrete–space-time systems.

Let S be the set of states of the discrete system (for instance a finite-state automaton), then S^* is the set of finite sequences of states, and S^ω is the set of infinite sequences of states. If we consider a run (or execution) of the system, ϕ, then this run can either be an infinite run or a finite run. We will call a finite run a *partial* run, because we assume that every finite run $\alpha \in S^*$ can be extended to an infinite run $\phi = \alpha\beta \in S^\omega$, where $\beta \in S^\omega$. We now formalise what a liveness property is.

Definition 2.11 (Liveness [Alpern and Schneider, 1985]). $P \subseteq S^\omega$ is a liveness property on a discrete–space-time system if

$$\forall \alpha \in S^* \, \exists \beta \in S^\omega : \alpha\beta \in P. \tag{2.3}$$

■

This formal definition can be put into words as *every finite run can always be extended to an infinite run which satisfies the liveness property.* The reason that this characterises the notion of liveness that "eventually something good happens" is because a property occurring eventually means that it cannot be disproved with a finite run of the system — it could always happen at some point in the future.

Alpern and Schneider [1985] also show that, using their definition, every property can be written as an intersection of a safety and a liveness property, by translating them to a topology on S^ω where safety properties are the closed sets and liveness properties are the dense sets. Later, Chang et al. [1992] gave a characterisation of liveness in terms of canonical temporal logic properties. For more history on the development of theory and proof methods around liveness, see Kindler [1994].

There are also other properties which can be put into a categorisation with safety and liveness, like progress (anything that is not a pure safety property), guarantee (also called inevitability), response (something keeps on happening), and persistence (eventually something always happens) [Baier and Kwiatkowska, 2000, Chang et al., 1992]. Guarantee, response and persistence properties are classes of liveness. All of these properties have their use in different kinds of systems, and all have different types of proof method which suit them best.

Methods for proving liveness are different to those for proving safety — this is part of the reason the two characterisations were initially made by Lamport [1977], who also proposed the method of proof lattices for proving liveness. This method is complex, and so other methods were proposed, by Flon and Suzuki [1981], Owicki and Lamport [1982] amongst others. More recent work includes that by Biere et al. [2002], who translate the problem of proving or disproving liveness into a reachability problem, so that safety proving tools can be used. Their idea is based on finding lasso-shaped counterexample traces using a state recording translation. A recent tool for proving liveness and termination of a program by counter-example guided abstraction refinement (CEGAR) is Terminator [Cook et al., 2005, 2007], which can automatically analyse a program to determine if it eventually does something good.

A related class of systems for which liveness has been considered is infinite-state systems. For example Bouajjani et al. [2005] discuss the problem of verifying liveness properties of systems with an infinite number of discrete states using the method of regular model checking, where the emptiness of certain automata for ω-regular languages are used to prove the properties. Liveness has also been used in continuous-space discrete-time systems, for instance by Cook et al. [2011] who prove stabilisation of biological systems.[6] Damm et al. [2005] also look at proving LTL properties for discrete-time hybrid systems, and show that their process terminates for robust non-linear hybrid systems.

2.4.2 Liveness in continuous and hybrid systems

In continuous-time systems, liveness has mainly been considered in timed automata and other discrete-space models. For instance, a formal framework for liveness in timed automata was defined by Segala et al. [1998], who also defined an embedding of an untimed version of the system to prove liveness. Practical proof of liveness in timed

[6]Stabilisation says that a system eventually settles down to a stable value.

automata is also well developed — tools like UPPAAL [Behrmann et al., 2004] and PRISM [Kwiatkowska et al., 2011] have the inbuilt ability to prove liveness properties, with PRISM also able to show with what probability a liveness property is true on a probabilistic timed-automaton or continuous-time Markov chain.

The situation in continuous–space-time systems is much less comprehensive, with well-developed results only in certain classes of hybrid systems. These classes include linear hybrid automata [Alur et al., 1996b], where the dynamics are always solvable, and also systems for which piecewise constant bounds on the derivatives can be given [Henzinger et al., 1998], where approximations of the reachable space are easy to find.

Some academics working in the area of hybrid systems consider time-bounded liveness properties instead of liveness properties on infinite time, and these include Behrmann et al. [2005], Girard and Zheng [2012]. Such bounded liveness properties say that "something good eventually happens within a certain period of time", and are effectively safety properties in the way they are proved. This means that methods already existing for safety proving in hybrid systems could potentially be used to prove them. However, there are some essential properties of systems which cannot be transformed into a time-bounded form, like infinite recurrence of states for instance. Hence, this thesis considers unbounded liveness properties, and uses linear temporal logic (LTL) to formulate them as it is expressive enough for our needs.

The main aim of this thesis is to consider practical methods for proving liveness in hybrid systems, as well as defining a formal framework. There is already some work on proving liveness in continuous and hybrid systems, with two main classes of properties considered. The first class considered is *inevitability properties*, which are those of the form $\Diamond\phi$ with ϕ a formula in propositional (not temporal) logic, and the second class is *region stability*, which is of the form $\Diamond\Box\phi$, where ϕ is again a formula in propositional logic. This region stability property is effectively a persistence property as defined in Section 2.4.1, but usually has the restriction that ϕ describes a contiguous set in the continuous state space.

A recent work which proves inevitability in hybrid systems is due to Duggirala and Mitra [2012], who focus particularly on proving inevitability over the discrete transitions in the system. An approach more focussed on the continuous dynamics is that of Sloth and Wisniewski [2011], who use Lyapunov functions to abstract a continuous system by timed automata which can then be used to prove inevitability properties of a continuous dynamical system. One of the contributions of this thesis uses a method which abstracts a continuous or piecewise-continuous dynamical system to a timed-automaton (Chapters 3 and 4), but we will consider how hyper-rectangles can be used to divide the space, rather than chosen functions.

Another way to prove inevitability properties is through using reachability, in a somewhat analogous approach to the typical proofs of safety in hybrid systems. To prove inevitability by reachability, we need to show that all trajectories eventually satisfy the desired formula ϕ, which we can think of as representing a set of the space. For proving safety properties, typically over-approximation of reachable spaces is used, as it is easier than under-approximations. However, to prove inevitability properties, that every trajectory eventually reaches the set described by ϕ, the only pair of time direction and space approximation which preserve the property under the approximation is to go backwards in time from the desired set and use under-approximations of the (backwards) reachable space. Making under-approximations of reachable spaces is not that well developed, but there are a few examples: Gentilini et al. [2007] propose a method to produce successive abstractions to verify all kinds of CTL properties on hybrid automata, including those which can use under-approximations to prove liveness properties, and Ghosh and Tomlin [2004a,b] use under-approximations to consider which sets reach specified steady states in biological models.

The second liveness property which has been considered in hybrid systems is the concept of region stability, which says that the trajectories of the system all eventually reach some region (or set) and stay there forever, although the trajectories can go in and out of this region a finite number of times before remaining there. This is called a stability property in dynamical systems terminology, as it is saying that the trajectories of the system eventually settle down to being in the region forever. The main body of work on proving this property is by the group of Andreas Podelski of the University of Freiburg, and their methods mostly use the idea of snapshot sequences to specify that only a finite amount of time of a trajectory can be spent outside of the region [Mitrohin and Podelski, 2011, Podelski and Wagner, 2006, 2007a,b,c]. Another method to prove region stability properties was proposed by Duggirala and Mitra [2011], who discuss how to use program analysis tools to analyse these kind of properties in hybrid automata. Bogomolov et al. [2010] also prove region stability by composing reachability analyses, dwell time, and extra variables.

Some work that considers more general temporal logic properties is by Batt et al. [2007], who discuss the way to prove CTL properties of genetic networks. In particular, they consider how to remove spurious time-converging behaviours from discrete abstractions of the systems, to ensure progress of time. This problem of progress of time is one that makes it very difficult to make abstractions of hybrid systems that can prove liveness. It is one reason why Chapters 3 and 4 consider abstractions to timed automata, which explicitly introduce time into the abstraction.

Liveness properties are characterised by the idea that, at any finite point in an execution, they could always be satisfied at some point in the future, or alternatively that the only type of execution which can disprove such a property is one of infinite length. In continuous space-time it can very difficult to be able to find infinite length paths, and so Chapter 5 proposes a new type of dynamically-aware property, which can disprove such liveness properties with a finite length path. This is the concept of *deadness*, which captures the idea that there could exist another property which, if it is true, implies that the liveness property can never hold in the system. Chapter 5 also considers a way to find such deadness properties automatically for certain kinds of hybrid systems, which is a step forward in proving liveness in hybrid systems.

Chapter 3

Proving inevitability in continuous dynamical systems

In this chapter we look at the problem of proving inevitability of continuous dynamical systems. An inevitability property says that a region of the state space will eventually be reached. This is a type of liveness property and is related to attractivity of sets in dynamical systems. We consider a method of Maler and Batt [2008] to make an abstraction of a continuous dynamical system to a timed-automaton, and show that if the timed-automaton it creates satisfies the equivalent inevitability property then the original system will satisfy inevitability. We then identify the problems that can occur with using the timed-automaton to prove inevitability if the state space splitting it is based on is not made by considering the dynamics of the system. In particular, the main problems we identify are that we can have pairs of states with two-way flow between them in the timed-automaton abstraction (giving infinite discrete transitions in finite time), and that the maximum time spent in a box of the splitting can be infinite. We define a method which solves these problems for the abstraction for a class of linear dynamical systems, and we show that the resulting timed abstraction proves inevitability.

The main contributions of this chapter are:

1. Proposing a dynamically-driven splitting of the state space of the system.
2. The proof that this splitting method creates a timed-automaton that will prove inevitability of the original system.

A preliminary version of this chapter was presented at FORMATS 2012 [Carter and Navarro-López, 2012].

3.1 Introduction

An *inevitability* property is a type of liveness property, and has the informal definition "the region L is eventually reached by all trajectories of the system", where L can represent any region in the state space of the system. In Section 2.3.2 we discussed the methods available for proving such properties — there are not many, and those that do exist are not that general. We wish to make inroads into the proof of such inevitability properties, by finding new methods to prove them, and to do so we will take the path of proof by abstraction to a simpler system. In Section 2.3.2 we said that abstractions suitable for proving liveness properties need to have progress of time enforced, and that the simplest abstraction with timing is the timed-automaton (TA). Also, reachability analysis on TA is decidable, and so inevitability on the TA will definitely be either proved or disproved, although what this means for the properties of the continuous system itself is dependant on how the abstraction is made.

We consider the method of Maler and Batt [2008] for abstracting a continuous system to a TA. The method is based on splitting the state space, and then analysing the flow across the boundaries to determine whether the TA has an edge between the two neighbouring states. The authors did not specify how the state space of a system should be split to create the TA, but stated that the accuracy of the model could be improved indefinitely by increasing the number of splits. However, in most dynamical systems splitting without considering the system's behaviour will not be very good at capturing the dynamics of the system, and we will require either a very large or an infinite number of splits to guarantee that we can verify properties of the system.

In this chapter we present a method for creating a *dynamically-driven* splitting of the state space of continuous dynamical systems, using known properties of the flow of the system to decide how to make the splits. We identify a terminating splitting method for a class of upper-triangular linear systems which ensures that the resulting timed-automaton will always prove the inevitability property. Our method is most closely related to that of Sloth and Wisniewski [2011], which abstracts continuous systems to timed automata using the idea of the method of Maler and Batt [2008], but Sloth and Wisniewski use Lyapunov functions to define the slices considered, whereas we use the original idea of constant variable slices.

3.2 The systems and method being considered

In this section, we consider all autonomous[1] n-dimensional continuous space-time dynamical systems, of the form

$$\dot{x} = f(x), \tag{3.1}$$

with $x = [x_1, \ldots, x_n]^T \in \mathbb{R}^n$ and $f(x) = [f_1(x), \ldots, f_n(x)]^T$, with $f : \mathbb{R}^n \to \mathbb{R}^n$ smooth. The state space of the system is assumed to be a finite box, which is defined by the limits $x \in [s_{1,-}, s_{1,+}) \times \ldots \times [s_{n,-}, s_{n,+}) = S \subseteq \mathbb{R}^n$. Let the region that we are interested in starting trajectories from be a set $\mathit{Init} \subseteq S$.

The method we consider is from Maler and Batt [2008]. It is an approximation method defined to abstract a continuous system to a TA, and is based on minimum and maximum velocities of a system defining bounds on the time taken to cross a certain distance in the system. The resulting TA is an over-approximation of the system, in the sense that every trajectory in the system is matched in time and space by one in the abstraction, but additional trajectories may be allowed in the abstraction (see Maler and Batt [2008] for more discussion of this).

3.2.1 Slicing the space and the discrete abstraction

The basic idea of Maler and Batt [2008] is to split the state space into slices by making splits along lines of the form $x_i = c$, where c is constant. These slices also define hyper-rectangles (which we refer to as *boxes*) of the space by the intersection of a slice in each dimension. Figure 3.1 shows a two-dimensional (2-D) example ($x \in \mathbb{R}^2$), with the middle slice in dimension 1 outlined by a dashed line, and the last slice in dimension 2 outlined by a dotted line. The box created by the intersection of these two slices is shaded. In the end, each of these boxes becomes a state of the timed-automaton.

[1]An autonomous dynamical system is one where the right-hand side of the equation does not explicitly depend on time.

A slice is a part of the state space S, restricted only in one dimension. In this work we allow slices to be of differing widths, defined by a vector of split points Verts_i in each dimension i. We will include the endpoints of the splitting vector, so that the first and last elements of Verts_i are $s_{i,-}$ and $s_{i,+}$, respectively. Let v_i be an index which indicates which slice we are considering in dimension i, and then the slice is given by

$$X_{i,v_i} = \big[s_{1,-}, s_{1,+}\big) \times \ldots \times \big[\mathit{Verts}_i(v_i), \mathit{Verts}_i(v_i+1)\big) \times \ldots \times \big[s_{n,-}, s_{n,+}\big) \subseteq S. \quad (3.2)$$

Slices are right-open so that they do not intersect, and so the set of slices for each i will form a partition of the state space S. The slices in one dimension being a partition of the state space means that: (1) the union of all slices in one dimension is the whole state space, and (2) the intersection of any two slices of the same dimension will be empty. If there are m_i slices in the i-th dimension this is:

$$\left(\bigcup_{j=1,\ldots,m_i} X_{i,j}\right) = S, \text{ and}$$
$$\forall\, 1 \le j\,, k \le m_i \ \big(j \ne k \ \Rightarrow\ X_{i,j} \cap X_{i,k} = \emptyset\big).$$

A *box* is defined by the intersection of a slice in each dimension. Let us choose the slice X_{i,v_i} in dimension i, which is the v_i-th slice in this dimension. Let us write $v = [v_1, \ldots, v_n]$ for the index of a box which is the intersection of the v_i-th slice in each dimension i. Then, the box is defined as

$$X_v = \bigcap_{i=1}^{n} X_{i,v_i} \subseteq S.$$

The set of all boxes in the partitioning of S is denoted by V.

Let us consider a box labelled by $v = [v_1, \ldots, v_i, \ldots, v_n]$ and define the neighbours of v by $\sigma^{\pm i}(v) = [v_1, \ldots, v_i \pm 1, \ldots, v_n]$ — notice that this function gives a box defined by the intersection of the same slices as v in all dimensions except dimension i, where the box defined by $\sigma^{\pm i}(v)$ is either in the slice directly above or directly below v. Boxes v and $\sigma^{\pm i}(v)$ therefore share an edge, hence why we call them neighbours, and we can define the facet (edge) between them as

$$F(v, \sigma^{\pm i}(v)) = \mathrm{cl}(X_v) \cap \mathrm{cl}(X_{\sigma^{\pm i}(v)}).$$

Note that this facet lies in an $(n-1)$-dimensional space, whereas the boxes are in n-dimensional space.

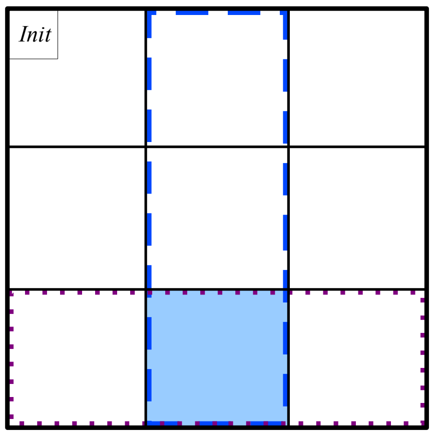


Figure 3.1: Partition the state space into boxes (rectangles in the 2-D case).

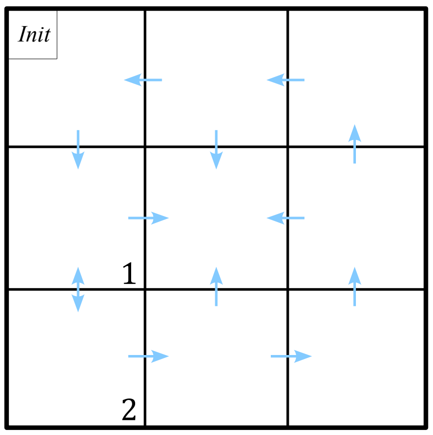


Figure 3.2: Calculate which directions the boundaries can be crossed in (denoted by arrows).

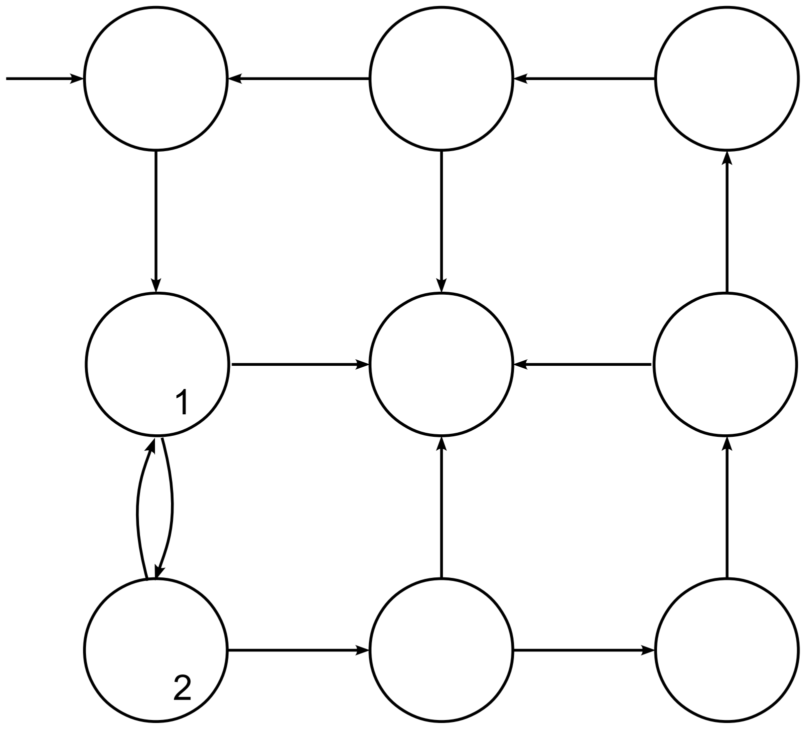


Figure 3.3: The finite automaton abstraction resulting from the method of Definition 3.1 applied to the dynamical system.

We will now make a finite-state automaton from this state space splitting, which will form the discrete structure of the timed-automaton. Each box of the splitting becomes a state in the finite-state automaton. To find possible crossings between these automaton states we consider the sign of the velocity on the facet between each pair of neighbouring boxes (see Fig. 3.2 to visualise the crossings in an example). We define the set of initial states in the automaton as those labelled by v's for which the related box contains some of the initial set, that is $X_v \cap \mathit{Init} \neq \emptyset$.

Definition 3.1 (Automaton abstraction [Maler and Batt, 2008])**.** The automaton $A = (V, V_0, \delta)$ is an automaton abstraction of the continuous system of Equation (3.1) if it consists of:

- A set of discrete states V labelled by vectors $v = [v_1, \dots, v_n]$, each related to a box of the continuous system's state space splitting.
- An initial set of states $V_0 \subseteq V$, where $V_0 = \{v \in V : X_v \cap \mathit{Init} \neq \emptyset\}$.
- A transition relation $\delta \subseteq V \times V$, where δ consists of neighbouring pairs of boxes $(v, v') \in V \times V$ where, if $v = [v_1, \dots, v_i, \dots, v_n]$ and $v' = [v_1, \dots, v_i + 1, \dots, v_n]$, then $\exists x \in F(v, v')$ such that $f_i(x) > 0$, or if v is as above with $v' = [v_1, \dots, v_i - 1, \dots, v_n]$ then $\exists x \in F(v, v')$ such that $f_i(x) < 0$. ■

This automaton abstraction says that if a facet can be crossed by the dynamics of the system, then the corresponding transition must exist in the relation δ. In particular, we take into account whether the facet is crossed in the positive or negative direction, and add a transition correspondingly. Note that there is nothing stopping a facet from being able to be crossed in both the positive and negative directions, so δ can contain both (v, v') and (v', v), as will be the case with the locations marked 1 and 2 in Fig. 3.3.

Definition 3.2 (Run of the automaton abstraction)**.** Let a run of the automaton abstraction be defined as the sequence of states that get passed through as it shadows the continuous dynamical system. We will write this as $\alpha = v^0, v^1, v^2, \dots$, where $v^0 \in V_0$ is the discrete state where the run α starts, with v^1 the next discrete state that this run passes through, and so on for the other v^i. ■

Let us consider any trajectory $x(t)$ of the continuous system and assume it starts in state v^0, so that the continuous state at time zero is in the box labelled by v^0, that is, $x(0) \in X_{v^0}$. The continuous state of the system evolves for some length of time still in the box X_{v^0} and then at time t_1 say, it crosses the facet $F(v^0, v^1)$. Immediately after this point in time, the automaton run is $\alpha = v^0, v^1$. As the continuous system trajectory evolves and crosses more facets, the automaton run gets the relevant states added to it. Notice that one run of the automaton is the abstraction of many continuous trajectories, and so this automaton abstraction is an over-approximation of the continuous system, in terms of the number of trajectories.

3.2.2 The timed-automaton abstraction

We now consider how Maler and Batt [2008] extend this to form a timed-automaton abstraction for the system. The extra thing we require is clocks to keep track of the times at which crossings are made in each dimension. In order to keep an over-approximation in terms of number of trajectories of the finite-state automaton abstraction, we must over-approximate maximum bounds on time of clocks, and under-approximate minimum bounds on clocks. This means, if we cannot be exact about limits, that we widen the range of times when we are allowed to take a transition, allowing more trajectories through. This keeps the over-approximation.

Firstly let us consider how to bound the time we spend in a state v of the timed-automaton, equivalent to a box X_v. Let d_i be the width of the box X_v in dimension i, then the maximal time that it can take to leave this box is over-approximated by the *box time*, defined as

$$\overline{t_v} = \min_{1 \le i \le n} \left(\frac{d_i}{\min(|f_i|) \text{ in box } X_v} \right). \tag{3.3}$$

If $\min(|f_i|) = 0$ for every dimension i in box X_v, then we define $\bar{t}_v = \infty$. We take the minimum over the dimensions here, as each dimension limits us in the amount of time we can spend in the box, and the minimum of these will force us to leave the box.

We can also limit the time spent in slices of the space, both above and below, in the positive and negative directions. Let us consider the slice X_{i,v_i} in dimension i, and let d_{i,v_i} be the width of this slice in dimension i. Also let $\underline{f_{i,v_i}} = \min_{x \in X_{i,v_i}} f_i$ be the minimum velocity in this slice, and $\overline{f_{i,v_i}} = \max_{x \in X_{i,v_i}} f_i$ be the maximum velocity.

	$\underline{t^+_{i,v_i}}$	$\overline{t^+_{i,v_i}}$	$\underline{t^-_{i,v_i}}$	$\overline{t^-_{i,v_i}}$
$0 < \underline{f_{i,v_i}} < \overline{f_{i,v_i}}$	$d_{i,v_i}/\overline{f_{i,v_i}}$	$d_{i,v_i}/\underline{f_{i,v_i}}$	∞	∞
$\underline{f_{i,v_i}} < \overline{f_{i,v_i}} < 0$	∞	∞	$-d_{i,v_i}/\underline{f_{i,v_i}}$	$-d_{i,v_i}/\overline{f_{i,v_i}}$
$\underline{f_{i,v_i}} < 0 < \overline{f_{i,v_i}}$	$d_{i,v_i}/\overline{f_{i,v_i}}$	∞	$-d_{i,v_i}/\underline{f_{i,v_i}}$	∞

Table 3.1: Minimum and maximum times that can be spent in slice i in the positive and negative directions (respectively).

Then the minimum ($\underline{t_{i,v_i}}$) and maximum ($\overline{t_{i,v_i}}$) times that can be spent in this slice in the positive (+) and negative (−) directions are given in Table 3.1 (*slice times*).

If we enter a slice X_{i,v_i} from the lower face in dimension i, then the minimum time we can take to leave by the opposite face is $\underline{t^+_{i,v_i}}$, and the maximum time to leave by the opposite face is $\overline{t^+_{i,v_i}}$, and similarly for entering from the upper face with $\underline{t^-_{i,v_i}}$ and $\overline{t^-_{i,v_i}}$. We use these values to bound timed-automaton clocks within slices of the space: there are two clocks per dimension, z_i^+ which bounds positive direction movements using $\underline{t^+_{i,v_i}}$ and $\overline{t^+_{i,v_i}}$ for each v_i, and z_i^- which bounds the negative direction movements. We also use a box clock z in the TA to satisfy the conditions on how long we can stay in each box, using (3.3). See Figure 3.4 for an example of the clock limits calculated on slices of the space and in each box.

We should make a note here about why we do not consider minimum times spent in a box. In order to consider minimum times spent, we need to know where we enter the box from, as if we start from one edge we might be able to exit straight away, but if we start from another we could remain in the box for a non-zero length of time. Because of this we would need to have one clock for each facet of the box which we could enter from, which adds $2n$ clocks to the timed-automaton. As the complexity of proving information about TAs increases exponentially with the number of clocks, it becomes unreasonable to have this many extra clocks for limited benefit — at the end of the day for progress through the automaton it is more useful to know the maximum amount of time we can spend to force a run of the timed-automaton to move on to another state when this maximum time is reached.

Let us now define this timed-automaton formally. The maximum time limits are used to form the invariant for each location, *forcing* a change in discrete state when they are reached, and the minimum time limits are used in the guard condition, *enabling* transitions to occur once they have been reached. We use the notation $\perp$ to indicate that a clock is inactive, in line with the general definition of a TA given in Definition 2.10 of Chapter 2.

Definition 3.3 (Approximating timed-automaton [Maler and Batt, 2008])**.** Given a dynamical system of the form of Equation (3.1), with finite state space S, its approximating timed-automaton is $\mathcal{TA} = (V, V_0, Z, I, \Delta)$ where V is a set of labels for the states as given in Def. 3.1, and $V_0 \subseteq V$ is the set of initial states as in Def. 3.1. We also define $Z = \{z, z_1^+, \ldots, z_n^+, z_1^-, \ldots, z_n^-\}$ as a set of clocks, and I as an invariant defined for every state $v = (v_1, \ldots, v_n)$ by

$$I_v = z < \overline{t_v} \wedge \bigwedge_{i=1}^{n} (z_i^+ < \overline{t_{i,v_i}^+}) \wedge (z_i^- < \overline{t_{i,v_i}^-})$$

with $z < \infty$ interpreted as true for each clock. The transition relation Δ consists of the following transition types $\delta = (v, g, \rho, v')$:

$$\delta_v^{+i} : (v,\ z_i^+ \geq \underline{t_{i,v_i}^+} \vee z_i^+ = \perp,\ z_i^+ := 0; z_i^- = \perp; z := 0,\ \sigma^{+i}(v)) \tag{3.4}$$

and

$$\delta_v^{-i} : (v,\ z_i^- \geq \underline{t_{i,v_i}^-} \vee z_i^- = \perp,\ z_i^- := 0; z_i^+ = \perp; z := 0,\ \sigma^{-i}(v)) \tag{3.5}$$

provided that such transitions are possible in the discrete automaton abstraction of Definition 3.1.

For each initial discrete state $v \in V_0$, the set we take as the initial region for clocks is

$$Z_v = \{0\} \times [0, \underline{t_{1,v_1}^+}) \times \ldots \times [0, \underline{t_{n,v_n}^+}) \times [0, \underline{t_{1,v_1}^-}) \times \ldots \times [0, \underline{t_{n,v_n}^-}), \tag{3.6}$$

so that all possible points in the box X_v are made possible as initial values in the timed-automaton abstraction. This preserves the over-approximation with respect to the number of trajectories included. ∎

Figure 3.5 shows an example of a TA resulting from abstraction of a continuous system. The location which gets the initial transition is the top left one, because that is where the initial region was contained in the state space diagram of Fig. 3.4. The initial region for clocks is defined by Equation (3.6), and uses the lower slice time bounds from Figure 3.4.

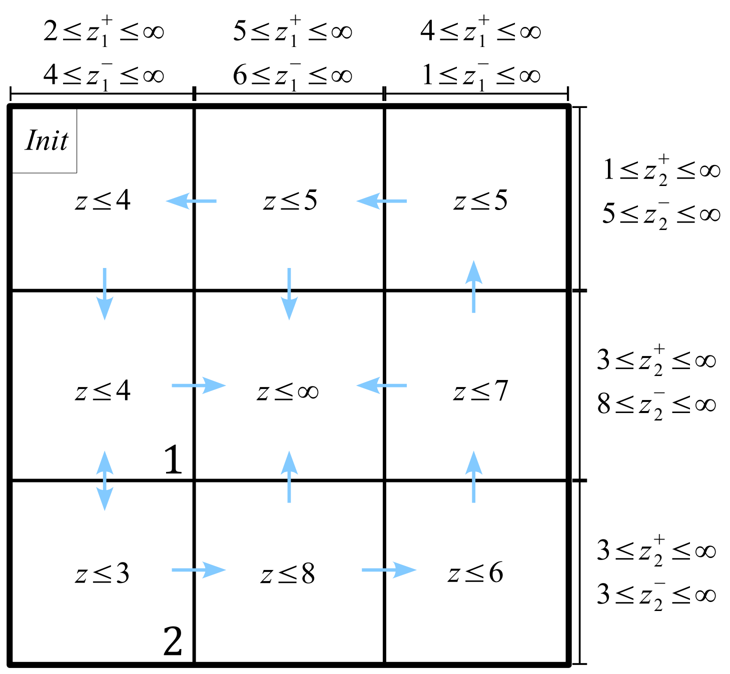

$2 \leq z_1^+ \leq \infty$
$4 \leq z_1^- \leq \infty$
$5 \leq z_1^+ \leq \infty$
$6 \leq z_1^- \leq \infty$
$4 \leq z_1^+ \leq \infty$
$1 \leq z_1^- \leq \infty$
Init
$z \leq 4$
$z \leq 5$
$z \leq 5$
$1 \leq z_2^+ \leq \infty$
$5 \leq z_2^- \leq \infty$
$z \leq 4$
$z \leq \infty$
$z \leq 7$
$3 \leq z_2^+ \leq \infty$
$8 \leq z_2^- \leq \infty$
1
$z \leq 3$
$z \leq 8$
$z \leq 6$
$3 \leq z_2^+ \leq \infty$
$3 \leq z_2^- \leq \infty$
2

Figure 3.4: Calculate the maximum box time by (3.3) and assign as limits to clock z, and calculate the minimum and maximum limits on the slice clocks in each slice by Table 3.1, and assign as limits to the clocks z_1^+, z_1^-, z_2^+, and z_2^-.

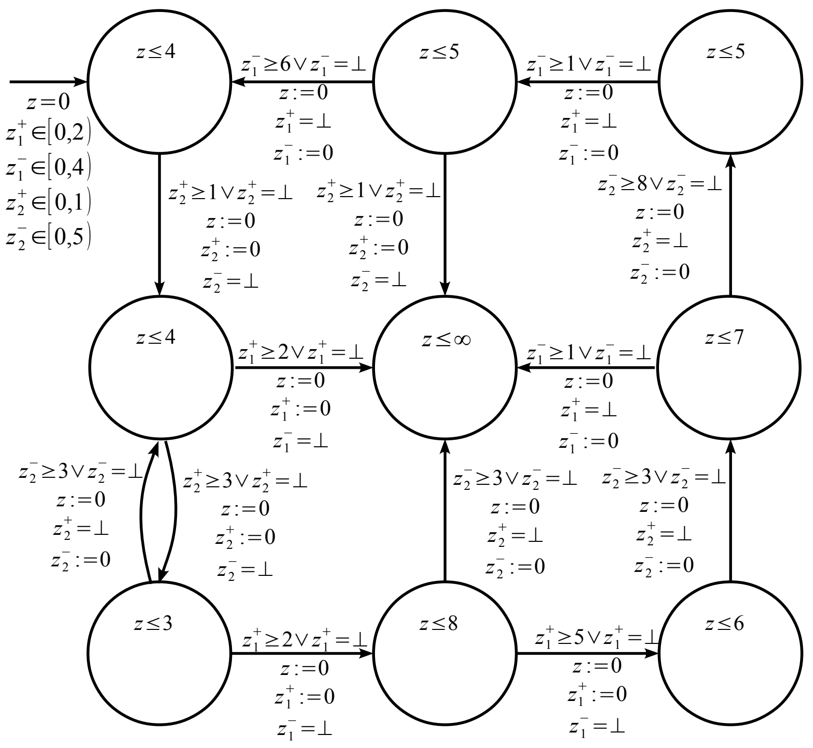

$z \leq 4$
$z \leq 5$
$z \leq 5$
$z = 0$
$z_1^+ \in [0,2)$
$z_1^- \in [0,4)$
$z_2^+ \in [0,1)$
$z_2^- \in [0,5)$
$z_1^- \geq 6 \vee z_1^- = \perp$ $z := 0$ $z_1^+ = \perp$ $z_1^- := 0$
$z_1^- \geq 1 \vee z_1^- = \perp$ $z := 0$ $z_1^+ = \perp$ $z_1^- := 0$
$z_2^+ \geq 1 \vee z_2^+ = \perp$ $z := 0$ $z_2^+ := 0$ $z_2^- = \perp$
$z_2^+ \geq 1 \vee z_2^+ = \perp$ $z := 0$ $z_2^+ := 0$ $z_2^- = \perp$
$z_2^- \geq 8 \vee z_2^- = \perp$ $z := 0$ $z_2^+ = \perp$ $z_2^- := 0$
$z \leq 4$
$z \leq \infty$
$z \leq 7$
$z_1^+ \geq 2 \vee z_1^+ = \perp$ $z := 0$ $z_1^+ := 0$ $z_1^- = \perp$
$z_1^- \geq 1 \vee z_1^- = \perp$ $z := 0$ $z_1^+ = \perp$ $z_1^- := 0$
$z_2^- \geq 3 \vee z_2^- = \perp$ $z := 0$ $z_2^+ = \perp$ $z_2^- := 0$
$z_2^+ \geq 3 \vee z_2^+ = \perp$ $z := 0$ $z_2^+ := 0$ $z_2^- = \perp$
$z_2^- \geq 3 \vee z_2^- = \perp$ $z := 0$ $z_2^+ = \perp$ $z_2^- := 0$
$z_2^- \geq 3 \vee z_2^- = \perp$ $z := 0$ $z_2^+ = \perp$ $z_2^- := 0$
$z \leq 3$
$z \leq 8$
$z \leq 6$
$z_1^+ \geq 2 \vee z_1^+ = \perp$ $z := 0$ $z_1^+ := 0$ $z_1^- = \perp$
$z_1^+ \geq 5 \vee z_1^+ = \perp$ $z := 0$ $z_1^+ := 0$ $z_1^- = \perp$

Figure 3.5: The TA resulting from the method of Maler and Batt [2008] applied to the dynamical system (3.1). In this case, no slice clock has an upper limit, so they do not appear inside locations.

Theorem 3.4 (Neo conservatism [Maler and Batt, 2008]). *For every trajectory $x(t)$ of (3.1) starting from a point $x(0) \in X_v$, there is at least one run ξ of $\mathcal{TA}$ starting from (v, Z_v) which shadows the trajectory of the continuous system in both time and space. That is, if the timed-automaton observes the continuous trajectory and evolves with it, it will allow the resulting trajectory to occur.*

This theorem says that every trajectory of the continuous space-time system has at least one matching trajectory in the timed-automaton. This is equivalent to saying the continuous system is over-approximated by the timed-automaton abstraction, in terms of the number of trajectories included. For the proof of this theorem see Maler and Batt [2008].

3.2.3 Preservation of inevitability

We now begin to think about how the method of Maler and Batt [2008] can be used to prove inevitability in the continuous system of (3.1). Let us write the inevitability property we wish to prove as $x(0) \in \mathit{Init} \Rightarrow \Diamond(x(t) \in L)$, where $\mathit{Init} \subseteq S$ and $L \subseteq S$ are sets in the state space. That is, we want to show that all trajectories that start in an initial set *Init* eventually will reach a set L, which we call the *live set*.

Let us now think about the timed-automaton abstraction created by the method of Maler and Batt [2008]. This method is an over-approximation of the continuous dynamical system in terms of the number of trajectories, as stated in Theorem 3.4. It would seem to be the case that if we can prove that all TA runs from the initial region eventually reach the live set L, we will have included all trajectories of the original system, and hence will have proved that all trajectories of the continuous system will reach the live set. However, there is a technicality around using this abstraction for proving inevitability of the original system, which is that we lose information about trajectories which evolve out of the finite state space.

Using a finite state space is essential to be able to have a finite TA abstraction both in terms of number of discrete states and values of time limits on the boxes, but it does create problems with eventual time limits. What is it that we are interested in when we talk about all trajectories eventually reaching a set? If a trajectory evolves across the edge of the state space, effectively becoming non-existent after a certain point in time, then do we wish to say this is an irrelevant trajectory, or would we prefer to use this trajectory as a counter-example to the inevitability property, as it does not reach the desired set? In some sense, either is valid, as we really may just be interested in what happens within the state space, and anything which evolves outside is not relevant. But, from the formal verification point of view, it does not sit well to ignore swathes of trajectories which do not technically satisfy the inevitability property.

In this thesis, we will assume that this problem is solved, either by us deciding we are not interested in trajectories which go outside of the state space, or by us selecting a state space such that no trajectories leave it (effectively a trapping region). When we consider linear systems in Section 3.4 onwards, we will assume that the state space is a trapping region, and we will show how we can over-approximate the state space for such linear continuous dynamical systems to ensure the over-approximation is a trapping region, hence preserving the over-approximation.

To prove that all runs of the TA abstraction reach L, we will need to have one (or more) boxes in the split space which are equivalent to L, so that the inevitability property becomes 'reach one of the selected states' in the TA. If we need to approximate the live set by a box X_l, then we should use an under-approximation in the abstraction, since we will then prove that all abstraction runs reach the box $X_l \subseteq L$, proving that all trajectories of the continuous system eventually reach the box X_l, and as $X_l \subseteq L$ this proves that all trajectories of the continuous system from the initial set *Init* reach the live set L.

Let us state and prove this assertion formally.

Theorem 3.5 (Preservation of inevitability)**.** *Consider a continuous space-time dynamical system of the form of Eq. (3.1), and let the desired live set in the continuous system be L. Let $X_l \subseteq L$ be a box labelled by the indexing vector l and let $\mathcal{TA}$ be a timed-automaton abstraction of this system made by the method of Maler and Batt [2008], with the box X_l as one of the boxes in the state space splitting. If it can be proved that all runs of $\mathcal{TA}$ reach the TA state l, then this implies that the live set L is reached by all trajectories of the continuous system, which we call* preservation of *inevitability.*

Proof. Let us assume that all runs of $\mathcal{TA}$ reach the TA state l. Then, Theorem 3.4 shows that the TA contains an over-approximation of the number of trajectories of the continuous dynamical system, hence all trajectories of the continuous system which start anywhere in $\cup_{v\in V_0} X_v$ will reach the box X_l, that is,

$$\left(x(0) \in \bigcup_{v\in V_0} X_v\right) \Rightarrow \Diamond\left(x(t) \in X_l\right). \tag{3.7}$$

Now, $v \in V_0$ includes all boxes which contain a part of the continuous initial set *Init*, which means that $\mathit{Init} \subseteq \cup_{v\in V_0} X_v$. Therefore, as (3.7) holds for all $x(0) \in \cup_{v\in V_0} X_v$, we can restrict the result to only the subset *Init*, giving

$$\left(x(0) \in \mathit{Init}\right) \Rightarrow \Diamond\left(x(t) \in X_l\right). \tag{3.8}$$

We also know that $X_l \subseteq L$, where L is the desired set for the continuous system, and as (3.8) holds for reaching the box X_l which forms part of L, we can say that

$$\left(x(0) \in \mathit{Init}\right) \Rightarrow \Diamond\left(x(t) \in L\right).$$

This is exactly the inevitability property we want to prove, and so the TA abstraction preserves inevitability. □

3.3 Problems for general continuous systems

If we want to use a TA abstraction to prove inevitability of a continuous system then once the abstraction is made we need to show that all possible runs of the TA will reach the desired location within finite time. This requires us proving two things about each possible run of the TA:

1. The run must pass through a finite number of states in the TA to reach the desired state v.

2. The run must leave every state on route to the desired v within a finite time.

If (1) is not true for some TA run ξ then we cannot guarantee that the desired state is ever reached by ξ, and hence we cannot guarantee that L is reached by every trajectory of the continuous system. And if (2) is not true for some TA run ξ then ξ can spend infinite time in some state which is not the desired state v, which does not guarantee that the desired state is reached *in finite time*. Hence we cannot prove that L is reached in finite time by every trajectory of the continuous system.

There are various reasons why the method of Maler and Batt [2008] does not satisfy these two key conditions, some of which we highlight in this section. We will also start to look at how to solve some of these problems.

One case when this TA abstraction does not work well to prove inevitability properties is when the trajectories of the system do not fit well with splitting based on crossing constant variable lines. A particular example of this is for 2-D linear systems with complex eigenvalues, where the trajectories of the system are spirals. Even when the real parts of the eigenvalues are negative and we know that the trajectories go inward by stability theory, the timed-automaton can allow flow round the edge of the split space without forcing us to move closer to the centre of the spiral (see for example Fig. 3.5, where the central state should be reached, but the TA allows flow through the eight states around the edge indefinitely). This problem is inherently one of the discrete splitting and the finite-state automaton (see Figs. 3.2 and 3.3), and nothing can be done to rectify this with the timings of the clocks in the TA.

Another problem with the discrete abstraction for general continuous dynamical systems is the fact that it can allow pairs of automaton locations where the trajectories can go both ways across the shared face (see locations marked 1 and 2 in Figs. 3.2 and 3.3). The timed-automaton does not restrict when these transitions can be taken, as box times are not directional and slice times only limit the time to reach the *opposite* face, so these pairs of locations introduce Zeno behaviour into the abstraction, that is, infinite transitions in finite time [Johansson et al., 1999]. This kind of behaviour prevents every run of the abstraction from reaching the desired final location, so the inevitability property cannot be proved.

Another potential problem with this method is related to the calculation of the limits for clocks in the TA abstraction. We do not guarantee that the maximum box times $\overline{t_v}$ will be *finite* values, due to the fact that if a zero velocity occurs in the box v in every dimension, then $d_i / \min(|f_i|) = \infty$ in every dimension, and so (3.3) will give an infinite value for $\overline{t_v}$. If there is one box in the abstraction which has an infinite box time, then any run which reaches this box will never be forced to leave this box even if the actual trajectories of the system would all leave it in finite time.

Some of the above problems can be remedied by choosing an appropriate method for splitting the state space. The method of Maler and Batt [2008] does not specify how we should choose the splitting, but just tells us the properties of the resulting TA when a choice has been made. As we see it, there are two ways to do such a splitting: either we make the splitting arbitrarily (systematically but not based on the system's dynamics) and rely on refinement to eventually capture enough information about a system, or we can use properties of the dynamics to choose where to split the system. The pros and cons of these are discussed below.

Arbitrary splitting. This approach does not rely on knowledge about the structure of the dynamical system, and so it can be used for complex systems no matter where the complexity comes from. The TA created by the abstraction method of Maler and Batt does approach the actual dynamics (theoretically) as more and more splits are made (see Maler and Batt [2008]). However, due to the two issues (1) that transitions both ways between pairs of automaton locations can exist, and (2) that there can be an infinite over-approximation of the time it takes to get across a box, the number of splits required is often very large, if not infinite, and so we obtain a huge number of locations in the timed-automaton.

Using system properties for splitting. In this approach we use the dynamics of the system we are looking at to automatically split the state space in a way which removes or reduces some of the problems associated with arbitrary splitting. In specific systems, it should be possible to make a splitting which can be proved to satisfy desirable properties, for instance that the liveness property is automatically satisfied. Even in systems where we cannot prove the liveness property immediately, it may be possible to at least have a much better starting point for refining the abstraction. The main problems with this idea are that a splitting method will only work for a certain class of systems, and that there are no automatic splitting methods in existence already; methods need to be designed for many different types of systems.

Given the considerations above, the contribution of this chapter will be to start the process of finding dynamically-driven automatic methods to create splittings. We will work from a theoretical basis to show that certain types of linear systems have a splitting which proves inevitability by the TA abstraction, with the idea that future work can extend these methods to be useful for more general systems, for instance nonlinear systems, piecewise-continuous systems, or hybrid systems. In Chapter 4 we will consider how this method can be extended for classes of piecewise-continuous systems and will analyse the difficulties involved.

3.4 Inevitability of upper-triangular linear systems

The problems we identified in the previous section are all issues that can stop the runs of the TA abstraction from reaching the desired state v in finite time, which prevents the TA abstraction from proving inevitability of the continuous system. In this section we will show that, in *linear* continuous systems, assuming only four properties of the abstraction is enough to guarantee that the abstraction will prove the inevitability property when it holds in the continuous system. Of these four assumptions, only two are challenging to satisfy, and we will show how to satisfy them for a sub-class of linear continuous systems in Section 3.5.

From now on we will consider the class of upper-triangular linear dynamical systems, with no input vector:

$$\dot{x} = Ax = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ 0 & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{n,n} \end{pmatrix} x. \tag{3.9}$$

We assume that the eigenvalues (the diagonal entries of A) are strictly negative. This type of system has a unique equilibrium point at zero,[2] and the negative eigenvalues mean that this point is stable and attractive, which is asymptotic stability. This means that some non-empty region L containing the equilibrium point must be reached within a finite time. From a dynamical systems perspective we know this property is true, but we want to prove the same property computationally — the long-term goal of this work is to prove inevitability properties of dynamical systems which cannot be proved by dynamical theory (due to complexity).

We note that the only real restriction on the class of linear systems is the existence of real strictly negative eigenvalues: for any linear system of the form $\dot{x} = Bx +$

[2]The equilibrium point $\overline{x}$ satisfies the equation $\dot{x} = 0$, that is $A\overline{x} = 0$. Invertibility of A means the only equilibrium point is at $\overline{x} = 0$.

b with B having real strictly negative eigenvalues and b any real vector, it can be transformed to $\dot{x} = Bx$ by moving the centre of the axis without loss of generality, and can then be transformed to an equivalent upper-triangular system by making the Schur decomposition. Hence Eq. (3.9) encompasses all linear dynamical systems with strictly negative real eigenvalues.

3.4.1 Over-approximating the state space by a trapping region

In Section 3.2.3 we briefly discussed the problem of what we assume about trajectories which go outside of the state space. If the given state space is a trapping region then we do not need to consider what will happen when a trajectory leaves, as no trajectory can ever cross the boundary once it has entered the region. However, if the state space is not a trapping region then we do not automatically have this property, but we may be able to expand the region of space we consider so that we do have a trapping state space, whilst also including all of the original trajectories of the system. Whatever state space we start off with, we now show that we can create one which is an over-approximation of the original space and also a trapping region, for dynamical systems of the form of (3.9) with negative real values on the diagonal. Through this we can use the TA abstraction to prove that all trajectories from the given initial region will reach the desired live set in finite time.

Let us start off with the given state space $S = [s_{1,-}, s_{1,+}) \times \ldots \times [s_{n,-}, s_{n,+})$, and we assume the equilibrium point (which we want to reach) is in this space, that is $0 \in S$. We would like to expand S to a region where we have only inwards trajectories across each $n-1$ dimensional edge (facet) of S. That is, we require for each i that $\dot{x}_i|_{x_i=s_{i,-}} \geq 0$ and $\dot{x}_i|_{x_i=s_{i,+}} \leq 0$ for all $x_j \in [s_{j,-}, s_{j,+})$, for all $j = 1, \ldots, n$ with $j \neq i$. Using an inductive method, we will now form an over-approximation of the region S which satisfies these requirements, which is therefore a trapping region.

Base case: In dimension n, the dynamics of the system are $\dot{x}_n = a_{n,n}x_n$ with $a_{n,n} < 0$. Hence, $\dot{x}_n > 0$ when $x_n < 0$, and $\dot{x}_n < 0$ when $x_n > 0$. As $0 \in S$, we know that $s_{i,-} < 0 < s_{i,+}$ for each $i = 1, \ldots, n$. For $i = n$ in particular, we have $s_{n,-} < 0$ so $\dot{x}_n|_{x_n=s_{n,-}} = a_{n,n}s_{n,-} > 0$ (and opposite for $\dot{x}_n$ at $s_{n,+}$), which is what we require in dimension n for a trapping region.

Inductive case: Let us assume that we have an over-approximating state space $S_{k-1} \supseteq S$ which has only inwards flow in all of the dimensions $n-k+1, \ldots, n$. We now consider the dimension $n-k$, where we have the limits $x_{n-k} \in [s_{n-k,-}, s_{n-k,+})$. We would like to find a new over-approximating state space $S_k \supseteq S_{k-1}$ with the only limits changing being those in dimension $n-k$, which become $x_{n-k} \in [s'_{n-k,-}, s'_{n-k,+})$, where these $s'_{n-k,\pm}$ are to be chosen. We also want these boundaries to be trapping, that is $\dot{x}_{n-k} \geq 0$ when $x_{n-k} = s'_{n-k,-}$, and $\dot{x}_{n-k} \leq 0$ when $x_{n-k} = s'_{n-k,+}$.

Let us look only at the lower face of S_k in dimension $n-k$. The equation for dynamics in dimension $n-k$ is

$$\dot{x}_{n-k} = a_{n-k,n-k}x_{n-k} + a_{n-k,n-k+1}x_{n-k+1} + \ldots + a_{n-k,n}x_n, \text{ with } a_{n-k,n-k} < 0.$$

We would like to choose the lower limit $s'_{n-k,-} \leq s_{n-k,-}$ such that $\dot{x}_{n-k} \geq 0$ when $x_{n-k} = s'_{n-k,-}$ for all combinations of values of other x_i's which are possible in S_{k-1}. So the problem we need to solve can be rephrased in terms of the minimum (strictly, the infimum in real space) value of the second part of the dynamics:

$$\text{find } s'_{n-k,-} \leq s_{n-k,+} \text{ such that } a_{n-k,n-k}s'_{n-k,-} + \inf_{x \in S_{k-1}} \left(\sum_{i=n-k+1}^{n} a_{n-k,i}x_i \right) \geq 0.$$

The solution to this problem which gives a minimum sized state space is when we take the 'equals' part of the inequality if this is smaller than the original boundary value $s_{n-k,-}$, giving us

$$s'_{n-k,-} = \min \left(s_{n-k,-}\, , \frac{-1}{a_{n-k,n-k}} \inf_{x \in S_{k-1}} \left(\sum_{i=n-k+1}^{n} a_{n-k,i}x_i \right) \right). \tag{3.10}$$

Using analogous reasoning for the upper limit we get

$$s'_{n-k,+} = \max \left(s_{n-k,+}\, , \frac{-1}{a_{n-k,n-k}} \sup_{x \in S_{k-1}} \left(\sum_{i=n-k+1}^{n} a_{n-k,i}x_i \right) \right), \tag{3.11}$$

where 'sup' is the supremum (the least upper bound), and 'inf' the infimum (the greatest lower bound).

Note that the only difference between S_{k-1} and S_k is that we have increased the size of the box in dimension x_{n-k}. As the dynamics of dimension i only depend on the dimensions with higher labels (that is $i+1, \ldots, n$), the direction of the flow across the edges of S_k in dimensions $n-k+1,\ldots,n$ will not have changed with the widening of dimension $n-k$, so will still be inwards. Hence, the edges of S_k in the dimensions $n-k, n-k+1,\ldots,n$ have only got inwards dynamics.

Conclusion: By the base case and inductive part, we can use this method to create an over-approximating state space S_n which has only inwards flow in each dimension, independently of whether the original state space S was trapping or not.

The result of this is that if the system given does not have only inwards flow, then the method just outlined can be used to find a wider state space which does. Because of this, from now on we will now assume that the state space S has only inwards flow, so we consider all possible trajectories of the original system when we consider the TA abstraction.

3.4.2 Assumptions about the timed-automaton

There are four assumptions we make about the splitting, some more challenging to achieve than others. In Sect. 3.5 we will define a method which can satisfy these assumptions for a subclass of the systems under study, which shows these are not unreasonable assumptions to make.

The first assumption is about where the equilibrium point occurs in relation to the splitting, and is here to give us one (and only one) box that we are interested in reaching for the inevitability property.

Assumption 3.6. There is exactly one box L in the splitting which contains the equilibrium point, and the equilibrium is not on the boundary of L. We will call L the *live box.*

This assumption implies that no split is made at $x_j = 0$ for any dimension j, as then the equilibrium would be on the boundary of a box in dimension j. This assumption is theoretically easy to satisfy, as in a linear system of the form of (3.9) there is only one equilibrium, and it is at $x = 0$. So, in order to satisfy this assumption we can simply not allow a split of the state space at any point where $x_i = 0$. This will force the equilibrium $x = 0$ to be in between two slice boundaries $x_i = l_i^- < 0$ and $x_i = l_i^+ > 0$ for every dimension i, giving us a live box $L = [l_1^-, l_1^+) \times \ldots \times [l_n^-, l_n^+)$ which satisfies Assumption 3.6 above. The challenge with this is to preserve the other assumptions below at the same time, but we will show how to do this in Sect. 3.5.

The second assumption specifies that there are a *finite* number of boxes in the abstraction, which is necessary for a useful abstraction. This is not really a major assumption on the abstraction as it simply requires us to have a finite number of splits in each dimension, and avoiding an infinite number of splits is one of the reasons we are finding a dynamically-driven splitting method. However, it is still worth stating this assumption explicitly.

Assumption 3.7. The automaton abstraction (Def. 3.1) of the system has a finite number of discrete states.

The next assumption has to do with how transitions are allowed in the abstracted system. We do not want it to be possible to keep transitioning between a pair of discrete states in the timed-automaton, as discussed in Section 3.3. This assumption will be one of the main drivers for the splitting method in Section 3.5.

Assumption 3.8. The continuous flow across any facet only occurs in one direction. That is, if the facet is $x_j = c$ with the other x_i's within some box limits, then the velocity $\dot{x}_j$ across this face will either be always $\dot{x}_j \geq 0$ or always $\dot{x}_j \leq 0$.

The fourth assumption is related to the timing constraints in the TA, ensuring we leave every box along a run within a finite time. This is the second major assumption which will drive the splitting method in Section 3.5.

Assumption 3.9. In each box X_v, except the live box L, the box time $\bar{t}_v$ is finite.

3.4.3 Similarity with respect to inevitability

We will show that every system of form (3.9) is proved to satisfy the inevitability property by any TA abstraction satisfying the assumptions above. In other words, we mean that every abstraction satisfying the assumptions is bisimilar with respect to inevitability to the original continuous system (provided the live box has not changed size). To do this we will prove that the discrete automaton abstraction (Def. 3.1) of the continuous system only has finite runs, and that the only location with no outgoing edges (hence the only possible final location) corresponds to the live box L. We then use Assumption 3.9 to guarantee that we only spent a finite amount of time in each box on the way, showing that L is reached in finite time. For ease of notation in the proofs, we will label boxes by b, by which we mean $b = X_v$ for some labelling vector v.

Theorem 3.10. *Assume we have an n-dimensional system of the form of (3.9) with strictly negative entries on the diagonal, and an automaton abstraction created by the method of Maler and Batt [2008] satisfying Assumptions 3.6–3.9. Then the automaton abstraction of the continuous system only has finite runs.*

Proof. Assume (for a contradiction) that we can find a run of infinite length in the automaton. As the automaton abstraction must have a finite number of locations by Assumption 3.7, any infinite run must go through at least one discrete location infinitely often. Let us assume (without loss of generality) that the infinite run starts in the state that will be repeated infinitely often — this only means that we will ignore the finite prefix to this state's first occurrence. Then for any move in the infinite run that is made in the k-th dimension in the positive direction, we must be able to find a corresponding move in the k-th dimension in the negative direction, and vice-versa. We now prove, by induction, that there is no dimension whose crossings can be part of an infinite run.

Base case: In the n-th dimension, the dynamics of the system is $\dot{x}_n = a_{n,n}x_n$ with $a_{n,n} < 0$. Hence, across any slice boundary in the n-th dimension, if $x_n > 0$ then $\dot{x}_n < 0$, and if $x_n < 0$ then $\dot{x}_n > 0$. Therefore crossing any n-th dimensional slice boundary can only be done in one direction, and so cannot be reversed as is necessary for this type of crossing to be present in the infinite run. Hence n-th dimensional crossings are not involved in the infinite run.

Inductive part: Assume that $n-k+1, \ldots, n$ dimensional crossings are not involved in the infinite run, and so the $x_{n-k+1}, \ldots, x_n$ variables are within one slice each for this run — let us assume (without loss of generality) that these slices containing the infinite run are defined by $x_i \in [c_{i-}, c_{i+})$ for $i = n-k+1, \ldots, n$. Let us consider a box b on the infinite run with limits $[b_{1-}, b_{1+}) \times \ldots \times [b_{n-k,-}, b_{n-k,+}) \times [c_{n-k+1,-}, c_{n-k+1,+}) \times \ldots \times [c_{n-}, c_{n+})$ where the $[c_{i-}, c_{i+})$ are the fixed slices we defined above. Then, for each box facet $x_{n-k} = c$ with $c \in \{b_{n-k,-}, b_{n-k,+}\}$ the derivative satisfies either always $\dot{x}_{n-k} \geq 0$ or always $\dot{x}_{n-k} \leq 0$ on this facet by Assumption 3.8. We will assume for ease of writing that the case $\dot{x}_{n-k} \geq 0$ holds on the box facet where $x_{n-k} = b_{n-k,+}$ and we will label the facet by F_{n-k} — the other three pairings of facet and derivative sign are analogous.

As the systems under consideration are upper-triangular, the dynamics of x_{n-k} has the form $\dot{x}_{n-k} = f(x_{n-k}, x_{n-k+1}, \dots, x_n)$ for a fixed linear function f of the variables $x_{n-k}, \dots, x_n$. On the facet F_{n-k}, we have assumed that $\dot{x}_{n-k} \geq 0$, and so

$$f(x_{n-k}, x_{n-k+1}, \dots, n_n) \geq 0 \text{ for } \begin{cases} x_{n-k} = b_{n-k,+}, \\ x_i \in [c_{i-}, c_{i+}) \quad \text{ for all } i = n-k+1, \dots, n. \end{cases} \tag{3.12}$$

Note that this statement does not use any of the facet-specific values of $x_1, \dots, x_{n-k-1}$, so (3.12) must also hold for all crossings of the slice boundary $x_{n-k} = b_{n-k,+}$ within the slices $x_i \in [c_{i,-}, c_{i,+})$, and this is where we have specified our infinite run will take place. Hence, this shows that any crossing of the slice boundary $x_{n-k} = b_{n-k,+}$ on the infinite run must be in the positive direction, and so cannot be reversed.

Now, all the assumptions we have made along the way have not reduced generality, in the sense that every other case has the same or analogous behaviour, which implies that *transitions in dimension $n-k$ do not appear in the infinite run.*

Conclusion: By the base case and inductive part, the infinite run through the automaton abstraction of the system cannot involve transitions in any dimension, which contradicts the existence of an infinite run. So the automaton abstraction of the n-dimensional system under the assumptions only has finite runs. □

Theorem 3.11. *Assume we have a system of form (3.9) with strictly negative diagonal entries, and an automaton abstraction created by the method of Maler and Batt [2008] satisfying Assumptions 3.6–3.9. Then the only location with no outgoing edges in the automaton abstraction corresponds to the box L containing the equilibrium point $\overline{x} = 0$.*

Proof. Consider a box $b = [b_{1,-}, b_{1,+}) \times \dots \times [b_{n,-}, b_{n,+})$ and assume it has no transitions out of it. Then all the existing transitions are inwards. It is not possible for whole box sides to have zero flow across them, due to not allowing splitting at $x_i = 0$ surfaces (by Assumption 3.6). Therefore the two opposite sides of a box in dimension i have opposite (inwards) flows, that is

$$\dot{x}_j|_{x_j = b_{j,-}} \geq 0, \tag{3.13}$$

$$\dot{x}_j|_{x_j = b_{j,+}} \leq 0. \tag{3.14}$$

Here, $\dot{x}_j|_{x_j=b_{j,-}}$ means "$\dot{x}_j$ evaluated at the point $x_j = b_{j,-}$". We will now prove by induction that $b_{j,-} < 0$ and $b_{j,+} > 0$ for all $j = 1, \ldots, n$.

Base case: In the n-th dimension, $\dot{x}_n = a_{n,n}x_n$ with $a_{n,n} < 0$. Equation (3.13) becomes $\dot{x}_n|_{x_n=b_{n,-}} = a_{n,n}b_{n,-} \geq 0$, which implies $b_{n,-} < 0$, as there are no splits at $x_n = 0$ by Assumption 3.6 and $a_{n,n} < 0$. Similarly (3.14) becomes $\dot{x}_n|_{x_n=b_{n,+}} = a_{n,n}b_{n,+} < 0$ implying $b_{n,+} > 0$.

Inductive case: Assume that $b_{j,-} < 0$ and $b_{j,+} > 0$ for all $j = n-k+1, \ldots, n$. Then let us consider the case when $j = n-k$, when

$$\dot{x}_{n-k} = a_{n-k,n-k}x_{n-k} + a_{n-k,n-k+1}x_{n-k+1} + \ldots + a_{n-k,n}x_n. \tag{3.15}$$

At $x_{n-k} = b_{n-k,-}$, with (3.13) this becomes

$$a_{n-k,n-k}b_{n-k,-} + a_{n-k,n-k+1}x_{n-k+1} + \ldots + a_{n-k,n}x_n \geq 0. \tag{3.16}$$

Now (3.16) must hold for all $x_j \in [b_{j,-}, b_{j,+})$ for every $j = n-k+1, \ldots, n$, and in particular it must hold for the worst case option, when every $a_{n-k,j}x_j \leq 0$ — as $b_{j,-} < 0$ and $b_{j,+} > 0$ this will hold for one of $x_j = b_{j,\pm}$, depending on the sign of $a_{n-k,j}$. So, in the worst case we need to satisfy

$$a_{n-k,n-k}b_{n-k,-} \geq 0, \tag{3.17}$$

which by the same argument as the base case implies that $b_{n-k,-} < 0$ (as $a_{n-k,n-k} < 0$ and there are no splits at $x_j = 0$ by Assumption 3.6). We can do a similar analysis on the equation for $\dot{x}_{n-k}|_{b_{n-k,+}}$ to prove that $b_{n-k,+} > 0$.

Conclusion: By base case and the inductive part, we know that the box b has $b_{j,-} < 0$ and $b_{j,+} > 0$ in every dimension j, so the box b contains the point $x = 0$, the equilibrium. Hence $0 \in b$, and so there is only one box with no outgoing edges and it is the box $b = L$. □

Theorem 3.12. *Assume we have a system of form (3.9) with strictly negative diagonal entries, and an automaton abstraction created by the method of Maler and Batt [2008] satisfying Assumptions 3.6–3.9. Then all trajectories of the TA get to the live box L in finite time, and so do all trajectories of the original system.*

Proof. Theorems 3.10 and 3.11 together imply that all runs of the TA lead to the box L containing the equilibrium point in a finite number of steps. Assumption 3.9 says that each box on this route is left within a finite time, and so the total time for any TA run to reach the box L is finite. By the over-approximation of number of trajectories, all trajectories of the original system reach the box L within finite time (and stay there as L is invariant). □

This proves that, for systems of the form of Section 3.4, a TA abstraction which satisfies the assumptions of Section 3.4.2 is bisimilar to the original system with respect to inevitability.

3.5 Dynamically-driven splitting method

In this section we define a method to split the state space of a continuous system such that the TA abstraction created from it satisfies the four assumptions for a subset of the upper-triangular linear systems. Together with the previous section this proves that we can automatically create a TA abstraction of such systems which proves inevitability. The method is shown to terminate for this particular class of systems, and the number of locations in the resulting abstraction is analysed.

3.5.1 The subclass of systems considered

In the previous section we saw that if a TA abstraction of a system of the form of (3.9) could satisfy four assumptions, then this TA abstraction would prove the inevitability property of getting close to an equilibrium. In this section we consider a subclass of (3.9), and propose a dynamically-driven splitting method for this class which will satisfy the four assumptions.

From now on we consider a subclass of upper-triangular linear systems of form (3.9), with negative real eigenvalues as before, with the extra condition that *in each row of the matrix, a maximum of one other non-zero entry is allowed.* These conditions mean that x_i's differential equation is in one of two forms, for each $i = 1, \ldots, n$, either

$$\dot{x}_i = a_{i,i}x_i \quad \text{for } a_{i,i} \text{ strictly negative, or} \tag{3.18}$$

$$\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j \quad \text{for } a_{i,i} \text{ strictly negative}, a_{i,j} \neq 0 \text{ and } i < j \leq n. \tag{3.19}$$

This class of systems does have restrictions, but allows various interesting possibilities. In particular, all 2-D systems with strictly negative real eigenvalues can be transformed by the Schur decomposition to an equivalent dynamical system of this form. The verification results for such 2-D systems (with state space limits and live box limits suitably transformed) will prove the desired properties about the original system. Higher-dimensional systems which can be of this form include systems modelling chains of behaviour, where each x_i's evolution only depends on itself and the element next in the chain. For example, simplified models of biological cascades can be expressed in this form [Heinrich et al., 2002]. There are also some piecewise-linear models of such cascades, where each element depends only on itself and the element before it: these can be modelled in the form (3.18) or (3.19) for each x_i and each region of dynamics. We will discuss the extension of the dynamically-driven splitting method to piecewise-linear systems of this form in Chapter 4.

If row i of the matrix has the form (3.18), this means that we have a constant value of $\dot{x}_i$ for any fixed x_i, no matter what the values of the other variables are. So the continuous flow across any facet in dimension i will always be in the same direction, that is Assumption 3.8 is always true on all box faces in dimension i. On the other hand, if row i has the second form (3.19), this means that if we try to satisfy Assumption 3.8 by separating $\dot{x}_i > 0$ from $\dot{x}_i < 0$ on a particular face $x_i = c$, we get a constant value of $x_j = -\frac{a_{i,i}c}{a_{i,j}}$ where the $\dot{x}_i = 0$ surface occurs through this face. If we choose to split at this point in the j-th dimension, we will create two new boxes where the $x_i = c$ facets have either $\dot{x}_i \geq 0$ or $\dot{x}_i \leq 0$, hence we can satisfy Assumption 3.8. This ease of selecting where to split is not available to us for general upper-triangular systems, and is what makes this special class better for automatic splitting.

Another property of systems of this class is that we can find a limit on the number of boxes with infinite time, and in fact we know exactly where these infinite-time boxes[3] can occur. This means that we can target those boxes with a specifically designed splitting method, in order to satisfy Assumption 3.9. We will look at these limits on infinite box time existence in Proposition 3.14 in Section 3.5.3.

3.5.2 The novel splitting method

The method we propose for splitting systems of the form described in Section 3.5.1 is described in Algorithm 3.1, and the idea of the method is demonstrated on a 2-D

[3]We will sometimes refer to 'boxes with infinite box time' as 'infinite-time boxes' for ease of writing.

Algorithm 3.1 Automatic splitting to prove inevitability

Input: Linear dynamical systems with each x_i's dynamics of the form of (3.18) or (3.19), with state space $S = [s_{1-}, s_{1+}) \times \ldots \times [s_{n-}, s_{n+})$, and live box $L = [l_{1-}, l_{1+}) \times \ldots \times [l_{n-}, l_{n+})$.

Output: A splitting of the system such that the TA abstraction proves inevitability of the live box.

```
1: for $i = 1, \ldots, n$ do                                              ▷ Step 1
2:     Initialise $C_i$ (the current splitting state in dimension $i$) to the unique subset of $x_i = l_{i-}, l_{i+}, s_{i-}, s_{i+}$.
3:     Initialise $N_i$ with the internal splits, $l_{i-}, l_{i+}$
4: end for
5: call FollowSplits                                                     ▷ Step 2
6: Calculate box times                                                   ▷ Step 3
7: $B \leftarrow$ list of boxes with infinite box time (except $L$)
8: while $B$ is non-empty do
9:     for $k = 1, \ldots, \text{length}(B)$ do
10:        $V \leftarrow$ get vertices of box $B(k)$
11:        $Z \leftarrow B(k)$
12:        call RemoveInfiniteTimes
13:    end for
14:    call FollowSplits
15:    $B \leftarrow$ new list of boxes with infinite box time (except $L$)
16: end while
```

Algorithm 3.2 `FollowSplits` sub-algorithm

Input: Current splitting state of the system C, with a finite list N of newly-made splits that need to be followed down the dimensions, and the dynamics of the system.

Output: A new splitting C and N with only one direction of flow across the faces of the boxes.

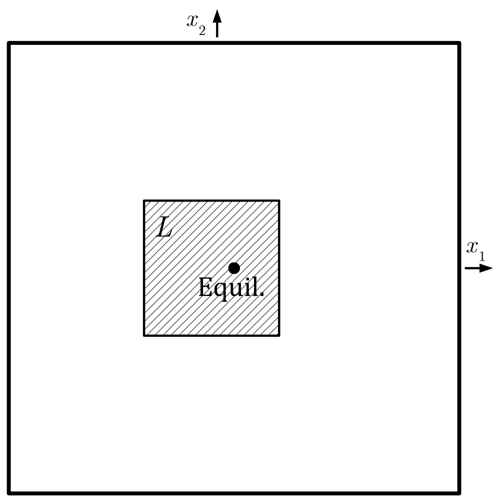
```
1: for $i = 1, \ldots, n - 1$ do
2:     for all $p \in N_i$ do
3:         $v \leftarrow$ find $\dot{x}_i$ velocity at vertices of the splitting surface $x_i = p$ in the region $S$
4:         if any$(v < 0)$ and any$(v > 0)$ then
5:             solve $\dot{x}_i = 0$ when $x_i = p$ giving $x_j = d$, for some $i < j \le n$.
6:             $C_j \leftarrow C_j \cup d$
7:             $N_j \leftarrow N_j \cup d$
8:         end if
9:         $N_i \leftarrow N_i \setminus \{p\}$
10:    end for
11: end for
12: $N_n = \emptyset$
```

Algorithm 3.3 `RemoveInfiniteTimes` sub-algorithm

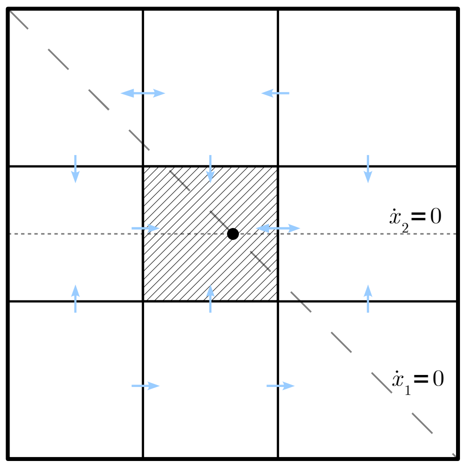
Input: Set of box vertices V, initial zero set Z, the dynamics $\dot{x} = Ax$, and the sets of current splitting state C and newly-made splits N.

Output: Extended splitting C and N if infinite time is removed.

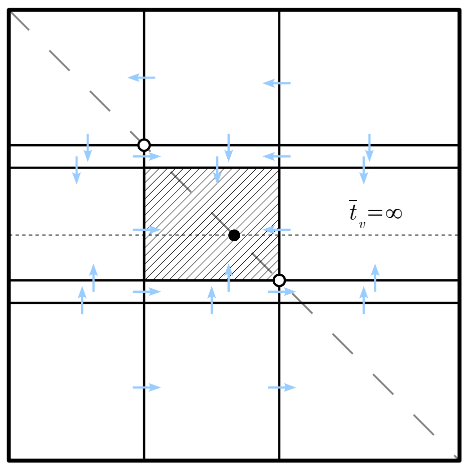
```
1:  for $i = n, n-1, \ldots, 1$ do
2:      if $\dot{x}_i$ has an off-diagonal entry then
3:          $j \leftarrow$ position of the off-diagonal entry in row $i$ of $A$
4:          $\begin{bmatrix} c_{i-} & c_{i+} \\ c_{j-} & c_{j+} \end{bmatrix} \leftarrow$ limits of the surface $\dot{x}_i = 0$ in the box
5:          if $c_{j-} > Z_{j+}$ then
6:              add split at $x_j = (c_{j-} + Z_{j+})/2$ (or nearby if this is $x_j = 0$)
7:              break loop
8:          else if $c_{j+} < Z_{j-}$ then
9:              add split at $x_j = (c_{j+} + Z_{j-})/2$ (or nearby if this is $x_j = 0$)
10:             break loop
11:         else if $c_{i-} > Z_{i+}$ then
12:             add split at $x_i = (c_{i-} + Z_{i+})/2$ (or nearby if this is $x_i = 0$)
13:             break loop
14:         else if $c_{i+} < Z_{i-}$ then
15:             add split at $x_i = (c_{i+} + Z_{i-})/2$ (or nearby if this is $x_i = 0$)
16:             break loop
17:         else
18:             $[Z_{i-}, Z_{i+}] \leftarrow [Z_{i-}, Z_{i+}] \cap [c_{i-}, c_{i+}]$
19:             $[Z_{j-}, Z_{j+}] \leftarrow [Z_{j-}, Z_{j+}] \cap [c_{j-}, c_{j+}]$
20:         end if
21:     else (row $i$ does not have an off-diagonal entry)
22:         if $Z_{i+} < 0$ then
23:             add split at $x_i = Z_{i+}/2$
24:             break loop
25:         else if $Z_{i-} > 0$ then
26:             add split at $x_i = Z_{i-}/2$
27:             break loop
28:         else
29:             $[Z_{i-}, Z_{i+}] \leftarrow \{0\}$ (point interval)
30:         end if
31:     end if
32: end for
```



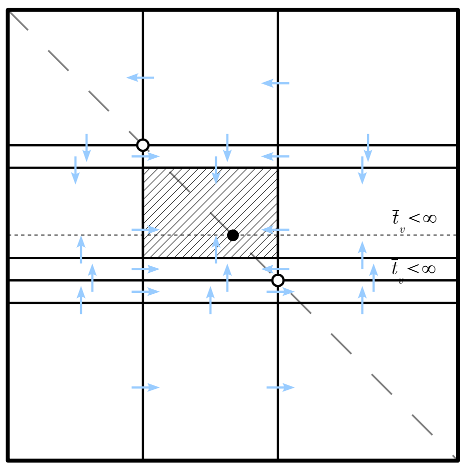
(1) Initial Setup: Given live box L containing the equilibrium point.



(2) After Step 1: Splits made based on the boundaries of the live box.



(3) After Step 2: Hollow dots indicate where $\dot{x}_1 = 0$ on x_1 constant lines. Splits made at these points.



(4) After Step 3: An extra split is added above the lower dot, removing infinite time in the box marked in (3).

Figure 3.6: Applying the splitting algorithm to the example $\dot{x}_1 = -x_1 - x_2$ and $\dot{x}_2 = -x_2$, with the given live box L defined slightly off centre around the equilibrium point $\overline{x} = 0$ (labelled by equil.). Arrows indicate the allowed directions of flow across box boundaries, and the shaded region indicates the live box L as it changes size.

example in Fig. 3.6. The idea is to firstly make an abstraction which satisfies Assumptions 3.6–3.8, and then remove the infinite time on boxes (to satisfy Assumption 3.9) whilst still preserving the first three assumptions. In order to do this there are three algorithms:

- Algorithm 3.1 drives the process, and consists of three steps. Step 1 (lines 1–4) initialises the splitting of the state space using the given state space and live box L, then step 2 (line 5) calls the algorithm `FollowSplits`, which ensures this initial splitting has only one-way crossings across box boundaries (satisfying Assumption 3.8). Lastly, step 3 (lines 6–16) iterates until all boxes have finite box times except the live box L, calling `FollowSplits` again to preserve Assumption 3.8.

- Algorithm 3.2 is called `FollowSplits`, and its purpose is to ensure there are no box facets which can have two-way flow across them. It works by finding the point on a splitting face $x_i = p$ (p constant) where the flow changes direction, adding a split at this point to separate the positive and negative flow on this facet. As mentioned at the end of Section 3.5.1, this splitting is possible because of the special nature of the class, so that every equation $\dot{x}_i = 0$ only depends on the value of x_i and possibly some x_j with $j > i$. So the additional split caused by each face $x_i = p$ is made at some point $x_j = d$, meaning that splits are always induced in the dimensions below the one we are currently considering.

- Algorithm 3.3 is called `RemoveInfiniteTimes`, and its purpose is to make splits within any box with infinite box time so that the resulting boxes in the new splitting have got finite box times. It does this by considering the limits of where each $n - 1$ dimensional surface $\dot{x}_i = 0$ exists in an infinite-time box, as we may be able to see that two surfaces $\dot{x}_i = 0$ and $\dot{x}_j = 0$ have separate limits of existence in the box for some dimension k. That is, they can never intersect in dimension k within the box, and so we can make a split in dimension k to separate them, and so ensure that one of the resulting boxes does not have $\dot{x}_i = 0$ passing through it, and the other box does not have $\dot{x}_j = 0$ passing through it.

To keep track of the splits that have been made in the system, we use the notation C_i for the set of split points in dimension i, including the endpoints $x_i = s_{i,-}$ and

$x_i = s_{i,+}$. The set N_i is the set of newly-made splits in C_i, which never includes the endpoints $s_{i,\pm}$. This set N_i feeds into `FollowSplits` to keep track of which splits could still have two-way flow, and `FollowSplits` processes these splits by dimension to remove two-way flow. We also use C to be the system's total current splitting state (the set of all C_i's), and we use N to be the total new splits in all dimensions. The letter B will be the list of boxes with infinite box time. We will also use V to be the set of vertices of a box, by which we mean the vectors describing the corners of the box, and we will use Z as a 'zero set', which will keep track of the parts of the box to have any $\dot{x}_i = 0$ surface passing through them — we will say Z_i^- be lower bound and Z_i^+ be upper bound in dimension i.

We will now show that this algorithm terminates and quantify the size of the resulting abstraction, then we will give an overview of the proof of why the abstraction satisfies Assumptions 3.6–3.9.

3.5.3 Termination and abstraction size

We split the proof of termination by each of the algorithms.

Termination and abstraction size for `FollowSplits`

Firstly we show that the `FollowSplits` sub-algorithm terminates, assuming it is passed finite sets of current C and newly-made N splits. `FollowSplits` consists of two for loops, the first clearly has a finite number of executions $(n - 1)$. The second iterates over a finite number of elements in the sub-list N_i, and each iteration removes an element from the current dimension list and possibly adds one to a lower dimension's list. Since this is done a finite number of times, each iteration of the inner loop is only done a finite number of times, and so `FollowSplits` terminates.

The number of splits that can be added by this is dependent on the size of the current and new splitting in each direction — let us say the size of the current splitting which is input to `FollowSplits` is $|C_i|$ for each dimension $i = 1 \dots, n$ and the size of the inputted newly-made splits is $|N_i|$ for $i = 1, \dots, n$. We will also denote the size of the revised current and newly-made splittings after `FollowSplits` as $|C_i'|$ and $|N_i'|$. The worst case of the size of splitting that `FollowSplits` creates occurs when each dimension causes a split in the dimension immediately after it, as the effect of these splits builds up, so the maximum number of splits after `FollowSplits` in each dimension $i = 1, \ldots, n$ is

$$|C_i'| = |C_i| + \sum_{j=1}^{i-1} |N_j|. \tag{3.20}$$

As we started with $|C_i|$ splits in each dimension i, we can see that potentially one new split can be made in dimension i for every inputted new split in the lower numbered dimensions.

Termination and abstraction size for `RemoveInfiniteTimes`

The sub-algorithm `RemoveInfiniteTimes` only consists of one for loop over a finite range and some other calculations, so each time it is called it will terminate. Also, it terminates by adding a maximum of one split in some dimension, so if C and N are the splits prior to calling `RemoveInfiniteTimes` then the maximum size of the splitting after calling this function is

$$\left.\begin{aligned} |C_i'| &= |C_i| + 1 \\ |N_i'| &= |N_i| + 1 \end{aligned}\right\} \text{ for some dimension } i. \tag{3.21}$$

Termination for the whole algorithm (Alg. 3.1)

We will now consider Algorithm 3.1 as a whole, which is where the most work is required. Clearly line 1 of Algorithm 3.1 performs a finite number of splits (a maximum of four in each dimension), so terminates. Then line 2 copies values internal to S from this set into another set (2 new splits per dimension). Then, line 5 simply calls FollowSplits on these initial splits, which terminates: the abstraction size after steps 1 and 2 of Algorithm 3.1 is

$$|C_i| = 2i + 2 \text{ splits in each of the dimensions } i = 1, \ldots, n. \tag{3.22}$$

For Step 3, lines 6–16, we must consider the while loop and the for loop inside it, which call the sub-algorithm `RemoveInfiniteTimes` for each infinite-time box. The for loop around the call to `RemoveInfiniteTimes` is over the number of boxes with infinite time, which will definitely be finite if there are a finite number of boxes in the abstraction. We have already shown that `RemoveInfiniteTimes` terminates, so to prove termination of the whole process it only remains to show that the while loop in Algorithm 3.1 is only looped a finite number of times.

We will prove in Proposition 3.17 that the while loop only needs to be looped a maximum of $n-1$ times to remove all boxes with infinite box time. However, to get to that proof we need some preliminary results, starting with quantifying the maximum number of infinite-time boxes. To show where the infinite-time boxes can occur we need the notion of the *offset* of a box, which will quantify how far away we are from the live box L in terms of the number of boxes in each dimension we must pass through to get to L. This notion will be essential to the proof of termination of the while loop.

Definition 3.13 (Offset of a box)**.** Let box b have limits $[b_{1-}, b_{1+}) \times \ldots \times [b_{n-}, b_{n+})$, and the live box L have limits $[l_{1-}, l_{1+}) \times \ldots \times [l_{n-}, l_{n+})$. In each dimension i, the *offset in dimension i* is the (unsigned) number of intervals we pass through to get from $[l_{i-}, l_{i+})$ to $[b_{i-}, b_{i+})$, including the live box interval in the count. We will use the functional form $\text{offset}(b, i)$ for the offset of box b in dimension i. ■

In other words, the offset quantifies how close we are to the central live box in any given dimension. For example, the live box L has $\text{offset}(L, i) = 0$ in every dimension i, and a box b with the limits $[l_{1+}, b_{1+}) \times [l_{2-}, l_{2+}) \times \ldots \times [l_{n-}, l_{n+})$ has $\text{offset}(b, 1) = 1$ and $\text{offset}(b, i) = 0$ for $i = 2, \ldots, n$.

We now use the concept of the offset of a box to show how far away boxes with infinite box time can be from the live box in each dimension.

Proposition 3.14. *Consider any dimension i_1 of the system and the chain of dimensions which take i_1 to a dimension with diagonal dynamics, that is the chain $i_1 \to i_2 \to \ldots \to i_k$ where $\dot{x}_{i_j} = a_{i_j,i_j} x_{i_j} + a_{i_j,i_{j+1}} x_{i_{j+1}}$ for each $j = 1, \ldots, k-1$, and $\dot{x}_{i_k} = a_{i_k,i_k} x_{i_k}$. Then the following hold:*

3.14.1 Let B be the set of boxes with infinite box time (excluding L), then any box $b \in B$ must satisfy

$$\text{offset}(b, i_j) \leq \text{offset}(b, i_{j+1}) + 1.$$

3.14.2 Any box $b \in B$ with infinite box time can be offset by at most $k-j$ boxes in the i_j-th dimension.

3.14.3 The maximum number of infinite-time boxes (excluding L) is

$$\left(\prod_{j=1,\ldots,n} 2j+1 \right) - 1.$$

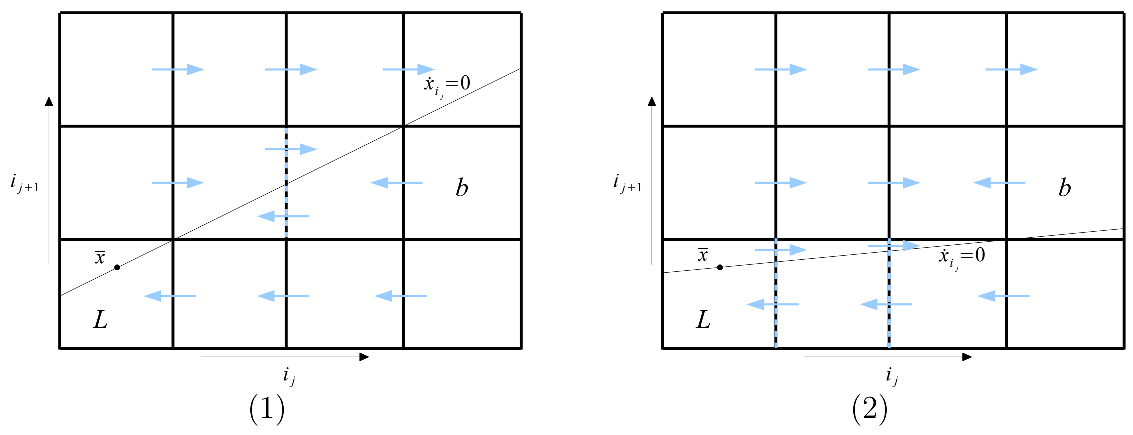


Figure 3.7: Possibilities for the projection of $\dot{x}_{i_j} = a_{i_j,i_j}x_{i_j} + a_{i_j,i_{j+1}}x_{i_{j+1}}$ to the x_{i_j}-$x_{i_{j+1}}$ plane, when offset$(b, i_j) = 3$ and offset$(b, i_{j+1}) = 1$. Diagram 1 shows the case if $\dot{x}_{i_j}$ enters the box b along the far facet $x_{i_{j+1}} = b_{i_{j+1},+}$ (actually only at the corner), which always causes two-way flow across one i_j-th dimensional facet, shown by the dotted line. Diagram 2 shows the option when we enter box b along the closer facet $x_{i_{j+1}} = b_{i_{j+1},-}$, causing $\dot{x}_{i_j} = 0$ to cross at least two facets on the way, giving two-way flow across these facets (shown by dotted lines).

Proof. We will prove these three parts in order.

3.14.1 Assume (for a contradiction) that we have a box b with infinite box time where offset$(b, i_j) >$ offset$(b, i_{j+1}) + 1$ for some j. Without loss of generality we assume that $b_{i_j-} > l_{i_j+}$ and $b_{i_{j+1}-} \geq l_{i_{j+1}+}$.[4] We consider the projection of this box to the x_{i_j}-$x_{i_{j+1}}$ plane, especially considering the dynamics of $\dot{x}_{i_j} = a_{i_j,i_j}x_{i_j} + a_{i_j,i_{j+1}}x_{i_{j+1}}$. The result for a special case is shown in Figure 3.7 to illustrate the idea. As the box b has infinite time, it must be possible for all the lines $\dot{x}_i = 0$ to pass through it. In particular the line $\dot{x}_{i_j} = 0$ must pass through it, with only two options for where it could enter from below because of the application of `FollowSplits`: (1) at the corner $x_{i_j} = b_{i_j-}$ and $x_{i_{j+1}} = b_{i_{j+1}+}$, or (2) anywhere along the facet for which $x_{i_{j+1}} = b_{i_{j+1}-}$, including at the corner $x_{i_j} = b_{i_j-}$ and $x_{i_{j+1}} = b_{i_{j+1}-}$. Either of these options force the line $\dot{x}_{i_j} = 0$ to have passed through the middle of some i_j-th dimension facet between the box L and b. This contradicts the assumption that step 2 has been completed. Hence offset$(b, i_j) \leq$ offset$(b, i_{j+1}) + 1$.

3.14.2 This is an inductive proof on the chain of dimensions. To start with, we consider the base case when we are at the end of the chain with diagonal dynamics $\dot{x}_{i_k} = a_{i_k,i_k}x_{i_k}$. The surface $\dot{x}_{i_k} = 0$ solves to give $x_{i_k} = 0$, which only occurs in the

[4]In this context, we are only assuming that $a_{i_j,i_{j+1}} > 0$, and we can just use the opposite box limits to make the analogous argument for $a_{i_j,i_{j+1}} < 0$.

slice of the space which contains the equilibrium point 0. Hence $\text{offset}(b, i_k) = 0$.

For the inductive part, assume the box b has an $\text{offset}(b, i_{j+1}) \leq k-(j+1) = k-j-1$ in dimension i_{j+1}. Then by Proposition 3.14.1, $\text{offset}(b, i_j) \leq \text{offset}(b, i_{j+1}) + 1 \leq k-j-1+1 = k-j$, proving that if Prop. 3.14.2 holds for dimension i_{j+1} then it holds for dimension i_j. By the base case and inductive proof we can see that Prop. 3.14.2 holds for all dimensions $i_1, i_2, \ldots, i_k$ in the chain.

3.14.3 To find the maximum number of boxes that can have infinite times, we must take the product of the allowed number of slices in each dimension (2 times the offset number plus one for the central slice), and subtract 1 to not count the live box L. In the worst case, the maximum number of infinite-time boxes is

$$\left(\prod_{j=1,\ldots,n} 2(n-j)+1\right) - 1.$$

This is the product of the first n odd numbers, subtracting 1 at the end, and reversing the order of the product gives the required result. □

Having now quantified the maximum number of infinite-time boxes, we will show that the while loop will terminate. Termination of the while loop only occurs if all the infinite-time boxes have been split in such a way that the (smaller) boxes which replace them have not got infinite box time. In order to remove the infinite time, we need to separate the positions where surfaces of zero derivatives can occur, so that one box does not have every surface of zero velocity passing through it. Let us define formally the notion of a separable and inseparable box to use in the next proposition.

Definition 3.15 (Separable and inseparable boxes)**.** Consider any box b which has infinite box time. If it is possible to make one split of a box b which makes the two sub-boxes have finite box times, we will call this box *separable.* If we cannot use only one split to produce finite box times on the resulting boxes, then we will call the box b *inseparable.* ■

The set of separable boxes are exactly those which cause a split to be made when the algorithm `RemoveInfiniteTimes` is used on each of them. Therefore, on the first run through the while loop, all initially separable boxes will be removed from the list B, shrinking the list for the next run of the loop. The idea of the proof that

Algorithm 3.1 removes all infinite times on boxes is to show that within a finite number of runs through the while loop every initially inseparable box has become separable. The method for proving this is to form a chain of neighbouring inseparable boxes with infinite time until eventually we get to an infinite-time box which is separable. Working backwards from the separable box we prove that splitting a separable box makes its neighbouring inseparable box become separable on the next run through the while loop. The chain of inseparable boxes is of maximum length $n - 1$ (proved in Prop. 3.17).

Working towards this proof we now prove a proposition in four parts, which develops the relationship of separable and inseparable boxes and the offset of a box.

Proposition 3.16.

3.16.1 If a box $b \neq L$ has infinite box time, then at least one dimension has a non-zero offset, and for any non-zero offset dimension i where $\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j$ and $\dot{x}_j = a_{j,j}x_j$, we have $\mathrm{offset}(b, i) = 1$.

3.16.2 All infinite-time boxes b with a non-zero offset in the i-th dimension where $\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j$ and $\dot{x}_j = a_{j,j}x_j$ are separable in dimension j.

3.16.3 Consider a box b with infinite time, and let i denote the highest-numbered dimension with non-zero offset. Let this dimension have dynamics $\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j$ for some j.[5] *Then, if b is inseparable there is another infinite-time box b' neighbouring it in dimension j, in the sense that* $\mathrm{offset}(b, j) = 0$ *and* $\mathrm{offset}(b', j) = 1$ *with all other offsets identical.*

Proof.

3.16.1 The only box where no dimensions are offset is the box L itself, which means that every box b which is not equal to L must have a non-zero offset in at least one dimension. By the proof of Prop. 3.14.2, we know that for all dimensions such that $\dot{x}_j = a_{j,j}x_j$, if b has infinite box time then $\mathrm{offset}(b, j) = 0$. If some

[5] *This is not a restriction: if a box has infinite time then all dimensions with diagonal dynamics, $\dot{x}_i = a_{i,i}x_i$, must have $x_i = 0$ as part of the box limits in this dimension, meaning the offset is zero. Hence, the contrapositive says that if the offset is non-zero then the dynamics must have a non-diagonal entry.*

dimension i is offset and has dynamics $\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j$ where j has the dynamics $\dot{x}_j = a_{j,j}x_j$, then Proposition 3.14.1 says that

$$\begin{aligned} \text{offset}(b,i) &\leq \text{offset}(b,j) + 1 \\ &\leq 0 \qquad\qquad\; +1 \\ &\leq 1 \end{aligned}$$

As we have assumed dimension i of box b has a non-zero offset, this also means offset$(b,i) \geq 1$, and combining these two equations gives us offset$(b,i) = 1$ for every offset dimension with dynamics of this form, proving the proposition.

3.16.2 By the proof of Prop. 3.16.1, we know that infinite-time boxes with $\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j$ and $\dot{x}_j = a_{j,j}x_j$ will have offset$(b,i) = 1$ and offset$(b,j) = 0$. Letting $b = [b_{1-}, b_{1+}) \times \ldots \times [b_{n-}, b_{n+})$, then the offset values imply that $0 \notin (b_{i-}, b_{i+})$ and $0 \in (b_{j-}, b_{j+})$. Solving $\dot{x}_i = 0$ for ranges x_i and x_j gives a solution box of $(x_i, x_j) \in [t_{i-}, t_{i+}] \times [t_{j-}, t_{j+}] \subseteq [b_{i-}, b_{i+}] \times [b_{j-}, b_{j+}]$. Notice that the pair $(x_i, x_j) = (0,0)$ is a solution of the equation $\dot{x}_i = 0$, and as the dynamics are linear there is a one-to-one mapping of the pairs (x_i, x_j) which satisfy the equation $\dot{x}_i = 0$. Hence there is no solution of this equation with $x_i \neq 0$ and $x_j = 0$, and so the box $[t_{i-}, t_{i+}] \times [t_{j-}, t_{j+}]$ cannot contain $x_j = 0$. However $\dot{x}_j = 0$ implies only that $x_j = 0$, and so the algorithm finds that the box b is separable by making a new split in dimension j (by either line 5 or 8 of Algorithm 3.3).

3.16.3 We will prove this part by construction of such a box b'. Let box b have limits $[b_{1-}, b_{1+}) \times \ldots \times [b_{n-}, b_{n+})$, and we know by Prop. 3.16.1 that offset$(b,i) = 1$ where i is the highest-numbered dimension with non-zero offset, so offset$(b,k) = 0$ for all $i < k \leq n$. If this box is inseparable then the limits created by each $\dot{x}_k = 0$ equation will have a non-zero region of intersection in every dimension. In particular, all $\dot{x}_k = 0$ lines which depend on x_j must have some region of intersection in the x_j variable space. As b is offset in dimension i but not dimension j, and $\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j$, we can see that $\dot{x}_i = 0$ only passes through the box b at a corner (see Figure 3.8) — let this corner have co-ordinates $(x_i, x_j) = (p_i, p_j) \in \{b_{i-}, b_{i+}\} \times \{b_{j-}, b_{j+}\}$. Then all equations $\dot{x}_k = 0$ which depend on the variable x_j must allow flow at this corner, that is $x_j = p_j$ must be

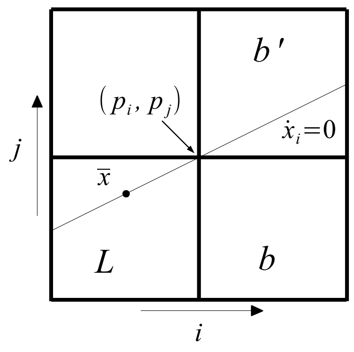


Figure 3.8: Assuming `FollowSplits` has been completed, the $n-1$ dimensional surface $\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j$ can only touch a corner of box b when $\text{offset}(b,i) = 1$ and $\text{offset}(b,j) = 0$.

contained in the region of solution of $\dot{x}_k = 0$ for such k. Note that this requires the equation $\dot{x}_j$ to have an off-diagonal entry, as if it did not then the only place $\dot{x}_j = 0$ would be satisfied would be at $x_j = 0$.

So, we define the new box b' as the neighbour of b obtained when we cross the boundary $x_j = p_j$. Since we have only changed the limits of x_j in this box, the positioning of surfaces $\dot{x}_k = 0$ only changes when a surface depends on the variable x_j. However, all surfaces dependent on x_j must have gone through the point $x_j = p_j$ in box b to make it inseparable, and so they also go through this point in the new box. As all of the zero equations which do not depend on x_j have not changed position relative to the new box b', they will pass through this new box in the same way as they passed through the old one b. Hence, all zero surfaces pass through the box b' and so *it has infinite box time*. Hence, we have constructed a box b' which neighbours b and has infinite box time.

□

We now bring these results together in a proposition which quantifies the maximum number of runs of the while loop of Algorithm 3.1, thus proving termination of this loop.

Proposition 3.17. *After $n-1$ runs through the while loop of Algorithm 3.1 there are no infinite-time boxes (except L) in the resulting abstraction. Hence the while loop terminates after $n-1$ iterations.*

Proof. (By induction on the number of runs through the while loop). The inductive hypothesis is that *the i-th run through the while loop separates (at least) all boxes where $n-i$ is the highest-numbered dimension with non-zero offset.*

Base case. By Proposition 3.16.2 all boxes with a maximum offset in dimension i where $\dot{x}_i = a_{i,i}x_i + a_{i,j}x_j$ and $x_j = a_{j,j}x_j$ are separable immediately. In particular, in the worst case all boxes b with maximum offset in dimension $n-1$ are separated (using a split in dimension n) on the first run of the while loop.

Induction. Assume that we have made $i-1$ runs through the while loop, and have separated (at least) all the boxes with maximum offset in dimensions $n-(i-1), \dots, n-1$. Then any remaining infinite-time boxes b have a maximum offset dimension of $n-i$. Now, Proposition 3.16.3 says "*if* b with maximum offset dimension i has infinite time and is inseparable *then* there is another box b' with higher offset dimension $j > i$ and infinite time" — taking the contrapositive of this we obtain "*if* every box b' with maximum offset direction greater than i does not have infinite time *then* any box b with maximum offset dimension i either does not have infinite time or is separable". Hence in our case when the maximum offset dimension is $n-i$, we use this contrapositive to say that as no infinite-time box b' with higher offset can occur, every box b with infinite time and maximum offset dimension $n-i$ must be separable. Hence, the i-th iteration of the while loop must separate all infinite-time boxes with maximum offset in dimension $n-i$, so that all boxes with maximum offset in dimension $n-i$ have finite box times after this step.

Putting together the base case and the inductive step, we can see that in the worst case there is one iteration of the while loop for every one of the $1, \dots, n-1$ dimensions of the system. Hence, we have proved that there are no boxes with infinite time (except L) after $n-1$ runs through the while loop. □

As the while loop of Algorithm 3.1 finishes when there are no longer any infinite-time boxes to consider, and we have now proved that after $n-1$ runs there are no longer any infinite-time boxes, we know that Algorithm 3.1 terminates.

Maximum size of the completed abstraction

To get a maximum number of splits made in each dimension by Algorithm 3.1, let us first consider the splits made by the while loop. Notice that the `FollowSplits` algorithm is called at the end of each iteration of the while loop, so to quantify the number of splits made by the while loop we should consider how many splits are made by `RemoveInfiniteTimes` and subsequently `FollowSplits` on each run through the loop. However, the extra splits made by `FollowSplits` do not increase the number of infinite-time boxes with any particular maximum-numbered dimension with non-zero offset, as Proposition 3.14 limits the number of boxes that can have this maximum-numbered offset dimension however the splits have been made. Hence, we can get an upper bound for the number of splits made by assuming that all the new splits by `RemoveInfiniteTimes` are made at once, and then `FollowSplits` is applied.

Each infinite-time box induces a maximum of one split to be made in the system, by a call to `RemoveInfiniteTimes` (see Eq. (3.21)). In the worst case for the `FollowSplits` algorithm, these splits are all made in the first dimension, which causes `FollowSplits` to track them down the dimensions, making a new split for each infinite box in every dimension. Now, we have quantified the number of infinite-time boxes (excluding L) in Proposition 3.14.3, and so the number of new splits added to the system during calls to `RemoveInfiniteTimes` is (in the worst case)

$$\left(\prod_{j=1,\dots,n} 2j+1\right) - 1 \text{ splits in dimension 1.} \tag{3.23}$$

The size of the splitting before step 3 of Alg. 3.1 started is given by Equation (3.22), and so the size of the splitting after we have considered step 1, step 2, and `RemoveInfiniteTimes` is given by

$$|C_1| = 4 + |N_1| \qquad |N_1| = \left(\prod_{j=1,\dots,n} 2j+1\right) - 1, \quad \text{in dimension 1,} \tag{3.24}$$

$$|C_i| = 2i+2, \qquad |N_i| = 0, \qquad \text{in other dimensions } i = 2, \dots, n. \tag{3.25}$$

Using (3.20), which quantifies the number of splits `FollowSplits` adds, we get a worst case total number of splits of

$$|C_i| = 2i + 1 + \left(\prod_{j=1,\dots,n} 2j+1\right) \text{ in each dimension } i,$$

which gives a worst case total number of slices in each dimension of

$$2i + \left(\prod_{j=1,\dots,n} 2j+1\right) \text{ in each dimension } i,$$

and multiplying these together for all dimensions gives us the maximum number of boxes in the TA abstraction:

$$\prod_{i=1,\dots,n} \left(2i + \left(\prod_{j=1,\dots,n} 2j+1\right)\right). \tag{3.26}$$

The maximum number of boxes is exponentially increasing with the number of dimensions in the system, so the method could be difficult to implement for higher-dimensional dynamical systems. However, at least we can guarantee that this finite limit on the size of the TA abstraction exists.

3.5.4 Satisfying the assumptions

We will now prove that Algorithm 3.1 creates an abstraction which satisfies Assumptions 3.6–3.9, that is:

1. There is only one equilibrium, in one box of the splitting, and it is not on the boundary of this box.
2. The automaton abstraction of the system has a finite number of discrete states.
3. The flow across each box facet is in only one direction.
4. Every box (except L) has finite time.

Proposition 3.18. *The TA abstraction of continuous systems of the special form in Equations (3.18) and (3.19) satisfies Assumption 3.6. That is, there is only one equilibrium of the system; it is inside but not on the boundary of the live box L.*

Proof. Step 1 of Alg. 3.1 creates one box containing the equilibrium $\overline{x} = 0$, under the original specification that the initial live box should include $\overline{x}$ (not on the boundary). Step 2 (the `FollowSplits` sub-algorithm) can change the size of the box containing the equilibrium, but cannot add splits exactly at $x_i = 0$ for any i (because of linearity of dynamics), so the box containing $\overline{x}$ does not have it on the boundary. Step 3 similarly can make splits which affect the box L, but these splits are chosen not to be at the equilibrium, and then when this new splitting is passed to `FollowSplits` we again cannot induce splits at the equilibrium. Hence the equilibrium is never on a boundary of a box, and so must be inside one box only. $\square$

Proposition 3.19. *The TA abstraction of continuous systems of the special form in Equations (3.18) and (3.19) satisfies Assumption 3.7. That is, the automaton abstraction of the system has a finite number of discrete states.*

Proof. The proof follows from the quantification of the number of boxes in Equation (3.26), which is a finite number when n is finite. □

Proposition 3.20. *The TA abstraction of continuous systems of the special form in Equations (3.18) and (3.19) satisfies Assumption 3.8. That is, the flow across each box facet is only in one direction.*

Proof. The `FollowSplits` sub-algorithm (Alg. 3.2) is designed to enforce this property. We will prove it by induction, with the inductive hypothesis "after iteration $i = k$ of the outer for loop in `FollowSplits`, flow only goes in one direction across every box facet defined by a constant value of x_j for all $1 \leq j \leq k$".

Base case. We want to show that the first iteration of the outer for loop separates the positive and negative flow across box faces defined by the equation $x_1 = c$ for some constant c. We need to consider the two possible types of dynamics separately.

1. If the dynamics of x_1 are of the diagonal form $\dot{x}_1 = a_{1,1}x_1$, then considering any dimension 1 facet defined by $x_1 = c$, say, we have $\dot{x}_1 = a_{1,1}c$ on the facet, which is constant. The `FollowSplits` algorithm does not add any splits for this box face, but the flow is already one-way across each facet, so no new splits are required.

2. If the dynamics of x_1 are of the non-diagonal form $\dot{x}_1 = a_{1,1}x_1 + a_{1,j}x_j$, then, on a facet $x_1 = c$, `FollowSplits` adds a split at the point c_j which solves $a_{1,1}c + a_{1,j}c_j = 0$. Without loss of generality, let us consider $a_{1,j} > 0$. Considering the new box facet where $x_j < c_j$ on $x_1 = c$ we find that $\dot{x}_1 = a_{1,1}c + a_{1,j}x_j < a_{1,1}c + a_{1,j}c_j = 0$, that is, $\dot{x}_1 < 0$. Similarly, the new box facet where $x_j \geq c_j$ on $x_1 = c$ always gives us $\dot{x}_1 \geq 0$. Hence, the new box facets in dimension 1 all have one-way flow across them each box face after the first iteration of the i loop of `FollowSplits`. The case for $a_{1,j} < 0$ is analogous.

Induction. Assume that we have done the first $k - 1$ iterations of the outer for loop and that flow only goes in one direction across every box face in the first $k - 1$

dimensions. Then we need to show that flow only goes in one direction across the k-th dimension and also that it does not interfere with the one-way flow of any of the previous $k-1$ dimensions. Showing that flow only goes in one direction in dimension k after `FollowSplits` is analogous to the base case for dimension 1, so we will just prove that after this is completed none of the previous dimensions' faces have two-directional flow.

When the outer for loop of `FollowSplits` is run for dimension k, new splits can only be induced in dimensions k_1 such that $k_1 > k$. Therefore `FollowSplits` does not introduce new splits in any of the previous dimensions j where $j \leq k$. Let us consider what happens to a box facet $x_j = c_j$ in a dimension $j \leq k$ when a split is made at $x_{k_1} = c_{k_1}$ in a dimension $k_1 > k$. Before the split, we know that either $\dot{x}_j \leq 0$ or $\dot{x}_j \geq 0$ on this box facet — let us assume $\dot{x}_j \geq 0$ without loss of generality. Let us also assume the limits of the box facet are given by $[b_{k_1,-}, b_{k_1,+})$ and that $b_{k_1,-} < c_{k_1} < b_{k_1,+}$ so that this facet is actually getting split into two parts. There are two cases depending if the dynamics of dimension j depend on the value of the dimension in which the splitting is made (dimension k_1).

Case 1. The dynamics of dimension j depend directly on dimension k_1, that is, $\dot{x}_j = a_{j,j}x_j + a_{j,k_1}x_{k_1}$. We know that $\dot{x}_j \geq 0$ for $x_j = c_j$ and all $x_{k_1} \in [b_{k_1,-}, b_{k_1,+})$, and so this will also hold when a smaller interval of x_{k_1} is used. Hence, the flow across each of the smaller facets is still one-way.

Case 2. The dynamics of dimension j do not depend directly on the dimension k_1 where the splitting is being made. So the value for $\dot{x}_j$ is calculated on the same range as before, and so still has one-way flow.

Hence, the flow across boundaries in all dimensions $j \leq k$ is still one-way across the new box facets when a box is split in dimension $k_1 > k$.

Conclusion. Putting together the base case with the inductive hypothesis, we conclude that after `FollowSplits` is completed the flow across each box facet is one-way. We now note that `FollowSplits` is the last function to be called which affects the splits made, and so the effect of one-way flow will persist at the end of the splitting algorithm. $\square$

Proposition 3.21. *The TA abstraction of continuous systems of the special form in equations (3.18) and (3.19) satisfies Assumption 3.9. That is, every box (except L) has a finite box time.*

Proof. We have showed that Algorithm 3.1 terminates, and in the process showed its while loop terminates. This while loop only stops when no boxes with infinite time remain, which means that every box has finite time. Hence, Assumption 3.9 is satisfied. □

3.5.5 The main result: inevitability by abstraction

We have now proved that Assumptions 3.6–3.9 of Section 3.4.2 are satisfied, and since we are considering a sub-class of that considered in Section 3.4, we can immediately use any results derived for the broader class. In particular, we can use the theorems of Section 3.4.3 which we have proved for the broader class of systems to write the result of this chapter as the following:

> If a TA is created by the method of Maler and Batt [2008] using the splitting of Alg. 3.1, then the proof of inevitability of L on the TA abstraction will prove the inevitability of L for the original system.

Let us state and prove this formally. The proof follows from the method of Algorithm 3.1 with Assumptions 3.6–3.9 of Section 3.4.2 and Theorems 3.10–3.12 of Section 3.4.3.

Corollary 3.22. *Given a continuous dynamical system with each x_i's dynamics of the form (3.18) or (3.19), and the TA abstraction created by the Algorithms 3.1–3.3, all possible trajectories of the TA will reach box L containing $\overline{x} = 0$, and the continuous system is inevitable with respect to region L.* ■

Proof. By Propositions 3.18–3.21 the TA abstraction satisfies Assumptions 3.6–3.9. Then Theorem 3.12 says that all TA trajectories reach L within finite time, and also that the live box is inevitable. □

3.6 Implementation details

We have made an implementation in MATLAB of our method for making TA abstractions of the special class of systems described in Section 3.5. We call the implementation the `ProveByTA` toolbox. This toolbox is available online at

`http://staff.cs.manchester.ac.uk/~navarroe/research/dyverse/liveness`

An overview of the flow of the method is given in Figure 3.9. The main part of the toolbox is the implementation of Algorithm 3.1, so we will firstly give a few details around how this has been implemented.

3.6.1 Implementation of the splitting algorithm

MATLAB is very suitable for implementing Algorithm 3.1, as there are many mathematical aspects to this algorithm which MATLAB can handle easily. One aspect which MATLAB handles natively is the use of matrices — as we are only considering linear systems, all the dynamics are represented in terms of matrices, so this is a very useful feature. The programming language of MATLAB is also very intuitive for the mathematician.

Probably the most difficult part of the implementation of Algorithm 3.1 is keeping track of the splitting state of the system, as there can be different numbers of splits in each dimension and so we cannot use a standard matrix to represent it, as we would like to do in MATLAB. Instead we use *cell arrays*, which are effectively matrices of a certain size with all the entries pointers to other data elsewhere. To store the splitting state, we have a cell array of size $1 \times n$, with each entry a vector of any length we wish listing the points in that dimension where a split has been made.

We also use cell arrays to store the velocity at each corner point, used in the sub-algorithm `FollowSplits`. The interesting point here is that we use n-dimensional cell arrays, rather than the standard two-dimensional ones. This is an impressive feature of MATLAB, helpful for when we are considering dynamical systems of higher dimension than 2.

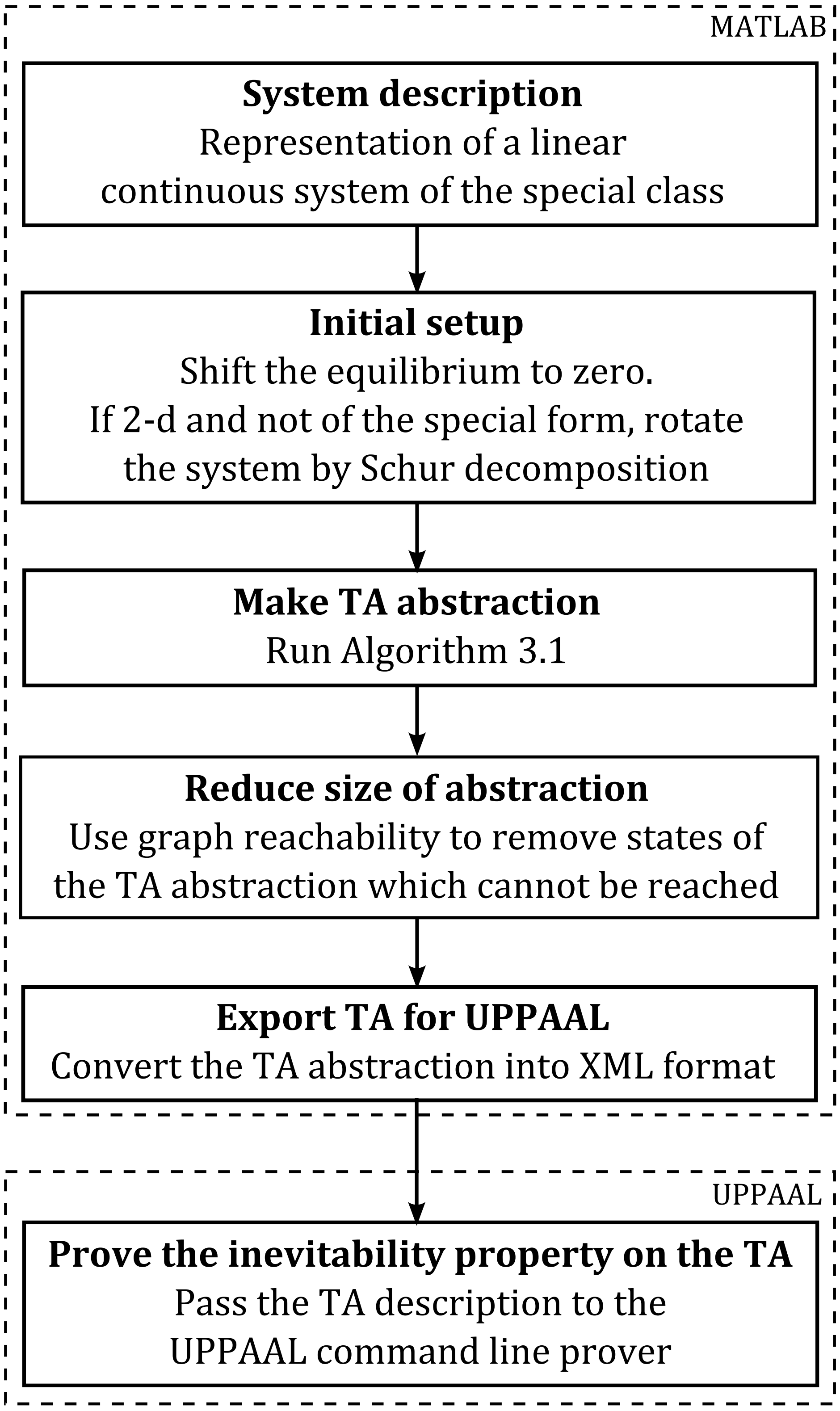

MATLAB
System description
Representation of a linear continuous system of the special class
Initial setup
Shift the equilibrium to zero.
If 2-d and not of the special form, rotate the system by Schur decomposition
Make TA abstraction
Run Algorithm 3.1
Reduce size of abstraction
Use graph reachability to remove states of the TA abstraction which cannot be reached
Export TA for UPPAAL
Convert the TA abstraction into XML format
UPPAAL
Prove the inevitability property on the TA
Pass the TA description to the UPPAAL command line prover

Figure 3.9: An overview of the ProveByTA toolbox.

3.6.2 Extra parts in the implementation

There are a few extra features which we have implemented to make this method usable as widely as possible. The first is the initial setup of the dynamical system, where the equilibrium gets shifted to zero if it is not already at zero. This means that any dynamical system of the form $\dot{x} = Ax + a$ can be considered, as long as A is of the special form of Section 3.5.1. The second part of this setup is for 2-D matrices, and was also discussed in Section 3.5.1. This part uses the fact that for any 2-D matrix with real negative eigenvalues we can use the Schur decomposition to make it an upper-triangular matrix, still with real strictly negative eigenvalues. This means that we can take as input any 2-D dynamical system of the form $\dot{x} = Ax + a$, provided the matrix A has real strictly negative eigenvalues.

When we use the Schur decomposition, we are effectively rotating the axes of the space of the system, and so we get a parallelogram-shaped state space for the upper-triangular system. To ensure we are considering all of the trajectories that could exist in the original system, we need to over-approximate this parallelogram-shaped state space by a rectangular one. In the implementation, this Schur decomposition and over-approximation of the resulting state space are triggered by an input flag being set to true.

We have also used a graph reachability algorithm on the abstraction to remove the states in the TA which are unreachable from the initial set in the discrete abstraction. This is so that we do not further process unreachable states, and also to decrease the size of the abstraction. The number of unreachable states in systems of the special form of Section 3.5.1 is typically large, because the dynamics of any dimension is either only dependant on itself or it is highly correlated with only one other dimension. In some cases we have seen roughly half of the states being removed by this reachability.

The final stage implemented in MATLAB is the conversion of the TA to a format suitable for UPPAAL [Behrmann et al., 2004] to take as input. This is important as it shows that the resulting TA does actually satisfy the inevitability property that we have claimed that it will. UPPAAL has a command-line prover which takes input files in XML format, and with a bit of persuasion we can get MATLAB to export in XML format too. This process involves firstly turning our splitting representation of the TA into a document object model (DOM), which is a language-independent way of representing data in a tree format. Once we have built this document object model, we can use a MATLAB function (`xmlwrite`) which will write it to an XML file in the format suitable for UPPAAL to use.

Once we have the XML file representing the structure of the TA abstraction of the continuous system, we call the UPPAAL stand-alone prover to prove that the inevitability property $(x(0) \in \mathit{Init}) \Rightarrow \Diamond(x(t) \in L)$ is satisfied, and hence that the original system satisfies the inevitability property of "all trajectories eventually get close to the equilibrium $\overline{x}$".

3.7 Example to demonstrate the implementation

We will now look at an example of a 2-D linear system to demonstrate the method as it is run by the MATLAB implementation. The system we consider is

$$\dot{x} = Ax = \begin{bmatrix} -1 & -1 \\ 0 & -1 \end{bmatrix} x, \tag{3.27}$$

where the state space is $S = [-5, 5) \times [-5, 5)$, the initial region is defined as $\mathit{Init} = [-5, -4) \times [-5, 5)$, and the initial live box is $L = [-2, 1) \times [-1.5, 1.5)$. This is precisely the example considered in Figure 3.6, when the splitting method was explained.

When we run the `proveByTA` toolbox on this example, it firstly gets checked to make sure it is of the required form, as otherwise the algorithm might never terminate. Next the initial splitting is made as in Step 1 of Alg. 3.1 (lines 1–4), which for this example gives the splitting after Step 1 to be

$$C_1 = \{-5, -2, 1, 5\}, \qquad N_1 = \{-2, 1\},$$
$$C_2 = \{-5, -1.5, 1.5, 5\}, \qquad N_2 = \{-1.5, 1.5\}.$$

Then, `FollowSplits` is called to give only one-way flow across boundaries, and so after this we have a splitting state of

$$C_1 = \{-5, -2, 1, 5\}, \qquad N_1 = \emptyset,$$
$$C_2 = \{-5, -1.5, -1, 1.5, 2, 5\}, \qquad N_2 = \emptyset.$$

The `proveByTA` toolbox then performs a state reachability to remove boxes unreachable from the initial set, so that we only consider the reachable boxes in the following steps. This removes 3 states of the timed-automaton in the example.

We then move on to Step 3 of Alg. 3.1, and find each of the infinite-time boxes in the current splitting — there is only one, which has the limits $[1,5) \times [-1,1.5)$. For this box, we initialise Z to be the closure of the whole box $Z = [1,5] \times [-1,1.5]$, and we call the `RemoveInfiniteTimes` sub-algorithm (Alg. 3.3). The first run of the for loop in Alg. 3.3 is for $i = 2$, which has diagonal dynamics and so the second part of the if statement is used (lines 21–29). As the initial limits of Z are set to the whole box, we have $Z_{2-} = -1$ and $Z_{2+} = 1.5$, and so the if and elseif statements of lines 22 and 25 are both false. Hence, we go to line 28 and set $Z_{2-} = Z_{2+} = 0$, giving $Z = [1,5] \times [0,0]$. The for loop is then iterated with $i = 1$, which has an off-diagonal entry in the dynamics, so the if statement of line 2 is true. Then we get that $j = 2$, and find the limits of the surface $\dot{x}_1 = 0$ in the box to be $[c_{j-}, c_{j+}] = [-1,-1]$ in dimension j, which means we take the elseif statement on line 8 and add a split at $x_2 = (-1+0)/2 = -1/2$. We then return to Alg. 3.1 having removed the only infinite-time box. `FollowSplits` is then called at line 12 of Alg. 3.1, with only one new split to follow, as $N_1 = \emptyset$ and $N_2 = \{-1/2\}$. As the only new split is in dimension 2 already, `FollowSplits` simply sets $N_2 = \emptyset$ and returns.

The final splitting of the system is given by

$$\begin{aligned} C_1 &= \{-5,-2,1,5\}, \\ C_2 &= \{-5,-1.5,-1,-0.5,1.5,2,5\}. \end{aligned}$$

The directions of flow between boxes created by this splitting is then calculated, and discrete reachability is performed once again to remove any unreachable states added in by the livebox splitting. In our example, no further states are removed. The maximum time that can be spent in a box is calculated, and everything is given to a method called `createXMLFile` in the `proveByTA` toolbox which rewrites the information into an XML file for input to UPPAAL.

Finally UPPAAL's stand-alone prover `verifyta` is called on the XML file of the abstraction, and the result is returned to MATLAB, which tells us `Liveness property proved!`. Figure 3.10 shows the TA abstraction for the example, when we view it in UPPAAL's graphical editor.

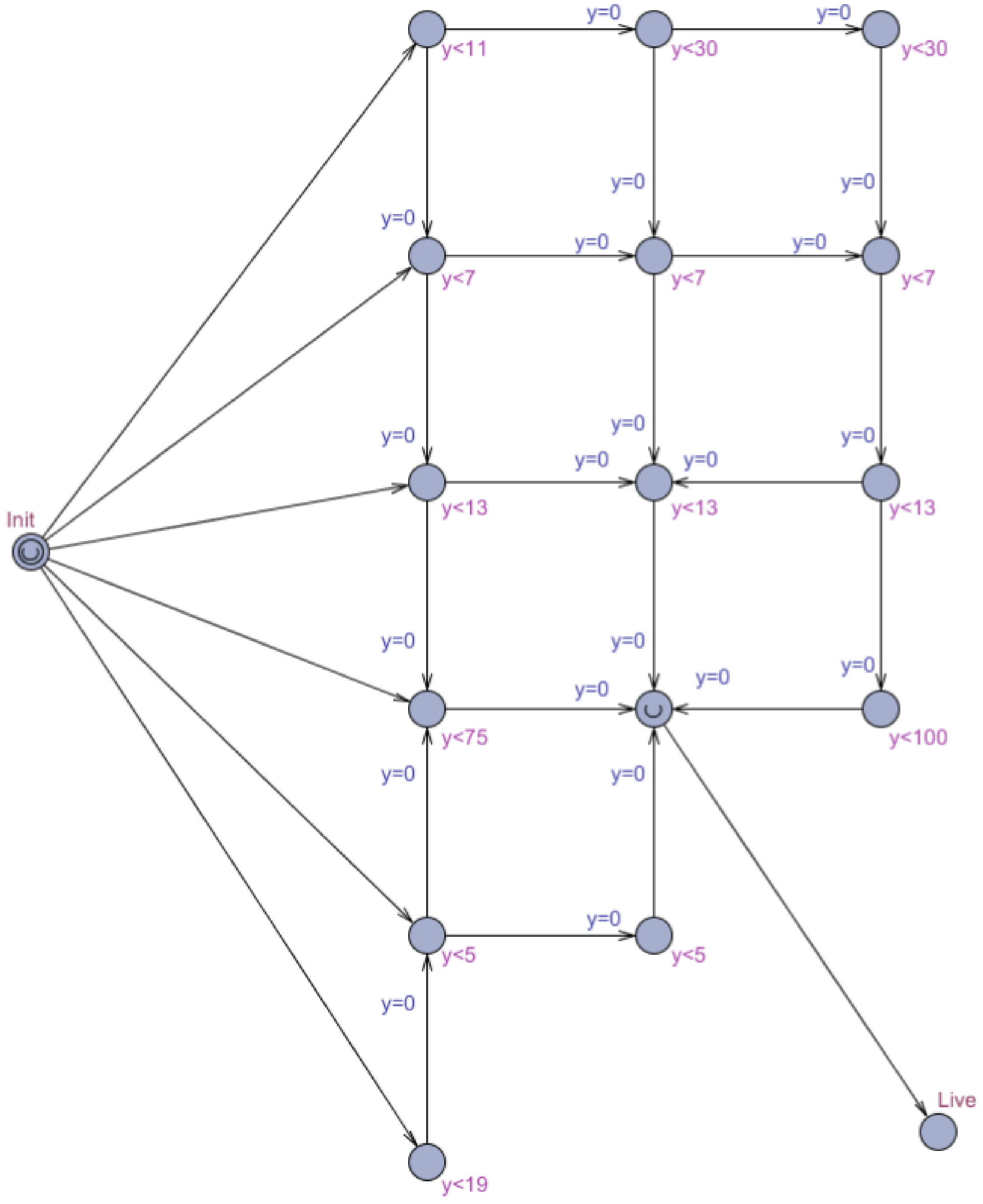


Figure 3.10: The timed-automaton abstraction of the 2-D example system of (3.27), viewed in UPPAAL's graphical editor. We just use the box clock in this model, and it is labelled by y. The "$y = 0$" statements are resets of the box clock on a transition, and the statements of the form "$y < 11$" are clock invariants on each state of the automaton. States containing a '$\cup$' are *urgent* states, where no time can pass.

3.8 Conclusions and future work

We have defined an algorithm for a class of linear systems which creates a splitting of the state space. When using this splitting to create a timed-automaton by the method of Maler and Batt [2008], we have shown that certain properties are true of the timed-automaton. Together these properties mean that the timed-automaton will prove inevitability of the original system reaching a set L around the equilibrium point $\overline{x}$. We have described the implementation of this algorithm, which automatically makes TA abstractions of systems with dynamics of the special class and then passes them to UPPAAL to prove their inevitability. The implementation also includes some extra features which can help the end-user, and also provide the capability to support larger systems.

Some future work on this method is to extend it for use in more general continuous systems. There are various problems to be overcome with such systems, one of which will be the termination of the splitting method, as the current method only terminates because of the special dynamics involved. Hence, part of the future work is to revise the splitting method to be more useful for more general linear systems and also nonlinear systems. In Chapter 4 we will show how it can be extended to a class of piecewise-continuous systems.

Chapter 4

Proving liveness in piecewise-linear dynamical systems

4.1 Introduction

In this chapter we will consider how to make timed-automaton (TA) abstractions of piecewise-linear (PWL) systems, by extending the method for continuous systems proposed in Chapter 3. We will consider the class of piecewise-linear systems extended in a sensible way from the special class of continuous systems that we studied in Chapter 3. We discuss the issues that occur on the boundaries between the regions with different dynamics, and show we can improve the way these issues affect the splitting by considering the analysis of sliding modes on these boundaries.

We will show that for the low-dimensional cases of these piecewise-linear systems the TA abstraction proves inevitability, and for higher-dimensional systems we will identify the problems that prevent the TA abstraction from proving inevitability. The work we present here is the first approach at proving such a TA abstraction can be used for proving inevitability of piecewise-linear systems, and as such it highlights a lot of problems which we might expect to occur in more general cases. We also discuss ways in which these problems could be approached in any future work in this area.

In Section 4.2 we introduce the class of piecewise-linear systems considered in this chapter, and give some background information about such systems. Then, Section 4.3 considers how to make a TA abstraction of a piecewise-linear system by extending the method for continuous systems given in Chapter 3. A new splitting method is then proposed in Section 4.4, and the size of the splitting is quantified in this section if we have a general result. In Section 4.5 we prove that the proposed splitting method creates a TA abstraction which proves inevitability for all one-dimensional (1-D) and two-dimensional (2-D) systems of the special piecewise-linear form. We then give some details of the implementation of the splitting method in Section 4.6, and show how a 2-D example system is proved by this implementation in Section 4.7. Lastly, in Section 4.8, we consider the problems and potential solutions for 3-D and higher-dimensional systems.

4.2 The type of systems considered

We consider the piecewise-linear extension of the special class of systems considered in Chapter 3. By piecewise-linear we mean that there is an arbitrary number m of different subsystems with dynamics of the special form, each with non-overlapping state spaces. Recall that the special form is $\dot{x} = Ax$, with $x \in \mathbb{R}^n$, and A an $n \times n$ upper-triangular matrix. The matrix A has the diagonal entries $a_{i,i} < 0$ for all $i = 1, \ldots, n$, with each row having one of the following forms:

$$\begin{aligned} \dot{x}_i = &\quad a_{i,i}x_i && \text{for } a_{i,i} \text{ negative, or} \\ \dot{x}_i = &\quad a_{i,i}x_i + a_{i,j}x_j && \text{for } a_{i,i} \text{ negative}, a_{i,j} \neq 0 \text{ and } i < j \leq n. \end{aligned} \tag{4.1}$$

We take m of these systems, each with dynamics $\dot{x} = A_k x$, $k = 1, \ldots, m$, where every A_k is of the special form of (4.1). The state space of each set of dynamics is given by a hyper-rectangle (box) $S_k \subset \mathbb{R}^n$ defined as

$$S_k = [s^k_{1-}, s^k_{1+}) \times \ldots \times [s^k_{i-}, s^k_{i+}) \times \ldots \times [s^k_{n-}, s^k_{n+}), \text{ for each } k = 1, \ldots, m. \tag{4.2}$$

We assume the S_k form a partition of a larger hyper-rectangle $S = [s_{1-}, s_{1+}) \times \ldots \times [s_{i-}, s_{i+}) \times \ldots \times [s_{n-}, s_{n+})$, by which we mean

$$\bigcup_{k=1,\ldots,m} S_k = S, \quad \text{and}$$

$$S_{k_1} \cap S_{k_2} = \emptyset \text{ for every } k_1, k_2 \in \{1, \ldots, m\} \text{ with } k_1 \neq k_2.$$

We will think of the box S as the state space of the whole piecewise-linear system, in the same way as S was the state space of the continuous system in Chapter 3. Note that the A_k matrices do not all have to be different: if two matrices are the same, $A_{k_1} = A_{k_2}$, then we effectively join their two state space boxes (S_{k_1} and S_{k_2}) together to make a potentially more complex shaped region of the state space with this set of dynamics. That is, the whole region $S_{k_1} \cup S_{k_2}$ has the dynamics $\dot{x} = A_{k_1}x$.

We will consider the inevitability property on this piecewise-linear system to be $(x(0) \in \mathit{Init}) \Rightarrow \Diamond(x(t) \in L)$, the same as the continuous case, where *Init* is an initial set of states and L is the live set which we wish to reach. We will assume that this live set is a box around the point $\overline{x} = 0$, which is the equilibrium of all the subsystems.

As with all dynamical systems, we can consider stability of piecewise-linear systems, and in Section 4.2.1 we will state a result which says that the class of upper-triangular piecewise-linear systems is stable and attractive, showing that the above inevitability property will be true in all piecewise-linear systems with subsystems of the form of (4.1). As well as stability, it can be useful to consider what happens on the boundaries between subsystems, and so in Section 4.2.2 we will discuss the method of sliding modes for defining dynamics on a boundary in piecewise-linear systems. We will use sliding modes in the TA abstraction method for piecewise-linear systems in Section 4.3.2.

4.2.1 Stability-related definitions

This special class of piecewise-linear systems can be proven to be stable and attractive (in the sense of Lyapunov). We will not make such a proof, but will take a proposition from Liberzon [2003] which states a stability result on a wider class of systems. We firstly define the concept of a stable matrix.

Definition 4.1 (Stable matrix [Liberzon, 2003])**.** A matrix A is said to be stable if all its eigenvalues have negative real part. ■

The following proposition says that all switched linear systems with subsystem dynamics described by such stable matrices are globally exponentially stable.

Proposition 4.2 ([Liberzon, 2003, pg. 35])**.** *If $\{A_q : q \in Q\}$ is a compact set of upper-triangular stable matrices, then the switched linear system $\dot{x} = A_q x$ is globally exponentially stable.* ■

Global exponential stability says that the trajectories are stable and attractive with exponential rate of attraction, which is a very strong stability result. We should note that we are actually only considering the stability of a smaller class of systems, with the restrictions:

1. Our set $\{A_k : k = 1, \dots, m\}$ is not only compact[1], but also finite;
2. The eigenvalues of our matrices are only real, not complex;
3. The switching between subsystems that we consider is not as general as 'switched systems', where switching can be made at a position in the space depending on parameters or inputs, and so potentially anywhere in the space. We only consider systems where the subsystems have non-intersecting state spaces and so switching between subsystems is made at the fixed areas in the space which define the boundaries between the subsystems' state spaces.

As a result of considering a smaller class of systems, we know that the systems we consider are globally exponentially stable, so all trajectories will converge to the equilibrium point at $x = 0$. This means we are in a similar position as we were in the continuous case, where we know the system is stable and attractive, but we now want to prove this by abstraction to a timed automaton.

4.2.2 Finding dynamics on a subsystem boundary

When we consider a boundary between two subsystems with different dynamics, there are four different ways in which dynamics at the boundary can interact, depending on the directions of the dynamics either side of the boundary. In the piecewise-linear systems we are considering, the boundaries are at a constant value in some dimension i: let us say the boundary we are considering is at $x_i = c$. Then, letting the dynamics below the surface be denoted by $\dot{x}^- = A^- x$, and above the surface by $\dot{x}^+ = A^+ x$, we have the following four options.

Case 1. $\dot{x}_i^- > 0$ and $\dot{x}_i^+ > 0$. When $x_i < c$, the dynamics are pushing the trajectories towards the surface $x_i = c$, and when $x_i > c$ the dynamics are pulling the

[1]A set is compact if every cover of it has a finite sub-cover, hence an already finite set will always satisfy this condition.

trajectories away from the surface, so all trajectories pass through the subsystem boundary $x_i = c$ in effectively zero time.

Case 2. $\dot{x}_i^- > 0$ and $\dot{x}_i^+ < 0$. Both sets of dynamics are pushing the trajectories towards the surface, so the trajectories must travel along the boundary in some fashion.

Case 3. $\dot{x}_i^- < 0$ and $\dot{x}_i^+ > 0$. Both sets of dynamics are pulling the trajectories away from the surface, so we would tend to think that trajectories which start *on* the boundary remain there forever, but trajectories which start *close* to the surface will not remain close forever.

Case 4. $\dot{x}_i^- < 0$ and $\dot{x}_i^+ < 0$. In the analogous situation to case 1, the trajectories are pushed through the surface in zero time.

Look ahead to Figure 4.1 for an illustration of these possibilities.

In Cases 1 and 4, where the dynamics either side are in the same direction, it is clear that the behaviour of the overall piecewise-linear system at this boundary is to go straight through the boundary in zero time. However, in Case 2 the situation is more complex, as we see that the dynamics of the system will tend towards the boundary from both sides, and so we expect the $\dot{x}_i$ dynamics to balance out on $x_i = c$, but the dynamics in other dimensions will force trajectories to slide *along* the boundary in some way. The situation is similar (in theory) in Case 3 — if the trajectory starts on the boundary it is possible for it to remain there because the dynamics both sides are trying to pull it away and therefore cancel each other out.

In the control community, the behaviour of piecewise-continuous systems at sliding boundaries between subsystems has been long considered, and various solutions have been proposed. However, they are usually based on the same idea, which is to take a convex combination[2] of the dynamics either side of the boundary to describe the dynamics when we slide along the boundary [Filippov, 1988]. A convex combination ensures that the dynamics along the boundary lie in between the dynamics either side. The methods used to find these equations along a boundary are grouped into the theory of *sliding modes*, where the 'sliding mode' is the name for the dynamics that we define along the sliding surface (the boundary).

[2]The vector f is a convex combination of vectors f_1 and f_2 if $f = c_1 f_1 + c_2 f_2$ with $0 \leq c_1, c_2, \leq 1$.

Two main methods used to describe these boundary dynamics are Filippov's continuation method [Filippov, 1988] and Utkin's equivalent control method [Utkin, 1992], but in our case they boil down to the same result, which is that the dynamics on the sliding surface $x_i = c$ are described by the equation $\dot{x}^s = f^s(x)$, where

$$f^s(x) = \frac{(A^-x)_i A^+x - (A^+x)_i A^-x}{(A^-x - A^+x)i}. \tag{4.3}$$

Here we use the notation $(\cdot)_i$ to denote the i-th component of the vector $(\cdot)$. The x in Equation (4.3) is assumed to have its i-th component set equal to c. The equation for $\dot{x}_j$ at the surface $x_i = c$ is obtained by taking the j-th component of f^s, giving

$$\dot{x}_j^s = \frac{(A^-x)_i (A^+x)_j - (A^+x)_i (A^-x)_j}{(A^-x - A^+x)i}.$$

In particular, the i-th component of $\dot{x}$ on the surface $x_i = c$ has the terms in the numerator cancelling out so that $\dot{x}_i^s = 0$. This says that Eq. (4.3) has been defined in such a way that the dynamics either side of the surface $x_i = c$ cancel out, which is exactly the effect we expect (trajectories travel along the surface).

For more information on sliding modes, see Utkin's book [Utkin, 1992], which has a thorough explanation of methods from the control perspective.

4.3 Extending the method for timed-automaton abstraction to piecewise-linear systems

In this section, we will start from the basis of having the method to make a TA abstraction for continuous systems of the special form of (3.18) and (3.19), and will think about the various problems which need to be overcome to extend it to the piecewise-linear case. There are two properties we want to preserve with any extension to piecewise-linear systems.

1. The first property to preserve is that the TA must over-approximate the number of trajectories of the original system. Then, if inevitability is proved in the TA it proves inevitability in the original system. For continuous systems, this was guaranteed by the method of Maler and Batt [2008] whatever splitting was chosen, and so in Section 4.3.1 we will extend the method of Maler and Batt to over-approximate the number of trajectories in piecewise-linear systems.

2. The second property we wish to preserve from the continuous case is the ability to choose a splitting which guarantees that the TA abstraction will prove inevitability if the original piecewise-linear system satisfied the inevitability property. This property guarantees that if inevitability holds then we can create a TA abstraction which also satisfies the inevitability property. In the continuous case we achieved this property by a careful choice of splitting of the state space, and so in Section 4.3.2 we will begin the discussion of the extension of this splitting, the continuation of which will encompass the rest of this chapter.

4.3.1 Making an over-approximating abstraction

In this section we will extend the method of Maler and Batt [2008], presented in Section 3.2, in order for it to make over-approximating TA abstractions of piecewise-linear systems, in terms of the number of trajectories included. Let us first pick out the three main elements of the method of Maler and Batt for continuous systems, which, given a splitting of the state space, does the following:

1. For each box facet $x_i = c$ between boxes with labels v and v', the Maler and Batt method considers the direction of the derivative $\dot{x}_i$ across the facet: if $\dot{x}_i \geq 0$ then only a transition in the positive direction (v to v') is added to the automaton abstraction, if $\dot{x}_i \leq 0$ then only a transition in the negative direction (v' to v) is added, and if $\dot{x}_i$ can have any sign then transitions in both directions are added.

2. For each box labelled by v in the splitting, the minimum absolute value of the derivative in every dimension is calculated as $\underline{\underline{f}}_i = \inf(|f_i|)$. The maximum time that can be spent in a box labelled by v is calculated as $\overline{t_v} = \min_{1 \leq i \leq n}(d_i / \underline{\underline{f}}_i)$ where d_i is the width of this box v in the i-th dimension.

3. For each slice limited by two splits $x_i = c$ and $x_i = c'$, the limits on the velocity across this slice are found, given by $\underline{f}_i \leq f_i \leq \overline{f}_i$. The values assigned to the slice times are defined by the relative values of these limits (see Table 3.1).

The main difference between a linear and a piecewise-linear system is that boundaries between different types of dynamics can occur throughout the space. If one of these boundaries $x_i = c$ between two subsystems passes through the interior[3] of a box then the possible values for the derivatives f_i inside the box and across the box facets will depend on both of the subsystems' dynamics in this box. Actually there can be multiple boundaries passing through a box, when different parts of the box are part of different subsystems, and so the derivatives will depend on the values of multiple sets of dynamics in the box. This affects the three elements of the abstraction in the following way.

1. For a box facet $F(v, v') \subset \{x : x_i = c\}$ between any two boxes v and v', we must look at the direction of the $\dot{x}_i$ derivatives in every subsystem that is defined on the facet. Any subsystem boundaries that pass through the interior of the box facet mean that the facet is divided into parts, with each part having dynamics defined by a different subsystem. To find the direction of flow allowed across the whole facet $F(v, v')$, we look at the direction of flow across the facet for each of the k subsystems which exist on a part of the facet: we consider what part of the facet a subsystem is defined on, and decide which directions the flow can go across the facet for this subsystem, using the method for continuous systems for each subsystem's dynamics. Putting these flow directions together we can decide whether the flow across the boundary is allowed to go in the positive direction v to v', the negative direction v' to v, both, or neither.

   There is also the possibility that the box facet coincides with a sub-system boundary, and then we should consider the derivatives for $\dot{x}_i$ when $x_i \to c^-$ in box v and also when $x_i \to c^+$ in box v' to assess in which directions transitions should be defined across the boundary.[4] Using the values of all of these derivatives over-approximates the number of transitions possible, as all the possible values are included in the abstraction. However, this step can include a lot of trajectories in the TA that are spurious (not present in the original system). We will discuss this problem more in the Section 4.3.2.

2. The box times for the TA give an overestimate of the maximum time taken to cross each box. In continuous systems, we found the minimum absolute velocity

[3]By $x_i = c$ *passing through the interior* we imply that this surface does not form one of the boundaries of the box.

[4]By $x_i \to c^-$ we mean take the one sided limit as x_i goes tends to c from below, and vice-versa for $x_i \to c^+$ being the limit from above.

for each dimension in the box $\underline{\underline{f_i}}$, and used these to quantify the maximum time to cross each dimension of the box with width d_i. We need to do the equivalent in piecewise-linear systems, ensuring that we over-approximate the time it takes to cross each dimension of the box. The complication with piecewise-linear systems occurs if parts of the box are in different subsystems. Let us consider a box labelled by v, with K the set of subsystems which form part of this box. Then, for each subsystem $k \in K$ we find the minimum and maximum velocities in this box in dimension i, which we denote

$$\underline{f_{i,v}^k} = \inf_{x \in (X_v \cap S^k)} f_i^k \text{ (the minimum)},$$
$$\overline{f_{i,v}^k} = \sup_{x \in (X_v \cap S^k)} f_i^k \text{ (the maximum)}.$$

Then to find the minimum and maximum velocity across all subsystems in this box in dimension i, we take the minimum or maximum across all subsystems in K (those which exist in part of the box v). This gives us the minimum and maximum velocity in the box v in dimension i of

$$\underline{f_{i,v}} = \min_{k \in K} \underline{f_{i,v}^k} \text{ (the minimum)},$$
$$\overline{f_{i,v}} = \max_{k \in K} \overline{f_{i,v}^k} \text{ (the maximum)}.$$

We can now define the minimum absolute velocity across the box v in dimension i, as

$$\underline{\underline{f_{i,v}}} = \begin{cases} \underline{f_{i,v}} & \text{if } 0 < \underline{f_{i,v}} \leq \overline{f_{i,v}}, \\ -\overline{f_{i,v}} & \text{if } \underline{f_{i,v}} \leq \overline{f_{i,v}} < 0, \\ 0 & \text{if } \underline{f_{i,v}} \leq 0 \leq \overline{f_{i,v}}. \end{cases} \tag{4.4}$$

With this equation for the maximum velocity in each dimension i, if d_i is the width of the box in this dimension then an over-approximation of the maximum time to cross the box v is the following (the same as for the continuous case).

$$\overline{t_v} = \min_{1 \leq i \leq n} \frac{d_i}{\underline{\underline{f_{i,v}}}}. \tag{4.5}$$

3. Similarly to the box times, calculation of slice times is found by taking the limits of the times in each of the subsystems in the areas of the slice that they are

defined, giving $\underline{f_i^k} \leq f_i^k \leq \overline{f_i^k}$ in each relevant subsystem k. Then we take the outer limits of these to give

$$\min_{k \in K} \underline{f_i^k} \leq f_i \leq \max_{k \in K} \overline{f_i^k}$$

as limits on the velocity that can be taken in this i-th dimensional slice. We can then use the slice times calculated by the entries in Table 3.1, which will then be outside limits on the time it can take to cross the slice we are looking at. So all of the trajectories of the original system are included in the TA abstraction.

All of these parts of the abstraction have over-approximated the number of trajectories allowed in the system, and so the whole abstraction method allows every trajectory of the original piecewise-linear system to exist in the TA abstraction. Hence the TA abstraction created using this extended version of the method of Maler and Batt [2008] will prove inevitability in the piecewise-linear system if we can show it does itself satisfy the inevitability property.

4.3.2 Removing spurious trajectories

We now consider how we have included spurious trajectories in the TA abstraction which did not exist in the original system. Removing all spurious trajectories will mean that the TA will satisfy the inevitability property if the original system does, meaning that we can guarantee the TA will prove inevitability if it is true.

Assume we have a box with a subsystem boundary $x_i = c$ going through the interior of it. Then if the dynamics of the two subsystems go in opposite directions in any dimension j, the equation for finding minimum absolute velocity, Eq. (4.4), will give an infinite time to cross the box in this dimension j. As we do not require continuity of the flow across the subsystem boundary, there is a higher chance of every dimension having flow in both positive and negative dimensions, meaning that the box time will be infinite even if every trajectory of the original system would leave the box labelled by v in finite time. There is also a problem with how to decide where to make a splitting to remove the two-way flow in the box if there is a discontinuity through it.

For these reasons, we have decided that it makes sense to put a split at each boundary of a subsystem, in order to be able to consider each box as only belonging to one subsystem, therefore having only one set of dynamics defined in it. This also means that some of the theory developed for splitting continuous systems to satisfy the assumptions of Section 3.4.2 should directly transfer over to the piecewise-linear case.

In the continuous case, we satisfied Assumption 3.6, which said that the equilibrium was in the interior of the live box, meaning that no splits could be made at $x_i = 0$ in any dimension. In the piecewise-linear case we would like to preserve the property of the equilibrium being in the interior of one box, which means that we cannot have any split made at $x_i = 0$ for any dimension $i = 1, \ldots, n$. Hence, if we are assuming a split is made at every subsystem boundary, we must make an additional restriction on the systems that we consider, which is that no subsystem should have a boundary at zero in any dimension. More formally, we impose the extra condition on Equation (4.2) that

$$s^k_{i\pm} \neq 0 \text{ for every dimension } i = 1, \ldots, n \text{ in all subsystems } k = 1, \ldots, m. \tag{4.6}$$

From the continuous splitting algorithm (Alg. 3.1) and the discussion above we can now see where the initial split points of the piecewise-linear system should be made:

- The system-wide state space limits given by S, that is $x_1 = \{s_{1-}, s_{1+}\}$, $x_2 = \{s_{2-}, s_{2+}\}$, …, $x_n = \{s_{n-}, s_{n+}\}$.
- The inputted live box limits given by L, that is $x_1 = \{l_{1-}, l_{1+}\}$, $x_2 = \{l_{2-}, l_{2+}\}$, …, $x_n = \{l_{n-}, l_{n+}\}$.
- The individual subsystem limits given by the S_k, that is $x_1 = \{s^k_{1-}, s^k_{1+}\}$, $x_2 = \{s^k_{2-}, s^k_{2+}\}$, …, $x_n = \{s^k_{n-}, s^k_{n+}\}$ for each $k = 1, \ldots, m$.

We will now consider what happens to the trajectories of the TA abstraction for the piecewise-linear system when we encounter a boundary between subsystems. We began this discussion in Section 4.2.2, where we gave a brief introduction to the theory of sliding modes. Considering a boundary surface where $x_i = c$, without loss of generality, whether we move through this surface or along this surface depends on the directions of the dynamics of x_i either side.

There are four possibilities for the dynamics on a boundary, as discussed in Section 4.2.2 and illustrated by the diagrams in Figure 4.1. Recall that we use the notation

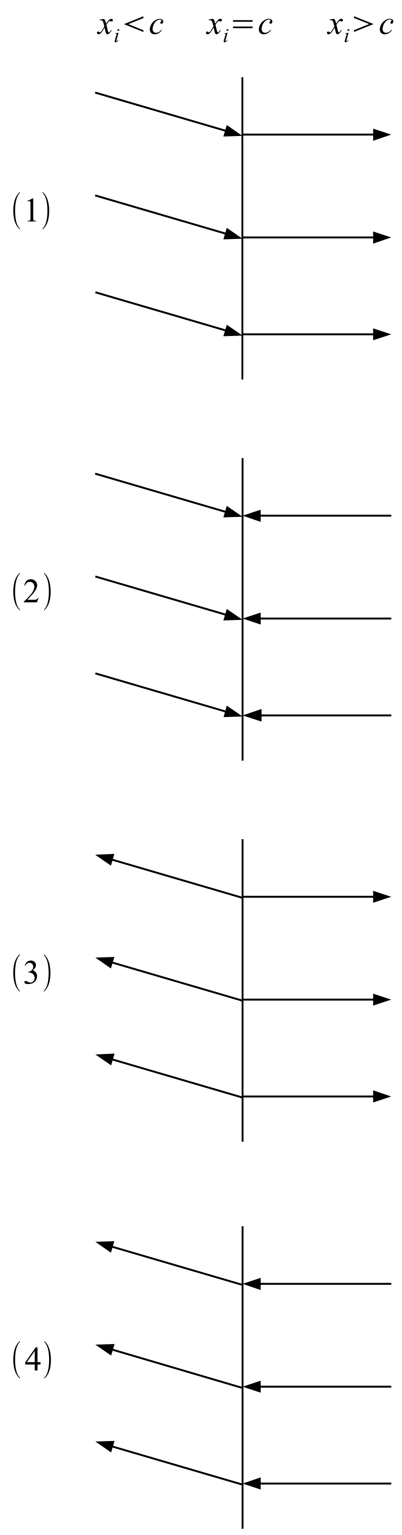


Figure 4.1: The four cases for how dynamics can evolve on the boundary between two subsystems.

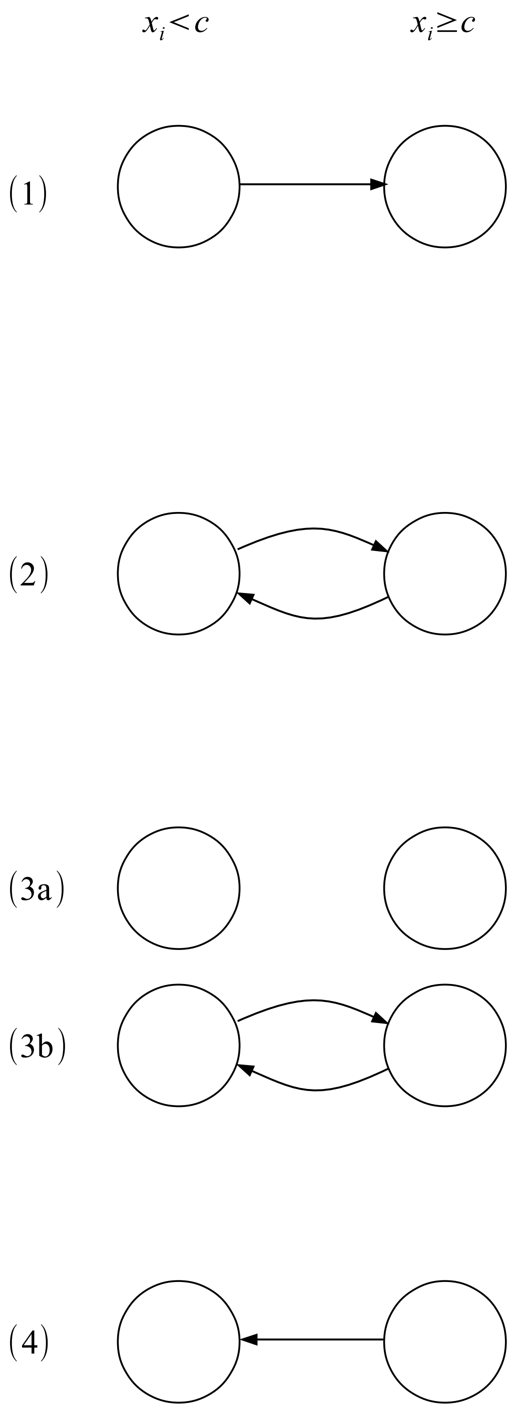


Figure 4.2: Timed-automaton abstraction states when we split at the boundary in each case.

$\dot{x}_i^-$ for the dynamics where $x_i \leq c$, and $\dot{x}_i^+$ for the dynamics where $x_i \geq c$. The four cases are the following.

Case 1. $\dot{x}_i^- > 0$ and $\dot{x}_i^+ > 0$. All trajectories pass through the boundary $x_i = c$ in effectively zero time (Figure 4.1 (1)).

Case 2. $\dot{x}_i^- > 0$ and $\dot{x}_i^+ < 0$. Both sets of dynamics are pushing the trajectories towards the surface, so the trajectories must travel along the boundary in some way. In this part of the space, $x_i = c$ is a stable sliding surface (Figure 4.1 (2)).

Case 3. $\dot{x}_i^- < 0$ and $\dot{x}_i^+ > 0$. Both sets of dynamics are pulling the trajectories away from the surface, so this boundary is an unstable sliding surface. That is, trajectories which start *on* the surface remain there forever, but trajectories which start *close* to the surface will not remain close forever (Figure 4.1 (3)).

Case 4. $\dot{x}_i^- < 0$ and $\dot{x}_i^+ < 0$. All trajectories are pushed through the surface in zero time (Figure 4.1 (4)).

When we place a split at the boundary point, the TA abstraction in each case becomes like that shown in Figure 4.2. Cases 1 and 4 are abstracted in a useful way, in that the TA abstraction captures discretely the behaviour that the trajectories go straight through the boundary from one state to the other (Figures 4.2 (1) and (4)). However, in Case 2, where we have a stable sliding surface, we have the limits $\lim_{x_i \to c^-} \dot{x}_i > 0$ and $\lim_{x_i \to c^+} \dot{x}_i < 0$. This means that when we consider the possible transitions on the boundary $x_i = c$ we see that flow is possible in both directions. So we end up with the automaton states as in Figure 4.2 (2), where we have introduced a pair of states which will form an infinite length discrete trace of the TA that does not enforce progress of time. This behaviour goes exactly against the idea of Assumption 3.8 in the continuous case and will prevent the TA abstraction from satisfying the inevitability property.

In the final case (Case 3), which transitions we define depend on how we define the splitting method. If we consider the flow on the boundary as directional limits from below and above then we will see that the two sides flow away from each other and will not add any transitions as shown in Fig. 4.2 (3a). However if we consider the limit of the dynamics on the boundary without considering explicitly the limits from below

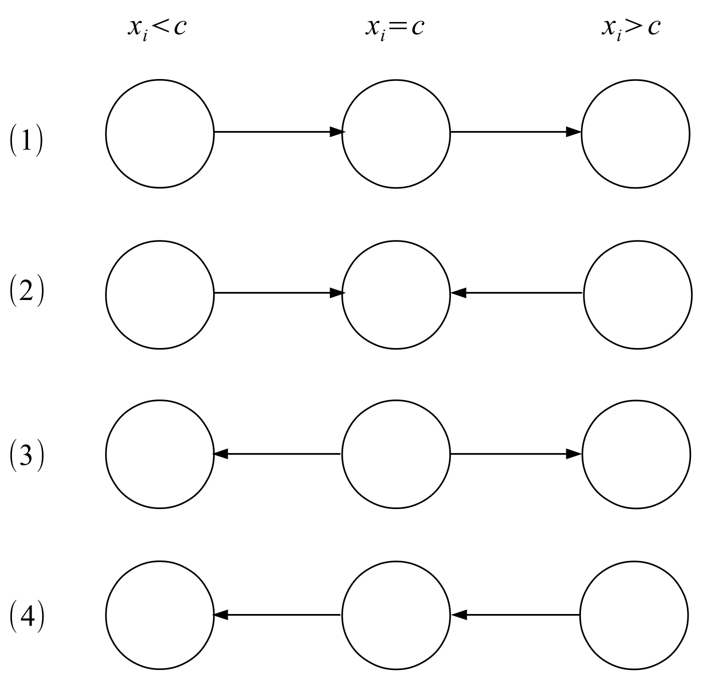


Figure 4.3: Timed-automaton abstraction states when we put two splits at the boundary in each case.

and above then the transitions are defined in both directions allowing an infinite TA run which does not progress time (Fig. 4.2 (3b)), as in Case 2. In our case, we have defined the subsystem state spaces as $x_i < c$ below and $x_i \geq c$ above, the option (3a) should capture the situation, as if a trajectory starts on the boundary $x_i = c$ then we assume it evolves by the dynamics $\dot{x}^+$.

Returning to the problem of stable sliding surface boundaries (Case 2), we see that there is no way of removing these pairs of states with two-way flow between them using only splits in dimensions $j \neq i$. However, we see that we can apply multiple splits at the boundary $x_i = c$ where the sliding occurs, to force a row of TA states *along the sliding surface*. We define two splits at each sliding surface position $x_i = c$, and give them an order, with the lower-ordered split taking the dynamics of the subsystem in the region $x_i < c$ and the upper split taking the dynamics of the subsystem in the region $x_i > c$. This gives us a row of TA states along the boundary, which immediately improves the nature of the discrete abstraction across the boundary, as illustrated in Figure 4.3 for each of the cases in Figures 4.1–4.2.

We now need to define what the dynamics are along the boundary, to give timing behaviour between the new states just introduced. In the section on sliding modes

(Sec. 4.2.2) we discussed the fact that the main methods are equivalent to the Filippov method [Filippov, 1988] in our case. For a sliding surface at $\dot{x}_i = c$, with dynamics either side of the surface denoted $\dot{x}^- = A^- x$ when $x_i < c$ and $\dot{x}^+ = A^+ x$ when $x_i > c$, with $A^\pm \in \{A_1, \ldots, A_m\}$, we can describe the dynamics on the surface by a convex combination of the dynamics of either side:

$$\dot{x}^s = f^s(x) = \frac{(A^- x)_i A^+ x - (A^+ x)_i A^- x}{(A^- x - A^+ x) i}. \tag{4.7}$$

The problem with using this equation in our special class of piecewise-linear systems is that the dynamics $\dot{x} = f^s(x)$ are not guaranteed to be linear. This is because the numerator is a nonlinear product of linear terms with a linear denominator — the two will only cancel out to make a linear term in very similar pairs of systems. Even if $f^s(x)$ is linear, it will not (in general) be of the special form of Equation (4.1). We do not wish to make the extra assumption of very restricted pairings of subsystems, and so think about how we can turn the result of (4.7) into linear dynamics of the special form.

The method of Maler and Batt [2008] uses precise dynamics to make under- and over-approximations of the time spent in a cube or slice of the system. However, as long as we preserve the under- or over-approximation of the dynamics we can use estimated dynamics to make these limits on time, as this simply increases the number of trajectories included in the TA (preserving the over-approximation of the number of trajectories). Since the Filippov dynamics f^s are a convex combination of the dynamics either side, that is $f^s(x) = c_1\, A^- x + c_2\, A^+ x$ with $0 \leq c_1, c_2 \leq 1$, the value of f^s at any particular point lies between the values of the dynamics either side. So the dynamics of the sliding surface $\dot{x} = f^s(x)$ satisfy the following inequality in each dimension $j = 1, \ldots, n$:

$$\min\left((A^- x)_j, (A^+ x)_j\right) \leq \; f^s_j \; \leq \max\left((A^- x)_j, (A^+ x)_j\right).$$

In particular this means that if the signs of $A^- x$ and $A^+ x$ are the same, then f^s has this same sign too. Hence, when looking at flow between states in the boundary for some cases we can guarantee that flow will only go in one direction between the states. And looking at each box in the boundary, if the limits of the dynamics in this box have the same sign throughout the box, then we can also use the smallest magnitude

$(A^{-}x)_j$	$(A^{+}x)_j$	Direction of flow allowed
> 0	> 0	$q_1 \rightarrow q_2$
> 0	< 0	$q_1 \leftrightarrow q_2$
< 0	> 0	$q_1 \leftrightarrow q_2$
< 0	< 0	$q_1 \leftarrow q_2$

Table 4.1: Transitions allowed by different flow directions between boxes along a sliding surface $x_i = c$. We consider a split line $x_j = d$ between two states, with q_1 satisfying $x_j < d \cap x_i = c$ and q_2 satisfying $x_j \geq d \cap x_i = c$.

$(A^{-}x)_j$	$(A^{+}x)_j$	$f^s_j(x)$	$\underline{\underline{f_j}}$
> 0	> 0	> 0	$\min((A^{-}x)_j, (A^{+}x)_j)$
> 0	< 0	$> 0, = 0, < 0$	0
< 0	> 0	$> 0, = 0, < 0$	0
< 0	< 0	< 0	$\min(-(A^{-}x)_j, -(A^{+}x)_j)$

Table 4.2: Minimum absolute velocity $\underline{\underline{f}}$ in dimension j in a box in a sliding surface $x_i = c$.

velocity to give us a longest time that can be spent in this box. We represent these ideas in Tables 4.1 and 4.2.

4.4 Method for splitting piecewise-linear systems

In this section we will summarise the changes that we have made to the splitting method for continuous systems (Algorithms 3.1–3.3) in order to make TA abstractions of piecewise-linear systems. The algorithms implement the method of Section 4.3.1, so create an over-approximating abstraction in terms of the number of trajectories included. The method for piecewise-linear systems also uses the ideas in Section 4.3.2 to remove some of the spurious trajectories that can get included in the splitting.

For continuous systems, the splitting algorithm we introduced (Algorithm 3.1) had broadly three parts:

1. The initial splitting, based on given “key lines” in the system.
2. The sub-algorithm `FollowSplits`, which ensured Assumption 3.8 (no two-way crossing of a box facet).
3. The method to remove infinite time on a box, which ensured Assumption 3.9

(only the live box L is allowed infinite time).

The algorithm for piecewise-linear systems has the same three parts, all of which have been changed somewhat from the continuous case. The ideas of the algorithm's parts are still to attempt to ensure the four assumptions established in Chapter 3, as these are key principles that an inevitability-proving abstraction must satisfy. However, in the piecewise-linear case we cannot necessarily ensure the assumptions, and also the assumptions may not be enough to prove completeness of the TA, so when we prove the inevitability of the TA abstraction (in certain cases), we will prove the desired theorems directly, without the intermediate method of the assumptions.

We have already discussed some of the changes, but now we present the method formally as Algorithms 4.1–4.4. The main routine in Algorithm 4.1 now has extra subroutines `RemoveSlidingInfiniteTimes` and `RemoveNormalInfiniteTimes` to make it clear what the structure of the algorithm is. We will now discuss the methods used for each of these algorithms.

4.4.1 Main splitting algorithm for piecewise-linear systems

In this section we will explain the main splitting algorithm for piecewise-linear systems, and we will begin to quantify the size of the resulting abstraction.

The main algorithm (Algorithm 4.1) starts by initialising the splitting state of the system. For our piecewise case, we use the following information to initialise the current splitting state C:

1. The system-wide state space limits given by S, that is $x_1 = \{s_{1-}, s_{1+}\}$, $x_2 = \{s_{2-}, s_{2+}\}$, …, $x_n = \{s_{n-}, s_{n+}\}$.
2. The inputted livebox limits given by L, that is $x_1 = \{l_{1-}, l_{1+}\}$, $x_2 = \{l_{2-}, l_{2+}\}$, …, $x_n = \{l_{n-}, l_{n+}\}$.
3. Each of the m subsystem's limits given by the S_k, that is $x_1 = \{s^k_{1-}, s^k_{1+}\}$, $x_2 = \{s^k_{2-}, s^k_{2+}\}$, …, $x_n = \{s^k_{n-}, s^k_{n+}\}$ for each $k = 1, \ldots, m$.
4. Extra split added for every position of a sliding surface, calculated by investigating every boundary between systems and seeing if the dynamics either side

Algorithm 4.1 Automatic splitting algorithm for piecewise-linear systems

Input: Piecewise-linear dynamical system with each subsystem's x_i dynamics of one of the forms of (4.1), with global state space $[s_{1-}, s_{1+}) \times \ldots \times [s_{n-}, s_{n+})$, subsystem state spaces S_k (see equation 4.2) for each $k = 1, \ldots, m$, and live box $L = [l_{1-}, l_{1+}) \times \ldots \times [l_{n-}, l_{n+})$.

Output: A splitting of the system such that the TA abstraction proves inevitability of the live box *in certain cases.*

```
1: for $i = 1, \ldots, n$ do                                                    ▷ Step 1
2:     Initialise $C_i$ (the current splitting state) to the unique subset of $x_i = l_{i-}, l_{i+}, s_{i-}, s_{i+}$, and $x_i = s^k_{i-}, s^k_{i+}$ for every subsystem $k$.
3:     $N_i \leftarrow C_i$ without the endpoints $s_{i-}$ and $s_{i+}$.
4:     Add an extra split to $C_i$ at every boundary where a sliding surface can occur.
5: end for
6: call FollowSplitsPWL                                                          ▷ Step 2
7: Calculate box times                                                           ▷ Step 3
8: $B \leftarrow$ list of boxes with infinite box time (except $L$)
9: while $B$ is non-empty do
10:     for $k = 1, \ldots, \text{length}(B)$ do
11:         $V \leftarrow$ get vertices of box $B(k)$
12:         $Z \leftarrow V$ (initialise the region where zero surfaces occur: let $Z^-_i$ be lower bound and $Z^+_i$ be upper bound in dimension $i$)
13:         if box $k$ is inside a sliding mode then
14:             call RemoveSlidingInfiniteTimes
15:         else (if box $k$ is part of a subsystem)
16:             call RemoveNormalInfiniteTimes
17:         end if
18:     end for
19:     call FollowSplitsPWL
20:     $B \leftarrow$ new list of boxes with infinite box time (except $L$)
21: end while
```

can go in opposite directions on this boundary. There will be two splits in C at the same point for each sliding surface.

We form C_i the current splitting state by taking the unique subset of the union of all the x_i points in items 1–3 of the list, and then duplicate points for the positions of the sliding mode surfaces (item 4).

The list N of splits that need to be followed by the `FollowSplitsPWL` algorithm is given by the unique set of C, with the system-wide limits removed, so N_i is just the unique subset of the x_i points from items 2 and 3 of the list. We want a unique set so we do not follow any split more than once, and like in the continuous case we do not follow splits on the outside edge of the system as they do not cause two-way flow on a facet. Since this step of Algorithm 4.1 consists of a for loop iterated a finite number n times (lines 1–5), with a list of simple assignments inside it, Step 1 of the algorithm will terminate.

To quantify the worst case size $|C|$ of the initial splitting we can say that in each dimension there are two splitting points for each of the subsystems, plus two for the live box (making $2m + 2$ in each dimension), which would include two splits at every boundary to allow for the sliding mode splits to be added in (item 4 in the list on page 125). Notice that the system-wide limits are included in this estimate since there must be at least one subsystem which touches each boundary of the system. However, we can do better than this in dimension n (and any other dimension where we know that no sliding modes exist): wherever a subsystem state space finishes (at $s^{k_1}_{n+}$ for example) either another subsystem will start (so $s^{k_1}_{n+} = s^{k_2}_{n-}$) or we will reach the edge of the whole system (so $s^{k_1}_{n+} = s_{n+}$). Hence we can remove at least half of the splits that are induced by S and the S_k combined before adding the two for the live box (giving $m + 3$ splits).

The calculation for $|N|$ is similar to the calculation for dimension n, as we remove all possible sliding modes and also the system-wide state space splits. Hence $|N| = m+1$.

Bringing these sizes together, we have that the worst case initial splitting size is

$$|C_i| = \begin{cases} 2m+2 & \text{where sliding modes exist in dimension } i \\ m+3 & \text{where sliding modes do not exist in dimension } i \end{cases} \tag{4.8}$$

$$|N_i| = m+1 \tag{4.9}$$

Step 2 of Algorithm 4.1 calls the `FollowSplitsPWL` algorithm, which we will discuss in the next section. This step aims to remove two-way crossings of box facets from the initial splitting. After this, Alg. 4.1 starts Step 3 which aims to remove the infinite box times. We cannot get such a tight limit for the maximum number of infinite-time boxes as the one deduced for continuous systems in Proposition 3.14.3. However, in the worst case we can limit it by the number of boxes in the splitting, subtracting one for the live box L. We can slightly improve this limit by remembering that in the n-th dimension of every subsystem the dynamics are diagonal, that is in every subsystem k we have $\dot{x}_n = a^k_{n,n}x_n$, and so infinite-time boxes can only occur in the central slice of the n-th dimension. This means that there are a maximum of

$$\left(\prod_{i=1,\ldots,n-1} (|C_i| - 1) \right) - 1 \tag{4.10}$$

boxes that have infinite time, excluding L. Note that we could try to extend the limit on the number of infinite-time boxes that we created in the continuous case (see Proposition 3.14), but as we have many subsystems there is no guarantee that this will create a smaller number than the simple limit given above. As we are only interested in the fact that the number of infinite-time boxes is finite, we will just use this simple estimate.

For each box that is part of a subsystem, or on a boundary between subsystems but not in a sliding mode, Step 3 does exactly the same routine as for the continuous case, we call `RemoveNormalInfiniteTimes` for the PWL case. However, if the box is part of a sliding mode then a new method is needed to calculate where we can put a split to remove the infinite time on the box, by calling `RemoveSlidingInfiniteTimes`.

4.4.2 `FollowSplitsPWL` sub-algorithm

In the continuous case, the sub-algorithm `FollowSplits` takes as input a list (N) of recent splits that have been made, and working from dimension 1 to dimension n calculates the new splits that need to be added to separate the occurrence of positive and negative derivatives on these new splitting surfaces. To update this process to piecewise-linear systems, we need to consider whether a split in N will induce a new split in each of the m subsystems.

The new `FollowSplitsPWL` algorithm is given in Algorithm 4.2. We now have three nested for loops, which select (respectively) the dimension of the split, the position of a split in this dimension, and the subsystem where this specific split can occur. Once we have chosen a specific splitting surface $x_i = p$ and a subsystem k to use, we find the values of $\dot{x} = A_k x$ at each vector in the corner set $V = \{s^k_{1-}, s^k_{1+}\} \times \ldots \times \{s^k_{i-1,-}, s^k_{i-1,+}\} \times p \times \{s^k_{i+1,-}, s^k_{i+1,+}\} \times \ldots \times \{s^k_{n-}, s^k_{n+}\}$, assigning these values to a variable $v = \{(A_k V_j)_i : V_j \in V\}$. If we have occurrences of both positive and negative values in v, then we must add a split in this subsystem at the point which makes $\dot{x}_i = 0$ with $x_i = p$. This induces a split in the dimension j that exists in the equation

Algorithm 4.2 `FollowSplitsPWL` sub-algorithm

Input: Current splitting state of the system $C = \{C_1, \dots, C_n\}$, the list $N = \{N_1, \dots, N_n\}$ of newly made splits in each dimension which need to be followed down the dimensions, and the dynamics of the system.

Output: A new splitting C with only one direction of flow across the faces of the boxes.

```
1: for $i = 1, \dots, n-1$ do
2:     for all $p \in N_i$ do
3:         for all subsystems $k$ where $p \in [s^k_{i-}, s^k_{i+})$ do
4:             $V \leftarrow$ the vertices of the splitting surface $x_i = p$ in the region $S_k$
5:             $v \leftarrow$ find the value of $\dot{x}_i = (A_k x)_i$ at all vertices in $V$
6:             if any($v < 0$) and any($v > 0$) then
7:                 $d \leftarrow$ value of $x_j$ which solves $\dot{x}_i = A_k x = 0$ when $x_i = p$ for some $i < j \leq n$
8:                 $C_j \leftarrow C_j \cup d$
9:                 $N_j \leftarrow N_j \cup d$
10:            end if
11:        end for
12:        $N_i \leftarrow N_i \setminus \{p\}$
13:    end for
14: end for
15: $N_n \leftarrow \emptyset$
```

for $\dot{x}_i = (A_k x)_i = a_{i,i}x_i + a_{i,j}x_j$, in the same way as the continuous `FollowSplits` method.

We now show the termination of `FollowSplitsPWL`, and also calculate its abstraction size. `FollowSplitsPWL` consists of three for loops, the first clearly has a finite number of executions $(n-1)$. The second iterates over the finite number of elements in the list N_i, and for a maximum of m iterations the third for loop can add an entry to one of the lower dimension's lists. The $p \in N_i$ loop then removes an element from the current dimension list N_i, so each iteration of p reduces the size of N_i by one. Since all these loops are over finite domains and all the calculations are terminating, `FollowSplitsPWL` terminates.

The number of splits that can be added by this is dependent on the number of subsystems m, on the size of the inputted current splitting in each direction ($|C_i|$ for $i = 1 \dots, n$), and also dependent on the size of the inputted newly-made splits list ($|N_i|$ for $i = 1, \dots, n$). The worst case is when each dimension causes a split in the dimension immediately after it in every subsystem, as the effect of these splits builds up. So the new worst case sizes $|C|'$ and $|N|'$ of the sets $|C|$ and $|N|$ are given by the following, for each dimension $i = 1, \ldots, n$:

$$|C_i|' = |C_i| + \sum_{j=1}^{i-1} m^{i-j}|N_j| \tag{4.11}$$

$$|N_i|' = 0. \tag{4.12}$$

4.4.3 `RemoveSlidingInfiniteTimes` sub-algorithm

This sub-algorithm attempts to separate the places where zero values in the derivatives can occur, specifically for boxes in a sliding surface. There are four cases requiring slightly different methods depending on whether the dynamics either side of the sliding surface have off-diagonal entries or not. For ease of reading, Algorithm 4.3 only shows the method for the most complex case where the dynamics both sides of the sliding mode have off-diagonal elements. The other three cases are similar, but less complex, and so we indicate their contents by ellipses (lines 25, 27, and 29).

In the most complex case, which is shown in the algorithm, we assume the dynamics above and below the sliding surface have off-diagonal entries. Then there are four possible regions where a certain variable x_i can have zero-valued sliding mode dynamics, that is $\dot{x}_i^s = 0$.

- The two regions where the derivatives either side can be zero: $x \in V$ such that $\dot{x}_i^- = 0$ (line 5), and $x \in V$ such that $\dot{x}_i^+ = 0$ (line 6).
- The two regions where the derivatives either side can have opposite signs, which allows zero derivative to occur in between them by some convex combination: $x \in V$ such that $\dot{x}_j^- > 0 \wedge \dot{x}_j^+ < 0$ (line 7), and $x \in V$ such that $\dot{x}_j^- < 0 \wedge \dot{x}_j^+ > 0$ (line 8).

These four regions are all calculated as intervals of the related variables. For instance, the limits of the $(n-1)$ dimensional surface $\dot{x}_i^- = a_{i,i}^- x_i + a_{i,j}^- x_j = 0$ in the box with vertices V are $[c_{i-}^1, c_{i+}^1]$ in dimension i and $[c_{j-}^1, c_{j+}^1]$ in dimension j, and similarly for the surface $\dot{x}_i^+ = a_{i,i}^+ x_i + a_{i,k}^+ x_k = 0$ we get limits $[c_{i-}^2, c_{i+}^2]$ on x_i and $[c_{k-}^2, c_{k+}^2]$ on x_k. We then do the same for the intersection of intervals where $\dot{x}_i^- > 0$ and $\dot{x}_i^+ < 0$, giving intervals of existence in x_i, x_j and x_k (denoted by the c^3's in Alg. 4.3) and similarly for $\dot{x}_i^- < 0$ and $\dot{x}_i^+ > 0$ (giving the c^4's).

Algorithm 4.3 `RemoveSlidingInfiniteTimes` sub-algorithm

Input: Set of box vertices V, initialised set Z where $\dot{x}_i = 0$ surfaces can occur in the box, the dynamics of the two boxes either side $\dot{x}^- = A^-x$ and $\dot{x}^+ = A^+x$, and the sets of current C and newly-made N splits.

Output: Extended C and N if infinite time is removed.

```
1: for $i = n, n-1, \dots, 1$ do
2:     if $(A^-x)_i$ and $(A^+x)_i$ both have off-diagonal entries then
3:         $j \leftarrow$ position of the off-diagonal entry in row $i$ of $A^-$
4:         $k \leftarrow$ position of the off-diagonal entry in row $i$ of $A^+$
5:         $\begin{bmatrix} c^1_{i-} & c^1_{i+} \\ c^1_{j-} & c^1_{j+} \end{bmatrix} \leftarrow$ limits of the surface $(A^-x)_i = 0$ in the box
6:         $\begin{bmatrix} c^2_{i-} & c^2_{i+} \\ c^2_{k-} & c^2_{k+} \end{bmatrix} \leftarrow$ limits of the surface $(A^+x)_i = 0$ in the box
7:         $\begin{bmatrix} c^3_{i-} & c^3_{i+} \\ c^3_{j-} & c^3_{j+} \\ c^3_{k-} & c^3_{k+} \end{bmatrix} \leftarrow$ intervals where $\dot{x}^-_i > 0$ and $\dot{x}^+_i < 0$ in the box
8:         $\begin{bmatrix} c^4_{i-} & c^4_{i+} \\ c^4_{j-} & c^4_{j+} \\ c^4_{k-} & c^4_{k+} \end{bmatrix} \leftarrow$ intervals where $\dot{x}^-_i < 0$ and $\dot{x}^+_i > 0$ in the box
9:         $[c_{i-}, c_{i+}] \leftarrow [\min_{p=1,2,3,4} c^p_{i-}, \max_{p=1,2,3,4} c^p_{i+}]$
10:        $[c_{j-}, c_{j+}] \leftarrow [\min_{p=1,3,4} c^p_{j-}, \max_{p=1,3,4} c^p_{j+}]$
11:        $[c_{k-}, c_{k+}] \leftarrow [\min_{p=2,3,4} c^p_{k-}, \max_{p=2,3,4} c^p_{k+}]$
12:        for $l = i, j, k$ do
13:            if $c_{l-} > Z_{l+}$ then
14:                add split at $x_l = (c_{l-} + Z_{l+})/2$ (or nearby if this is $x_l = 0$)
15:                break the $i$ loop
16:            else if $c_{l+} > Z_{l-}$ then
17:                add split at $x_l = (c_{l+} + Z_{l-})/2$ (or nearby if this is $x_l = 0$)
18:                break the $i$ loop
19:            end if
20:        end for
21:        $[Z_{i-}, Z_{i+}] \leftarrow [Z_{i-}, Z_{i+}] \cap [c_{i-}, c_{i+}]$
22:        $[Z_{j-}, Z_{j+}] \leftarrow [Z_{j-}, Z_{j+}] \cap [c_{j-}, c_{j+}]$
23:        $[Z_{k-}, Z_{k+}] \leftarrow [Z_{k-}, Z_{k+}] \cap [c_{k-}, c_{k+}]$
24:    else if $(A^-x)_i$ has off-diagonal entry, $(A^+x)_i$ does not then
25:        ...(analogous to above, removed for brevity)
26:    else if $(A^+x)_i$ has off-diagonal entry, $(A^-x)_i$ does not then
27:        ...(analogous to above)
28:    else (if $(A^-x)_i$ and $(A^+x)_i$ do not have off-diagonal entries)
29:        ...(analogous to above)
30:    end if
31: end for
```

Algorithm 4.4 `RemoveNormalInfiniteTimes` sub-algorithm

Input: Set of box vertices V, initialised set Z where $\dot{x}_i = 0$ surfaces can occur in the box, the dynamics of this box $\dot{x} = Ax$, and the sets of current C and newly-made N splits.

Output: Extended C and N if infinite time is removed.

```
 1: for $i = n, n-1, \dots, 1$ do
 2:     if $\dot{x}_i = (Ax)_i$ has an off-diagonal entry then
 3:         $j \leftarrow$ position of the off-diagonal entry in row $i$ of $A$
 4:         $\begin{bmatrix} c_{i-} & c_{i+} \\ c_{j-} & c_{j+} \end{bmatrix} \leftarrow$ limits of the surface $(Ax)_i = 0$ in the box
 5:         if $c_{j-} > Z_{j+}$ then
 6:             add split at $x_j = (c_{j-} + Z_{j+})/2$ (or nearby if this is $x_j = 0$)
 7:             break loop
 8:         else if $c_{j+} > Z_{j-}$ then
 9:             add split at $x_j = (c_{j+} + Z_{j-})/2$ (or nearby if this is $x_j = 0$)
10:             break loop
11:         else if $c_{i-} > Z_{i+}$ then
12:             add split at $x_i = (c_{i-} + Z_{i+})/2$ (or nearby if this is $x_i = 0$)
13:             break loop
14:         else if $c_{i+} > Z_{i-}$ then
15:             add split at $x_i = (c_{i+} + Z_{i-})/2$ (or nearby if this is $x_i = 0$)
16:             break loop
17:         else
18:             $[Z_{i-}, Z_{i+}] \leftarrow [Z_{i-}, Z_{i+}] \cap [c_{i-}, c_{i+}]$
19:             $[Z_{j-}, Z_{j+}] \leftarrow [Z_{j-}, Z_{j+}] \cap [c_{j-}, c_{j+}]$
20:         end if
21:     else (row $i$ does not have an off-diagonal entry)
22:         if $Z_{i+} < 0$ then
23:             add split at $x_i = Z_{i+}/2$
24:             break loop
25:         else if $Z_{i-} > 0$ then
26:             add split at $x_i = Z_{i-}/2$
27:             break loop
28:         else
29:             $[Z_{i-}, Z_{i+}] \leftarrow \{0\}$ (point interval)
30:         end if
31:     end if
32: end for
```

We then over-approximate these intervals in each of the dimensions i, j and k by taking the minimum and maximum of the relevant intervals in each dimension (lines 9–11). The resulting intervals $[c_{l-}, c_{l+}]$ (on the variables $l = i, j, k$) are then compared to the current limits for where we can have minimum absolute velocity equal to zero (lines 12–20). If we can say that the previous limits in the box Z do not intersect with the new limits we have calculated, then we can put a split to separate places where zero absolute velocity can occur, removing the infinite time on this box. If we do not break the loop by making a split then we shrink the region Z (lines 21–23) for use in later iterations of i on this box.

We will not prove the termination and abstraction size of this sub-algorithm here, as we can only do so in certain classes of systems. We will look at some of these classes in Section 4.5.

4.4.4 `RemoveNormalInfiniteTimes` sub-algorithm

We include this section for completeness, but we note that this algorithm is identical to `RemoveInfiniteTimes` presented for continuous systems in Section 3.5.2. We will prove the termination and abstraction size of this algorithm for specific types of piecewise-linear systems.

4.5 A timed-automaton abstraction which proves inevitability

We will now prove that Algorithm 4.1, described in the previous section, creates a TA abstraction which is complete (with respect to proving inevitability) for particular sub-classes of the special class described in Section 4.2. In the continuous case, when we proved inevitability we had an intermediate representation of four assumptions, which proved theorems on the TA abstraction. However, the proof of these theorems using the assumptions relied heavily on the continuous nature of the dynamics, and so we cannot transfer them straight across to the piecewise-linear case. Because of this, we will not make use of the intermediate representation for this piecewise-linear case, but will prove the theorems directly.

For ease of reading, we rewrite here the three theorems that we want to prove.

Theorem 4.3. *Assume we have a piecewise-linear n-dimensional system with each subsystem of the form (4.1) in each dimension, and no subsystem boundaries at $x_i = 0$ for any dimension i. Then a timed-automaton abstraction created by the method of Algorithms 4.1–4.4 only has finite traces.*

Theorem 4.4. *Assume we have a piecewise-linear n-dimensional system with each subsystem of the form (4.1) in each dimension, and no subsystem boundaries at $x_i = 0$ for any dimension i. Then a timed-automaton abstraction created by the method of Algorithms 4.1–4.4 has only one location with no outgoing edges, corresponding to the live box L containing the equilibrium point $\overline{x} = 0$.*

Theorem 4.5. *Assume we have a piecewise-linear n-dimensional system with each subsystem of the form (4.1) in each dimension, and no subsystem boundaries at $x_i = 0$ for any dimension i. Then a timed-automaton abstraction created by the method of Algorithms 4.1–4.4 satisfies that all trajectories of the TA get to the live box L in finite time, and so do all trajectories of the original system.*

We will now prove these theorems for the one-dimensional (1-D) and two-dimensional (2-D) sub-classes of piecewise-linear systems defined in Section 4.2, and in Section 4.8 we will analyse the problems which still remain for proving the higher-dimensional versions of these systems. Proving these results in 1-D and 2-D is a novel result, and shows that this method has potential in piecewise-linear systems, and piecewise-continuous systems more generally.

4.5.1 1-D systems

We will firstly consider the case of piecewise-linear systems in 1-D, which will give some insight into why the lower-dimensional systems can be abstracted by this method.

Piecewise-linear 1-D systems of the special form consist of subsystems k with dynamics $\dot{x}_1 = a_k x_1$ for $a_k < 0$, with each subsystem defined in the region $x_1 \in S_k = [s^k_{1-}, s^k_{1+})$.[5] As we only have one variable x_1 in the system, we can only make splitting points on this variable. Hence, to find sliding surfaces we are only interested in the

[5]Note that the vector of variables in this case is a scalar, since $x = [x_1]$.

values of $\dot{x}_1$ at boundaries $x_1 = c \neq 0$ between different subsystems. Choose any two of these adjacent subsystems, say k_1 and k_2 with shared boundary $x_1 = c$, then their derivatives on the boundary are $\dot{x}_1^{k_1} = a_{k_1}c$ and $\dot{x}_1^{k_2} = a_{k_2}c$ respectively. Now both $a_{k_1} < 0$ and $a_{k_2} < 0$, so

$$\dot{x}_1^{k_1} < 0 \text{ and } \dot{x}_1^{k_2} < 0 \qquad \text{if } c > 0, \tag{4.13}$$

$$\dot{x}_1^{k_1} > 0 \text{ and } \dot{x}_1^{k_2} > 0 \qquad \text{if } c < 0. \tag{4.14}$$

Hence, for any particular boundary $x_1 = c$, the derivatives $\dot{x}_1^{k_1}$ and $\dot{x}_1^{k_2}$ always have the same sign, and so *no sliding surfaces can occur in 1-D systems.* We should also note that this reasoning applies to the n-th dimension of any n-D system of this form, since the dynamics of x_n is only dependent on itself in all subsystems, with exactly the form of the 1-D systems.

Termination and abstraction size for 1-D systems

We will now show termination of Algorithm 4.1 and its sub-algorithms for the 1-D sub-class of the systems of form (4.1) with no subsystem boundaries at $x_i = 0$ for any dimension i, and we will quantify the resulting abstraction size. The initial splitting is made in lines 1–5, and takes the unique subset of the values $l_{1-}, l_{1+}, s_{1-}, s_{1+}, s_{1-}^k, s_{1+}^k$ for $k = 1 \dots, m$. As we know this does not have any sliding modes, we know the worst case abstraction size after this initial splitting is $|C_1| = m + 3$ and $|N_1| = m + 1$ by equations (4.8) and (4.9).

Algorithm 4.2 (`FollowSplitsPWL`) is called next, at line 6 of Alg. 4.1. As we only have one dimension (which is effectively dimension n), the outer for loop is not run and we go immediately to the last line of `FollowSplitsPWL` that removes all the entries in N_1. Hence our splitting after this sub-algorithm is that same as it was before it, and `FollowSplitsPWL` obviously terminates.

We then get to Step 3 of Alg. 4.1, which starts by calculating the box times for the boxes we have created, and then finds the list of boxes which have infinite time, not including L. For a 1-D system of this form, infinite time can occur in any box where the line $\dot{x}_1 = 0$ can exist, but as $\dot{x}_1 = a_{1,1}x_1$ with $a_{1,1} < 0$ the zero surface only exists when $x_1 = 0$, which is defined to be only in the live box L. Hence there are no infinite-time boxes apart from L, and so the list B is empty. Therefore the while loop is not run at all, and the algorithm terminates with the same splitting that we started with after the initial splitting, which is of size $|C_1| = m + 3$.

Proving that the TA abstraction can show inevitability in 1-D systems

To show that this method creates a TA abstraction which can prove inevitability, we need to prove Theorems 4.3–4.5. Firstly, we show Theorem 4.3, that the automaton abstraction of the piecewise-linear system only has finite traces.

Proof of Theorem 4.3 for 1-D systems. If we assume that there is an infinite length trace in a 1-D piecewise-linear system of this form, then as we have a finite number of TA states, there must be at least one box b which is passed through an infinite number of times. As we discussed in the continuous case, this necessitates it is possible to take a transition in (say) the positive direction out of the box b across the boundary $x_1 = c$, and that at some later point in the infinite trace it is possible to take a transition in the negative direction across $x_1 = c$.

The dynamics in each subsystem are $\dot{x}_1 = a^k_{1,1}x_1$ for $a_{1,1} < 0$, and so when $x_1 > 0$ we have $\dot{x}^k_1 < 0$ for every subsystem k, and similarly when $x_1 < 0$ we have $\dot{x}^k_1 > 0$ for every k. Hence, when we cross $x_1 = c$, if $c > 0$ then $\dot{x}_1 < 0$ for every k and a transition is only possible in the negative direction for the whole of time, and if $c < 0$ then $\dot{x}_1 > 0$ and a transition is only possible in the positive direction for all of time. This means it is not possible to reverse a transition made in dimension 1 (the only dimension we have), there are no infinite length traces, and all traces of the automaton abstraction are finite. □

Now we will show Theorem 4.4, that the only location with no outgoing edges the automaton abstraction corresponds to the box L containing the equilibrium point $x_1 = 0$.

Proof of Theorem 4.4 for 1-D systems. As we have shown in the previous proof, the dynamics of all the subsystems go in the same directions across the whole space, which is positive when x_1 is negative, and negative when x_1 is positive. Hence, for any box which does not contain $x_1 = 0$ the two boundaries of this box have the same sign, the flow across them goes in the same direction, and so one of these edges has flow into the box and one has flow out of the box. For the box L which contains $x_1 = 0$, the lower boundary is at negative x_1 and so has positive (inwards) flow, and the upper boundary is at positive x_1 and so has negative (inwards) flow. Hence the only location of the automaton abstraction with no outgoing edges corresponds to the box L around $x_1 = 0$. $\square$

We now prove the main result, that the Algorithms 4.1–4.4 create a TA abstraction whose trajectories all get to the live box L in finite time, and so do the trajectories of the original system.

Proof of Theorem 4.5 for 1-D systems. The automaton abstraction of the TA only has finite traces by Theorem 4.3 and so must reach a state where no further transitions can be taken, which by Theorem 4.4 must be the state corresponding to the live box L. As we have proved termination of the algorithms in the 1-D case (Sec. 4.5.1), we know that all infinite-time boxes must have been removed to make the while loop of Algorithm 4.1 terminate, and so all boxes have finite time. Hence, putting together the finite trace with a finite time spent in each state on the TA trace, we see that the box L is reached in finite time from anywhere in the TA. Hence we have proved that all trajectories of the original system get to the live box L in finite time, since the TA over-approximates the number of trajectories in the system. $\square$

4.5.2 2-D systems

We will now consider the 2-D systems of the special form, where the variable vector is $x = [x_1, x_2]^T$. Each subsystem k can have dynamics with one of two forms,

$$\dot{x} = \begin{bmatrix} a^k_{1,1} & 0 \\ 0 & a^k_{2,2} \end{bmatrix} x, \qquad \text{or} \quad \dot{x} = \begin{bmatrix} a^k_{1,1} & a^k_{1,2} \\ 0 & a^k_{2,2} \end{bmatrix} x, \tag{4.15}$$

with $a^k_{1,1} < 0$, $a^k_{2,2} < 0$, and $a^k_{1,2} \in \mathbb{R} \setminus \{0\}$. We again assume each subsystem has rectangular state space S_k, and that these state spaces form a partition of a larger rectangular system-wide state space S.

As discussed for the 1-D case, the 2nd dimension in these 2-D upper-triangular systems cannot have sliding surfaces occurring, and so they can only occur along lines of constant x_1 in the system. Considering a stable sliding surface at $x_1 = c_1$ and $x_2 \in [c_{2-}, c_{2+})$ between subsystems k_1 (below) and k_2 (above): this sliding surface has

$\dot{x}_1^{k_1} > 0$ and $\dot{x}_1^{k_2} < 0$ along it. We know that the sliding mode dynamics along the surface are $\dot{x}_1 = 0$, with $\dot{x}_2$ having a value between the values of $\dot{x}_2^{k_1} = a_{2,2}^{k_1} x_2$ and $\dot{x}_2^{k_2} = a_{2,2}^{k_2} x_2$. Whilst within this surface, the only transitions between boxes that can be taken are in the x_2 direction, and so we must consider the value of $\dot{x}_2$ at a boundary point $x_2 = c \in [c_{2-}, c_{2+})$, which gives us $\dot{x}_2$ between $a_{2,2}^{k_1} c$ and $a_{2,2}^{k_2} c$ on the box facet $x_2 = c$. In the same way as for the 1-D case, these values are either both positive or both negative, and so *every transition between boxes existing along a stable sliding surface can only occur in one direction.*

In a similar way we can consider the case of an unstable sliding surface $x_1 = c_1$, where (if k_1 is below and k_2 above) the derivatives have the values $\dot{x}_1^{k_1} < 0$ and $\dot{x}_1^{k_2} > 0$ on this surface. In this case all transitions out of the sliding surface in the positive and negative x_1 directions are allowed to occur, and no transitions inwards in the x_1 directions are allowed. Considering the x_2 direction transitions, the process is the same as that for the stable sliding mode, and so *every transition between boxes existing along the unstable sliding surface can only occur in one direction.*

We will now prove termination and abstraction size of Algorithm 4.1 applied to this class of systems, and then show that the abstraction satisfies the three theorems.

Termination and abstraction size in 2-D systems

To show termination of Algorithm 4.1 and its sub-algorithms we start by considering the initial splitting, which takes the unique subset of the values $l_{i-}, l_{i+}, s_{i-}, s_{i+}, s_{i-}^k, s_{i+}^k$ and then adds an extra value in dimension 1 for every possible location of a sliding mode. As quantified in equations (4.8) and (4.9), the maximum sizes of the C and N sets after the initial splitting are given by the following:

$$|C_1| = 2m + 2, \qquad |N_1| = m + 1, \tag{4.16}$$

$$|C_2| = m + 3, \qquad |N_2| = m + 1, \tag{4.17}$$

where m is the number of subsystems present in the system. As this is a finite number of assignments, Step 1 of Alg. 4.1 terminates.

Algorithm 4.2 is called next to follow the new splits given in the set N. As discussed in the 1-D case, the splits in the last (2nd) dimension are not followed by the `FollowSplitsPWL` algorithm, but the ones in the 1st dimension induce splits in the 2nd dimension. Using the quantification in Section 4.4.3 we know that this sub-algorithm terminates, with maximum sizes afterwards given by

$$|C_1| = 2m + 2, \qquad |N_1| = 0, \tag{4.18}$$

$$|C_2| = m + 3 + m * (m + 1) = m^2 + 2m + 3, \qquad |N_2| = 0. \tag{4.19}$$

The algorithm then enters Step 3 of Alg. 4.1 which firstly finds the list of infinite-time boxes — as we are in 2-D there can be a maximum of $2m$ boxes with infinite time, by equation (4.10). We can also show an extra condition in the 2-D case.

Proposition 4.6. *There are no infinite-time boxes on the stable or unstable sliding surfaces in 2-D systems.*

Proof. We discussed above that all sliding surfaces in 2-D systems of this form are in the 1st dimension, so that a box in a sliding surface has limits of the form $[b_1, b_1] \times [b_{2-}, b_{2+})$. Let us assume for a contradiction that there is a box on such a sliding surface which has infinite time. Then both of the equations $\dot{x}_1 = 0$ and $\dot{x}_2 = 0$ must be able to be satisfied in the box, and in fact the equation $\dot{x}_1 = 0$ is true at all points on a sliding surface. However, we know that $\dot{x}_2^k = a_{2,2}^k x_2$ for each subsystem $k = 1, \ldots, m$, and as the dynamics on the sliding surface are a convex combination of the subsystems either side, we have

$$\dot{x}_2^s = \left(c^{k_1} a_{2,2}^{k_1} + c^{k_2} a_{2,2}^{k_2}\right) x_2,$$

with $c^{k_1}, c^{k_2} \geq 0$ and $c^{k_1} + c^{k_2} = 1$. Now, $a_{2,2}^k < 0$ for all k and so $c^{k_1} a_{2,2}^{k_1} + c^{k_2} a_{2,2}^{k_2} < 0$. Hence, if $\dot{x}_2 = 0$ on the sliding surface, then this implies that $x_2 = 0$ must exist in the sliding surface, and so we must have $0 \in [b_{2-}, b_{2+})$. At the point $x_1 = b_1$ and $x_2 = 0$ which must be in the box, $\dot{x}_1^{k_1} = a_{1,1}^{k_1} b_1$ and $\dot{x}_1^{k_2} = a_{1,1}^{k_2} b_1$. However as $a_{1,1}^{k_1} < 0$ and $a_{1,1}^{k_2} < 0$ these two derivatives have the same sign, which contradicts the assumption that this point is in the sliding surface. Hence, in the 2-D systems of the form of Section 4.2, there is no box on a sliding surface that has infinite time. □

This result restricts the possible number of infinite-time boxes given in (4.10) to only $m + 1$, where m is the number of subsystems. It also means that Step 3 of Algorithm 4.1 will never call the sub-algorithm `RemoveSlidingInfiniteTimes` (Alg. 4.3), as there are no infinite times on sliding surfaces. This helps with the termination proof, as we only need to prove termination of `RemoveNormalInfiniteTimes` (Alg. 4.4).

In the continuous case, we showed that termination of the removal of infinite times depended on how far offset the boxes were, and in particular we derived that all infinite-time boxes with offset in dimension $n-1$ are always separable in dimension n on the first run through the while loop (see Proposition 3.16.2). When we are only working in 2-D this means that (in the continuous case) every box with infinite time is immediately separable in dimension 2. The piecewise-linear case extends this directly to say that every infinite-time box in every subsystem is immediately separable on the first run through the while loop, as the proof does not require any properties of where in the system this box occurs but only that it is not a box which forms part of the boundary of a subsystem. Hence every infinite-time box in the initial TA abstraction of a piecewise-linear system of the special form is separable on the first run through the while loop of Algorithm 4.1. Therefore one run suffices to remove these infinite-time boxes, so the while loop terminates, and thus Step 3 of Alg. 4.1 terminates. Step 3 of this algorithm results in at most one split in dimension 2 being made for each infinite-time box, giving a splitting of maximum size (before `FollowSplitsPWL` in line 19) of

$$|C_1| = 2m+2, \qquad |N_1| = 0, \tag{4.20}$$

$$|C_2| = m^2+2m+3+m+1 = m^2+3m+4, \qquad |N_2| = m+1. \tag{4.21}$$

When `FollowSplitsPWL` is then called, all the new splits in N are only in dimension 2 (the n-th dimension) and so do not induce any new splits, and so Algorithm 4.1 terminates with a final splitting of maximum size

$$|C_1| = 2m+2, \tag{4.22}$$

$$|C_2| = m^2+3m+4. \tag{4.23}$$

Proving that the TA abstraction can show inevitability in 2-D systems

To show that this method creates a TA abstraction which proves inevitability of 2-D piecewise-linear systems of the form of (4.1), we need to prove the three theorems.

Firstly, we prove Theorem 4.3, that the automaton abstraction of the system only has finite traces.

Proof of Theorem 4.3 for 2-D systems. Assume there is an infinite length trace in the automaton abstraction. Then for every transition that is made in the positive i direction in this trace there must be a corresponding transition made in the negative i direction. As shown in the 1-D piecewise-linear case the n-th dimension of the system has transitions in one direction only for each split value $x_n = c$, and so cannot be involved in an infinite trace. So in 2-D the infinite length trace must only involve infinite loops of transitions in dimension 1.

As x_2 is fixed within one slice for this infinite trace, the only way a transition could be reversed is if there are a pair of boxes in this slice which have transitions going between them in both directions in dimension 1. There are five cases we need to consider to prove that this pair cannot occur.

Case 1. The two boxes are within the same subsystem. Then by continuity of the dynamics and because `FollowSplitsPWL` has completed we deduce that flow can only go in one direction across the boundary between the boxes, so there cannot be transitions going in both directions across this boundary.

Case 2. The two boxes are in different subsystems. In this case we assume there is a boundary between the boxes which is not 'doubled' in the splitting, that is there is no sliding surface on this boundary. As there is no sliding surface on the boundary, this presupposes that the dynamics across the shared boundary go in the same direction, so there cannot be transitions going in both directions across this boundary.

Case 3. One box is in a stable sliding surface, the other next to the surface. Let the stable sliding surface exist at $x_1 = c$, then the x_1 derivatives can only flow inwards to the sliding surface box, and so this stops a transition into the sliding surface being reversed. Hence there cannot be transitions going in both directions across a boundary.

Case 4. One box is in an unstable sliding surface, the other next to the surface. This is an analogous proof to the stable sliding surface, except that flow can only leave the sliding surface (rather than only enter). Hence there cannot be transitions going in both directions across a boundary.

Case 5. One box is in a transversal sliding surface, the other next to the surface. A transversal sliding surface is a boundary region where the two dynamics either side are flowing in the same direction, and would be caused in the abstraction by a stable/unstable sliding surface being present somewhere else on this surface. The fact that the flow only goes one-way by definition again stops any transition from being reversed as is necessary for an infinite trace to occur.

Hence there is no possible pairing of boxes for which an infinite trace of the automaton abstraction can be found, and so all traces of the automaton abstraction of the 2-D piecewise-linear systems of the special form are finite. □

Next we must prove Theorem 4.4, which says that the only location with no outgoing edges corresponds to the box L containing the point $x = 0$.

Proof of Theorem 4.4 in 2-D systems. Firstly note that a location with no outgoing edges must have infinite time, as it will never be left, and the box time assigned is an over-approximation of the time taken. However, we have proven termination of this algorithm, which requires that all of the infinite-time boxes (except L) have been removed, so a location with no outgoing edges can only correspond to box L. □

Proof of Theorem 4.5 in 2-D systems. To prove the main result, that Algorithms 4.1–4.4 create a TA abstraction whose trajectories all get to the live box L in finite time we can use the same proof as for 1-D systems, simply replacing "1-D" with "2-D". □

Hence, we have now proved that the proposed these algorithms work to create a TA abstraction that proves inevitability for 1-D and 2-D systems. In Section 4.8 we will move on to consider the problems which arise when we try to prove the same properties about 3-D and higher-dimensional systems.

4.6 Implementation details

The algorithms of Section 4.4 have been implemented in MATLAB, using the useful mathematical functions it contains. This implementation is called the `PWproveByTA` toolbox, and is available online at

`http://staff.cs.manchester.ac.uk/~navarroe/dyverse/liveness`

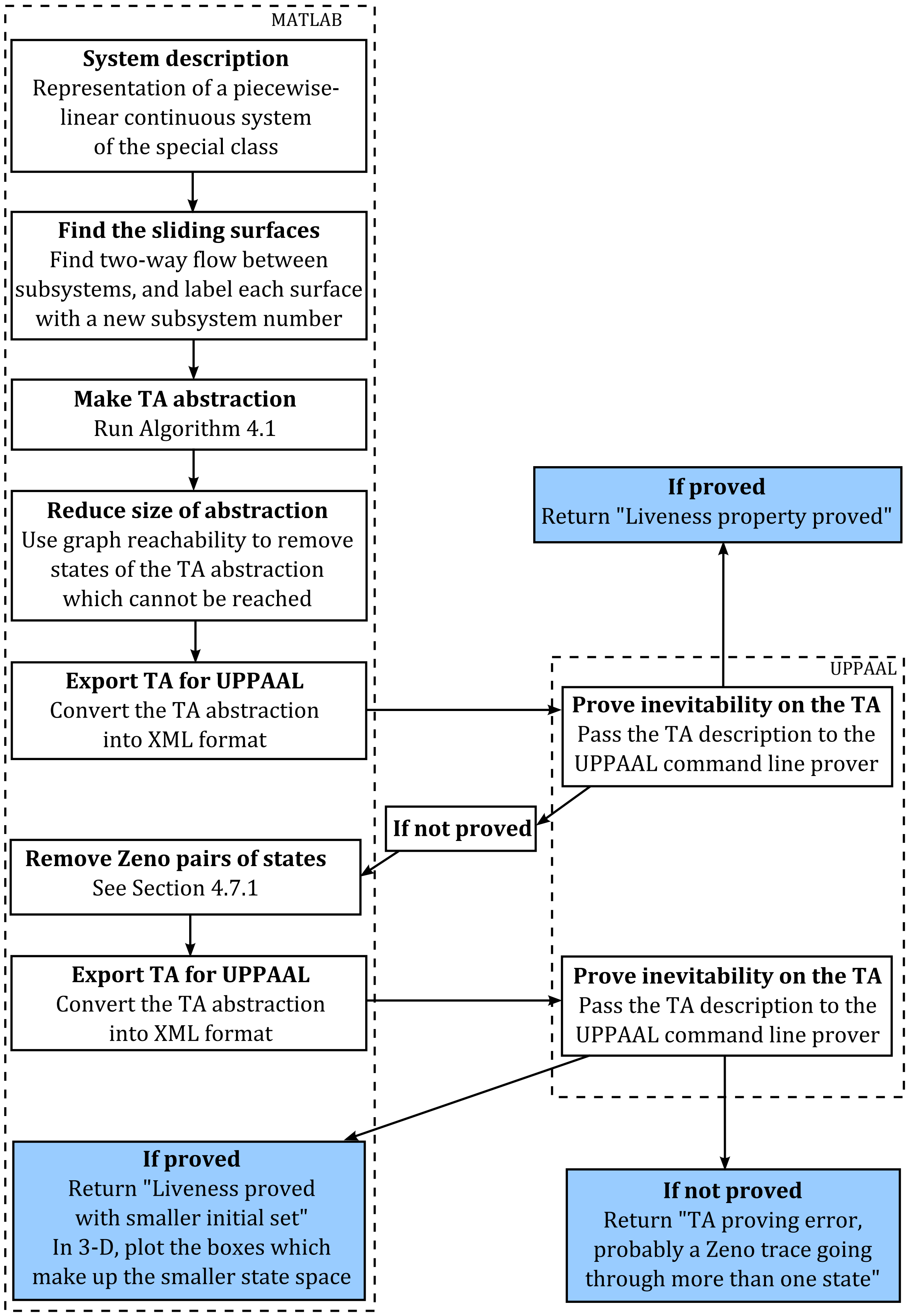

MATLAB
System description
Representation of a piecewise-linear continuous system of the special class
Find the sliding surfaces
Find two-way flow between subsystems, and label each surface with a new subsystem number
Make TA abstraction
Run Algorithm 4.1
Reduce size of abstraction
Use graph reachability to remove states of the TA abstraction which cannot be reached
Export TA for UPPAAL
Convert the TA abstraction into XML format
UPPAAL
Prove inevitability on the TA
Pass the TA description to the UPPAAL command line prover
If proved
Return "Liveness property proved"
If not proved
Remove Zeno pairs of states
See Section 4.7.1
Export TA for UPPAAL
Convert the TA abstraction into XML format
Prove inevitability on the TA
Pass the TA description to the UPPAAL command line prover
If proved
Return "Liveness proved with smaller initial set"
In 3-D, plot the boxes which make up the smaller state space
If not proved
Return "TA proving error, probably a Zeno trace going through more than one state"

Figure 4.4: An overview of the method for the `PWprovebyTA` toolbox. Output states are shaded.

An overview of the flow of the method is given in Figure 4.4. Most of the first part of this method is implemented in a very similar way to the continuous case, so refer to the discussion in Section 3.6.1 for implementation details for these. However, there are a couple of parts unique to this implementation which we should pick out.

4.6.1 Finding sliding surfaces

The first task that the `PWproveByTA` implementation does after being given the system description is to consider every boundary between subsystems, and work out whether a sliding surface exists anywhere on it. That is, for each pair of subsystems, we first work out whether they share a boundary, and if they do then we calculate where they have opposite derivatives on this boundary. This is not too difficult to implement as the dynamics are linear and of the form of (4.1), so that each set of dynamics can only change sign on one straight line through the surface.

If we find that a sliding surface occurs on a boundary which forms part of $x_i = c$, say, then we add this value c to a list of points which we must duplicate in the splitting in dimension i. We also add a new subsystem number to the system representation, and specify the dynamics on this new subsystem with the two sets of dynamics from the subsystems above and below. There is an assumption built into the implementation that if a subsystem has two sets of dynamics defined then these are the limiting values for the actual dynamics in this subsystem, which is what we want on sliding surfaces.

4.6.2 Removing Zeno pairs of states

This method is a response to a problem which will be discussed more in Section 4.8.1. The idea of the method is that it removes pairs of discrete states which still have two-way flow between them after the algorithm first terminates: this is what we consider to be a Zeno pair of states, as the automaton could get stuck in this pair of states just transitioning between them forever without progressing time. The method to remove these Zeno pairs also removes all discrete states which could lead to these Zeno pairs, by performing a backwards reachability on the discrete states of the automaton. This leaves us with the part of the state space which never leads to a Zeno pair.

After these Zeno pairs and their predecessors are removed, the TA abstraction is again converted to XML format for input to UPPAAL, and we ask UPPAAL to prove the inevitability property on this TA. If it succeeds in proving the property, and the system is 3-D, then we use MATLAB to plot the boxes which make up the part of the original state space which has been proved to be inevitable. We will show this method in an example in Section 4.8.1.

4.7 Example: 2-D system with four subsystems

We will now look at an example of a 2-D piecewise-linear system of the special class, to demonstrate the method and its output. The system we consider is

$$\begin{aligned}
\dot{x} = A_1 x &= \begin{bmatrix} -1 & -1 \\ 0 & -1 \end{bmatrix} x, \quad S_1 = [-5,5) \times [-5,-1), \\
\dot{x} = A_2 x &= \begin{bmatrix} -1 & 1 \\ 0 & -1 \end{bmatrix} x, \quad S_2 = [1,5) \times [-1,5), \\
\dot{x} = A_3 x &= \begin{bmatrix} -1 & -3 \\ 0 & -1 \end{bmatrix} x, \quad S_3 = [-1,1) \times [-1,5), \\
\dot{x} = A_4 x &= \begin{bmatrix} -1 & 2 \\ 0 & -1 \end{bmatrix} x, \quad S_4 = [-5,-1) \times [-1,5),
\end{aligned} \tag{4.24}$$

where the global state space is $S = [-5,5) \times [-5,5)$, the initial region is defined as $\mathit{Init} = [-5,-4) \times [-5,5)$, and the initial live box is $L = [-2,1) \times [-1.5,1.5)$.

When we run the `PWproveByTA` implementation on this example, the initial splitting created by step 1 of Alg. 4.1 is

$$\begin{aligned}
C_1 &= \{-5,-2,-1,-1,1,1,5\}, & N_1 &= \{-2,-1,-1,1,1\}, \\
C_2 &= \{-5,-1.5,-1,1.5,5\}, & N_2 &= \{-1.5,-1,1,1.5\}.
\end{aligned}$$

Due to the repeated values in C_1 we can deduce that there are sliding surfaces along at least part of $x_1 = -1$ and $x_1 = 1$.

Next the implementation calls the `FollowSplitsPWL` algorithm, and ensures we only have one-way flow across boundaries, giving a splitting state of

$$\begin{aligned}
C_1 &= \{-5,-2,-1,-1,1,1,5\}, \\
C_2 &= \{-5,-2,-1.5,-1,-0.5,-0.3333,0.3333,0.5,0.6667,1,1.5,2,5\}.
\end{aligned}$$

The `PWproveByTA` toolbox then performs a state reachability analysis, to remove those boxes which are not reachable from the initial set *Init*. Before this analysis there are 72 boxes, with 18 removed from consideration by the reachability, leaving only 54 boxes to consider for removal of infinite-time boxes.

Of these 54 boxes, only two have got infinite box time, excluding the live box L. The first has the limits $[-1,-1] \times [-0.3333, 0.3333]$, which is part of the surface $x_1 = -1$, but is not the part which has sliding dynamics. Algorithm 4.4 is called with this box as target and the set where $\dot{x}_1 = 0$ surfaces can occur is initialised to the limits of the box, $Z = [-1,-1] \times [-0.3333, 0.3333]$. On the first iteration of the i loop in Alg. 4.4, we have $i = 2$ where the dynamics are diagonal, so we go straight to line 21. We do not satisfy the if or elseif statements on lines 22 and 25, so we go to line 28 and set $[Z_{2-}, Z_{2+}] = [0, 0]$. So $Z = [-1,-1] \times [0, 0]$. The second iteration of the i loop has off-diagonal dynamics, and so the if statement on line 2 is true. We get values of $[c_{2-}, c_{2+}] = [0.3333, 0.3333]$ on line 4, which then makes the if statement on line 5 true. Then on line 6, we make a split at $x_2 = (0.3333 + 0)/2 = 0.1667$.

The second infinite-time box, with limits $[1, 1] \times [-0.3333, 0.3333]$, is now considered. In a very similar fashion to the previous box, we make a split at $x_2 = -0.1667$.

Control is then returned to Alg. 4.1 having removed the two infinite-time boxes. `FollowSplitsPWL` is then called at line 19 of Alg. 4.1, with two new splits to follow, as $N_1 = \emptyset$ and $N_2 = \{0.1667, -0.1667\}$. As these new splits are in dimension 2 already, `FollowSplitsPWL` simply sets $N_2 = \emptyset$ and returns.

Hence, the final splitting of the system is given by

$$C_1 = \{-5, -2, -1, -1, 1, 1, 5\},$$
$$C_2 = \{-5, -2, -1.5, -1, -0.5, -0.3333, -0.1667, 0.1667, 0.3333, 0.5, 0.6667, 1, 1.5, 2, 5\}.$$

The directions of flow between boxes created by this splitting is then calculated, and discrete reachability is performed once again to remove any unreachable states added in by the livebox splitting. We end up with 64 boxes to form the TA abstraction states, and we calculate the maximum time that can be spent in a box. The XML file for input to UPPAAL is created to represent the abstraction using the `createXMLFile` method from the `proveByTA` toolbox, which rewrites the information into an XML file for input to UPPAAL.

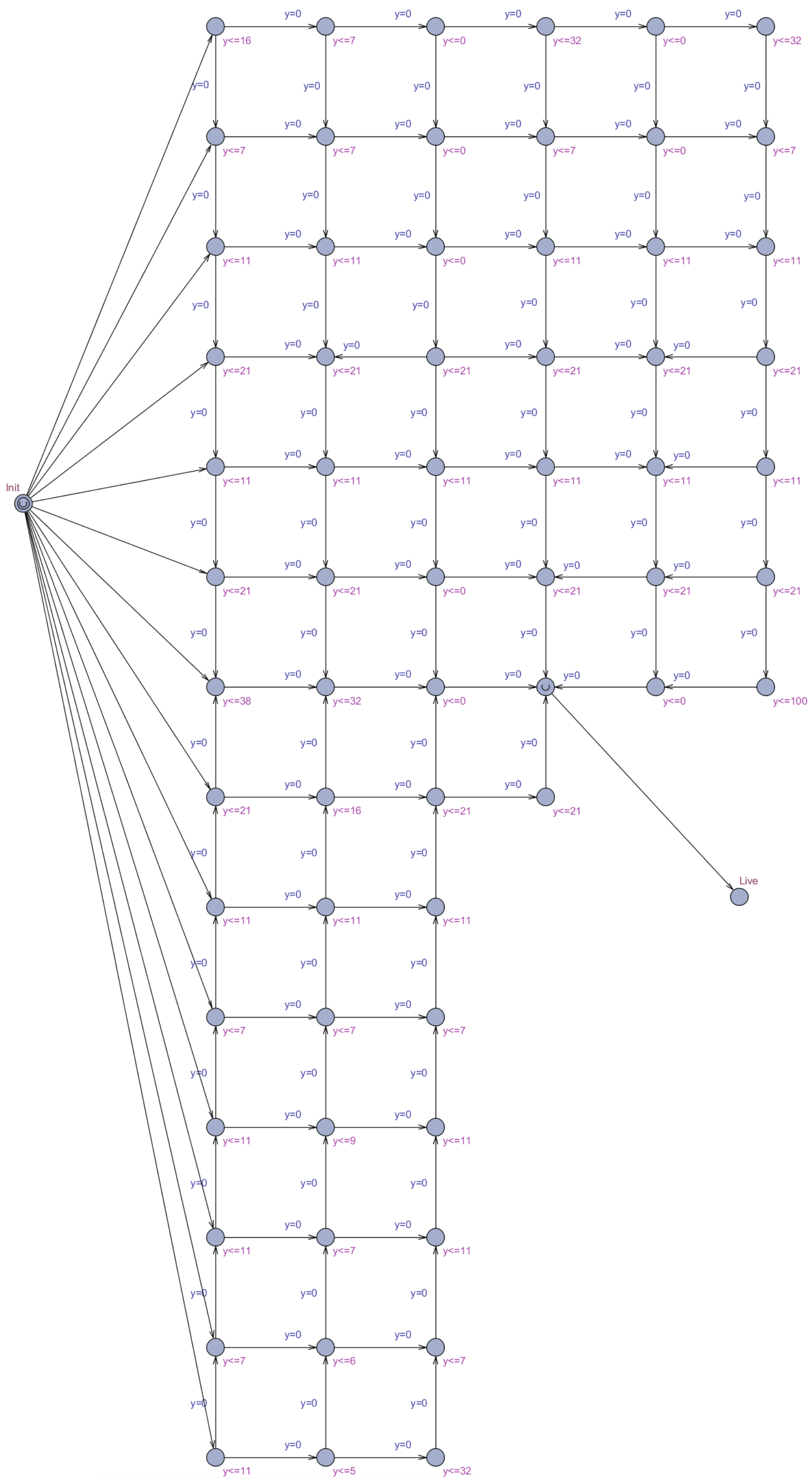


Figure 4.5: The timed-automaton abstraction of the 2-D piecewise-linear example system of (4.24), viewed in UPPAAL's graphical editor. We just use the box clock in this model, and call it y. The "$y = 0$" statements are resets of the box clock on a transition, and the statements of the form "$y < 11$" are clock invariants on each state of the automaton. States containing a '$\cup$' are *urgent* states, where no time can pass.

The UPPAAL stand-alone prover `veriftyta` is called, to prove the inevitability property of reaching the live set $L = [-1, 1) \times [-0.1667, 0.1667)$. UPPAAL proves this property and returns to MATLAB, and the `PWproveByTA` toolbox then returns `Liveness property proved!` to the user, showing that the example of (4.24) satisfies the inevitability property.

The resulting TA which UPPAAL sees can be viewed in UPPAAL's graphical editor, and it is shown in Figure 4.5.

4.8 Analysis and potential solutions for 3-D and higher-dimensional systems

We will now consider what happens when the algorithms we have presented are used for higher-dimensional systems. We will illustrate each issue with a specific example of a 3-D system which will help to analyse the problem we are considering, and can also help to suggest the potential ways in which this problem can be addressed in the TA abstraction, to get some useful results from the abstraction. Firstly, we show that two-way flow across facets can be a problem which is not addressed by the algorithms we have proposed in 3-D systems, but also show that we find the subset of the initial set which does not lead to this two-way flow in the TA abstraction.

4.8.1 Zeno behaviour in pairs of states

When we consider the piecewise-linear systems of the class in Section 4.2 in 3-D or higher, there is now the possibility that the `FollowSplitsPWL` algorithm will not be able to separate some pairs of boxes where the transitions between them go in both directions. In particular, some pairs of boxes within a sliding mode boundary will have flow in both directions which cannot be separated, due to the use of the derivatives either side to over-approximate and under-approximate the value of the derivative across a boundary. This two-way flow leads to pairs of TA states causing infinite length runs with no guarantee of time progressing, which is Zeno behaviour. Formally, the definition of Zeno behaviour is that an infinite number of discrete transitions occur in a finite length of time.

Let us consider this problem more formally. Let a stable or unstable sliding surface be at $x_i = c_i$ $(i < n)$ between two subsystems k_1 and k_2 with dynamics $\dot{x} = A_{k_1}x$ and $\dot{x} = A_{k_2}x$ respectively. Let us consider two neighbouring boxes defined within this sliding surface, b_1 and b_2. Let the boundary between these two boxes be at $x_j = c_j$, so that the limits of the boxes are identical apart from the limit of x_j. The allowed transitions between the two boxes are defined by the signs of $\dot{x}_j^{k_1}$ and $\dot{x}_j^{k_2}$ in the boundary facet $F = \mathrm{cl}(b_1) \cap \mathrm{cl}(b_2)$ which forms part of the surface defined by $x_j = c_j$. If there is some point $x \in F$ for which $\mathrm{sgn}(\dot{x}_j^{k_1}) = -\mathrm{sgn}(\dot{x}_j^{k_2})$ then our algorithm obtains a negative lower limit and a positive upper limit for the value of $\dot{x}_j$ at this point on the boundary $x_j = c_j$ between the boxes b_1 and b_2. Hence there is no way of separating the positive and negative values of the derivatives using `FollowSplitsPWL`. This means that the TA abstraction allows an infinite length discrete trace going between the pair of states b_1 and b_2 forever.

An example which demonstrates this problem is the 3-D system given below.

$$\dot{x} = A_1 x = \begin{bmatrix} -1 & -1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & -1 \end{bmatrix} x, \qquad S_1 = [-5, 5) \times [-5, 5) \times [-5, -1),$$

$$\dot{x} = A_2 x = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & -2 \end{bmatrix} x, \qquad S_2 = [-5, 5) \times [-5, -1) \times [-1, 5),$$

$$\dot{x} = A_3 x = \begin{bmatrix} -1 & 0 & -3 \\ 0 & -1 & 4 \\ 0 & 0 & -1 \end{bmatrix} x, \qquad S_3 = [-5, 1) \times [-1, 5) \times [-1, 5),$$

$$\dot{x} = A_4 x = \begin{bmatrix} -1 & 2 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & -3 \end{bmatrix} x, \qquad S_4 = [1, 5) \times [-1, 5) \times [-1, 5),$$

where the global state space is $S = [-5, 5) \times [-5, 5) \times [-5, 5)$, the initial region is defined as *Init* $= S$ (the whole space), and the initial live box is $L = [-2, 1) \times [-1.5, 1.5) \times [-1.5, 1.5)$.

When the implementation of the abstraction algorithms is applied to this case it terminates (implying that the infinite-time boxes are removed), with an abstraction

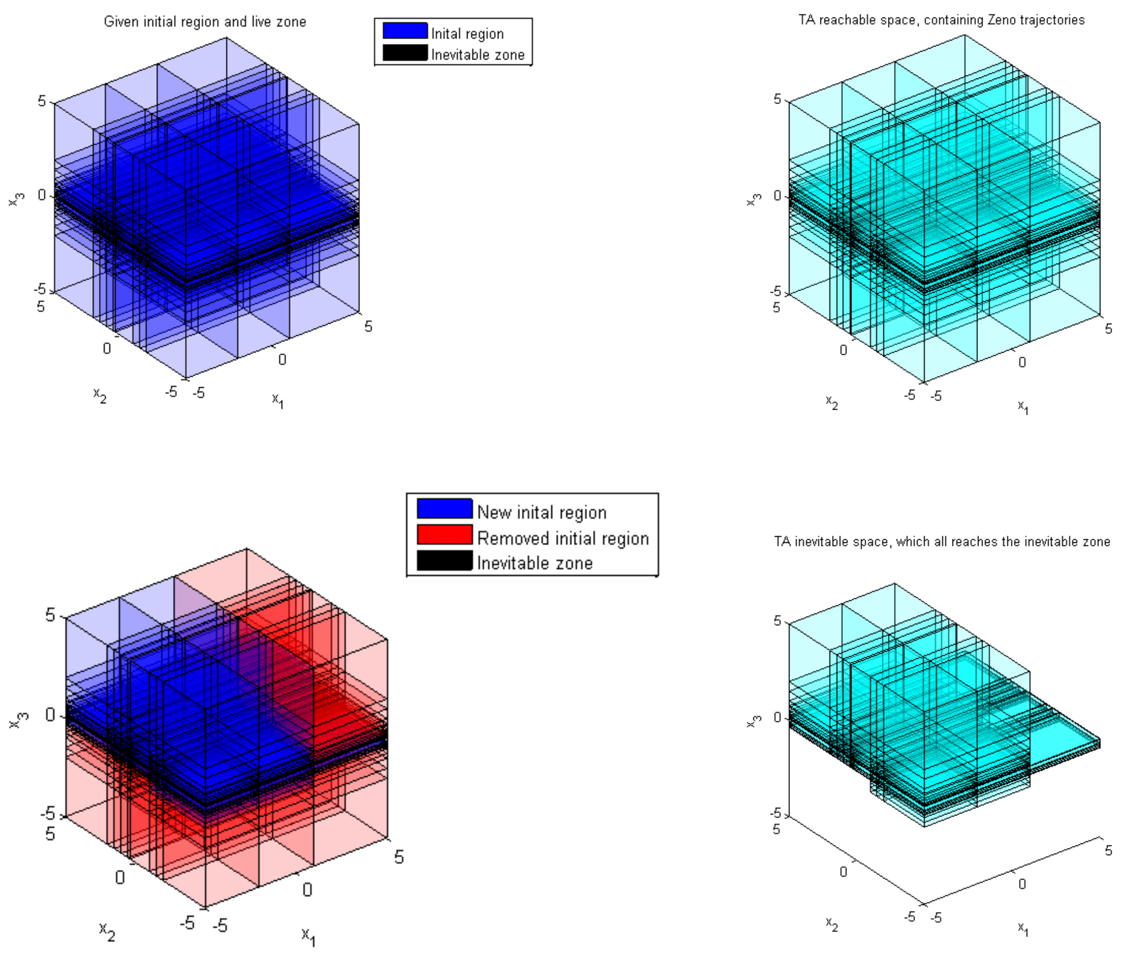


Figure 4.6: The MATLAB diagrams showing the revised initial and reachable sets from which we can prove inevitability of the live set L. Here the "inevitable zone" is the live box L, which is in the middle of these diagrams so not visible.

of size $|C_1| = 5$, $|C_2| = 11$, and $|C_3| = 24$. When the resulting TA is tested for inevitability using UPPAAL a counterexample trace is found that does not reach the live box L. Looking at the TA we discover that there are 35 pairs of transitions which go in opposite directions between a pair of states: these transitions involve 57 states out of the 920 states in the system.

As we discussed above, the proposed algorithms will always lead to this Zeno pairs problem in examples with both positive and negative values at a single point on a sliding surface, and this two-way flow cannot be removed with the `FollowSplitsPWL` algorithm. There is not an obvious way of solving this problem in the current framework, so instead we wish to consider what we can do to give a useful result to the user from the implementation of the algorithms in such cases.

We advocate finding a smaller initial set from which these Zeno pairs of states cannot be reached, meaning that all of this smaller initial set will reach the live box L. The resulting TA on this smaller state space can then be proved to be inevitable, meaning that the original system on this smaller space is inevitable. This removal of Zeno states has been implemented as part of the MATLAB implementation, so that a new state space is found which we can prove reaches the live box L in the TA abstraction. The new initial set and reachable space are not necessarily rectangular after this process, so for 3-D systems the implementation returns 3-D plots of the boxes which make up the revised initial set and the reachable space from which inevitability can be proved. The drawings returned for the example discussed above are shown in Figure 4.6.

4.8.2 Zeno behaviour in cycles of states

The second problem that we see in higher-dimensional piecewise-linear systems is being able to find a group of states in the automaton abstraction that can be traversed in a cycle with no guarantee of progress of time. This type of behaviour in an abstraction is generally caused by subsystems interacting in a certain way. The best way to explain this is to look at a 3-D example:

$$\dot{x} = A_1 x = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & -1 \end{bmatrix} x, \qquad S_1 = [-5, 2) \times [2, 5) \times [-5, 5),$$

$$\dot{x} = A_2 x = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & -1 \end{bmatrix} x, \qquad S_2 = [2, 5) \times [2, 5) \times [-5, 5),$$

$$\dot{x} = A_3 x = \begin{bmatrix} -1 & -1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & -1 \end{bmatrix} x, \qquad S_3 = [2, 5) \times [-5, 2) \times [-5, 5),$$

$$\dot{x} = A_4 x = \begin{bmatrix} -1 & -1 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & -1 \end{bmatrix} x, \qquad S_4 = [-5, 2) \times [-5, 2) \times [-5, 5),$$

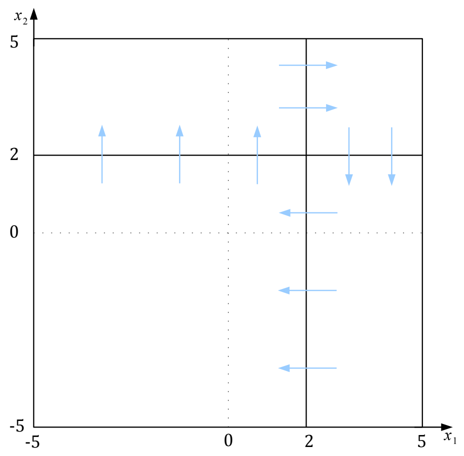


Figure 4.7: Cyclical behaviour: The flow around the system in the slice $x_3 \in [1, 2)$.

where the global state space is $S = [-5, 5) \times [-5, 5) \times [-5, 5)$, the initial region is defined as $Init = S$ (the whole space), and the initial live box is $L = [-2, 1) \times [-1.5, 1.5) \times [-1.5, 1.5)$.

This example has been designed so that there is a cycle of length 4 around the joins between the systems, in particular when we are in the slice $x_3 \in [1, 2)$ the four boxes around the point $(x_1, x_2) = (2, 2)$ will have cyclic behaviour, as demonstrated by Figure 4.7 which shows the x_1 and x_2 flow in this slice $x_3 \in [1, 2)$.

When we attempt to prove an example like this with the MATLAB implementation, Algorithms 4.1–4.4 are completed and then the resulting TA is passed to UPPAAL to attempt to verify the inevitability property. UPPAAL returns an answer saying that the property is not satisfied, and then the MATLAB code removes any states that reach a Zeno pair in the automaton abstraction (as discussed in the previous section), and passes the resulting automaton to UPPAAL for proving again. However, UPPAAL again returns that the inevitability property is not satisfied, and so the implementation returns the message `TA proving error, probably a Zeno trace going through more than 2 states`.

For future work to deal with these cycles better, we could implement a method to remove these Zeno cycles and the states that lead to them, in a similar way to removing the Zeno pairs in the previous section. To automatically find Zeno cycles in the TA we would firstly need to detect cycles in the automaton abstraction, and then assess the time taken along a cycle to see if it could be a Zeno cycle (in our case, a Zeno cycle requires that the time taken to make the cycle could be zero). Although this would give the user a clearer idea of which parts of the TA abstraction could reach the desired region, in cases where these cycles are inherent in the piecewise-linear system (like the example above), there is a possibility that we could end up removing most of the state space, so it is debatable how useful this would be. However, it could give some useful information to the user of the algorithm.

4.9 Conclusions and future work

In this chapter we have extended the splitting method of Chapter 3 to a piecewise-linear class of systems, with each subsystem of the special upper-triangular form of Chapter 3. The piecewise-linear class which we have considered form a natural extension of the continuous systems studied in Chapter 3, and so some of the theory for those continuous systems carried over to parts of the piecewise-linear systems. The splitting method proposed for this class of piecewise-linear systems was shown to give a TA abstraction which proved inevitability of the original system in the 1-D and 2-D cases.

The 3-D and higher-dimensional cases were also discussed, and we have demonstrated some problems that occur with the proposed algorithms in these cases. We proposed solutions to at least give some helpful information to the user in the case which involves infinite length paths of discrete states (Zeno behaviour) caused by pairs of states allowing two-way flow. The solution proposed uses the TA abstraction found by the algorithm, and then eliminates those states with two-way flow and their predecessors to give the part of the space which the TA can guarantee satisfies the inevitability property.

Future work on this algorithm falls into two categories:

1. The first area of future work is to improve the method for the special class of systems to make it work for 3-D and higher-dimensional systems. The main focusses for this need to be the reasons why the method in this chapter has issues with such systems. For instance, the cases of Zeno pairs and cycles of states caused by the layout of the space may have their abstractions improved by not making splits at the subsystem boundaries, which was one assumption we made quite early on (Sec. 4.3.2). There may also be problems around the

occurrence of boxes with infinite time which are not removed by the splitting method as it stands. The proof of removal of infinite time in the continuous case relied on every inseparable infinite-time box having a neighbouring box with greater offset which also has infinite time. In piecewise-linear systems we cannot guarantee that such a neighbouring box exists, and so we may not be able to remove the infinite time on a box, so this needs to be investigated.

2. The second area for future work it to extend this method to create useful abstractions for more complex classes of piecewise-linear, piecewise-continuous, and hybrid systems. As there are so many open ends for 3-D and higher-dimensional systems of the special form at the moment, the first area of future work should be explored first to make more general abstractions. However, there is an immediate area of research in looking at 1-D and 2-D piecewise-continuous systems with more general dynamics in the subsystems, to see how this method extends to those cases.

Chapter 5

Deadness: disproving liveness in hybrid dynamical systems

In this chapter we consider general hybrid dynamical systems, looking at liveness properties and how to automatically disprove them on hybrid systems. Liveness properties are defined as those which cannot be disproved by an execution of finite length, but this can be difficult as infinite length executions are very hard to find in most systems, and in continuous–space-time systems especially. If we could gain some knowledge about the future of a finite execution then we may be able to disprove the liveness property without actually having to find the whole infinite execution. The idea of this chapter is that we can stop an execution if we know it will never satisfy the liveness property, due to some dynamical knowledge about the future of this execution.

For this purpose, we will define a new type of dynamically-aware property which can disprove such liveness properties with a finite length trace. This is the concept of *deadness*, which captures the idea that there could exist another property which, if it is true, implies that the liveness property can never hold in the system. After defining deadness formally (Sect. 5.4), we introduce an algorithm to find deadness conditions for hybrid automata (Sect. 5.5). The algorithm works for special cases of liveness and deadness. The novel idea of the algorithm is that the verification procedure is guided by stability-like properties of the equilibria present in the system. For this, the notion of hybrid-space equilibria is introduced. In Section 5.6 we will then discuss how to implement this algorithm and how to use it to prove deadness (therefore disproving liveness), and describe the implementation that has been made.

The next section (Sect. 5.1) shows the problem addressed by this chapter in more detail, by means of a motivating example. In Sect. 5.7 we will return to the example to examine the properties and methods presented.

Some of the results of this chapter have been submitted for publication in Applied Mathematics and Computation [Carter and Navarro-López, 2013].

5.1 A motivating example

To motivate the definitions in this chapter, we will consider the following discontinuous system, which can be modelled as a hybrid system. It is a simplified oilwell vertical drillstring that exhibits multiple equilibria and periodic oscillations [Navarro-López and Carter, 2011]:

$$\begin{aligned}\dot{x}_1 &= \frac{1}{J_\mathrm{r}}\left[-(c_\mathrm{t}+c_\mathrm{r})x_1 - k_\mathrm{t}x_2 + c_\mathrm{t}x_3 + u\right],\\ \dot{x}_2 &= x_1 - x_3,\\ \dot{x}_3 &= \frac{1}{J_\mathrm{b}}\left[c_\mathrm{t}\,x_1 + k_\mathrm{t}\,x_2 - (c_\mathrm{t}+c_\mathrm{b})x_3 - T_{\mathrm{f_b}}(x_3)\right].\end{aligned} \tag{5.1}$$

Here x_1 and x_3 are the angular velocities of the top-rotary system and the bit, respectively, and x_2 is the difference between the two angular displacements. We combine these variables into a state vector $\boldsymbol{x} = (x_1,\, x_2,\, x_3)^\mathrm{T} \in \mathbb{R}^3$. The input torque $u > 0$ and the weight on the bit $W_\mathrm{ob} > 0$ are two varying parameters with $u, W_{ob} \in \mathbb{R}$. The discontinuous friction torque is $T_{\mathrm{f_b}}(x_3) = f_\mathrm{b}(x_3)\mathrm{sign}(x_3)$, where

$$f_\mathrm{b}(x_3) = W_\mathrm{ob}R_\mathrm{b}\left[\mu_{\mathrm{c_b}} + (\mu_{\mathrm{s_b}} - \mu_{\mathrm{c_b}})\exp^{-\frac{\gamma_\mathrm{b}}{v_\mathrm{f}}|x_3|}\right]. \tag{5.2}$$

Here $R_\mathrm{b} > 0$ is the bit radius; $\mu_{\mathrm{s_b}},\ \mu_{\mathrm{c_b}} \in (0,1)$ are the static and Coulomb friction coefficients associated with the bit; and $0 < \gamma_\mathrm{b} < 1$ and $v_\mathrm{f} > 0$ are constants. The Coulomb and static friction torque are $T_{\mathrm{c_b}}$ and $T_{\mathrm{s_b}}$, respectively, with $T_{\mathrm{c_b}} = W_\mathrm{ob}R_\mathrm{b}\mu_{\mathrm{c_b}}$, $T_{\mathrm{s_b}} = W_\mathrm{ob}R_\mathrm{b}\mu_{\mathrm{s_b}}$. When the system's state is at the discontinuity point, $x_3 = 0$, we define T_{f_b} by Utkin's equivalent control method for sliding modes [Utkin, 1992] as

$$T_{\mathrm{f_b}}(\boldsymbol{x}) = u_\mathrm{eq}(\boldsymbol{x}) = c_\mathrm{t}\,x_1 + k_\mathrm{t}\,x_2 - (c_\mathrm{t}+c_\mathrm{b})\,x_3. \tag{5.3}$$

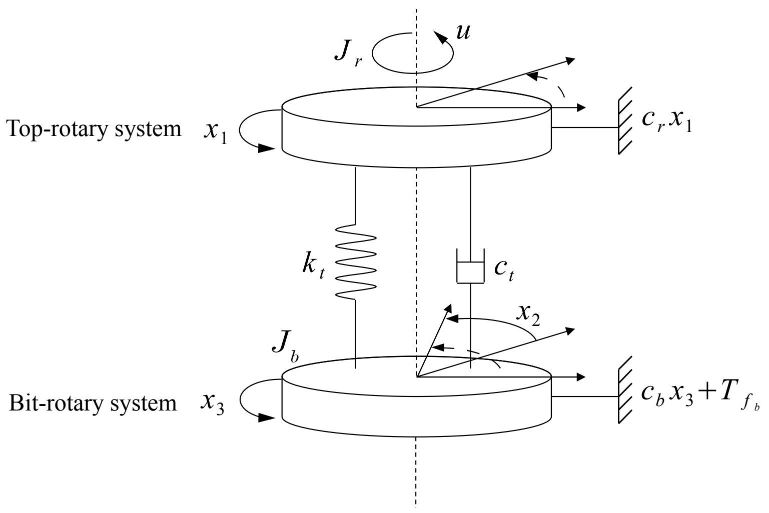


Figure 5.1: The model for the drillstring. The curved dashed arrows indicate the angular displacement of the top and bit, and x_2 is the difference between them.

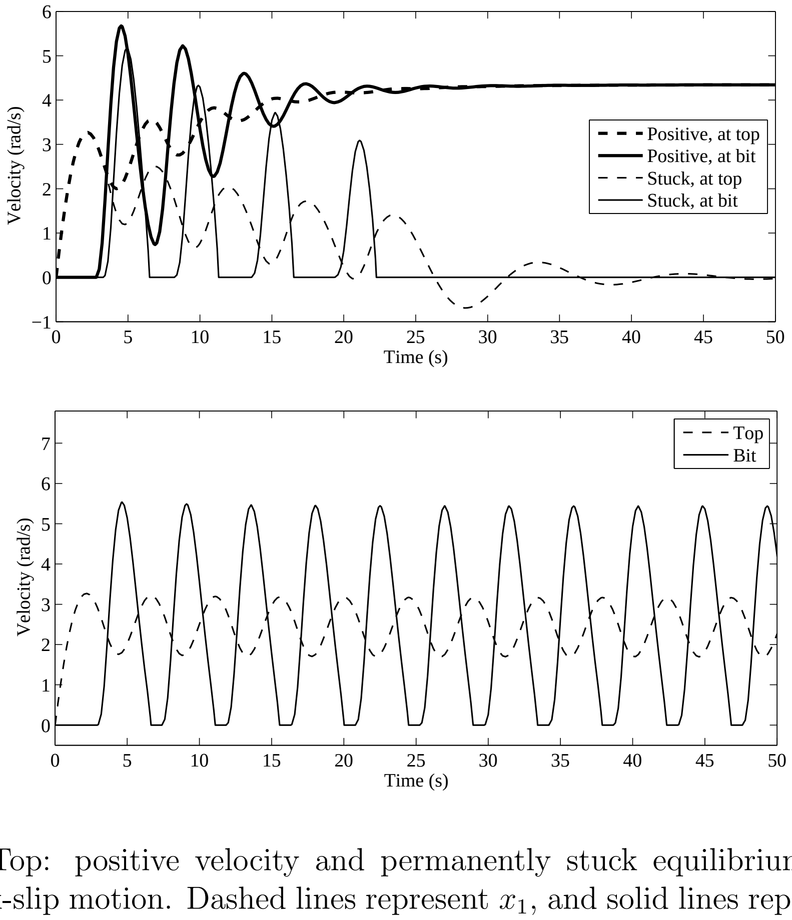


Figure 5.2: Top: positive velocity and permanently stuck equilibrium behaviours. Bottom: stick-slip motion. Dashed lines represent x_1, and solid lines represent x_3. For these particular trajectories, u = 6000 N m with W_{ob} variable: high W_{ob} (= 59208 N) results in the stuck behaviour, low W_{ob} (= 50000 N) results in the positive velocity behaviour, and medium W_{ob} (= 53018 N) causes the periodic stick-slip behaviour.

Typical parameter values used in this chapter are:

$$
\begin{aligned}
c_t &= 172.3067\ \mathrm{N\,m\,s/rad}, & J_b &= 471.9698\ \mathrm{kg\,m^2}, & \mu_{c_b} &= 0.5,\\
c_r &= 425\ \mathrm{N\,m\,s/rad}, & J_r &= 2122\ \mathrm{kg\,m^2}, & \mu_{s_b} &= 0.8,\\
c_b &= 50\ \mathrm{N\,m\,s/rad}, & R_b &= 0.155575\ \mathrm{m}, & \gamma_b &= 0.9,\\
k_t &= 861.5336\ \mathrm{N\,m/rad}, & u &= 6000\ \mathrm{N\,m}, & \nu_f &= 1\ \mathrm{rad/s}.
\end{aligned}
$$

This system exhibits a rich collection of behaviours depending on the competing ‘strengths’ of two locally attractive equilibrium points: one with $x_3 = 0$ and one with $x_3 > 0$. The values of (u, W_{ob}) vary the relative attractivity of the two equilibria, resulting in three main behaviour patterns:

- **Positive velocity equilibrium**: the bit velocity x_3, converges to a positive equilibrium value and $x_1 = x_3$.
- **Permanent stuck bit**: the bit stops rotating after some period of time and never starts again.
- **Stick-slip motion**: the bit velocity x_3 oscillates between zero and a positive velocity.

Figure 5.2 shows these three behaviour patterns.

Analysing dynamical patterns of this system is very hard, as it is with many non-linear hybrid systems. We can identify these three behaviours by simulation, but it is very difficult to know which one will actually be present in the system for any given set of parameters (see Navarro-López and Cortés [2007] for the dynamical analysis of this model). Consequently, this model of a drillstring is an ideal candidate for new methods of analysis, in particular formal verification. In Section 5.7 we will return to this example and show how we can disprove that the positive velocity equilibrium is globally attractive by using the permanent stuck bit equilibrium to create a deadness property.

5.2 Preliminary definitions

In this section we introduce the key definitions which we make use of in this chapter. We will firstly give those definitions relating to hybrid systems, and then we will introduce the two concepts of safety and liveness for such systems.

5.2.1 Hybrid automata

We already introduced the concept of hybrid automata in Section 2.2.1, Definition 2.8. Hybrid automata are a useful model of hybrid dynamical systems, since they explicitly show the interaction between the continuous and the discrete parts of the system. Let us recall the definition of a hybrid automaton and the hybrid state space of such an automaton.

Definition 5.1 (Hybrid automaton [Johansson et al., 1999])**.** A hybrid automaton is a collection

$$H = (Q, E, \mathcal{X}, Dom, \mathcal{F}, Init, G, R)$$

that models a hybrid system, where

- Q is a finite set of locations.
- $E \subseteq Q \times Q$ is a finite set of edges called transitions or events.
- $\mathcal{X} \subseteq \mathbb{R}^n$ is the continuous state space.
- $Dom : Q \to 2^{\mathcal{X}}$ is the location domain (sometimes called an invariant). It assigns a set of continuous states to each discrete location $q \in Q$, thus, $Dom(q) \subseteq \mathcal{X}$.
- $\mathcal{F} = \{f_q(x) : \; q \in Q\}$ is a finite set of vector fields describing the continuous dynamics in each location, such that $f_q : \mathcal{X} \to \mathcal{X}$. Each $f_q(x)$ is assumed to be Lipschitz continuous on the location domain for q in order to ensure that the solution exists and is unique.
- $Init \subseteq \bigcup_{q \in Q} q \times Dom(q) \subseteq Q \times \mathcal{X}$ is a set of initial states.
- $G : E \to 2^{\mathcal{X}}$ is a guard map. G assigns to each edge a set of continuous states; this set contains the states which enable the edge to be taken.
- $R : E \times \mathcal{X} \to 2^{\mathcal{X}}$ is a reset map for the continuous states for each edge. It is assumed to be non-empty, so that the dynamics can only be changed, not destroyed. ∎

Definition 5.2 (Hybrid state space [Johansson et al., 1999])**.** The hybrid state space is the set defined by

$$\mathcal{Z} \equiv \bigcup_{q \in Q} q \times Dom(q) \subseteq Q \times \mathcal{X}.$$

That is, the set of all pairs (q, x) which the hybrid automaton allows to exist. A *hybrid state* z is a member of the hybrid state space, or $z = (q, x) \in \mathcal{Z}$. A *hybrid set* W is a subset of $\mathcal{Z}$, that is $W \subseteq \mathcal{Z}$. ∎

We should consider how such a hybrid automaton can evolve, by firstly defining the hybrid time trajectory, and secondly defining the execution of the hybrid automaton on such a time trajectory.

Definition 5.3 ([Johansson et al., 1999])**.** A *hybrid time trajectory* $\tau = \{I_i\}_{i=0}^N$, with $N \in \mathbb{N}_0$, where $\mathbb{N}_0$ is the set natural numbers including zero, is a finite or infinite sequence of intervals of the real line, such that

- for all $0 \leq i < N$, $I_i = [t_i, t_i']$ with $t_i \leq t_i' = t_{i+1}$;
- if $N < \infty$, either $I_N = [t_N, t_N']$ with $t_N \leq t_N' < \infty$, or $I_N = [t_N, t_N')$ with $t_N < t_N' \leq \infty$.

The set of all hybrid time trajectories is denoted by $\mathcal{T}$. ■

For ease of notation we will use $t \in \tau$ as a shorthand for 'there is some i such that $t \in [t_i, t_i'] \in \tau$' for any $\tau \in \mathcal{T}$. We will also always write the final interval in the sequence as a closed interval $[t_N, t_N']$, but the reader may substitute $[t_N, t_N')$ if required.

When considering a finite part of an infinite time trajectory, also called a *partial* hybrid time trajectory, we will use the notation $\tau_{0,p}$, $0 \leq p \leq N$, $p < \infty$. This consists of the intervals $\{I_i\}_{i=0}^p$, where $I_i = [t_i, t_i']$ for $0 \leq i < p$, and $I_p = [t_p, t_p'']$ with $t_p \leq t_p'' \leq t_p'$ and $t_p'' < \infty$.

We now define the execution of the system on τ. The idea is that continuous flow of the hybrid automaton occurs in every interval $[t_i, t_i']$ (when this interval is of non-zero length), and discrete transitions occur to take the end of one interval $[t_i, t_i']$ to the start of the next one $[t_{i+1}, t_{i+1}']$. This captures the behaviour of the hybrid automaton perfectly, allowing continuous flow in one location, taking us to a point when we make a discrete transition to another location, to continue continuous motion again.

Definition 5.4 (Valid Execution [Johansson et al., 1999])**.** An execution ϕ of a hybrid automaton H is a collection $\phi = (\tau, z)$ with hybrid time trajectory $\tau = \{[t_i, t_i']\}_{i=0}^N \in \mathcal{T}$, and $z : \tau \to \mathcal{Z}$ a product of mappings $q : \tau \to Q$ and $x : \tau \to \mathcal{X}$, satisfying

1. Initial condition: $z(t_0) = (q(t_0), x(t_0)) \in \mathit{Init}$.
2. Continuous evolution: for all i such that $t_i < t_i'$, it is the case that for $t \in [t_i, t_i']$, $q(t)$ is constant and $x(t)$ is Lipschitz continuous and differentiable, $x(t) \in \mathit{Dom}(q(t))$, and the evolution is described by $\dot{x}(t) = f_{q(t)}(x(t))$ for $t \in [t_i, t_i')$.

3. Discrete transitions: for all $i \in \{0, 1, \ldots, N-1\}$, an edge $e = (q(t'_i), q(t_{i+1})) \in E$ exists for which $x(t'_i) \in G(e)$, and $x(t_{i+1}) \in R(e, x(t'_i))$.

The set of all executions of H from the initial set *Init* is defined by $\mathcal{E}_{H,\mathit{Init}}$. We will also use the shorthand $\mathcal{E}_H$ for the set of all executions of H with initial set equal to $\mathcal{Z}$ (the whole space). ■

For any hybrid time trajectory τ, note that $z(t'_i)$ is not necessarily equal to $z(t_{i+1})$, due to the implicit dependence of q and x on the interval of τ being considered. In general, at least the discrete state will change, so that $q(t'_i) \neq q(t_{i+1})$.

We now introduce the notion of a *future unaware execution*, which is a sequence of hybrid states ϕ in the hybrid state space of the hybrid automaton H, where ϕ does not have to follow the dynamics of H.

Definition 5.5 (Future unaware execution)**.** A future unaware execution ϕ of a hybrid automaton H is a collection $\phi = (\tau, z)$ with hybrid time trajectory $\tau = \{[t_i, t'_i]\}_{i=0}^N \in \mathcal{T}$, and $z : \tau \to 2^{\mathcal{Z}}$ is a product of multivalued mappings $q : \tau \to 2^Q$ and $x : \tau \to 2^{\mathcal{X}}$, satisfying

1. Initial Condition: $z(t_0) = (q(t_0), x(t_0)) \in \mathit{Init}$.

2. Continuous evolution: $x(t)$ is continuous and $x(t) \in \mathit{Dom}(q(t))$ in every interval $t \in [t_i, t'_i]$ for $0 \leq i \leq N$.

3. Discrete transitions: for all $i \in \{0, 1, \ldots, N-1\}$, ϕ jumps from one location $q(t'_i)$ to another $q(t_{i+1})$; this jump does not necessarily follow the guards. The continuous state x can be reset to any value in the new location domain $\mathit{Dom}(q(t_{i+1}))$.

The set of all future unaware executions of H that start from the initial set *Init* is denoted by $\mathcal{E}_{U,\mathit{Init}}$, and the set with initial set $\mathcal{Z}$ (the whole space) is denoted by $\mathcal{E}_U$.■

These future unaware executions are a generalisation of the notion of Kleene closure of the set of symbols in a finite-state automaton [Hopcroft et al., 2007]. In both cases we know that when we consider the actual structure of the finite-state or hybrid automaton, the resulting strings/executions will exist as a subset of the symbol set closure or the future unaware set respectively. Another way of thinking about it is that the future unaware executions are all the executions possible given only a hybrid state space in which they occur.

We now define a useful order on hybrid time trajectories and hybrid automaton executions, and then use it when we classify executions into types.

- $\tau = \{I_i\}_{i=0}^{N} \in \mathcal{T}$ is a *prefix* of $\tau' = \{J_i\}_{i=0}^{M} \in \mathcal{T}$, denoted $\tau \preceq \tau'$, if either they are identical or τ is finite with $M \geq N$, $I_i = J_i$ for all $i = 0, \dots, N-1$ and $I_N \subseteq J_N$.
- $\phi = (\tau, z)$ is a *prefix* of $\phi' = (\tau', z')$, denoted $\phi \preceq \phi'$, if $\tau \preceq \tau'$ and $z(t) = z'(t)$ for all $t \in \tau$.
- ϕ is a *strict prefix* of ϕ', denoted $\phi \prec \phi'$, if $\phi \preceq \phi'$ and $\phi \neq \phi'$.

Definition 5.6 ([Johansson et al., 1999])**.** Taking $0 \leq N \leq \infty$, we define an execution $\phi = (\tau, z)$ to be:

- *finite* if τ is a finite sequence ending with a finite interval, that is $\tau = \tau_{0,N}$ with $t'_N < \infty$ and $N < \infty$.
- *infinite* if τ is a finite sequence ending with an infinite interval *or* an infinite sequence, that is $\tau = \tau_{0,N}$ with either $t'_N = \infty$ or $N = \infty$.
- *maximal* if $\nexists \phi'$ with $\phi \prec \phi'$. ■

We will assume in this chapter that the hybrid automaton is non-blocking (see Johansson et al. [1999]), so that maximal executions of the system are always infinite. The assumption of a non-blocking automaton simply rules out behaviours for which the model 'gets stuck' after a finite length of time. In general these will be artefacts of the mathematical model and not realistic behaviours — this is because a real world system does not stop after a finite length of time, instead time continues and forces the system to keep evolving in some way. Hence, except in very specific cases, an infinite execution will always occur, so it is realistic for the model to be non-blocking.

We now classify some properties of the executions. Firstly notice that the set of valid executions of H is a subset of the class of future unaware executions, or $\mathcal{E}_H \subseteq \mathcal{E}_U$. The set of all executions with particular initial condition $z(t_0) \in \mathcal{Z}$ is denoted by $\mathcal{E}_{H,z(t_0)}$ for valid executions of H, and by $\mathcal{E}_{U,z(t_0)}$ for the set of future unaware executions. The valid executions of H from a set of initial conditions *Init* is denoted by $\mathcal{E}_{H,\mathit{Init}}$, and similarly $\mathcal{E}_{U,\mathit{Init}}$ for future unaware. We also define $\mathcal{E}_H^F$ and $\mathcal{E}_H^\infty$ as, respectively, the sets of all finite and infinite executions of H (similarly $\mathcal{E}_U^F$ and $\mathcal{E}_U^\infty$

for future unaware executions). These can also be combined: the set of finite valid executions which start from $z(t_0) \in \mathcal{Z}$ is $\mathcal{E}^F_{H,z(t_0)}$ for example.

5.2.2 Safety and liveness

In this section we will define the concepts of safety and liveness, which are descriptors for logical properties. In this chapter we do not use any particular logic, but it must be a future-time temporal logic which can express both safety and liveness properties on hybrid systems, for example linear temporal logic (LTL) or computation tree logic (CTL), or many of the other logics discussed in Section 2.3.4. We assume that we have syntax and semantics[1] defined for the logic we are using, and define the satisfaction relation by the following.

Definition 5.7 (Satisfaction by Infinite Executions)**.** Consider the hybrid automaton H, and the formula φ defined in some temporal logic on H. An infinite (possibly future unaware) execution of the hybrid automaton $\phi = (\tau, z) \in \mathcal{E}^\infty_U$ is said to satisfy φ and is denoted by $\phi \vDash \varphi$, if and only if $z(t)$ satisfies the semantics of the formula φ in the logic. ■

This definition only defines how infinite executions can satisfy a property. It is necessary for the definition of deadness that we know how finite executions satisfy logical properties, so we define *chattering semantics* for expressions of the form $\phi_{0,p} \vDash \varphi$, where the last value of the execution, $z(t''_p) = (q(t''_p), x(t''_p))$, is repeated for the rest of time. This is a sensible semantics for continuous-time properties, where there is no notion of a 'next state' in the execution.

Definition 5.8 (Satisfaction by Finite Executions)**.** Given a finite execution $\phi_{0,p}$, we define the *chattering extension* to this execution by $\phi^{\mathrm{c}}_{0,p} = (\tau_{\mathrm{c}}, z_{\mathrm{c}}) \in \mathcal{E}^\infty_U$ where $\tau_{\mathrm{c}} = \{[t_0, t'_0], \ldots, [t_{p-1}, t'_{p-1}], [t_p, \infty)\}$, and $z_{\mathrm{c}}(t) = z_{0,p}(t)$ for $t \in \tau_{0,p}$, and $z_{\mathrm{c}}(t) = z(t''_p)$ for $t \in [t''_p, \infty)$. Then, a finite execution can satisfy a formula φ by considering the equivalence

$$(\phi_{0,p} \vDash \varphi) \;\equiv\; (\phi^{\mathrm{c}}_{0,p} \vDash \varphi). \tag{5.4}$$

■

[1]Syntax is the symbols we can use and the way these symbols can be combined, and semantics is the meaning of these symbols in the logic.

Note that this execution extended by chattering is not necessarily a valid execution of the hybrid automaton H, but will definitely belong to the set of future unaware executions. Since this is only a convention for satisfaction of logical formulae we do not worry too much about the real world meaning.

We will now formally define a safety property for hybrid systems, where intuitively the idea is that 'nothing bad ever happens'. The formal definition is based on the fact that if an infinite execution is not safe, then there must have been a point in time at which the "bad thing" happened. Safety was originally defined for discrete-time systems by Leslie Lamport in [Paul et al., 1985], although we use the definition by Alpern and Schneider [1985]. Note that our generalisation of this definition to hybrid systems uses future unaware executions, as whether a logical property is classified as safety is independent of the dynamics of the hybrid automaton — it only requires a hybrid state space to define the property on.

Definition 5.9. A formula S defined on the hybrid state space $\mathcal{Z}$ is a *safety property* iff for all future unaware infinite executions that do not satisfy S a finite prefix can be found for which all infinite extensions do not satisfy S, or more formally

$$\forall \phi \in \mathcal{E}_U^\infty \ \big(\phi \nvDash S \ \Rightarrow \ \exists \phi_{0,p} \in \mathcal{E}_U^F \ \phi_{0,p} \prec \phi \ \forall \phi' \in \mathcal{E}_U^\infty (\phi_{0,p} \prec \phi' \Rightarrow \phi' \nvDash S)\big). \tag{5.5}$$

■

Safety properties in hybrid systems have typically been reduced to invariance properties, which say that 'some bad set is never reached'. In the drillstring example, for instance, a safety property could be that we never want to get negative velocity on the drill bit (so that it is always drilling forwards into the ground).

Liveness is a property which says that 'something good eventually happens'. It has been considered before in discrete systems, mostly in the context of verification of such systems [Baier and Kwiatkowska, 2000, Bouajjani et al., 2005, Owicki and Lamport, 1982], but has not been formally considered in hybrid dynamical systems. Here we define liveness in the context of hybrid automata, using the ideas of Alpern and Schneider [1985]. The formal definition uses the fact that if 'something good eventually happens' and at some finite point in an execution it has not already happened, then it must still be possible to satisfy the liveness property at some point in the future.

We again use future unaware executions, as the concept of liveness is independent of the actual dynamics of H.

Definition 5.10. A formula $\overline{L}$ defined on the hybrid state space $\mathcal{Z}$ is a *liveness property* iff every finite future unaware execution of H can be extended to an infinite execution which satisfies $\overline{L}$, or more formally

$$\forall \phi_{0,p} \in \mathcal{E}_U^F \; \exists \phi \in \mathcal{E}_U^\infty \; \big(\phi_{0,p} \prec \phi \wedge \phi \vDash \overline{L}\big). \tag{5.6}$$

■

The key idea of liveness properties is that they *cannot be directly disproved by a finite execution*, as at any finite time point we do not know what will happen in the future, and so the 'good thing' could still happen. In dynamical systems, the idea of liveness is most clearly related to achieving a desired goal, whether it be that of reaching a useful set of the state space or that of tending to a periodic cycle of states.

We now define the way that the definitions of safety and liveness properties translate into descriptors for executions in the hybrid automaton.

Definition 5.11. An infinite execution $\phi \in \mathcal{E}_U^\infty$ is called *safe* with respect to safety property S iff it satisfies the safety property, or $\phi \vDash S$. A hybrid automaton H is *safe* iff $\forall \phi \in \mathcal{E}_{H,Init}^\infty \; (\phi \vDash S)$. ■

Definition 5.12. An infinite execution $\phi \in \mathcal{E}_U^\infty$ is *live* with respect to liveness property $\overline{L}$ iff it satisfies the liveness property, or $\phi \vDash \overline{L}$. A hybrid automaton H is *live* iff $\forall \phi \in \mathcal{E}_{H,Init}^\infty \; \big(\phi \vDash \overline{L}\big)$. ■

5.3 Relating liveness to stability-type properties

In this section we will look at how a typical stability-type property of hybrid dynamical systems relates to a simple liveness property. This will motivate the definition of deadness in the next section. We will use the notation of a *ball of radius* $r > 0$ *around a point* $p \in \mathbb{R}^n$, defined by

$$B(r, p) = \{x \in \mathbb{R}^n : \|x - p\| < r\},$$

where $\|\cdot\|$ denotes the 2-norm on the Euclidean space $\mathbb{R}^n$.

We looked at the dynamical systems properties of stability and attractivity in Section 2.1.2 for continuous dynamical systems, and we also discussed stability results in hybrid dynamical systems in Section 2.2.5. We introduced the notion of an equilibrium point as a point in space at which an execution of the the system, if it starts at the point, does not change its position in space as time evolves. Let us denote an equilibrium point of a hybrid automaton as $\overline{z} \in \mathcal{Z}$.

Let us consider the property of attractivity on hybrid systems, and relate it to a liveness property. Attractivity says that every trajectory which starts within a certain range of an equilibrium point will tend towards that equilibrium, reaching it eventually (in possibly infinite time or an infinite number of discrete transitions). Here we just consider global attractivity. The formal definition for global attractivity in hybrid automata is

$$\overline{x} \text{ is } \textit{globally attractive} \;\Leftrightarrow \forall \phi = (\tau, z) \in \mathcal{E}^{\infty}_{H,\mathit{Init}},\ \lim_{t \to t_\infty} x(t) = \overline{x},$$

with $t_\infty = \sum_i (t'_i - t_i)$, the final time in the execution ϕ.

In continuous systems global attractivity is considered for all initial conditions in the state space, and theoretically this is how the definition should be considered in hybrid automata. However, due to the complexity of hybrid systems we would not usually expect every execution starting from every point in the system to converge to one equilibrium, so global attractivity of an equilibrium is not very useful with *Init* $= \mathcal{Z}$. This is why we have included the initial set *Init* in the allowed executions for global attractivity, as we are more likely to be interested in whether the equilibrium is attractive given a certain set of likely initial conditions, *Init* $\subseteq \mathcal{Z}$. This still allows for the case when *Init* $= \mathcal{Z}$ should we require it.

Rewriting the limit function with its classical definition gives us

$$\overline{x} \text{ is globally attractive } \Leftrightarrow$$
$$\Big[\forall \phi = (\tau, z) \in \mathcal{E}^{\infty}_{H,\mathit{Init}} \Big(\forall \epsilon > 0\, \exists \delta > 0 \big[t \in B(\delta, t_\infty) \Rightarrow x(t) \in B(\epsilon, \overline{x})\big]\Big)\Big]. \quad (5.7)$$

That is, a hybrid automaton is globally attractive if for any execution, we can select any small ball around $\overline{x}$ and guarantee if we go far enough in time that we will enter this selected ball.

This attractivity property does not lend itself to computational proof very easily, as it involves two different quantifiers intricately linked. However, we can consider a weaker condition, an inevitability property:

$$\forall \phi = (\tau, z) \in \mathcal{E}^{\infty}_{H,Init}\ \exists \delta > 0 \big[t \in B(\delta, t_\infty) \Rightarrow x(t) \in B(\epsilon, \overline{x})\big],$$

for some chosen ϵ. Rewriting in linear temporal logic this becomes

$$\forall \phi = (\tau, z) \in \mathcal{E}^{\infty}_{H,Init}\ \Diamond[x(t) \in B(\epsilon, \overline{x})]. \tag{5.8}$$

This property says that all trajectories eventually reach some set of the space described by a ball around $\overline{x}$.

In Equation (5.8) we have simply restricted (5.7) by considering only one ϵ, and so (5.7) $\Rightarrow$ (5.8). An equivalent statement of this is the contrapositive:

$$\neg(5.8)\ \Rightarrow\ \neg(5.7). \tag{5.9}$$

Now $\neg(5.8) \Leftrightarrow\ \exists \phi = (\tau, z) \in \mathcal{E}^{\infty}_{H}\ \Box[x(t) \notin B(\epsilon, \overline{x})]$, which says that for (5.8) to be false we only require one infinite execution which never enters $B(\epsilon, \overline{x})$. So (5.9) says that finding one execution which disproves the liveness property $\Diamond[x(t) \in B(\epsilon, \overline{x})]$ will disprove Equation (5.7) which expresses global attractivity of the hybrid automaton.

So $\neg(5.8)$ requires at least one infinite execution to not satisfy the liveness property $\Diamond[x(t) \in B(\epsilon, \overline{x})]$. However, as mentioned before, finding infinite executions of hybrid automata is very hard, so we wish to define a property on finite executions which will imply that $\Diamond[x(t) \in B(\epsilon, \overline{x})]$ is not true for some set of infinite executions. If we can define and use such a property this will mean that we can disprove a global attractivity property by finding one finite execution.

In the next section we will define deadness as such a property for disproving liveness properties with finite executions, and we will use it to disprove global attractivity of the positive velocity equilibrium in the drillstring example in Section 5.7.

5.4 Defining deadness

Once we have specified a desired liveness property, we wish to verify whether it is actually true, and this is where the complexity arises. Liveness properties are complex to verify, since it has to be shown that for all possible initial conditions and all possible executions from these initial conditions some property holds at some point in the future. If we cannot prove liveness, a sensible plan is to attempt to disprove it, and this could in turn disprove much more general properties, like global attractivity (as discussed in Section 5.3). However, even disproving liveness properties involves finding counterexample executions of infinite length, as finite length executions could always be extended to something that satisfies the liveness property (by definition). Finding infinite length executions is especially hard in hybrid dynamical systems, due to the need to accurately represent a path in *continuous* space and time.

For this reason, we propose using another property alongside the liveness property which will disprove liveness with a finite execution if it is proved to be true. We call this property *deadness*: it is a concept related to dead states in automata theory. Note that, unlike safety and liveness, this property is related only to the valid executions of the hybrid automaton, not the more abstract future unaware versions.

Definition 5.13. A formula D on a hybrid automaton H is a *deadness property for liveness property* $\overline{L}$ iff any finite execution which satisfies D but not $\overline{L}$ cannot be extended to an infinite execution which satisfies $\overline{L}$, or more formally

$$\forall \phi_{0,p} \in \mathcal{E}_H^F \big(\phi_{0,p} \vDash (D \wedge \neg\overline{L}) \ \Rightarrow\ \forall \phi \in \mathcal{E}_H^\infty \left(\phi_{0,p} \prec \phi \ \Rightarrow\ \phi \nvDash \overline{L}\right)\big). \tag{5.10}$$

We can define a deadness property for H with respect to the initial set *Init* by taking instead $\phi_{0,p} \in \mathcal{E}_{H,Init}^F$ and $\phi \in \mathcal{E}_{H,Init}^\infty$ in (5.10). ■

Intuitively, we are formalising the idea that 'when we are dead, we cannot be alive again'. As deadness is a property that is defined on finite executions, we must evaluate $\phi_{0,p} \vDash (D \wedge \neg\overline{L})$ with the chattering extension defined in Def. 5.8, that is $\phi_{0,p} \vDash (D \wedge \neg\overline{L}) \ \equiv\ \phi_{0,p}^{\mathrm{c}} \vDash (D \wedge \neg\overline{L})$.

From their definition, deadness properties are those which can be satisfied by a finite execution, or more formally by the chattering infinite extension of this execution (Def. 5.8). The idea is that once some point in space has been reached the deadness property becomes true and would not become false again if the execution always remained at this point. Some properties which could be deadness properties (given in metric temporal logic (MTL)) are $\Diamond P$ (eventually the set described by P is reached), $\Diamond_{[0,\overline{t}]} P$ (within $\overline{t}$ seconds a set is reached), and $\Diamond\Box_{[0,\overline{t}]} P$ (eventually the set P is reached and the trajectory then remains there for $\overline{t}$ seconds).

We now define the way that the definition of deadness properties translates into descriptors for executions and the hybrid automaton.

Definition 5.14. A finite execution $\phi_{0,p} \in \mathcal{E}^F_U$ is called *dead* if it satisfies the deadness property D but not the liveness property $\overline{L}$, or $\phi_{0,p} \vDash (D \wedge \neg\overline{L})$. A hybrid automaton H is said to be *dead* if there exists at least one finite execution of H which is dead after starting in $Init$, that is

$$H \text{ is dead} \Leftrightarrow \exists \phi_{0,p} \in \mathcal{E}^F_{H,Init}(\phi_{0,p} \vDash (D \wedge \neg\overline{L})). \tag{5.11}$$

■

Lemma 5.15. *If the hybrid automaton H is dead with respect to a liveness property $\overline{L}$ and a deadness property D, then H is not live.* ■

Proof. Assume there exists $\phi_{0,p} = (\tau_{0,p}, z) \in \mathcal{E}^F_{H,Init}$ such that $\phi_{0,p} \vDash (D \wedge \neg\overline{L})$. Then for all extended executions $\phi \in \mathcal{E}^\infty_{H,Init}$, $\phi \nvDash \overline{L}$ or $\phi_{0,p} \not\prec \phi$ by Def. 5.13. As $\phi_{0,p}$ is a finite execution, it is not maximal in H, and so (by non-blocking of H) there must be at least one infinite execution ϕ' such that $\phi_{0,p} \prec \phi'$, so $\phi' \nvDash \overline{L}$. Now, this is a contradiction to the required condition for liveness of H in Def. 5.12, so the hybrid automaton H is not live. □

It is important to understand that the concept of deadness only tells us about liveness if it is proven — if deadness does not hold then this does not prove whether H is live or not. We are introducing a framework to help prove more instances of properties on hybrid systems, as using a deadness property can increase our ability to find counterexamples to liveness properties, by only requiring finite executions of the hybrid automaton to be found. We can use our dynamical knowledge of the hybrid automaton to create these deadness properties, so that finite executions are enough to disprove liveness — this is a new idea for formal verification of hybrid automata.

5.5 Finding deadness properties for hybrid systems

In this section we will give one method for how deadness properties can be found, using dynamical properties of the hybrid system we are interested in. The idea is to find invariant sets which trap the executions of the hybrid automaton away from where

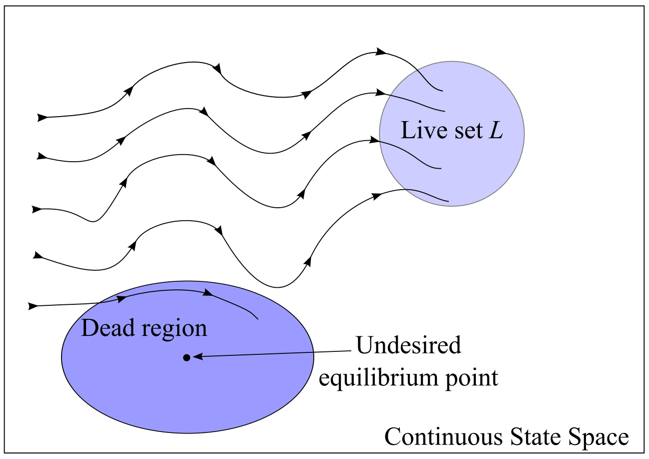


Figure 5.3: Proving that the live set L is not reached by every execution, using a deadness property based around an equilibrium point. For clarity, only the continuous space is shown here.

the desired behaviour occurs, and then the deadness property is to show that such an invariant set is reached in finite time. Towards this goal, we give the definition of a hybrid invariant set.

Definition 5.16. A hybrid set $W \subseteq \mathcal{Z}$ is a *hybrid invariant set* for the hybrid automaton H iff all executions starting in W remain there for all time, or more formally

$$\forall \phi = (\tau, z) \in \mathcal{E}_{H,W}\ \forall t \in \tau\ (z(t) \in W). \qquad \blacksquare$$

A hybrid invariant set is a part of the hybrid state space $\mathcal{Z}$ which the executions of the system cannot leave once they have entered it. As a hybrid invariant set can include parts of more than one discrete location, executions which are trapped inside the set can still make both discrete and continuous transitions.

The method we propose finds a deadness property for a liveness property that says we reach some desired hybrid set $L \subseteq \mathcal{Z}$ in the space, which is an inevitability property. We will refer to this desired set L as the *live set*. The deadness property is to reach any of a selection of invariant sets which trap the dynamics away from L, which is also an inevitability property. Figure 5.3 shows the idea of this property in a visual way.

We make use of the fact that hybrid dynamical systems can have multiple equilibrium points, some of which we wish to tend to and some we want to avoid. The algorithm we present does not consider more general types of limiting behaviour that can create invariant sets, although these could be added with different methods for finding the invariant sets. The other kinds of limiting behaviours are discussed in the conclusion to this chapter. We use here the notion of equilibria in *hybrid-space*, so that a point $(\overline{q}, \overline{x})$ can be an equilibrium of the hybrid automaton rather than a state vector $\overline{x}$ simply in continuous space.

Definition 5.17. $\overline{z} = (\overline{q}, \overline{x}) \in \mathscr{Z}$ is a *hybrid-space equilibrium* of the hybrid automaton H if both of the following conditions hold:

1. $\overline{x} \in Dom(\overline{q})$ and $f_{\overline{q}}(\overline{x}) = 0$.

2. $\overline{x} \notin G(e)$ for any $q \in Q$ such that $e = (\overline{q}, q) \in E$. ■

These hybrid-space equilibria effectively consider the dynamics in each location separately, only allowing a hybrid state to be considered an equilibrium if no continuous or discrete dynamics can change the value of an execution that starts at that hybrid state. This notion of equilibrium should be contrasted with the typical notion used in switched and hybrid systems, where equilibria are defined as points in continuous-space only, common to all locations where they exist. More formally, a continuous-space equilibrium $\overline{x} \in \mathbb{R}^n$ is defined as having (1) $f_q(\overline{x}) = 0$ for every $q \in Q$ where $\overline{x} \in Dom(q)$, and (2) if $\overline{x}$ ever occurs in a guard condition then it must be reset to itself (although the discrete location could change).

The continuous-space definition of equilibrium has evolved in the theory of stability of switched systems, where switching could happen at any time in the system. In such a context the only sensible idea of equilibrium will allow discrete motion to happen at an equilibrium point, as switching could happen whilst at the point. However, as we are considering the class of hybrid systems for which hybrid automata are a natural representation, and not the class of switched systems, it makes just as much sense to consider equilibria that can trap executions of the hybrid automata in one location only.

When we consider these hybrid-space equilibria we can use the methods of Lyapunov functions for continuous systems (introduced in Section 2.1.2) to analyse the dynamics of each location. In particular, we can find invariant sets of each set of discrete dynamics, which will be invariant sets of the hybrid automaton if they do not intersect with any guard conditions.

Algorithm 5.1 Finding dead sets to disprove that all executions reach L

Input: Hybrid automaton H, and a live set $L \subseteq \mathcal{Z}$ we would like to reach.
Output: A hybrid set $W \subseteq \mathcal{Z}$ which, if reached, will disprove the liveness property $\Diamond L$.

```
1: $W \leftarrow \emptyset$ (initialise the hybrid dead region)
2: for all $q \in Q$ do
3:     $EQ_q \leftarrow$ find all solutions of equation $f_q(x) = 0$ in set $Dom(q)$
4:     for all $\overline{x} \in EQ_q$ do
5:         if $\overline{x} \in L$ then
6:             remove $\overline{x}$ from $EQ_q$
7:             continue (to next $\overline{x}$)
8:         else if $f_q$ is nonlinear and $\overline{x}$ is not locally asymptotically stable then
9:             remove $\overline{x}$ from $EQ_q$
10:            continue (to next $\overline{x}$)
11:        else if $f_q$ is linear and $\overline{x}$ is unstable then
12:            remove $\overline{x}$ from $EQ_q$
13:            continue (to next $\overline{x}$)
14:        else
15:            for all $e \in E$, with $e = (q, p)$ for any $p$ do
16:                if $\overline{x} \in G(e)$ then
17:                    remove $\overline{x}$ from $EQ_q$
18:                    break loop (and go to next $\overline{x}$ in $EQ_q$)
19:                end if
20:            end for
21:        end if
22:        $V \leftarrow$ Lyapunov function for the linearised dynamics of $f_q$ around $\overline{x}$ in
   domain $Dom(q)$
23:        $W_V \leftarrow$ invariant set of nonlinear dynamics $f_q$ created from $V$
24:        $W \leftarrow$ add $(q, W_V)$ to the collection of dead sets
25:    end for
26: end for
```

Given this discussion, we can now explain the method that we propose for finding deadness properties on a hybrid system described by a hybrid automaton. The method is given in Algorithm 5.1.

In particular, for each location $q \in Q$ of the hybrid automaton, the algorithm first finds all equilibrium points, disregarding any equilibrium point in the desired set L. Then the unstable equilibria are disregarded: for locations with nonlinear dynamics only locally asymptotically stable equilibria are kept, and for linear dynamics all stable equilibria are kept.[2] Every guard condition that allows executions to leave q is then

[2]These conditions are because we follow Lyapunov's indirect method, where we can only ensure that the stability of the nonlinear system is the same as that of the linearised system if the equilibrium is asymptotically stable. That is, the real part of the eigenvalues are strictly less than zero.

tested to see if $\overline{x}$ can satisfy it, and if so then $\overline{x}$ is disregarded. We are then left only with equilibria of the type of Definition 5.17, with all these equilibria locally stable or asymptotically stable.

For each hybrid-space equilibrium point, the method then proceeds to find a Lyapunov function for the linearised dynamics about this equilibrium point, which is then optimised to be a Lyapunov function V for the nonlinear dynamics. This creates a invariant set in the continuous space $W_V \subseteq \mathcal{X}$ which traps all trajectories close to the equilibrium $\overline{x} \in \mathcal{X}$ which exists in location $q \in Q$. We can add this continuous invariant set to a hybrid invariant set by $W = W \cup (q \times W_V)$ and this will create a larger hybrid invariant set W.

Repeating this process in every location $q \in Q$ around each of the stable equilibrium points in this location gives us a hybrid invariant set which, if reached, disproves the liveness property. Therefore, the deadness property is reaching the hybrid invariant set W.

5.6 Implementing the method to disprove liveness

We will now discuss how we have implemented Algorithm 5.1 to automatically find dead sets and how deadness can then be proved. We show how each part has been implemented and also discuss other methods for achieving the same ends. Most of the implementation has been made using MATLAB and its Symbolic Math Toolbox.

The input to the implementation is given as a MATLAB structure describing the hybrid automaton in terms of the properties of each location (domain, dynamics, initial set and live set) and the properties of each transition (guard and reset).

5.6.1 Finding the stable equilibria in each location of the hybrid automaton (lines 3–21)

It is a relatively simple task to find the equilibria of a set of dynamics, as we must just solve the equation $f_q(x) = 0$ for values of x. There are numerical methods for finding such equilibria, which are typically iterative optimisation algorithms like steepest descent minimisation methods. As we would like to find all of the equilibrium points it makes sense to solve the equations symbolically, as would be done by hand.

For this purpose we have used the Symbolic Math Toolbox from MATLAB, but other symbolic mathematics engines could be used (Mathematica or Maple, for instance).

For each location q we solve the equation $f_q(x) = 0$ to get possible equilibria $\overline{x} \in EQ_q$ using the symbolic *solve* function from the Symbolic Math Toolbox. We use the symbolic *subs* function to substitute the found equilibria into (1) the domain equation, to check the equilibrium is in the domain, (2) the guard conditions, to check it is not in a guard, and (3) the live set L, to check it is not a desired point. We then test the eigenvalues of the Jacobian of the dynamics at $\overline{x}$ to find the stability of the point, and remove those not locally stable — this uses the symbolic MATLAB functions *Jacobian* and *eig*, along with a linearity test for the dynamics. We get left with sets of locally stable equilibria EQ_q for each $q \in Q$.

5.6.2 Creating an invariant set around a stable equilibrium point (lines 22–23).

The method we use for creating an invariant set around a given locally stable equilibrium point is due to Davison and Kurak [1971], and relies on finding a quadratic Lyapunov function for this equilibrium point. There are various other methods available for finding Lyapunov functions and regions of attraction numerically in continuous systems (for instance Flashner and Guttalu [1988], Ohta et al. [1993], Ratschan and Smaus [2006], Vandenberghe and Boyd [1993]), but we selected the method by Davison and Kurak [1971] because it can deal with arbitrary nonlinear systems to create a simple quadratic Lyapunov function. The method is made up of four parts:

1. A Lyapunov function is found for the system when it is linearised about the equilibrium point;

2. This function for the linearised dynamics is given an arbitrarily small radius, and then the radius is optimised so it has as large an area as possible;

3. The quadratic Lyapunov function itself is optimised to have the largest area possible for the nonlinear dynamics using the result of step 2 as the starting point;

4. The resulting function is checked with a very fine mesh to make sure it is actually

a Lyapunov function, and if not we start from step 2 again, using a finer mesh of vectors for the optimisation.

We now give a closer look at the implementation for each of these steps.

Step 1. Solving for a Lyapunov function of a linear vector field $\dot{x} = Ax + b$ involves solving the *Lyapunov equation*, $A^T P + PA = -Q$, where Q is chosen and positive definite, and P is positive definite. This gives us a Lyapunov function for this linearisation of the form $V(x) = (x - \overline{x})^T P(x - \overline{x})$. There are many algorithms available to solve such Lyapunov equations, for example Hammarling [1982], and the `Lyap` method from the Control System Toolbox in MATLAB. We use the highly efficient method `f08qh` from the NAG toolbox for MATLAB in our implementation.

Step 2. Given this $V(x)$ from the first step, we start with $V(x) < \epsilon$ for some small ϵ (we have used $\epsilon = 10^{-5}$ in our implementation). We then used `fminsearchcon`, a constrained optimisation method obtained from the MATLAB file exchange, to expand this to a larger set $V(x) < \epsilon_1$. This optimisation is made subject to the larger set still being a Lyapunov function for the nonlinear dynamics, and not intersecting the guards or the live set L. This optimisation is achieved through using randomised vectors spanning the space, and so the set found will be slightly different every time the algorithm is run.[3]

Step 3. We can then try to find a more optimal trapping set created by a new Lyapunov function $V_1(x) = (x - \overline{x})^T P_1(x - \overline{x}) < 1$, using $V(x) < \epsilon_1$ as a starting point for the optimisation. The optimisation is made over the area of the set enclosed by the equation $V_1(x) < 1$, with the same constraints as for step 2.

Step 4. The boundary $V_1(x) = 1$ is then tested to make sure that the flow across the boundary is always inwards, and it is also checked that it does not intersect with the guard conditions or live set. This is achieved by testing the values of the functions at every point of a very fine mesh over the surface.

[3]The set can be checked to be a valid invariant by checking the value of the derivative on the boundary using a theorem prover such as MetiTarski [Akbarpour and Paulson, 2010], although this has not been implemented.

We should briefly discuss where numerical errors could creep in to this method. The first place is at the beginning of step 2 of the optimisation, where the Lyapunov function is given a very small radius. The correctness of this step depends on $V(x) = \epsilon$ being a trapping set for the nonlinear system: if $\dot{V}(x) > 0$ anywhere on the surface $V(x) = \epsilon$ then this is not a trapping set for the nonlinear dynamics. We have selected a very small epsilon which should be fine for most systems, but automatic checks can be built to make sure that this initial set is actually an invariant set for the nonlinear dynamics, and ϵ can be reduced accordingly if not.

The second place that numerical errors can affect the method is in the optimisation, where we rely on having a mesh of vectors over the surface of the Lyapunov function to make sure we are optimising within the set where the function creates a invariant set for the nonlinear dynamics. If there are not enough vectors then important bumps and spikes in the flow could be missed and the optimisation could continue even though the condition $\dot{V}(x) \leq 0$ has been breached. However, this problem is reduced by step 4 of the method, where the optimisation is repeated with more vectors if it does not create a suitable Lyapunov function after one iteration. Davison and Kurak [1971] suggest suitable numbers of vectors to start the optimisation with for different dimensional systems, which we have used in our implementation.

5.6.3 Proving the deadness property

In order to prove the deadness property, we need to find at least one execution of the hybrid automaton which eventually reaches one of the dead sets. This is effectively a *satisfiability* (SAT) problem on the hybrid system: SAT problems are of the form 'given a system and a logical specification, is there an execution of the system which satisfies the specification?' We have the SAT problem 'given our hybrid automaton H and the specification 'eventually reach a dead set', is there an execution of H which satisfies the specification?'

In the domain of computer science, specialised solvers are used for finding a solution of a SAT problem, and such solvers are many and varied in type. In the domain of hybrid systems and real-arithmetic SAT solving, there are only a few solvers available, including iSAT [Fränzle et al., 2007] and ABsolver [Bauer et al., 2007]. Of these, iSAT is the most well developed for hybrid systems and has a easy-to-use input method,

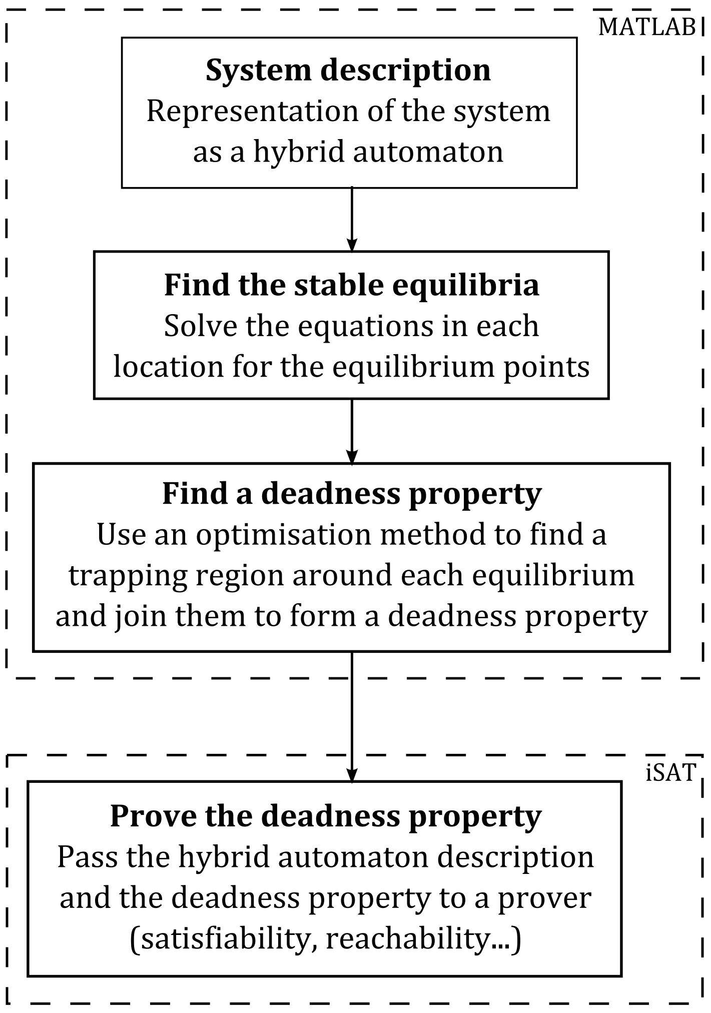

MATLAB
System description
Representation of the system as a hybrid automaton
Find the stable equilibria
Solve the equations in each location for the equilibrium points
Find a deadness property
Use an optimisation method to find a trapping region around each equilibrium and join them to form a deadness property
iSAT
Prove the deadness property
Pass the hybrid automaton description and the deadness property to a prover (satisfiability, reachability...)

Figure 5.4: An overview of the method to disprove liveness by proving deadness in a hybrid system.

so we make use of this solver in this chapter. It is based on bounded model checking [Biere et al., 1999], relying on a fixed time-step discretisation of the dynamics of the hybrid automaton, so that on the k-th iteration iSAT will look at all executions of length $k\Delta t$, where Δt is the length of the time step.

The input to iSAT is a file containing a representation of the hybrid automaton, with a fixed-step discretisation of the continuous dynamics. The file also defines a target set for the automaton executions — for our case it will be a mixed logic and inequality constraint specifying the dead sets that have been calculated by the implementation of Algorithm 5.1. The possible outputs for iSAT are 'unsatisfiable', 'satisfiable' (with the satisfying solution returned), and 'unknown' (with a candidate solution returned). The candidate solution offered by iSAT when it returns with status 'unknown' could still be a satisfying execution, but the 'highly incomplete deduction calculus' [AVACS H1/2 iSAT Developer Team, 2010] means that iSAT cannot always decide whether it is.

Using a fixed time-step discretisation of the dynamics can create problems with fast-changing dynamical systems (whether continuous or hybrid), as a large numerical error can be built up as iSAT approximates the flow. This is a problem which can only be helped by using fewer or smaller time steps — we will look at some examples which work in the example in the next section.

5.7 The drillstring example

We now return to the example that we discussed in Section 5.1. This is the model of the drillstring, which has three possible long-term behaviour patterns, driven by two equilibrium points that can be locally attractive. The 'strength' of the attractivity of each equilibrium point is dependent on two parameters: the driving torque at the top of the drill u, and the weight on the bit W_{ob}. We wish to tend to the locally attractive positive velocity equilibrium so that the drill is constantly making progress into the ground.

We consider the hybrid automaton model of the drillstring given in Fig. 5.5. All the continuous dynamics have the form of (5.1), with only the term T_{f_b} changing with the location as defined below (see (5.2) and (5.3) for definitions of f_b and u_{eq}):

$$T_{f_b}(\boldsymbol{x}) = \begin{cases} f_b(x_3) & \text{if } q = q_1, \\ -f_b(x_3) & \text{if } q = q_2, \\ u_{eq}(\boldsymbol{x}) & \text{if } q = q_3. \end{cases}$$

For ease of notation, we also use the following three sets:

$$\begin{aligned} S_1 &= \{\boldsymbol{x} \in \mathbb{R}^3 : x_3 > 0 \vee (x_3 = 0 \wedge u_{eq}(\boldsymbol{x}) > T_{sb})\}, \\ S_2 &= \{\boldsymbol{x} \in \mathbb{R}^3 : x_3 < 0 \vee (x_3 = 0 \wedge u_{eq}(\boldsymbol{x}) < -T_{sb})\}, \\ S_3 &= \{\boldsymbol{x} \in \mathbb{R}^3 : x_3 = 0 \wedge (|u_{eq}(\boldsymbol{x})| \leq T_{sb})\}. \end{aligned}$$

We will look at disproving the property of global attractivity of the positive velocity equilibrium. As discussed in Section 5.3, global attractivity being true implies that the

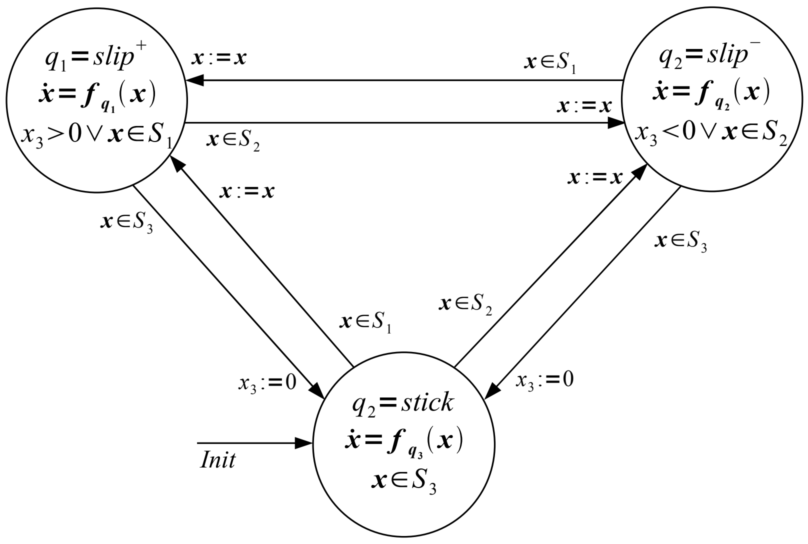


Figure 5.5: The hybrid automaton for the drillstring example [Navarro-López and Carter, 2011].

liveness property 'a small sphere L around the positive velocity equilibrium is eventually reached' is also true. In fact this liveness property is an inevitability property, so we can use Algorithm 5.1 to find deadness properties in order to prove that, for some selection of parameters, we will never get close to the desired equilibrium. This will imply that the equilibrium is not globally attractive. We have tested our prototype implementation of the algorithm on this example with values of $W_{ob} = 59000\,\mathrm{N}$ and $u = 6000\,\mathrm{N\,m}$: from simulation we expect that the executions will have the 'stuck' behaviour for this value of W_{ob} (i.e. be attracted to the undesired equilibrium).

The algorithm analyses each location in turn, starting with q_1, the positive velocity location. We give a summary of the results of the implementation.

- Analyse $f_{q_1}(\boldsymbol{x}) = 0$ in the set $Dom(q_1) = S_1$. Find two equilibrium points at $\overline{x}_1 = \overline{x}_3 > 0$ as the two solutions of the equations $u - (c_r + c_b)\overline{x}_3 - f_b(\overline{x}_3) = 0$, and $\overline{x}_2 = (u - c_r\overline{x}_3)/k_t$. The first one found is at $[2.07, 5.94, 2.07]^T$ and is unstable in the linearisation, and the second one is at $[1.62, 6.17, 1.62]^T$ and is the desired equilibrium inside the specified live set L, so move on.

- Analyse $f_{q_2}(\boldsymbol{x}) = 0$ in the set $Dom(q_2) = \{\boldsymbol{x} \in \mathcal{X} : \boldsymbol{x} < 0 \vee (x_3 = 0 \wedge u_{eq}(\boldsymbol{x}) < -T_{sb})\}$. Find no solutions of the equation with the current parameters, and so move on.

- Analyse $f_{q_3}(\boldsymbol{x}) = 0$ in the set $Dom(q_3) = \{\boldsymbol{x} \in \mathcal{X} : \boldsymbol{x} = 0 \wedge (|u_{eq}(\boldsymbol{x})| < T_{sb})\}$.

Find an equilibrium point at $\overline{x}_1 = \overline{x}_3 = 0$ and $\overline{x}_2 = u/k_t = 6.96$. It is not the desired equilibrium, but it is stable for the chosen parameter values. It is not in any of the guards from location q_3 for our parameters, so we want to find an invariant set around it to use as a dead set.

- In location q_3, we have 'lost a dimension', due to $x_3 = 0$ always being true, with dynamics $\dot{x}_3 = 0$. Our implementation automatically detects this by looking at the conditions for the domain (in particular $x_3 = 0$), and as the dynamics are linear we can allow the resulting zero eigenvalue whilst keeping stability in the x_1-x_2 plane. Let us write $\boldsymbol{y} = (x_1, x_2)^T$ for the new variables after projection into the $x_3 = 0$ plane. The implementation then solves the Lyapunov equation for the restricted dynamics of location q_3 to get a first guess at a Lyapunov function:

$$V(\boldsymbol{y}) = (\boldsymbol{y} - \overline{\boldsymbol{y}})^T \begin{pmatrix} 6.15 & 1.23 \\ 1.23 & 2.84 \end{pmatrix} (\boldsymbol{y} - \overline{\boldsymbol{y}}),$$

  where $\overline{\boldsymbol{y}} = (0, 6.96)^T$. Although this function would not prove attractivity of the equilibrium point in 3-dimensions, we can use it to provide a trapping set for the executions, which is what we are actually interested in.

- Starting from $V(\boldsymbol{y}) < 10^{-5}$, extend this to a larger set by using the optimisation method of Davison and Kurak [1971] which we have implemented, taking the domain of q_3 and the non-satisfaction of the guards on edges out of q_3 as extra constraints. The method gives an invariant set of

$$W_{q_3} \equiv \left\{ \boldsymbol{y} : (\boldsymbol{y} - \overline{\boldsymbol{y}})^T \begin{pmatrix} 0.652 & 0.110 \\ 0.110 & 0.410 \end{pmatrix} (\boldsymbol{y} - \overline{\boldsymbol{y}}) < 1 \right\},$$

  in location q_3, which forms a hybrid invariant set of $W = \{(q, x) \in \mathcal{Z} : q = q_3,\ (x_1, x_2)^T \in W_{q_3},\ x_3 = 0\}$ to be the dead set for the hybrid automaton.[4]

This algorithm has created a deadness condition (reaching an invariant set in q_3), which, if we can find at least one execution from the initial set *Init* to satisfy it, will disprove the liveness property of all executions of the hybrid automaton tending to the desired equilibrium.

[4]When these results are replicated, slightly different results will be obtained due to the randomised allocation of vectors that are used in the optimisation.

We have attempted to find a satisfying execution using iSAT [Fränzle et al., 2007]. For the drillstring example, iSAT returns 'unknown' for all initial conditions tried, however close or far away from the dead set they are. This may be to do with the fact that the dead set we want to reach is in a lower-dimensional subspace, and it is therefore difficult to numerically check that the candidate solution offered is allowable. However, in some cases where iSAT returns 'unknown', the candidate solution offered actually is a satisfying dead execution, which we can check by substituting the last time point of the candidate solution into the equation for the dead set which we are attempting to satisfy. These cases that return valid candidate solutions are typically achieved by choosing an initial condition close to the dead set (in terms of time separation), which allows us to specify a much smaller time step.

For instance, choosing an initial condition of $x_1 = 0.09$, $x_2 = 8.5$, $x_3 = 0.0002$ and $q = q_1$, with time step $\Delta t = 0.01$ s iSAT finds a dead execution in 11 steps (1 discrete transition, then 10 time steps), ending at a set of points $x_1 \in (0.02580196, 0.02580197)$, $x_2 \in (8.50609831, 8.50609832)$, $x_3 \in [0, 0]$ and $q = q_3$. Substituting these values into W, the equation for the hybrid invariant set, we find that these last points satisfy W, and so we can conclude that this is a dead execution. Therefore, the hybrid automaton H is dead for this initial condition. This is an interesting result, because (with these parameters) we know that the drillstring cannot recover from such a dead execution, and so cannot obtain the desired behaviour of getting close to the desired equilibrium. Consequently, we know with certainty that there are some initial states which we should not use to start execution of the system, and can avoid them in order to satisfy the liveness property that we reach the desired equilibrium.

For a more general result, we specify an initial set by taking intervals around the point initial condition given above, so that $x_1 \in (0.08, 0.1)$, $x_2 \in (8, 9)$, $x_3 \in (0.0001, 0.0003)$ and $q = q_1$. Then, with the same time step, iSAT finds the shortest execution (in number of steps) which starts in this initial set and reaches the set W: actually iSAT finds an execution which makes only 1 discrete transition and no time steps, starting anywhere in the set $x_1 \in [0.09749999, 0.09859424]$, $x_2 \in [8.02374976, 9.0240449]$, $x_3 \in [0.00009999, 0.00030001]$, $q = q_1$ and ending anywhere in the set $x_1 \in [0.09749999, 0.09875001]$, $x_2 \in [8.02374976, 9.0240449]$, $x_3 \in [0, 0]$, $q = q_3$, which is inside the dead set W. Again this is returned as 'unknown' but checking the result by substitution into W shows that it is a dead execution, and we can conclude that the hybrid automaton H is dead for this set of initial states. This is even more helpful than the previous result, as we have obtained a particularly bad execution which can satisfy the deadness condition with one discrete transition, which is a initial point we should definitely avoid.

5.8 Conclusions and future work

In this chapter we have defined a new logical property called deadness, which makes use of the known dynamics of a hybrid automaton to disprove a related liveness property. We have defined an algorithm to find such deadness properties automatically when proving inevitability properties, a class of liveness which says that eventually a set is reached. The algorithm uses invariant sets around undesired equilibria as 'dead sets' which disprove liveness if they can be proven to be reached. Since there are methods available to implement all of the steps of the algorithm, we have made a prototype implementation in MATLAB and iSAT (a hybrid system SAT solver) and have tested it on a model of an oilwell drillstring. A dead set was calculated for this model, and the deadness condition was proven to hold for selected initial conditions using iSAT, although the resulting executions had to be checked manually.

We have not solved the problem of finding dead sets for every type of behaviour that can occur in a hybrid automaton, so future work will focus on three different types of invariant-creating behaviour that can occur in a hybrid system. Mainly:

1. Equilibrium points where the executions can keep jumping between locations whilst still at the same continuous point. For this kind of continuous-space equilibrium, using common or multiple Lyapunov functions [Branicky, 1994, Liberzon, 2003] for the hybrid automaton may be more suitable for finding dead sets.

2. Trapping sets caused by stable periodic orbits. We have not discussed this kind of trapping set, as it is difficult to find periodic orbits in an automated way. However, there are methods which find such periodic orbits and their trapping sets [Guckenheimer, 2001], which could potentially be used to find more deadness conditions.

3. Convergence to an equilibrium point caused by discrete transitions (like that which occurs in switching controllers for electronic circuits [Hejri and Mokhtari, 2008]). This kind of stability is caused only by the existence of the discrete transitions, and is characterised by oscillating executions of the system which may (or may not) converge to an equilibrium.

Another strand of future work is to improve the methods of proving such deadness properties, in particular through better SAT solvers for hybrid systems, or through using already developed reachability algorithms.

Chapter 6

Conclusions and future work

This thesis has proposed new contributions in the area of proving liveness properties in hybrid dynamical systems. The methods that we have proposed can be broadly classed as *dynamically-driven formal verification methods*, as we make use of information we can gather from the dynamics of the system to guide the formal verification. As the number of results that previously existed in this area is small, the scope for research is still vast, but the results of this thesis should provide a springboard for future work.

The key contributions of this thesis are in two main classes, the first to propose and investigate methods for proving liveness properties, and the second to define the concept of deadness and demonstrate how it can be used to disprove liveness. Together these results give a new framework for consideration of liveness properties in hybrid dynamical systems.

6.1 Summary of conclusions

In Chapter 3 we considered an algorithm proposed by Maler and Batt [2008], which defines an abstraction method for a continuous dynamical system based on splitting the state space of the system into boxes (hyper-rectangles). The abstraction obtained from this algorithm is a timed-automaton which over-approximates the original continuous system in terms of the number of trajectories included. This over-approximation means that we can use the abstraction to show that all trajectories of the continuous dynamical system reach a desired set, thus proving an inevitability property (a class of liveness). However, Maler and Batt did not propose how to split the state space of the dynamical system into boxes, but only showed how to make an over-approximating abstraction from some given splitting.

Given the great unexplored potential of the method of Maler and Batt for proving inevitability, we looked at what can prevent it creating useful inevitability-proving abstractions. Using this analysis, we proposed a method for creating a splitting of the state space for a class of linear continuous dynamical systems, and proved that it causes the resulting timed-automaton abstraction to always prove inevitability. In this way we showed that the special class of linear continuous systems we considered are equivalent to the timed-automaton abstraction with respect to proving inevitability. This result of equivalence with respect to inevitability is very interesting, as it holds out hope for abstraction methods that are capable of proving liveness in more general systems.

As the aims of this thesis are working towards proving liveness in hybrid dynamical systems, we next turned our attention to how this abstraction method could extend prove inevitability in piecewise-linear dynamical systems. In Chapter 4 we proposed the extension to the method of Maler and Batt [2008] to create timed-automaton abstractions of piecewise-linear dynamical systems, which preserved the key property of over-approximation of the number of trajectories included in the system. We then proposed a splitting method for piecewise-linear systems where each subsystem is of the special form of Chapter 3.

The splitting method proposed for this class of piecewise-linear systems was shown to give a timed-automaton abstraction which proved inevitability of the original system in the one-dimensional and two-dimensional cases. This result shows that we have equivalence with respect to inevitability, like in the continuous system case, which again holds out some hope for abstraction methods capable of proving liveness in systems with interactions of continuous dynamics and discrete transitions. We demonstrated some problems that occur with the proposed algorithms in the cases of three-dimensional and higher-dimensional systems, and proposed solutions which aim to help the user as much as possible when the abstraction does not prove inevitability of all the trajectories coming from the originally defined initial set. Used in more complex systems, this information could help a user to see why their system may not satisfy the inevitability property, and to redesign the system accordingly.

Picking up on the idea of showing when a system does not satisfy a liveness property, we defined a logical property called *deadness* in Chapter 5. This property is a new concept in formal verification of liveness in hybrid systems, and aims to makes use of the known dynamics of a hybrid system to disprove a liveness property without the need to find an infinite length execution of the hybrid system. Using deadness properties in the process of proving liveness is a new approach to proving such properties, but showing that a liveness property is not satisfied can be just as useful as showing that it is, as a definitive answer is usually the most helpful when analysing a system.

We also showed that the property of deadness is not just an abstract concept, but proposed a method to find such deadness properties automatically for given inevitability properties. The algorithm uses the dynamics of the system, along with key ideas of Lyapunov stability theory to find invariant sets around undesired equilibria which we call ‘dead sets’. Once a trajectory reaches a dead set, it is trapped there forever, and so we know that reaching a dead set will disprove the desired liveness property.

6.2 Limitations of the thesis

The work presented in this thesis is a first step towards automated verification of liveness properties in hybrid dynamical systems, and so there are limitations on the results. We will now outline these limitations and the key assumptions made in this work.

Firstly, in Chapters 3 and 4 there is a degree of restriction on the type of dynamical systems considered, as they are special classes of linear or piecewise-linear systems. Most of the systems where we would require the formal verification of a liveness property are those which we cannot analyse with the classical Lyapunov stability theory — typically nonlinear systems — and therefore cannot be analysed using the method in its current form.

The second key assumption, also found in Chapters 3 and 4, is that of a finite state space for the timed-automaton abstraction method. This assumption is essential to make a useful timed-automaton abstraction, as an infinitely-sized state space would give us infinite time bounds on the clocks in the timed-automaton abstraction. Most dynamical systems can be restricted to a very large, but finite, state space, and so this should not be a problem in general. However, there may be systems where an infinite state space is required, in which case the use of this method may be limited.

The third key assumption is made in the method proposed in Chapter 5 for finding dead sets in hybrid systems. The method proposed uses information about the equilibrium points of each location of the hybrid automaton separately, which is easy to obtain. However, the inherent nature of most hybrid systems relies on interactions between different subsystems, and so we could be missing a lot of useful deadness properties by not considering this more complex behaviour. These were not considered because new methods are required to automatically detect these types of behaviour, which was outside the scope of this thesis.

Finally, we only consider inevitability properties in the practical methods of this thesis, and not the whole class of liveness. The wider class of liveness encompasses a very wide range of properties, which, in dynamical systems, will require different proving methods to those for inevitability.

We see these limitations as possible areas for future work, and so in the next section we will draw together all the future work we foresee as a result of this thesis.

6.3 Future work

There is inevitably room for future work based on this thesis, with some work to address the limitations set out in Section 6.2 and other work to extend the results that have been obtained. We will discuss the main areas in which we can see potential by considering the results of each chapter.

In the continuous case of Chapter 3, future work on the splitting method is to extend it for use in more general continuous systems. There are various problems to be overcome with such systems, one of which will be the termination of the splitting method, as the current method only terminates because of the special dynamics involved. Hence, part of any future work is to revise the splitting method to be more useful for more general linear systems and also nonlinear systems. Initially, making an investigation into exactly why the systems of the special class can have these complete abstractions will guide our thoughts on how to extend this to more general systems. In addition, it is not realistic to have complete abstractions for general continuous dynamical systems, that is, we would not expect all timed-automaton abstractions to always prove inevitability when it is true in the original system. Therefore, future work should also focus on what can and cannot be achieved by a splitting along constant variable lines in a general dynamical system.

Looking in another direction, it may be more appropriate to look at key surfaces in the system to use to divide the space, rather than simply constant variable lines defining hyper-rectangles. This direction has already started to be explored by Sloth and Wisniewski [2013], who look at using Lyapunov functions to create the splitting of the state space of the system.

Moving on to look at the splitting method in Chapter 4, we see that the work falls into two categories:

1. The first area of future work is to improve the method for the special class of systems to make it work for three-dimensional and higher-dimensional systems. The main focusses for this need to be the reasons why the method in this chapter has issues with such systems. For instance, the cases of Zeno pairs and cycles of states caused by the layout of the space may have their abstractions improved by not making splits at the subsystem boundaries, which was one assumption we made quite early on (Sec. 4.3.2). There may also be problems around the occurrence of boxes with infinite time which are not removed by the splitting method as it stands. The proof of removal of infinite time in the continuous case relied on every inseparable infinite time box having a neighbouring box with greater offset which also has infinite time. In piecewise-linear systems we cannot guarantee that such a neighbouring box exists, and so we may not be able to remove the infinite time on a box. This needs to be investigated further.

2. The second area for future work is to extend this method to create useful abstractions for more complex classes of piecewise-linear, piecewise-continuous, and hybrid systems. As there are so many open ends for three-dimensional and higher-dimensional systems of the special form at the moment, the best immediate area of research is to look at one-dimensional and two-dimensional piecewise-continuous systems with more general dynamics in the subsystems, to see how this method extends to those cases.

Considering the method of Chapter 5, which finds deadness conditions by finding dead sets, future work should focus on the other types of behaviour which can cause invariants in the system. There are three main types of invariant-creating behaviour that can occur in a hybrid system which we have not considered in this thesis:

1. Equilibrium points where the executions can keep jumping between locations whilst still at the same continuous point. For this kind of continuous-space equilibrium, the methods of common or multiple Lyapunov functions [Branicky, 1994, Liberzon, 2003] for the hybrid automaton may be more suitable for finding dead sets.

2. Trapping sets caused by stable periodic orbits. We have not discussed this kind of trapping set, as it is difficult to find periodic orbits in an automated way. However, there are methods which find such periodic orbits and their trapping sets [Guckenheimer, 2001], which could potentially be used to find more deadness conditions.

3. Convergence to an equilibrium point caused by discrete transitions (like that which occurs in switching controllers for electronic circuits [Hejri and Mokhtari, 2008]). This kind of stability is caused only by the existence of the discrete transitions, and is characterised by oscillating executions of the system which may (or may not) converge to an equilibrium.

Another strand of future work is to improve the methods of proving such deadness properties, in particular through better SAT solvers for hybrid systems, or through using already developed reachability algorithms.

Bibliography

A. Abate, A. D'Innocenzo, G. Pola, M. D. Di Benedetto, and S. S. Sastry. The concept of deadlock and livelock in hybrid control systems. Technical report, University of California at Berkeley, 2006.

A. Abate, M. Prandini, J. Lygeros, and S. Sastry. Probabilistic reachability and safety for controlled discrete time stochastic hybrid systems. *Automatica*, 44(11):2724–2734, 2008.

A. Abate, A. D'Innocenzo, M. D. Di Benedetto, and S. S. Sastry. Understanding deadlock and livelock behaviors in hybrid control systems. *Nonlinear Analysis: Hybrid Systems*, 3:150–162, 2009.

A. Abate, J.-P. Katoen, J. Lygeros, and M. Prandini. Approximate model checking of stochastic hybrid systems. *European Journal of Control*, 16(6):624–641, 2010.

E. Ábrahám-Mumm, U. Hannemann, and M. Steffen. Verification of hybrid systems: Formalization and proof rules in PVS. In *Proceedings of the Seventh IEEE International Conference on Engineering of Complex Computer Systems*, pages 48–57, 2001.

K. Ajami, S. Haddad, and J. Ilie. Exploiting symmetry in linear time temporal logic model checking: One step beyond. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science (LNCS)*, pages 52–67. Springer, 1998.

B. Akbarpour and L. C. Paulson. Applications of MetiTarski in the verification of control and hybrid systems. In R. Majumdar and P. Tabuada, editors, *Hybrid Systems: Computation and Control (HSCC 2009)*, volume 5469 of *Lecture Notes in Computer Science (LNCS)*, pages 1–15. Springer, 2009.

B. Akbarpour and L. C. Paulson. MetiTarski: an automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.

B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21: 181–185, 1985.

B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

M. Althoff, O. Stursberg, and M. Buss. Reachability analysis of linear systems with uncertain parameters and inputs. In *Proceedings of the 46th IEEE Conference on Decision and Control*, pages 726–732, 2007.

K. Altisen and S. Tripakis. Implementation of timed automata: An issue of semantics or modeling? In P. Pettersson and W. Yi, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS 2005)*, volume 3829 of *Lecture Notes in Computer Science (LNCS)*, pages 273–288. Springer, 2005.

R. Alur. Timed automata. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science (LNCS)*, pages 8–22. Springer, 1999.

R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science (LNCS)*, pages 74–106. Springer, 1991.

R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1): 181–203, 1994.

R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993a.

R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In

R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science (LNCS)*, pages 209–229. Springer, 1993b.

R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43:116–146, 1996a.

R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996b.

R. Alur, T. A. Henzinger, and H. Wong-Toi. Symbolic analysis of hybrid systems. In *Proceedings of the 36th IEEE Conference on Decision and Control*, pages 702–707, 1997.

R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, 211(1-2):253–273, 1999.

R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.

R. Alur, S. La Torre, and P. Madhusudan. Perturbed timed automata. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control (HSCC 2005)*, volume 3414 of *Lecture Notes in Computer Science (LNCS)*, pages 70–85. Springer, 2005.

R. Alur, T. Dang, and F. Ivancic. Counter-example guided predicate abstraction of hybrid systems. *Journal of Theoretical Computer Science*, 354(2):250–271, 2006.

F. Amato, R. Ambrosino, C. Cosentino, and G. De Tommasi. Finite-time stabilization of impulsive dynamical linear systems. *Nonlinear Analysis: Hybrid Systems*, 5(1): 89–101, 2011a.

F. Amato, R. Ambrosino, G. De Tommasi, and A. Merola. Estimation of the domain of attraction for a class of hybrid systems. *Nonlinear Analysis: Hybrid Systems*, 5 (3):573–582, 2011b.

P. J. Antsaklis, J. A. Stiver, and M. D. Lemmon. Hybrid system modelling and autonomous control systems. In R. Grossman, A. Nerode, A. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science (LNCS)*, pages 366–392. Springer, 1993.

M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: a case study. In O. Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science (LNCS)*, pages 171–185. Springer, 1997.

E. Asarin, T. Dang, and O. Maler. The d/dt tool for verification of hybrid systems. In *Computer Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science (LNCS)*, pages 365–370. Springer, 2002.

AVACS H1/2 iSAT Developer Team. *iSAT Quick Start Guide*, 2010. Available online at `http://isat.gforge.avacs.org`. Accessed on 26th July 2012.

C. Baier and M. Kwiatkowska. On topological hierarchies of temporal properties. *Fundamenta Informaticae*, 41(3):259–294, 2000.

A. Baños and A. Barreiro. *Reset Control Systems.* Advances in Industrial Control. Springer, 2012.

R. H. Bartels and G. W. Stewart. Solution of the matrix equation ax + xb = c [f4]. *Communications of the ACM*, 15(9):820–826, 1972.

G. Batt, C. Belta, and R. Weiss. Model checking liveness properties of genetic regulatory networks. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science (LNCS)*, pages 323–338. Springer, 2007.

A. Bauer, M. Pister, and M. Tautschnig. Tool-support for the analysis of hybrid systems and models. In *Design, Automation and Test in Europe (DATE '07)*, pages 924–929, 2007.

G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science (LNCS)*, pages 33–35. Springer, 2004.

G. Behrmann, K. Larsen, and J. Rasmussen. Beyond liveness: Efficient parameter synthesis for time bounded liveness. In P. Pettersson and W. Yi, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS 2005)*, volume 3829 of *Lecture Notes in Computer Science (LNCS)*, pages 81–94. Springer, 2005.

A. Bemporad and M. Morari. Control of systems integrating logic, dynamics, and constraints. *Automatica*, 35(3):407–427, 1999.

J. Bengtsson and W. Yi. Timed automata: semantics, algorithms and tools. In *Lectures in Concurrency and Petri Nets: Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science (LNCS)*, pages 87–124. Springer, 2004.

A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science (LNCS)*, pages 193–207. Springer, 1999.

A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In *Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science (LNCS)*, pages 415–418. Springer, 1996.

F. Blanchini. Set invariance in control. *Automatica*, 35:1747–1767, 1999.

S. Bogomolov, C. Mitrohin, and A. Podelski. Composing reachability analyses of hybrid systems for safety and stability. In *Automated Technology for Verification and Analysis (ATVA 2010)*, volume 6252 of *Lecture Notes in Computer Science (LNCS)*, pages 67–81. Springer, 2010.

A. Bouajjani, A. Legay, and P. Wolper. Handling liveness properties in (ω-) regular model checking. *Electronic Notes in Theoretical Computer Science*, 138(3):101–115, 2005.

P. Bouyer, U. Fahrenberg, K. G. Larsen, and N. Markey. Timed automata with observers under energy constraints. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC'10)*, 2010.

M. Bozga, S. Graf, and L. Mounier. IF-2.0: a validation environment for component-based real-time systems. In E. Brinksma and K. Larsen, editors, *Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science (LNCS)*, pages 630–640. Springer, 2002.

M. S. Branicky. Stability of switched and hybrid systems. In *Proceedings of the IEEE Conference on Decision and Control*, pages 3498–3503, 1994.

M. S. Branicky. *Studies in hybrid systems: modeling, analysis, and control.* PhD thesis, Massachusetts Institute of Technology, 1995.

M. S. Branicky. Multiple Lyapunov functions and other analysis tools for switched and hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):475–482, 1998.

M. S. Branicky. Introduction to hybrid systems. *Handbook of Networked and Embedded Control Systems*, pages 91–116, 2005.

M. S. Branicky, V. S. Borkar, and S. K. Mitter. A unified framework for hybrid control: model and optimal control theory. *IEEE Transactions on Automatic Control*, 43(1): 31–45, 1998.

L. M. Bujorianu. *Stochastic reachability analysis of hybrid systems.* Communications and Control Engineering. Springer, 2012.

C. Cai, A. R. Teel, and R. Goebel. Smooth Lyapunov functions for hybrid systems part I: existence is equivalent to robustness. *IEEE Transactions on Automatic Control*, 52(7):1264–1277, 2007.

C. Cai, A. R. Teel, and R. Goebel. Smooth Lyapunov functions for hybrid systems part II: (pre)asymptotically stable compact sets. *IEEE Transactions on Automatic Control*, 53(3):734–748, 2008.

L. P. Carloni, R. Passerone, A. Pinto, and A. L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1):1–193, 2006.

R. Carter and E. M. Navarro-López. Dynamically-driven timed automaton abstractions for proving liveness of continuous systems. In M. Jurdziński and D. Ničković, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS 2012)*, volume 7595 of *Lecture Notes in Computer Science (LNCS)*, pages 59–74. Springer, 2012.

R. Carter and E. M. Navarro-López. Disproving liveness in hybrid dynamical systems. *Applied Mathematics and Computation*, 2013. Manuscript submitted for publication.

A. Casagrande, C. Piazza, A. Policriti, and B. Mishra. Inclusion dynamics hybrid automata. *Information and Computation*, 206(12):1394–1424, 2008.

E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In W. Kuich, editor, *International Colloquium on Automata, Languages and Programming (ICALP 92)*, volume 623 of *Lecture Notes in Computer Science (LNCS)*, pages 474–486. Springer, 1992.

Z. Chaochen, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science (LNCS)*, pages 36–59. Springer, 1993.

A. Chutinan and B. H. Krogh. Computational techniques for hybrid system verification. *IEEE Transactions on Automatic Control*, 48:64–75, 2003.

E. M. Clarke. The Birth of Model Checking. In *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science (LNCS)*, pages 1–26. Springer, 2008.

E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC '87)*, pages 294–303, 1987.

E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

E. M. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *International Journal of Foundations of Computer Science*, 14 (4):583–604, 2003a.

E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, O. Stursberg, and M. Theobald. Verification of hybrid systems based on counterexample-guided abstraction refinement. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science (LNCS)*, pages 192–207. Springer, 2003b.

B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *Static Analysis Symposium*, volume 3672 of *Lecture Notes in Computer Science (LNCS)*, pages 87–101. Springer, 2005.

B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *Proceedings of the 34th annual ACM symposium on Principles of programming languages (POPL '07)*, pages 265–276, 2007.

B. Cook, J. Fisher, E. Krepska, and N. Piterman. Proving stabilization of biological systems. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, volume 6538 of *Lecture Notes in Computer Science (LNCS)*, pages 134–149. Springer, 2011.

J. Cortés. Discontinuous dynamical systems: A tutorial on solutions, nonsmooth analysis, and stability. *IEEE Control Systems Magazine*, pages 36–73, 2008.

E. A. Cross and I. M. Mitchell. Level set methods for computing reachable sets of systems with differential algebraic equation dynamics. In *American Control Conference*, pages 2260–2265, 2008.

W. Damm, G. Pinto, and S. Ratschan. Guaranteed termination in the verification of ltl properties of non-linear robust discrete time hybrid systems. In *Automated Technology for Verification and Analysis (ATVA 2005)*, volume 3707 of *Lecture Notes in Computer Science (LNCS)*, pages 99–113. Springer, 2005.

W. Damm, H. Dierks, S. Disch, W. Hagemann, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. *Science of Computer Programming*, 77(10-11):1122–1150, 2012.

T. Dang, C. Le Guernic, and O. Maler. Computing reachable states for nonlinear biological models. *Theoretical Computer Science*, 412(21):2095–2107, 2011.

T. X. Dang. *Vérification et synthèse des systèmes hybrides.* PhD thesis, Institut National Polytechnique de Grenoble, 2000.

Z. Dang. Pushdown timed automata: a binary reachability characterization and safety verification. *Theoretical Computer Science*, 302:93–121, 2003.

Z. Dang, T. Bultan, O. H. Ibarra, and R. A. Kemmerer. Past pushdown timed automata and safety verification. *Theoretical Computer Science*, 313(1):57–71, 2004.

E. Davison and E. Kurak. A computational method for determining quadratic Lyapunov functions for non-linear systems. *Automatica*, 7(5):627–636, 1971.

J. M. Davoren and A. Nerode. Logics for hybrid systems. *Proceedings of the IEEE*, 88:985–1010, 2000.

C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proceedings of the DIMACS/SYCON workshop on Verification and Control of Hybrid systems III*, volume 1066 of *Lecture Notes in Computer Science (LNCS)*, pages 208–219. Springer, 1996.

M. De Wulf, L. Doyen, N. Markey, and J.-F. Raskin. Robust safety of timed automata. *Formal Methods in System Design*, 33:45–84, 2008.

R. Decarlo, M. S. Branicky, S. Pettersson, and B. Lennartson. Perspectives and results on the stability and stabilizability of hybrid systems. *Proceedings of the IEEE*, 88 (7):1069–1082, 2000.

W. Denman, B. Akbarpour, S. Tahar, M. H. Zaki, and L. C. Paulson. Formal verification of analog designs using metitarski. In A. Biere and C. Pixley, editors, *Formal Methods in Computer Aided Design (FMCAD 2009)*, pages 93–100, 2009.

P. S. Duggirala and S. Mitra. Abstraction refinement for stability. In *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS 2011)*, 2011.

P. S. Duggirala and S. Mitra. Lyapunov abstractions for inevitability of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC 2012)*, pages 115–123, 2012.

E. A. Emerson. The beginning of model checking: a personal perspective. In *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science (LNCS)*, pages 27–45. Springer, 2008.

E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages, and Programming*, volume 85 of *Lecture Notes in Computer Science (LNCS)*, pages 169–181. Springer, 1980.

E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.

A. F. Filippov. *Differential equations with discontinuous right-hand sides.* Springer, 1988.

H. Flashner and R. Guttalu. A computational approach for studying domains of attraction for non-linear systems. *International Journal of Non-Linear Mechanics*, 23(4):279–295, 1988.

L. Flon and N. Suzuki. The total correctness of parallel programs. *SIAM Journal of Computing*, 10(2):227–246, 1981.

M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

G. Frehse. Phaver: Algorithmic verification of hybrid systems past HyTech. *International Journal on Software Tools for Technology Transfer (STTT)*, 10:263–279, 2008.

R. Gentilini, K. Schneider, and B. Mishra. Successive abstractions of hybrid automata for monotonic CTL model checking. In S. Artemov and A. Nerode, editors, *Logical Foundations of Computer Science*, volume 4514 of *Lecture Notes in Computer Science (LNCS)*, pages 224–240. Springer, 2007.

R. Ghosh and C. J. Tomlin. An algorithm for reachability computations on hybrid automata models of protein signaling networks. In *43rd IEEE Conference on Decision and Control (CDC)*, pages 2256–2261, 2004a.

R. Ghosh and C. J. Tomlin. Symbolic reachable set computation of piecewise affine hybrid automata and its application to biological modelling: Delta-Notch protein signalling. *Systems biology*, 1(1):170–183, 2004b.

A. Girard and G. Zheng. Verification of safety and liveness properties of metric transition systems. *ACM Transactions on Embedded Computing Systems*, 11(S2):54:1–54:23, 2012.

A. Girard, A. A. Julius, and G. J. Pappas. Approximate simulation relations for hybrid systems. *Discrete Event Dynamic Systems*, 18(2):163–179, 2008.

P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem.* PhD thesis, University of Liege, 1994.

R. Goebel, R. G. Sanfelice, and A. R. Teel. Hybrid dynamical systems: robust stability and control for systems that combine continuous-time and discrete-time dynamics. *IEEE Control Systems Magazine*, 29:28–93, 2009.

S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science (LNCS)*, pages 72–83. Springer, 1997.

L. Grüne. Subdivision techniques for the computation of domains of attractions and reachable sets. In *Proceedings of Nonlinear Control Systems (NOLCOS 2001)*, pages 762–767, 2001.

J. Guckenheimer. Computing periodic orbits. In J. L. Lumley, editor, *Fluid Mechanics and the Environment: Dynamical Approaches*, volume 566 of *Lecture Notes in Physics*, pages 107–119. Springer, 2001.

H. Guéguen and J. Zaytoon. On the formal verification of hybrid systems. *Control Engineering Practice*, 12:1253–1267, 2004.

W. M. Haddad and V. Chellaboina. Dissipativity theory and stability of feedback interconnections for hybrid dynamical systems. *Mathematical Problems in Engineering*, 7(4):299–335, 2001.

H. Haimovich and M. M. Seron. Componentwise ultimate bound and invariant set computation for switched linear systems. *Automatica*, 46(11):1897–1901, 2010.

N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. *Lecture Notes in Computer Science*, 864:223–237, 1994.

S. J. Hammarling. Numerical solution of the stable, non-negative definite Lyapunov equation. *IMA Journal of Numerical Analysis*, 2(3):303–323, 1982.

K. X. He and M. D. Lemmon. Lyapunov stability of continuous-valued systems under the supervision of discrete-event transition systems. In *Hybrid Systems: Computation and Control (HSCC'98)*, volume 1386 of *Lecture Notes in Computer Science (LNCS)*, pages 175–189. Springer, 1998.

W. P. M. H. Heemels, M. K. Çamlıbel, A. J. van der Schaft, and J. M. Schumacher. *On the Existence and Uniqueness of Solution Trajectories to Hybrid Dynamical Systems*, pages 391–422. Springer, 2003.

R. Heinrich, B. G. Neel, and T. A. Rapoport. Mathematical models of protein kinase signal transduction. *Molecular Cell*, 9(5):957–970, 2002.

M. Hejri and H. Mokhtari. Global hybrid modeling and control of a buck converter: a novel concept. *International Journal of Circuit Theory and Applications*, 37:968–986, 2008.

T. A. Henzinger. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

T. A. Henzinger. Hybrid automata with finite bisimulations. In *International Conference on Automata, Languages and Programming (ICALP 95)*, volume 944 of *Lecture Notes in Computer Science (LNCS)*, pages 324–335. Springer, 1995.

T. A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, 1996.

T. A. Henzinger and P.-H. Ho. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):540–554, 1998.

T. A. Henzinger, Z. Manna, and A. Pnueli. Towards refining temporal specifications into hybrid systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science (LNCS)*, pages 60–76. Springer, 1993.

T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:110–122, 1997.

T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-toi. Beyond HyTech: hybrid systems analysis using interval numerical methods. In N. Lynch and B. H. Krogh, editors, *Hybrid Systems: Computation and Control (HSCC 2000)*, volume 1790 of *Lecture Notes in Computer Science (LNCS)*, pages 130–144. Springer, 2000.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM symposium on Principles of programming languages (POPL'02)*, pages 58–70, 2002.

O. Hijab. *Introduction to calculus and classical analysis.* Undergraduate Texts in Mathematics. Springer, third edition, 2011.

J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesley, 3rd edition, 2007.

K. H. Johansson, M. Egerstedt, J. Lygeros, and S. Sastry. On the regularization of Zeno hybrid automata. *Systems & Control Letters*, 38(3):141–150, 1999.

M. Johansson and A. Rantzer. Computation of piecewise quadratic Lyapunov functions for hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):555–559, 1998.

A. Kapur, T. A. Henzinger, Z. Manna, and A. Pnueli. Proving safety properties of hybrid systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 1994)*, volume 863 of *Lecture Notes in Computer Science (LNCS)*, pages 431–454. Springer, 1994.

F. Kerber and A. van der Schaft. Compositional analysis for linear control systems. In *Hybrid Systems: Computation and Control (HSCC 2010)*, pages 21–30, 2010.

H. K. Khalil. *Nonlinear Systems*. Prentice Hall, 3rd edition, 2002.

E. Kindler. Safety and liveness properties: a survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.

F. Klaedtke, S. Ratschan, and Z. She. Language-based abstraction refinement for hybrid system verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2007)*, volume 4349 of *Lecture Notes in Computer Science (LNCS)*, pages 151–166. Springer, 2007.

X. D. Koutsoukos, K. X. He, M. D. Lemmon, and P. J. Antsaklis. Timed petri nets in hybrid systems: stability and supervisory control. *Discrete Event Dynamic Systems*, 8:137–173, 1998.

S. Kowalewski. Introduction to the analysis and verification of hybrid systems. In *Modelling, Analysis, and Design of Hybrid Systems*, volume 279 of *Lecture Notes in Control and Information Sciences (LNCIS)*, pages 153–171, 2002.

R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2:255–299, 1990.

M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. *Information and Computation*, 205(7):1027–1077, 2007.

M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science (LNCS)*, pages 585–591. Springer, 2011.

S. Lafortune and C. G. Cassandras. *Introduction to discrete event systems*. Springer, 2nd edition, 2008.

L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2), 1977.

L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

S. Lasota and I. Walukiewicz. Alternating timed automata. *ACM Transactions on Computational Logic*, 9(2):1–27, 2008.

D. Lester and P. Gowland. Using PVS to validate the algorithms of an exact arithmetic. *Theoretical Computer Science*, 291(2):203–218, 2003.

D. R. Lester. The worlds shortest correct exact real arithmetic program? *Information and Computation*, 216:39–46, 2012.

D. Liberzon. *Switching in Systems and Control*. Birkhäuser, 2003.

D. Liberzon and A. S. Morse. Basic problems in stability and design of switched systems. *IEEE Control Systems Magazine*, 19(5):59–70, 1999.

A. M. Lyapunov. *The general problem of the stability of motion*. PhD thesis, Moscow University, 1892. Reprinted in English in the International Journal of Control, 55(3), 1992.

O. Maler and G. Batt. Approximating continuous systems by timed automata. In J. Fisher, editor, *Formal Methods in Systems Biology (FMSB 2008)*, volume 5054 of *Lecture Notes in Computer Science (LNCS)*, pages 77–89. Springer, 2008.

Z. Manna and H. B. Sipma. Deductive verification of hybrid systems using STeP. In *Transformation-Based Reactive Systems Development*, volume 1231 of *Lecture Notes in Computer Science (LNCS)*, pages 22–43. Springer, 1997.

A. S. Matveev and A. V. Savkin. *Qualitative theory of hybrid dynamical systems.* Birkhauser, 2000.

I. Mitchell and C. J. Tomlin. Level set methods for computation in hybrid systems. In N. A. Lynch and B. H. Krogh, editors, *Hybrid Systems: Computation and Control (HSCC 2000)*, volume 1790 of *Lecture Notes in Computer Science (LNCS)*, pages 310–323. Springer, 2000.

I. M. Mitchell. Comparing forward and backward reachability as tools for safety analysis. In *HSCC 2007*, volume 4416 of *Lecture Notes in Computer Science (LNCS)*, pages 428–443. Springer, 2007.

S. Mitra. *A verification framework for hybrid systems.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2007.

C. Mitrohin and A. Podelski. Composing stability proofs for hybrid systems. In U. Fahrenberg and S. Tripakis, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS 2011)*, volume 6919 of *Lecture Notes in Computer Science (LNCS)*, pages 286–300. Springer, 2011.

E. Mojica-Nava, N. Quijano, N. Rakoto-Ravalontsalama, and A. Gauthier. A polynomial approach for stability analysis of switched systems. *Systems & Control Letters*, 59(2):98–104, 2010.

E. M. Navarro-López. Hybrid modelling of a discontinuous dynamical system including switching control. In *2nd IFAC Conference on Analysis and Control of Chaotic Systems (CHAOS09)*, 2009.

E. M. Navarro-López and R. Carter. Hybrid automata: An insight into the discrete abstraction of discontinuous systems. *International Journal of Systems Science. Special issue on Variable Structure Systems Methods for Control and Observation of Hybrid Systems*, pages 1883–1898, 2011.

E. M. Navarro-López and D. Cortés. Avoiding harmful oscillations in a drillstring through dynamical analysis. *Journal of Sound and Vibration*, 307:152–171, 2007.

J. Oehlerking and O. Theel. Decompositional construction of Lyapunov functions for hybrid systems. In R. Majumdar and P. Tabuada, editors, *Hybrid Systems: Computation and Control (HSCC 2009)*, volume 5469 of *Lecture Notes in Computer Science (LNCS)*, pages 276–290. Springer, 2009.

Y. Ohta, H. Imanishi, L. Gong, and H. Haneda. Computer generated Lyapunov functions for a class of nonlinear systems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 40(5):343–354, 1993.

A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verification of linear hybrid systems. In D. L. Dill, editor, *Computer Aided Verification (CAV '94)*, volume 818 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1994.

J. Ouaknine and J. Worrell. On the decidability of metric temporal logic. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 188–197, 2005.

S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

S. Owre, J. M. Rushby, and N. Shankar. PVS: a prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE'92)*, volume 607 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 748–752. Springer, 1992.

M. Paul, H. J. Siegert, M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider, editors. *Distributed systems: methods and tools for specification. An advanced course*, volume 190 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1985.

S. Pettersson and B. Lennartson. Stability and robustness for hybrid systems. In *Proceedings of the 35th IEEE Conference on Decision and Control*, pages 1202–1207 vol.2, 1996.

A. Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *Journal of Logic and Computation*, 20(1):309–352, 2008.

A. Platzer. *Logical analysis of hybrid systems: proving theorems for complex dynamics*. Springer, 2010.

A. Platzer and J.-D. Quesel. KeYmaera: a hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.

A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th International IEEE Symposium on the Foundations of Computer Science (FOCS 1977)*, pages 46–57, 1977.

A. Podelski and S. Wagner. Model checking of hybrid systems: from reachability towards stability. In *Hybrid Systems: Computation and Control (HSCC 2006)*, volume 3927 of *Lecture Notes in Computer Science (LNCS)*, pages 507–521. Springer, 2006.

A. Podelski and S. Wagner. A method and a tool for automatic verification of region stability for hybrid systems. Technical report, Max-Planck-Institut für Informatik, 2007a.

A. Podelski and S. Wagner. Region stability proofs for hybrid systems. In *Formal Modeling and Analysis of Timed Systems (FORMATS'07)*, volume 4763 of *Lecture Notes in Computer Science (LNCS)*, pages 320–335. Springer, 2007b.

A. Podelski and S. Wagner. A sound and complete proof rule for region stability of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC 2007)*, volume 4416 of *Lecture Notes in Computer Science (LNCS)*, pages 750–753. Springer, 2007c.

S. Prajna and A. Jadbabaie. Safety verification of hybrid systems using barrier certificates. In R. Alur and G. J. Pappas, editors, *Hybrid Systems: Computation and Control (HSCC 2004)*, volume 2993 of *Lecture Notes in Computer Science (LNCS)*, pages 477–492. Springer, 2004.

S. Prajna, A. Papachristodoulou, P. Seiler, and P. A. Parrilo. SOSTOOLS and its control applications. In D. Henrion and A. Garulli, editors, *Positive Polynomials in Control*, volume 312 of *Lecture Notes in Control and Information Sciences (LNCIS)*, pages 273–292. Springer, 2005.

S. Prajna, A. Jadbabaie, and G. J. Pappas. A framework for worst-case and stochastic safety verification using barrier certificates. *IEEE Transactions on Automatic Control*, 52(8):1415 –1428, 2007.

A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In *Computer Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science (LNCS)*, pages 95–104. Springer, 1994.

N. Ramdani and N. S. Nedialkov. Computing reachable sets for uncertain nonlinear hybrid systems using interval constraint-propagation techniques. *Nonlinear Analysis: Hybrid Systems*, 5(2):149–162, 2011.

S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *ACM Transactions on Embedded Computing Systems*, 6(1):573–589, 2007.

S. Ratschan and Z. She. Providing a basin of attraction to a target region of polynomial systems by computation of Lyapunov-like functions. *SIAM Journal on Control and Optimization*, 48(7):4377–4394, 2010.

S. Ratschan and J.-G. Smaus. Verification-integrated falsification of non-deterministic hybrid systems. In *Proceedings of the 2nd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 371–376, 2006.

S. Sankaranarayanan. Automatic invariant generation for hybrid systems using ideal fixed points. In *Hybrid Systems: Computation and Control (HSCC 2010)*, pages 221–230, 2010.

R. Segala. *Modeling and verification of randomized distributed real-time systems.* PhD thesis, Laboratory for Computer Science, Massachussetts Institute of Technology, 1995.

R. Segala, R. Gawlick, J. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141:119–171, 1998.

G. F. Simmons. *Differential equations with applications and historical notes*. McGraw-Hill, 1972.

C. Sloth and R. Wisniewski. Verification of continuous dynamical systems by timed automata. *Formal Methods in System Design*, 39(1):47–82, 2011.

C. Sloth and R. Wisniewski. Complete abstractions of dynamical systems by timed automata. *Nonlinear Analysis: Hybrid Systems*, 7(1):80–100, 2013. Special issue on IFAC World Congress 2011.

C. Sloth, G. J. Pappas, and R. Wisniewski. Compositional safety analysis using barrier certificates. In *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control (HSCC '12)*, pages 15–24, 2012.

J. A. Stiver, X. D. Koutsoukos, and P. J. Antsaklis. An invariant-based approach to the design of hybrid control systems. *International Journal of Robust and Nonlinear Control*, 11:453–478, 2001.

O. Stursberg, S. Kowalewski, and S. Engell. On the generation of timed discrete approximations for continuous systems. *Mathematical and Computer Modelling of Dynamical Systems: Methods, Tools and Applications in Engineering and Related Sciences*, 6(1):51–70, 2000.

P. Tabuada. *Verification and control of hybrid systems: a symbolic approach*. Springer, 2009.

K.-C. Tai. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In *International Conference on Parallel Processing*, volume 2, pages 69 –72, 1994.

A. Tiwari and G. Khanna. Series of abstractions for hybrid automata. In *Hybrid Systems: Computation and Control (HSCC 2002)*, volume 2289 of *Lecture Notes in Computer Science (LNCS)*, pages 465–478. Springer, 2002.

A. Tiwari and G. Khanna. Nonlinear systems : approximating reach sets. In R. Alur and G. J. Pappas, editors, *Hybrid Systems: Computation and Control (HSCC 2004)*, volume 2993 of *Lecture Notes in Computer Science (LNCS)*, pages 600–614. Springer, 2004.

C. Tomlin, G. J. Pappas, and S. Sastry. Conflict resolution for air traffic management: a study in multiagent hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):509–521, 1998.

C. J. Tomlin, I. M. Mitchell, A. M. Bayen, and M. Oishi. Computational techniques for the verification of hybrid systems. *Proceedings of the IEEE*, 91:986–1001, 2003.

S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18:25–68, 2001.

V. I. Utkin. *Sliding modes in control and optimization.* Springer, 1992.

A. van der Schaft and H. Schumacher. *An introduction to hybrid dynamical systems*, volume 251 of *Lecture Notes in Control and Information Sciences (LNCIS).* Springer, 2000.

L. Vandenberghe and S. Boyd. A polynomial-time algorithm for determining quadratic Lyapunov functions for nonlinear systems. In *Proceedings of the European Conference on Circuit Theory and Design*, pages 1065–1068, 1993.

L. Vu and D. Liberzon. Common Lyapunov functions for families of commuting nonlinear systems. *Systems & Control Letters*, 54:405–416, 2005.

H. Ye, A. N. Michel, and L. Hou. Stability theory for hybrid dynamical systems. *IEEE Transactions on Automatic Control*, 43(4):461–474, 1998.

J. Zhao and D. J. Hill. Dissipativity theory for switched systems. *IEEE Transactions on Automatic Control*, 53(4):941–953, 2008.

Appendix A

Glossary of key notation

Symbols

$\perp$	Means ‘false’. Indicates a clock is inactive in a timed-automaton.
$(Ax)_i$	The i-th component of Ax.
$\dot{x}$	The derivative of x with respect to time, $\dot{x} = dx/dt$.
$\|x\|$	2-norm of a vector x.

Latin letters

A	Matrix in $\mathbb{R}^{n\times n}$.
$B(r,p)$	Ball of radius $r \geq 0$ around a point $p \in \mathbb{R}^n$.
b	A box in the TA splitting, or a vector in $\mathbb{R}^n$.
c	Constant.
$\mathrm{cl}(\cdot)$	Closure of the set represented by “$\cdot$”.
C	Current splitting state of a system.
d_i	Width of some box v in dimension i.
$\mathcal{E}_H$	Set of executions of the hybrid automaton H.
$\mathcal{E}_U$	Set of future-unaware executions of a hybrid automaton.
f	Continuous function.
$F(v,v')$	Facet between two boxes v and v'.
H	Hybrid automaton.
i,j,k	Integers.
L	Live set: a region of the state space that we wish to reach.
$\overline{L}$	A liveness property.
m	An integer, the number of subsystems.
n	An integer, the number of dimensions in the system.

$\mathbb{N}_0$	The set of natural (counting) numbers, including zero: $\{0, 1, 2, 3, \ldots\}$.
N	Set of newly-made splits.
q	Discrete state of a hybrid system.
$\mathbb{R}$	Euclidean real continuous space.
$\mathbb{R}_{\geq 0}$	Euclidean real numbers greater than or equal to zero.
$\mathbb{R}^n$	n-dimensional euclidean space.
S	Continuous state space of a continuous or piecewise-continuous system.
t	Time. $t \in \mathbb{R}$, $t \geq 0$.
$\overline{t_v}$	Maximum limit on the time spent in a box.
V	Set of discrete states in a timed-automaton, or the vertex set of a box.
V_0	Set of initial discrete states.
$V(x)$	Lyapunov function.
W	A hybrid set, $W \subseteq \mathcal{Z}$.
x	Vector of continuous-space variables, $x = [x_1, \ldots, x_n]$.
x_i	The i-th dimension variable, i-th element of x.
$\dot{x}^{\pm}$	Derivative as we approach some boundary from above (+) or below (-).
$\dot{x}^s$	Sliding mode dynamics on some subsystem boundary.
$\overline{x}$	An equilibrium point of the system.
$x(t)$	Time-dependent trajectory of a continuous system.
X_{i,v_i}	The v_i-th slice in dimension i.
X_v	Box labelled by $v = [v_1, \ldots, v_n]^T$ the intersection of X_{i,v_i} in each dimension.
Z	Set of clocks in a timed-automaton, or the region where a surface $\dot{x}_i = 0$ can exist in a box.
$\mathcal{Z}$	Hybrid state space.

Greek letters

α	Run of a finite-state automaton.
ϕ	Execution of a hybrid automaton.
$\phi_{0,p}$	Execution on $\tau_{0,p}$.
$\sigma^{\pm i}(v)$	Box neighbouring v in the positive (+) or negative (-) direction.
τ	Hybrid time trajectory.
$\tau_{0,p}$	Finite part of a trajectory: $\tau_{0,p} = \{[t_i, t'_i]\}_{i=0}^{p}$.
ξ	Run of a timed-automaton.