Verifying LTL Specifications for Discrete-Time Dynamical Systems

Bachelor Thesis

Erik Voogd, University of Groningen

First supervisor
Dr. ir. B. Besselink
University of Groningen

Second supervisor
Dr. J.A. Perez Parra
University of Groningen

July 19, 2018

Abstract

Linear-time temporal logic (LTL) is a class of logics whose formulas can express global, eventual, or repeated satisfaction of properties. We explore how to use LTL formulas for specifying properties of Discrete-time Dynamical Systems (DDS) on a continuous domain, and to verify them. To this end, a finite abstraction from the infinite system is constructed. The main result is that we can rigorously verify an LTL formula for a DDS using this abstraction and tools proposed in the literature.

Contents

1 Introduction

Control theory for discrete-time dynamical systems is generally concerned with the analysis of properties like stability and robustness, which describe how the trajectory of a system will converge to equilibrium points, and stay close to it under perturbations. These asymptotic properties alone, however, are not sufficient to characterize a wide variety of desired criteria for systems. Consider, for example, a system that models the traffic flow of a signalized intersection. The criterion that the waiting time for two successive green lights should be no longer than, say, forty seconds, does not refer to asymptotic behavior, and can therefore not be expressed in terms of the traditional control objectives of stability and robustness.

These kinds of specifications are required for many other types of systems, particularly for *cyber-physical* systems. A cyber-physical system can be thought of as a group of communicating computing devices that interact with the physical world. The study of these systems has become increasingly important over the last few decades, as pointed out in [8], and typical examples are found in areas such as robot operation, medical monitoring, control of traffic flow, and smart buildings.

A way to express the more complex criteria is through so-called *Linear-time Temporal Logic* (LTL), whose language can formally encode system specifications [1]. LTL specifications can however not immediately be verified for dynamical systems that arise from physical or engineering processes. This is because such systems typically evolve on an infinite, continuous state space, whereas LTL has been developed for systems with a finite number of states. In this thesis, we will therefore discuss a method for verifying LTL specifications for discrete-time dynamical systems. Specifically, the continuous state space domain is partitioned so as to construct a *finite state abstraction* (FSA, or simply *abstraction*). Each element of the partition, which is a subset of the continuous domain, will be a state of the FSA. With this abstraction, logically expressed system specifications can be then rigorously verified.

Results by the authors of [2] and [3] are used for constructing the abstractions using mixed monotonicity and interval partitions. The construction of automata from LTL formulas is based on the translation procedure presented in [5] and the online tool that the authors published. The aim of this thesis is then to show a constructive way of verifying LTL specifications for a discrete-time dynamical system, and to prove that it is correct.

In Section 2, we introduce a discrete-time dynamical system and the aim of this research, after which we will present in Section 3 the language that is used to formulate system specifications. Section 4 introduces a general framework called a *transition system*, which will be used in Section 5 to construct a finite abstraction. After that, we present the notion of a *Büchi automaton* in Section 6, which can accept the same language as a given LTL formula. This section will also define a *product automaton*, which can be used to verify specifications for systems with a finite number of states. Every section includes illustrative examples. In Section 7 we present the main results, and Section 8 contains a summary and a discussion of the results. Appendix C contains a case study showing how the theory can be applied in practice.

2 Problem Statement

In this section we present some notations and the model of our discrete-time dynamical system. Because we will also introduce notions of other types of systems in this article, we will refer to this system as the *continuous*, or the original system, where the word continuous will refer to the state space domain.

A discrete-time dynamical system without input control or disturbance can be modeled as

$$x[t+1] = F(x[t]),$$

where $x[t] \in \mathcal{X}$ for some domain $\mathcal{X} \subset \mathbb{R}^n$. The infinite sequence $x[\cdot] = x[0]x[1]\dots$ defines a trajectory of the system starting at some $x[0] = x_0$. The ith component of a state is indicated with a superscript, so we write $x = (x^1, ..., x^n)$.

If W is a possibly infinite set, then the powerset 2^W denotes the set of all possible subsets of W. If W is finite, the number of elements it contains is denoted by $|W|$.

We are looking to verify system specifications, which requires us to observe the system in some sense. We therefore assume the existence of a map $o : \mathcal{X} \to 2^O$ belonging to the system, for some finite set of observations O. The system we consider is then described by

$$\Sigma : \begin{cases} F : \mathcal{X} \to \mathcal{X}, \\ o : \mathcal{X} \to 2^O. \end{cases} \tag{1}$$

A trajectory $x[\cdot]$ then gives rise to an observed infinite output word $y[\cdot] = o(x[\cdot])$ by $y[k] = o(x[k])$ for $k \in \mathbb{Z}_{\geq 0}$, where each $y[k]$ is a set of observations. A word that repeats some finite sequence of substates $z_1 \dots z_n$ ad infinitum is denoted by $(z_1 \dots z_n)^\omega$. In particular, $z^\omega = zzz\dots$.

Example 1 shows how a map o and a set O can be defined in practice. In the next section, we define a logic that can express system specifications in a formula ϕ using the set of observations O as atomic propositions. In this article we aim to verify that a given Σ satisfies a given formula ϕ.

Example 1. Suppose that Σ in (1) describes the movement of a robot, where $\mathcal{X} = \mathbb{R}^2$. The robot's environment, as illustrated in Figure 1, is labeled by E. We further have a dangerous region D, and two regions with specific purposes A and B. The set of observations of Σ is $O = \{A, B, D, E\}$. Then, if $x = (x^1, x^2)$, an observation map o for Σ can be given by

$$\begin{aligned} A \in o(x), &\text{ iff } 3 \leq x^1 < 4 \text{ and } 3 \leq x^2 < 4, \\ B \in o(x), &\text{ iff } 0 \leq x^1 < 1 \text{ and } 0 \leq x^2 < 1, \\ D \in o(x), &\text{ iff } 1 \leq x^1 < 3 \text{ and } 1 \leq x^2 < 3, \\ E \in o(x), &\text{ iff } 0 \leq x^1 < 6 \text{ and } 0 \leq x^2 < 4, \end{aligned}$$

Be aware that o maps to *sets* of observations. For example, $o(2, 2) = \{D, E\}$, and $o(7, 5) = \emptyset$.

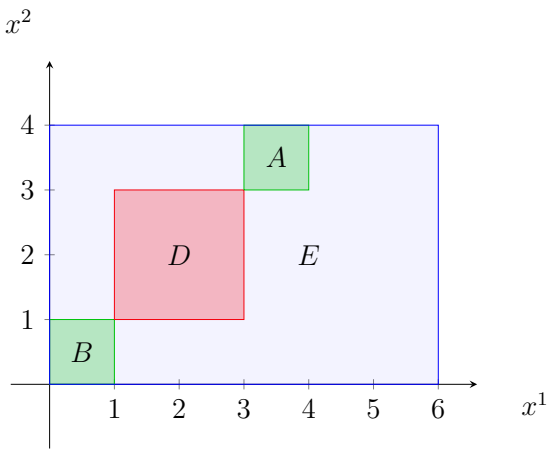

Figure 1: Labeled regions for a robot environment. See also Examples 1 and 2.

3 Linear Temporal Logic

Simply put, a logic consists of syntax and semantics, which will be formally defined in this section. After that, the syntax is conveniently extended, to show how we can express very useful specifications.

3.1 Syntax

The syntax of a logic merely describes how formulas are built. Formulas in Linear-time Temporal Logic (LTL) are built from logical operators, temporal operators, and atomic propositions. In the context of this thesis, the atomic propositions will be elements of the finite set of *observations* O belonging to Σ in (1).

Definition 1. A propositional *Linear Temporal Logic formula* over a given set of observations O is recursively defined as follows:

- $\top$ is a formula;
- Every element $p \in O$ is a formula;
- If ϕ and ψ are formulas, then $\neg\phi,\ \phi \wedge \psi,\ \phi \vee \psi,\ \bigcirc\phi$, and $\phi\ U\ \psi$ are also formulas.

Whereas the classical logical operators should be familiar to the reader, the new temporal operators $\bigcirc$ and U might not be. For now, it suffices to know that they are pronounced as "next" and "until", respectively. An example of an LTL formula is $\neg(\top U(p_1 \wedge (p_2 \vee \bigcirc p_3)))$, where each $p_i \in O$. At this point, the formula has no meaning, because we have yet to define the semantics.

3.2 Semantics

The semantics of a logic models the interpretation of formulas with a set of rules on how to evaluate them. The evaluation can be regarded as a mapping to *true* or *false*, and we say that a formula is *satisfied* if it is evaluated as true. In LTL, temporal operators will make claims about observations or formulas in future states. Knowing this, it would seem natural to evaluate LTL formulas over an

infinite *word* $\xi = y_0 y_1 \ldots$, where each y_k is a (possibly empty) set of observations. The subscripted ξ_k will then denote the infinite suffix of ξ starting at position $k \geq 0$, so $\xi_k = y_k y_{k+1} \ldots$.

Definition 2. The satisfaction of a formula ϕ over a set of observations O by an infinite word $\xi = y_0 y_1 \ldots$ at position $k \in \mathbb{Z}_{\geq 0}$, denoted by $\xi_k \models \phi$, is defined recursively as follows:

- $\xi_k \models \top$,
- $\xi_k \models p$ iff $p \in y_k$, for $p \in O$,
- $\xi_k \models \neg\phi$ iff not $\xi_k \models \phi$,
- $\xi_k \models \phi \wedge \psi$ iff $\xi_k \models \phi$ and $\xi_k \models \psi$,
- $\xi_k \models \phi \vee \psi$ iff $\xi_k \models \phi$ or $\xi_k \models \psi$,
- $\xi_k \models \bigcirc\phi$ iff $\xi_{k+1} \models \phi$,
- $\xi_k \models \phi \; U \; \psi$ iff there exists $j \geq k$ such that $\xi_j \models \psi$ and $\xi_i \models \phi$ for all $k \leq i < j$.

An infinite word ξ satisfies a formula ϕ, written $\xi \models \phi$, iff $\xi_0 \models \phi$. The *language* of a formula ϕ, written $\mathscr{L}_\phi$ is the set of all infinite words that satisfy ϕ. Two formulas ϕ and ψ are *equivalent*, written $\phi \equiv \psi$, if $\mathscr{L}_\phi = \mathscr{L}_\psi$.

The first item models the formula for true in such a way that it is always satisfied at any point in an infinite word. Indeed, we want this formula to always be evaluated as true. An observation p is satisfied if it is in the current set of observations. The classical logical operators for negation, conjunction, and disjunction are modeled in the familiar way. The circle in "$\bigcirc\phi$" means that ϕ should be satisfied by the suffix directly following the current set of observations. A formula $\phi \; U \; \psi$ is satisfied if ϕ remains satisfied *until* ψ is satisfied, and ψ should indeed be satisfied at some point in the future. If, specifically, $\psi = p$ for some $p \in O$, and $p \in y_k$, then the formula is satisfied. To see this, put $j = k$, then, since $p \in y_j$, indeed $\xi_j \models p$ and $\xi_i \models \phi$ for all i in the half-open interval $k \leq i < j$, which is empty. This means that if p is satisfied now, then $\phi \; U \; p$ is also satisfied.

3.3 Derived Formulas

The operator U allows us to express that formulas be satisfied *eventually*. Indeed, $\top$ is always satisfied, and so $\top \; U \; \psi$ is satisfied iff ψ is fulfilled at some point in the future. This makes way for a useful and widely applied unary operator, which we denote by "$\Diamond$". In some texts, the operator is denoted by F, for *future*. Another useful operator is "$\Box$", in some texts denoted G for *global*, which says that a formula should *always* be satisfied. Equivalently, the negation of the formula should never be satisfied at any point in the future. We can extend the syntax as follows:

$$\begin{aligned}
\bot &:= \neg\top, && \text{(false)} \\
\phi \rightarrow \psi &:= \neg\phi \vee \psi, && \text{(conditional)} \\
\phi \leftrightarrow \psi &:= (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi), && \text{(biconditional)} \\
\Diamond\phi &:= \top \; U \; \phi, && \text{(eventually)} \\
\Box\phi &:= \neg\Diamond\neg\phi, && \text{(always)} \\
\phi \; R \; \psi &:= \neg(\neg\phi \; U \; \neg\psi). && \text{(release)}
\end{aligned}$$

Defined this way, it is easily verified with the semantics in Definition 2 that the formula $\bot$ is always evaluated as false. The classical operators for conditional and biconditional should be familiar. The *release* operator is defined as the dual of U, which means that also $\neg(\neg\phi\ R\ \neg\psi) \equiv \phi\ U\ \psi$. This dual relation is completely analogue to the relation between $\wedge$ and $\vee$, better known as De Morgan's laws. The intuition for a formula $\phi\ R\ \psi$ is that ψ must always be satisfied, but it might be that ϕ is satisfied at some point, and this is when the necessity for the satisfaction of ψ is *released*. Note that, as opposed to U, with R it is allowed that ϕ is never satisfied, in which case ψ should always be true. Indeed, $\Box\psi \equiv \bot\ R\ \psi$, as can be checked using the syntax. The power of the derived formulas is illustrated with a practical example.

Example 2. Consider again the robot dynamics from Example 1, with $O = \{A, B, D, E\}$, graphically presented in Figure 1. Assume that the robot is to deliver packages repeatedly from A to B, always staying in E, and always avoiding the danger zone D. This specification can be expressed in the formula

$$\phi_1 = \Box E \wedge \Box\neg D \wedge \Box\Diamond A \wedge \Box\Diamond B.$$

In words, the last two conjuncts say that the robot should "always eventually" reach (always return to) region A, and also always eventually reach region B. We can refine the criterion by realizing that the robot should not return to B as long as it has not got a package from A. This can be expressed as

$$\phi_2 = \Box E \wedge \Box\neg D \wedge \Box\Diamond B \wedge \Box(B \rightarrow \bigcirc(\neg B\ U\ A)).$$

The last conjunct makes sure that it is always the case that the robot is not in B, or, if it is, in the next state it will not be in B until it has been to A. The reader can verify that the following infinite words do *not* satisfy ϕ_2:

$$\begin{aligned}\xi &= (\{E\})^\omega = \{E\}\{E\}\{E\}\{E\}\dots \\ \xi' &= (\{E\}\{E, B\}\{E, A\}\{E, D\})^\omega \\ \xi'' &= (\{E, B\}\{E, B\}\{E, A\})^\omega\end{aligned}$$

A sequence that does satisfy ϕ_2 is given by $\zeta = (\{E, B\}\{E\}\{E, A\}\{E\})^\omega$. Note that evaluating the words formally with the semantics is a job far from trivial.

3.4 Problem Revisited

The problem statement that was described in Section 2 can now be made formal, using the following definition.

Definition 3. Let Σ be a system with F, o, and O as in (1), and let ϕ be an LTL formula over O. Then Σ *satisfies* ϕ, written $\Sigma \vdash \phi$, if for every initial x_0, $y[\cdot] \in \mathscr{L}_\phi$, where $y[t] = o(x[t])$ and $x[t+1] = F(x[t])$ for all $t \geq 0$, and $x[0] = x_0$.

In other words, all infinite words of observations from the system must satisfy the formula for every initial x_0.

Problem 1. Given a pair (Σ, ϕ), where Σ is the system in (1) and ϕ an LTL formula as in Definition 1, verify whether $\Sigma \vdash \phi$, using a constructive procedure.

The system Σ has trajectories that run through a real state space, where the number of states is infinite. To answer the question whether Σ satisfies a given LTL formula does not have a trivial answer. Tools and theory from computer science will be of great help in working towards a practical solution. This is why the next section introduces the notion of a *transition system*. A transition system is a general framework that is widely used in theoretical computer science, in particular formal methods.

4 Transition Systems

This section defines a transition system, and the reader will be familiarized with it by use of some examples. Transition systems can have an infinite structure, which makes it difficult to compare them. For this purpose, we will also define a relation on states and systems called *simulation*.

4.1 Definition and Examples

A transition system with an observation map is defined as follows.

Definition 4. A transition system is a tuple $T = (S, U, \delta, O, o)$ where

- S is a set of states,
- U is a set of inputs,
- $\delta : S \times U \to 2^S$ is the transition map,
- O is a set of observations,
- $o : S \to 2^O$ is an observation map.

Each of the sets S, U, and O can be either finite or infinite. A transition system is called finite if all three are finite, and infinite otherwise. Furthermore, it is *non-blocking* if $|\delta(s, u)| > 0$ for all $s \in S$ and $u \in U$. This means that every state has at least one possible transition. Lastly, a system is *deterministic* if $|\delta(s, u)| \leq 1$ for all s, u. This means that if a transition system is both non-blocking *and* deterministic, its transition map δ might be considered as a mapping to S, rather than to 2^S. We illustrate Definition 4 with an example.

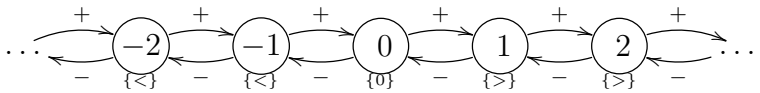

Figure 2: Graphical representation of an infinite transition system. See Example 3. The observations are written below each state.

Example 3. The system presented graphically in Figure 2 is captured by the infinite transition system $T = (S, U, \delta, O, o)$, where

- $S = \mathbb{Z}$,

- $U = \{+, -\}$,
- $\delta(s, u) = \begin{cases} s+1 & \text{if } u = + \\ s-1 & \text{if } u = -, \end{cases}$
- $O = \{<, 0, >\}$,
- and $o(s) = \begin{cases} \{>\} & \text{if } x > 0 \\ \{<\} & \text{if } x < 0 \\ \{0\} & \text{otherwise.} \end{cases}$

Given a state $n \in \mathbb{Z}$, the system jumps to state $n+1$ if it gets the symbol $'+'$ as input, or to $n-1$ if the input is the symbol $'-'$. Note that it is both non-blocking and deterministic. The observation map only gives information about the sign of the number that a state represents. In Figure 2, the observations are written below each state.

Similar to a trajectory in a continuous system, we refer to a *run* of the transition system by $s[\cdot] = s[0]s[1]s[2]\ldots$. A run is defined by an infinite input sequence $u[\cdot] = u[0]u[1]\ldots$ such that $s[k+1] \in \delta(s[k], u[k])$ for all $k \geq 0$ and some chosen $s[0] = s_0$. Note that one input sequence can define multiple runs if the transition system is non-deterministic. The following example shows that discrete-time dynamical systems as in (1) can be naturally represented as an infinite transition system.

Example 4. The discrete-time dynamical system Σ in (1) is captured by the infinite transition system $T = (S, U, \delta, O, o)$, where O and o are already given, and

- $S = \mathcal{X} \subset \mathbb{R}^n$,
- $U = \{0\}$ (or any singleton),
- and $\delta(s, 0) = \{F(s)\}$.

The set of inputs U is taken as a singleton, because the transitions in the system should not depend on input, as the continuous system does not have a control input.

Example 4 can easily be extended to model discrete-time dynamical systems with inputs, i.e., systems of the form

$$x[t+1] = F(x[t], u[t]),$$

where each $u[t] \in \mathcal{U}$. This possibly infinite $\mathcal{U}$ will then be the set of inputs for the transition system. There exist techniques to develop control strategies that make sure that systems with input satisfy a given LTL formula ϕ. For example, in [2], a case study on traffic flow shows how to synthesize a control strategy that ensures that the waiting queues for the traffic lights will never be too long.

This article however is restricted to verification of formulas only, and since the discrete set U belonging to the system T in Example 4 is a singleton, we can omit it. We will therefore be mainly interested in the infinite transition system

$$T_\Sigma = (\mathcal{X}, F, O, o), \tag{2}$$

where $S = \mathcal{X}$, O, and o are still as in Definition 4, but $\delta = F$ maps to $\mathcal{X}$ instead of $2^{\mathcal{X}}$, since the system is non-blocking and deterministic. For every initial $x_0 \in \mathcal{X}$, a run of T_Σ gives rise to the same infinite output word $y[\cdot] = y[0]y[1]y[2]\dots$ as Σ in (1), which should be verified against the specification ϕ as in Definition 3. Then Σ satisfies ϕ if and only if T_Σ satisfies ϕ, for which we also write $T_\Sigma \vdash \phi$, as in Definition 3.

4.2 Simulation

In the next section, we transform the infinite transition system T_Σ into a *finite abstraction*. Intuitively, this will cause a loss of some details of the original system. In the process, we must however maintain the aspects that are important for our analysis. Specifically, all trajectories in the continuous system have to correspond to one in the abstraction. Moreover, since we are verifying an LTL formula over the observations that are done, it is necessary that the corresponding trajectory in the abstraction produces the same infinite output word as in the original continuous system. This motivates the following definition.

Definition 5. Let $T = (S, U, \delta, O, o)$ be a transition system. A relation $\mathcal{R} \subset S \times S$ between states is called a *simulation* if for every pair $(s, t) \in \mathcal{R}$ and all $u \in U$,

(i) s and t are observationally equivalent, written $o(s) = o(t)$,

(ii) if $s' \in \delta(s, u)$, then there exists $t' \in \delta(t, u)$ such that $(s', t') \in \mathcal{R}$.

We say that a state t *simulates* s, if there exists a simulation relation $\mathcal{R}$ such that $(s, t) \in \mathcal{R}$.

States as elements of the relation $\mathcal{R}$ in Definition 5 are defined in terms of transitions to other states, but the relation has no base case. This gives rise to an infinite, circular-like definition, a concept known formally as a *coinductive* definition. Simulations are known to be preorder relations, which is easy to establish, see, for example, [9].

Simulation of states is useful in this context only when comparing two different transition systems.

Definition 6. Let $T = (S, U, \delta, O, o)$ and $\hat{T} = (\hat{S}, \hat{U}, \hat{\delta}, \hat{O}, \hat{o})$ be transition systems. We say that $\hat{T}$ *simulates* T if $\hat{O} = O$ and $\hat{U} = U$, and there exists a simulation $\mathcal{R} \subset S \times \hat{S}$, such that for every $s \in S$, $(s, t) \in \mathcal{R}$ for some $t \in \hat{S}$.

In other words, $\hat{T}$ simulates T if there exists a relation that shows that every state in T is simulated by a state in $\hat{T}$. The corresponding observation map to both transition systems is used to see whether the states are observationally equivalent. Specifically, $o(s) = \hat{o}(t)$ if $(s, t) \in \mathcal{R}$. The following lemma is an important result for satisfying LTL formulas.

Lemma 1. If T and $\hat{T}$ are transition systems, and $\hat{T}$ simulates T, then every trajectory of T corresponds to some trajectory of $\hat{T}$ with the same infinite output word.

Proof. This is a direct consequence of Definitions 5 and 6. ☺

The following example defines a transition system $\hat{T}$ that simulates T in Example 3.

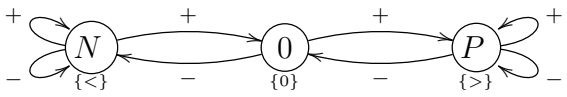

Figure 3: Graphical representation of a finite transition system that simulates the infinite transition system from Example 3. See Example 5

Example 5. Consider the finite transition system $\hat{T} = (\hat{S}, \hat{U}, \hat{\delta}, \hat{O}, \hat{o}\}$, where

- $\hat{S} = \{N, 0, P\}$,
- $\hat{U} = \{+, -\}$,
- $\hat{O} = \{<, 0, >\}$,
- $\hat{o}(s) = \begin{cases} \{<\} & \text{if } s = N, \\ \{0\} & \text{if } s = 0, \\ \{>\} & \text{if } s = P, \end{cases}$
- $\hat{\delta}$ is given by the transition table:

u	$+$	$-$
$\hat{\delta}(0, u)$	$\{P\}$	$\{N\}$
$\hat{\delta}(P, u)$	$\{P\}$	$\{0, P\}$
$\hat{\delta}(N, u)$	$\{0, N\}$	$\{N\}$

The system is non-deterministic, and is presented graphically in Figure 3. The symbols P and N represent the collection of positive and negative integers respectively. Also, let $T = (S = \mathbb{Z}, U, \delta, O, o)$ be the infinite transition system from Example 3. Observe that $\hat{O} = O$ and $\hat{U} = U$, and define the relation

$$\mathcal{R} := \{(x, N) \mid x \in \mathbb{Z}_{<0}\} \cup \{(0, 0)\} \cup \{(x, P) \mid x \in \mathbb{Z}_{>0}\}$$

Then for all $s \in S$, there is a pair (s, t) for some $t \in \hat{S}$. We will now verify that $\mathcal{R}$ is a simulation by considering the two properties in Definition 5 seperately.

(i) Indeed, for every pair $(s, t) \in \mathcal{R}$, we have $o(s) = \hat{o}(t)$.

(ii) Here, we verify (ii) for every $(s, t) \in \mathcal{R}$ and $u \in U$ by a case analysis on the union of the three sets.

- First, consider the case where $(s, t) = (0, 0)$. For $u = +$, s' must be 1, $\hat{\delta}(0, +) = \{P\}$ and indeed $(1, P) \in \mathcal{R}$. For $u = -$, s' must be -1, $\hat{\delta}(0, -) = \{N\}$, and we also confirm that $(-1, N) \in \mathcal{R}$.
- Consider now $(s, t) \in \{(x, N) \mid x \in \mathbb{Z}_{<0}\}$. If $u = -$, take the only two possible transitions, and observe $(s - 1, N) \in \mathcal{R}$. For $u = +$, $\delta(s, +) = \{s + 1\}$ and $\hat{\delta}(N, +) = \{0, N\}$. If $s + 1 = 0$, pick $t' = 0$ as the next state, and confirm $(0, 0) \in \mathcal{R}$. Otherwise, $s + 1 < 0$, pick $t' = N$, and indeed, $(s + 1, N) \in \mathcal{R}$.
- The case where $(s, t) \in \{(x, P) \mid x \in \mathbb{Z}_{>0}\}$ is similar.

We conclude that $\mathcal{R}$ is a simulation, and hence, the infinite, deterministic transition system T is simulated by the finite, non-deterministic system $\hat{T}$.

In Example 5, we saw a finite transition system simulating an infinite transition system. In exchange, the finite system was constructed to allow non-deterministic behaviour. The next section introduces a way of constructing a finite, non-deterministic transition system that simulates T_Σ in (2). This is done because satisfying LTL formulas for finite systems is more viable than for infinite systems.

5 Finite State Abstractions

First, the general way of constructing a finite abstraction is shown in this section. After that, we present methods and assumptions to make the construction computationally more feasible and efficient. Finally, we show that, although an abstraction simulates T_Σ, the converse is not true in general. This has some unfavorable consequences that we will explore and partially solve.

5.1 Construction

When constructing a finite state abstraction from T_Σ in (2), we discretize the continuous state space $\mathcal{X}$ by constructing a partition.

Definition 7. For a (possibly infinite) set $\mathcal{X}$, a partition of $\mathcal{X}$ is $\{\mathcal{X}_q\}_{q\in Q}$, where Q is finite and

(i) $\mathcal{X}_q \cap \mathcal{X}_{q'} = \emptyset$ for all $q, q' \in Q$,

(ii) $\bigcup_{q\in Q} \mathcal{X}_q = \mathcal{X}$

Each $\mathcal{X}_q$ is called a *part* of $\mathcal{X}$.

In other words, all the parts are subsets of $\mathcal{X}$, and every point in $\mathcal{X}$ is represented by exactly one of the parts. Intuitively, the relation between $\mathcal{X}$ and Q identifies the relation between the transition system T_Σ and its abstraction, and it is the key in going from an infinite to a finite number of states.

Definition 8. Let T_Σ be as in (2). A non-deterministic *finite state abstraction* of T_Σ is a tuple $A = (Q, \delta, O, o_A)$ with Q finite, such that $\{\mathcal{X}_q\}_{q\in Q}$ is a partition of $\mathcal{X}$, and $\delta : Q \to 2^Q$ is the transition map defined such that

$$\forall x \in \mathcal{X} : \text{ if } x \in \mathcal{X}_q \text{ and } F(x) \in \mathcal{X}_{q'}, \text{ then } q' \in \delta(q), \tag{3}$$

and $o_A : Q \to 2^O$ is the observation map such that

$$\forall x \in \mathcal{X} : \text{ if } x \in \mathcal{X}_q \text{ then } o(x) = o_A(q).$$

The set $\delta(q)$ is called the *one-step reachable state set* of $\mathcal{X}_q$.

The finite state abstraction of T_Σ is another transition system, where the input is again omitted. Note that, defined this way, the observation map of Σ puts certain restrictions on the form of the partition. Specifically, if two states of the orginal system are not observationally equivalent, they cannot be in the same part of the partition by definition. We say that a partition of $\mathcal{X}$ *respects* the observation map o of Σ if it allows for the construction of a finite state abstraction.

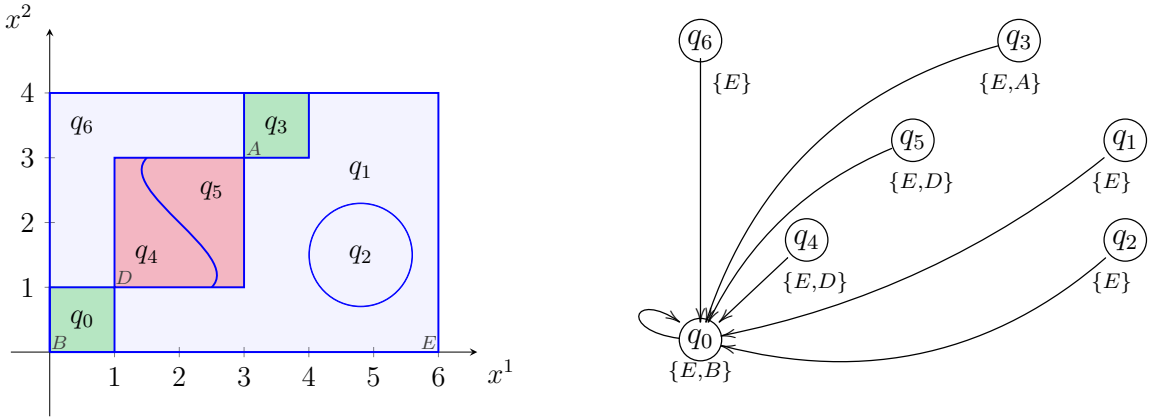

Figure 4: Graphical representations for Example 6: arbitrary partition (a) that respects the observation map o, and finite state abstraction (b) of the robot environment in Figure 1 using $F(x) = (0.5, 0.5)$. Note that for every part of the partition, no two points have different observation sets. The observations are written below the states in the abstraction.

An abstraction A approximates the original system, and the construction of an abstraction can be seen as a trade-off between infinite states in the deterministic system T_Σ, and non-determinism in the finite abstraction. Coarse partitioning of $\mathcal{X}$ will cause more non-determinism, and fine partitioning will limit non-deterministic behaviour.

The following lemma is an important result of Definition 8, because it reduces the question described in Problem 1 to satisfying a specification for a finite abstraction, rather than for the continuous system.

Lemma 2. If A is a finite state abstraction of T_Σ as in Definition 8, then A simulates T_Σ.

Proof. This is proven by providing an explicit simulation relation $\mathcal{R} \subset \mathcal{X} \times Q$, such that every $x \in \mathcal{X}$ is in it. Consider the relation

$$\mathcal{R} := \{(x, q) \in \mathcal{X} \times Q \mid x \in \mathcal{X}_q\}.$$

It follows immediately that the relation $\mathcal{R}$ then satisfies properties (i) and (ii) in Definition 5 by the way A is constructed. Hence, $\mathcal{R}$ is a simulation relation. ☺

We let the notation for Σ or T_Σ satisfying an LTL formula ϕ as in Definition 3 extend naturally to writing $A \vdash \phi$ for the abstraction A satisfying ϕ.

Example 6. Consider again the system Σ describing the dynamics of a robot, as given in Example 1, but now assume that $\mathcal{X} = [0, 6) \times [0, 4)$. Then, a partition that respects the observation map is given in Figure 4a.
Let F of Σ be given by the constant function $F(x) = (0.5, 0.5)$, i.e., F sends every state to a point in $\mathcal{X}_{q_0}$, where $q_0 \in Q$. Then the finite state abstraction for F is the transition system as shown in Figure 4b.
The reader can verify that the finite state abstraction satisfies, for example, the formulas $\bigcirc B$ and $\Box E \wedge \bigcirc\Box B$.

5.2 Computation

Calculating the one-step reachable set of states for each state in the abstraction is a major challenge. This is because we have to evaluate F at an infinite number of points in $\mathcal{X}$ if we want to determine δ for the abstraction. To make the procedure more feasible, we introduce the notion of a *grid* and we restrict the system Σ to the class of *monotone* systems.

Defining monotonicity of functions requires that the domain $\mathcal{X}$ has an order. Assume for simplicity that $\mathcal{X}$ is a subset of the positive orthant of $\mathbb{R}^n$, which is denoted here by $\mathbb{R}^n_{\geq 0}$. Let the partial order on $\mathcal{X} \subset \mathbb{R}^n_{\geq 0}$ then be given by: $x \leq y$ if and only if $x^i \leq y^i$ for all $i \in \{1, \ldots, n\}$, and $x, y \in \mathcal{X}$. Note that not every pair of states in $\mathcal{X}$ can be compared in this way. Take as an example the two states $(0, 1)$ and $(2, 0)$ in $\mathbb{R}^2_{\geq 0}$. A system (1) is monotone, if the function $F : \mathcal{X} \to \mathcal{X}$ preserves an order on $\mathcal{X}$. This means that F is either non-increasing or non-decreasing in its entire domain.

In [2], two different algorithms are presented to compute the one-step reachable state sets. The most efficient one assumes that $\mathcal{X} \subset \mathbb{R}^n$ can be partitioned by a grid. This intuitively means that every dimension of $\mathcal{X}$ is split into subintervals, and the cartesian product of subintervals, one from each dimension, will make up a part of the partition.

Definition 9. Let $\mathcal{X} \subset \mathbb{R}^n_{\geq 0}$. For each $i \in \{1, \ldots, n\}$, let $N_i > 0$ and $\gamma^i = (\gamma^i_1, \ldots, \gamma^i_{N_i+1})$ be a strictly increasing sequence. A *gridded partition* of $\mathcal{X}$ is a partition of hyperrectangles $\{\mathcal{I}_q\}_{q \in Q}$, where $q = (k_1, \ldots, k_n)$ and $\mathcal{I}_q = \prod_{i=1}^n [\gamma^i_{k_i}, \gamma^i_{k_i+1})$.

The sequences γ^i divide each state space dimension into exactly N_i subintervals $[\gamma^i_j, \gamma^i_{j+1})$ for $j \in \{1 \ldots N_i\}$. A state $q \in Q$ can be identified by a sequence $(k_1, \ldots, k_n)$, where each k_i is an index for the sequence γ^i.

A hyperrectangle $\mathcal{I}_q$ can also be written as $[a_q, b_q)$, where the ith coordinate of a_q and b_q are determined by k_i belonging to $q = (k_1, \ldots, k_n)$. Each hyperrectangle $\mathcal{I}_q$ then denotes the subset $\{x \in \mathcal{X} \mid a_q \leq x < b_q\}$. The following example illustrates how a gridded partition can be constructed.

Example 7. Consider the robot environment of Examples 1 and 6 on $\mathcal{X} = [0, 6) \times [0, 4)$. To construct a gridded partition that respects the observation map, we can pick, for example, $N_1 = 4$ and $N_2 = 3$, and we let $\gamma^1 = (0, 1, 3, 4, 6)$ and $\gamma^2 = (0, 1, 3, 4)$. Figure 5a shows what the partition looks like.

Then, for computing the one-step reachable set of states, consider a hyperrectangle $\mathcal{I}_q = [a_q, b_q)$ from the grid. Let $F(\mathcal{I}_q) = \{F(x) \mid x \in [a_q, b_q)\}$ naturally denote the image of the box. Note that if $\delta(q) = \{q' \in Q \mid \mathcal{I}_{q'} \cap F(\mathcal{I}_q) \neq \emptyset\}$ then property (3) in Definition 8 would indeed be satisfied. Now define the hyperrectangle G_q as

$$G_q := \{x \in \mathcal{X} \mid F(a_q) \leq x < F(b_q)\}.$$

Note that by monotonicity of F, we have $a_q \leq x < b_q$ implies $F(a_q) \leq F(x) < F(b_q)$ for all $x \in \mathcal{X}$. Hence, for every part $\mathcal{I}_q$, it holds that $F(\mathcal{I}_q) \subset G_q$. We therefore say that G_q *overapproximates* the image of $\mathcal{I}_q$. Then, monotonicity can be used to compute the abstraction very efficiently, by defining the one-step reachable state set in the following way:

$$\delta(q) := \{q' \in Q \mid \mathcal{I}_{q'} \cap G_q \neq \emptyset\}.$$

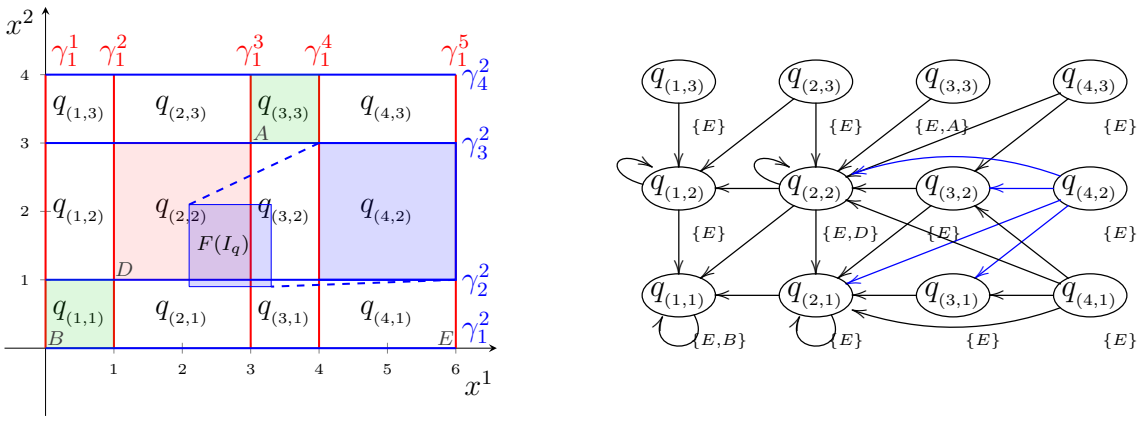

Figure 5: The constructed grid (a) of Example 7 and a finite state abstraction (b) of the robot dynamics using $F(x) = Cx$ in Example 8. The grid respects the observation map. Each state is identified by a pair of indices for both dimensions. The sets of observations for each state are written below the states.

In other words, the mapping of two corner points of a box are computed and compared to corner points of every part of the partition. Now only two points need to be evaluated by F for each state in the finite abstraction, instead of infinitely many.

The following example uses monotonicity to show how the grid from Example 7 is used to construct a finite state abstraction.

Example 8. With the gridded partition from Example 7, and using the function $F(x) = Cx$ with

$$C = \begin{pmatrix} 0.5 & 0.1 \\ 0.1 & 0.5 \end{pmatrix},$$

we construct the finite state abstraction $A = (Q, \delta, O, o_A)$. The map F is monotone, and the system is stable. We have

$$Q = \{q_{(1,1)}, q_{(2,1)}, q_{(3,1)}, q_{(4,1)}, q_{(1,2)}, q_{(2,2)}, q_{(3,2)}, q_{(4,2)}, q_{(1,3)}, q_{(2,3)}, q_{(3,3)}, q_{(4,3)}\}.$$

For $q = q_{(4,2)}$, we have the two corner points $a_q = (4, 1)$ and $b_q = (6, 3)$. Using monotonicity of F, we efficiently compute G_q as an overapproximation of $F(I_q)$ and conclude that the image intersects with at most four other boxes, so we set

$$\delta(q_{(4,2)}) := \{q_{(2,1)}, q_{(3,1)}, q_{(2,2)}, q_{(3,2)}\}.$$

Repeating this computation for all states, we find the finite state abstraction as presented in Figure 5b. Notice that state $q_{(4,2)}$ has exactly four outgoing arrows. Notice also how all state transition tend to go to $q_{(1,1)}$, and that this is indeed the only state that does not have any transitions going anywhere but to itself. This is expected, because the system is stable, and converges to the origin $(0, 0)$ in the continuous system.

5.3 Mixed Monotonicity

The previous result provides a huge computational advantage, but to require monotonicity is also a significant restriction on the variety of systems that we can use. We therefore present a generalization of monotonicity.

Definition 10. A system (1) is called *mixed monotone*, if there exists a *decomposition function* $f : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$, such that the following three properties hold:

1. $F(x) = f(x, x)$ for all $x \in \mathcal{X}$,
2. For all $x_1, x_2, y \in \mathcal{X}$ such that $x_1 \leq x_2 : f(x_1, y) \leq f(x_2, y)$,
3. For all $x, y_1, y_2 \in \mathcal{X}$ such that $y_1 \leq y_2 : f(x, y_1) \geq f(x, y_2)$.

In words, the system is called mixed monotone if the function F can be decomposed into two parts, one of which is non-decreasing, the other non-increasing. From this it immediately follows that every monotone F is also mixed monotone. Namely, we can let $f(x, y)$ ignore one of its arguments and define it to be simply $F(x)$ or $F(y)$, depending on whether F is non-decreasing or non-increasing. The following example provides an intuition of mixed monotonicity.

Example 9. Consider a simple one-dimensional system $G(x) = x \cdot 2^{-x}$ on the domain $\mathbb{R}_{\geq 0}$. Observe that $G(0) = 0, G(2) = \frac{1}{2}$, and $G(4) = \frac{1}{4}$. Since $0 < 2 < 4$, but $G(0) < G(4) < G(2)$, the system is not monotone. Now, let $g(x, y) = x \cdot 2^{-y}$, then $G(x) = g(x, x)$, and g is indeed non-decreasing in its first, and non-increasing in its second argument. Hence, this one-dimensional system is mixed monotone.

Let F of Σ in (1) be a mixed monotone function with a decomposition function $f(x, y)$, and let $\mathcal{I}_q$ be a hyperrectangle of a gridded partition. Notice that $F(x) = f(x, x)$ by definition, and that

$$f(a_q, x) \leq f(x, x) \leq f(b_q, x), \text{ for all } x \in \mathcal{X}. \tag{4}$$

We also have $f(x, b_q) \leq f(x, x) \leq f(x, a_q)$ by Definition 10, and in particular $f(a_q, b_q) \leq f(a_q, x)$ and $f(b_q, x) \leq f(b_q, a_q)$. Together with (4), this gives $f(a_q, b_q) \leq f(x, x) \leq f(b_q, a_q)$ for all $x \in \mathcal{I}_q$. Now define the hyperrectangle H_q as

$$H_q := \{x \in \mathcal{X} \mid f(a_q, b_q) \leq x \leq f(b_q, a_q)\}. \tag{5}$$

Then it follows that $F(\mathcal{I}_q) \subset H_q$. Mixed monotonicity can thus be used to compute the abstraction by defining the one-step reachable state set in the following way:

$$\delta(q) := \{q' \in Q \mid \mathcal{I}_{q'} \cap H_q \neq \emptyset\}. \tag{6}$$

Since, $F(\mathcal{I}_q)$ is contained in H_q, property (3) of Definition 8 is still satisfied.

We have thus shown a way to compute the one-step reachable set of states for each state in the finite abstraction very efficiently for the class of mixed monotone dynamical systems. This class is significantly larger than the class of monotone systems.

5.4 Removing Spurious Self-loops

A result of Lemma 1 and 2 is that every trajectory in T_Σ, and hence in Σ, corresponds to a trajectory in the abstraction A. The converse is however not necessarily true: the abstraction A may define runs that do not correspond to any trajectory in the original continuous system. These runs are in some sense not authentic, and are therefore referred to as *spurious* trajectories.

The most common form of a spurious trajectory in a finite abstraction is one that stays in the same state forever. A state has a *self-loop* when its part in the partition of the continuous domain intersects with its own image. If, however, the continuous system does not define any trajectory that stays within this part of the partition indefinitely, then the self-loop in the abstraction is called a *spurious self-loop*. The state that corresponds to this part of the partition is referred to as a *stuttering* state.

Example 10. The abstraction constructed in Example 8, depicted in Figure 5b, defines an infinite trajectory $q[\cdot] = (q_{(2,2)})^\omega$. Since the origin $(0, 0)$ of the continuous system with $F(x) = Cx$ is stable, this is a spurious trajectory. The self-loop of $q_{(2,2)}$ is then also spurious, and $q_{(2,2)}$ is a stuttering state. Similarly, $q_{(1,2)}$ and $q_{(2,1)}$ in the example are stuttering.

Finding a run of the finite state abstraction that does not satisfy a given LTL formula ϕ does not prove that Σ does not satisfy ϕ. This is a consequence of the existence of spurious trajectories. Identifying these trajectories is in general a difficult problem, but we illustrate here how to find and eliminate the ones that are caused by spurious self-loops.

To remove all spurious self-loops, we first put $Q' := Q$, and we will extend the set of states in this set Q'. If $q \in Q$ has a spurious self-loop, we neutralize it by creating a new state $q_s \in Q'$, whose only incoming transition will be that of q. We then compute the intersection of $\mathcal{I}_q$ with the overapproximation of its image (5), and put $\mathcal{I}_{q_s} := \mathcal{I}_q \cap H_q$, which will be a hyperrectangle. For $\mathcal{I}_{q_s}$ as a subset of $\mathcal{I}_q$, we compute the overapproximation of the image again. This image will then determine all outgoing transitions of the new state q_s. Specifically,

$$\delta(q_s) := \{q' \in Q \mid \mathcal{I}_{q'} \cap H_{q_s} \neq \emptyset\}.$$

Notice that then $\delta(q_s) \subset \delta(q)$ and $o(q_s) = o(q)$. After the creation of this new state, the self-loop of q can safely be removed, without annihilating the simulation property of Lemma 2. If now $q \in \delta(q_s)$, there is still a spurious trajectory. This trajectory is not a self-loop anymore, but rather a run that jumps back and forth between q and q_s. The procedure should be applied again in a similar way on the transition from q_s to q.

This approach results in a *refined partition* $\{\mathcal{X}_q\}_{q \in Q'}$, for a finite set Q', such that $Q \subset Q'$. For each $q' \in Q'$ then, $\mathcal{X}_{q'} \subset \mathcal{X}_q$ for some q in the original Q, where $\mathcal{X}_{q'} = \mathcal{X}_q$ if and only if $q' = q$. Each state $q' \in Q'$ such that $q' \notin Q$ has only one incoming transition by construction.

This algorithm describes how to eliminate the spurious self-loops, but not how to detect them. Notice however that with some minor modifications we can turn this approach into a detection algorithm. Namely, on each iteration we intersect $\mathcal{I}_q$ with a recursive overapproximation of the image of F. When this intersection becomes empty, the trajectory in the continuous system can not stay there indefinitely, and we conclude that the self-loop is spurious. If this does not happen for a sufficiently large number of iterations, we can safely assume that the transition is not a spurious self-loop. This procedure is presented more formally in Algorithm 1.

Data : Finite State Abstraction $A = (Q, \delta, O, o_A)$ of a system Σ based on a partition $\{\mathcal{I}_q\}_{q \in Q}$. System Σ has mixed monotone F with decomposition function f.
Input : State q with a candidate spurious self-loop.
Output: Returns $True$ if q has a spurious self-loop.

iter := 1;
$x_1 := a_q$;
$x_2 := b_q$;
while iter $\leq$ MAX_ITER **do**
  $y_1 := f(x_1, x_2)$;
  $y_2 := f(x_2, x_1)$;
  if $y_1 \leq b_q \wedge a_q \leq y_2$ **then**
    $(x_1, x_2) :=$ intersect$([y_1, y_2], [a_q, b_q])$;
    iter := iter $+1$;
  else
    return $True$;
  end
end
return $False$;

Algorithm 1: Detecting spurious self-loops in a finite state abstraction. Note that if $False$ is returned, there is no guarantee that the self-loop of q is not spurious.

For an LTL formula ϕ that does not contain the "next" operator $\bigcirc$, we can simply remove spurious self-loops. Intuitively, this is because subsequent occurrences of one set of observations in an infinite word are ignored by definition of the operator U, which is the only remaining temporal operator. Even though this will mean that there will be trajectories of the original system Σ that do not correspond to a run of the abstraction, the necessary implication still holds. This is made formal in the following proposition.

Proposition 1. Let T_Σ be as in (1) and A be its finite state abstraction as in Definition 8. If A' is a transition system equal to A, but with all spurious self-loops removed, and ϕ is an LTL formula without the "next" operator $\bigcirc$, then

$$A' \vdash \phi \implies T_\Sigma \vdash \phi$$

Proof. See Appendix A. ☺

Note that the derived formulas "release" (R), "eventually" $(\Diamond)$ and "always" $(\Box)$ can still be verified after removing spurious self-loops, since they do not contain the "next" operator $(\bigcirc)$.

6 Verification of LTL Formulas

We have seen how we can formulate system specifications for dynamical systems in Linear-time Temporal Logic. The dynamical system Σ can be seen as a transition system, and we have presented a way to make a finite abstraction of this system. In this section, we will exploit a fact we

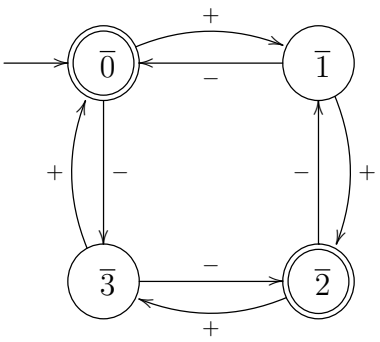

Figure 6: Finite state automaton representing the finite cyclic group of integers modulo four. The initial state is $0 \pmod 4$, as can be seen by the unlabeled incoming arrow. The states $0 \pmod 4$ and $2 \pmod 4$ are final states, and have a double circle around them to show this.

know from the literature [5, 6], namely that any LTL formula can be expressed as a particular kind of transition system, called a *Büchi automaton*. The reader will first be introduced with the general notion of a *finite state automaton*. Finally, we present the *Büchi product automaton*, which is the concept that is used to check if a system specification is satisfied.

6.1 Finite State Automaton

A finite state automaton is very similar to a transition system. The difference in this context is that an automaton produces no output, but it has a designated initial state, and a designated set of final states.

Definition 11. A (possibly non-deterministic) *finite state automaton* is a tuple $M = (Q, q_0, U, \delta, \mathcal{F})$, where

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- U is the input alphabet,
- $\delta : Q \times U \to 2^Q$ is the transition map, and
- $\mathcal{F} \subset Q$ is the set of final states.

The input alphabet is sometimes referred to as the set of transition labels. Because of non-determinism, finite words $u[0] \dots u[n]$, for some n, will determine a number of runs $p \geq 0$. For each run $q[0] \dots q[n+1]$ of the (possibly empty) collection of runs of M determined by $u[0] \dots u[n]$, each $q[0] = q_0$, and $q[k+1] \in \delta(q[k], u[k])$ for $0 \leq k \leq n$. An input word is said to be *accepted* by M, if at least one of the runs ends in an accepting state, so $q[n+1] \in F$ for some run $q[0] \dots q[n+1]$. The empty input word, denoted by ϵ, is accepted if and only if $q_0 \in F$.

The collection of all finite words that M accepts is called the *language* of M, and is denoted by $\mathscr{L}_M$. The language $\mathscr{L}_M$ is a subset of U^*, which is informally defined here as the collection of all finite words over the alphabet U.

Example 11. Consider the finite state automaton $M = (Q, q_0, U, \delta, \mathcal{F})$ as shown in Figure 6, where

- $Q = \mathbb{Z}/\mathbb{Z}_4$ is the group of integers modulo four,
- $q_0 = 0 \pmod 4$ is the initial state,
- $U = \{+, -\}$ is the input alphabet,
- $\delta(q, u) = \begin{cases} q + 1 & \text{if } u = + \\ q - 1 & \text{if } u = -, \end{cases}$
- and $\mathcal{F} = \{0 \pmod 4, 2 \pmod 4\} \subset Q$.

Here, addition and subtraction by δ are done modulo four. The automaton M is deterministic and non-blocking. It accepts for example the empty input word ϵ, and the words "$++$", "$+-$", "$--$", "$++++$", "$+--+$". The words "$+$", "$-$", "$+-+$" are examples that are not accepted by M. In fact, in general we have

$$\mathscr{L}_M = \{w \in U^* \mid \text{length of } w \text{ is even}\}.$$

Set inclusion from right to left can be shown using induction on the length of the word, where the base case is the empty word ϵ, and the inductive step is a case analysis of which pair of symbols, "$++$", "$+-$", "$-+$", or "$--$", was added to the word.

6.2 Büchi Automata

A Büchi automaton is a particular kind of finite state automaton. The major difference between Büchi automata and finite state automata is that the semantics are defined over infinite input words, rather than over finite words. This is very convenient, since satisfaction of LTL formulas is also defined over infinite words.

It is known that every LTL formula ϕ has a corresponding Büchi automaton capable of accepting the same language. Intuitively, output sequences of the system Σ in (1) should be the input of the Büchi automaton if we want to check that Σ satisfies the specification ϕ. If an output word generated by a trajectory of Σ is then accepted, this trajectory of Σ satisfies the formula ϕ. For this reason, it is no coincidence that the input alphabet has been changed to 2^O, where O is as in (1), in the following definition:

Definition 12. A possibly non-deterministic Büchi automaton is a tuple $B = (Q, Q_0, 2^O, \delta, \mathcal{F})$, where

- Q is the set of states,
- $Q_0 \subset Q$ is the set of initial states,
- 2^O is the input alphabet,
- $\delta : Q \times 2^O \to Q$ is the transition function, and
- $\mathcal{F} \subset Q$ is the set of *accepting* states.

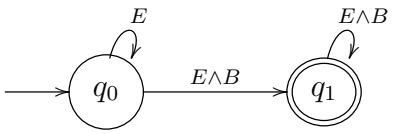

Figure 7: Büchi automaton accepting the same language as $\phi = \Box E \wedge \Diamond\Box B$ in Example 12. The boolean formulas represent the set of sets that enable the transitions.

A Büchi automaton will accept sets of observations $y[k]$ as input, where $y[k] \in 2^O$ for all $k \geq 0$. A transition with a subset of O as input is possible if and only if the conjunction of all observations in it is evaluated as true. Input and trajectories are defined completely analogue to regular finite state automata, only the input words $y[\cdot]$ and the runs $q[\cdot]$ are infinite instead of finite.

An important note is that we changed the name of $\mathcal{F} \subset Q$ to the set of *accepting* states, rather than final states. This is because acceptance is defined over infinite runs, and infinite runs evidently have no last state. Acceptance for Büchi automatons is defined as follows:

Definition 13. Let $B = (Q, Q_0, 2^O, \delta, \mathcal{F})$ be a Büchi automaton and let $\inf(q[\cdot])$ denote the set of states ocurring infinitely often in $q[\cdot]$. Then $q[\cdot]$ is called an *accepting run* of B, if $\inf(q[\cdot]) \cap \mathcal{F} \neq \emptyset$. An infinite input word $y[\cdot]$ is *accepted* by B if it defines at least one accepting run.

We say that *at least* one accepting run should exist for an input, because the automaton is non-deterministic. Intuitively, a run of B that is not accepting either reaches a state q and an input y for which $\delta(q, y) = \emptyset$, or it enters a cycle of states and none of the states are in $\mathcal{F}$.

The collection of all infinite input words that is accepted by a Büchi automaton B is denoted by $\mathscr{L}_B$. As shown in [5] and [6], there exists a Büchi automaton for every LTL formula, such that they accept exactly the same language. The authors of [5] published an online software tool that constructs a corresponding Büchi automaton when providing it with an LTL formula. A Büchi automaton that accepts the same language as an LTL formula ϕ is denoted by B_ϕ. An example is given as follows.

Example 12. Consider the robot as in Examples 1, 7, and 8, with $O = \{A, B, D, E\}$. We wish to verify the LTL formula

$$\phi = \Box E \wedge \Diamond\Box B.$$

A Büchi automaton accepting the same language as ϕ is $B_\phi = (Q, Q_0, \delta, 2^O, \mathcal{F})$, where

- $Q = \{q_0, q_1\}$ is the set of states,
- $Q_0 = \{q_0\}$ is the set of initial states,
- The input alphabet is $\{\emptyset, \{A\}, \{B\}, \{D\}, \{E\}, \{A, B\}, \ldots, \{A, B, D, E\}\}$,
- δ is defined such that

$$\begin{array}{ll} q_0 \in \delta(q_0, y) & \text{iff } E \in y \\ q_1 \in \delta(q_0, y) & \text{iff } E \in y \text{ and } B \in y \\ q_0 \notin \delta(q_1, y) & \text{for all } y \\ q_1 \in \delta(q_1, y) & \text{iff } E \in y \text{ and } B \in y \end{array}$$

- and $F = \{q_1\}$ is the set of accepting states.

The automaton is shown in Figure 7. It is non-deterministic, and it blocks for any input y that does not contain E, as reflected by the conjunct $\Box E$. For the transitions going into the single accepting state, observation B is necessary, as reflected by the conjunct $\Diamond\Box B$. Note that non-determinism allows a run to stay in q_0, even if E and B are both true. More specifically, for the input $\{E, B\}$ we have $\delta(q_0, \{E, B\}) = \{q_0, q_1\}$. In Figure 7, the set of sets that enable a transition is represented by a boolean formula. For example, going from q_0 to q_1, $E \wedge B$ is written to represent all elements of 2^O that contain both E and B: $\{E, B\}, \{E, B, A\}, \{E, B, D\}$ and $\{E, B, A, D\}$.

6.3 Product Automaton

Here we define the product automaton of a finite state abstraction A and a Büchi automaton B_ϕ for an LTL formula ϕ. The product automaton will define all runs of A that satisfy ϕ. The set of states for the product automaton is the cartesian product of the two sets of states, and a transition from one pair of states to another is only possible under two conditions. First, for the two states of A in the two pairs, the transition must have been possible in A. Second, for the two states of B_ϕ, the transition must have been possible in B_ϕ given the input that is the output of A in the current state.

Definition 14. An uncontrolled *Büchi product automaton* $P = A \otimes B$ of a finite state abstraction $A = (Q_A, \delta_A, O, o)$ as in Definition 8 and a Büchi automaton $B = (Q_B, Q_{0B}, 2^O, \delta_B, \mathcal{F}_B)$ as in Definition 12, is a tuple $P = (Q_P, Q_{0P}, \delta_P, \mathcal{F}_P)$, where

- $Q_P = Q_A \times Q_B$ is the set of states,
- $Q_{0P} = Q_A \times Q_{0B} \subset Q_P$ is the set of initial states,
- $\delta_P : Q_P \to Q_P$ is the unlabeled transition function such that

$$\delta_P((q_A, q_B)) = \{(q'_A, q'_B) \in Q_P \mid q'_A \in \delta_A(q_A) \text{ and } q'_B \in \delta_B(q_B, o(q_A)), \text{ and} \tag{7}$$

- $\mathcal{F}_P = Q_A \times \mathcal{F}_B$ is the set of accepting states.

The Büchi product automaton is a specific Büchi automaton, where the input is a singleton, and therefore omitted. For this reason, it is also referred to as the *uncontrolled* Büchi product automaton, because every run is non-deterministically defined by the same infinite input word.

The next lemma formalizes that runs of a product automaton $P = A \otimes B_\phi$ correspond to a trajectory of A whose output satisfies ϕ, and conversely, that every such run of A is indeed in P. This is a very significant result, because we will have obtained a finite state automaton that contains the necessary and sufficient information that we need in order to know whether a given system Σ satisfies an LTL formula ϕ.

To prove the lemma, we need some extra notations. We define the mappings $\alpha : Q_P \to Q_A$ as $(q_A, q_B) \mapsto q_A$ and $\beta : Q_P \to Q_B$ as $(q_A, q_B) \mapsto q_B$. Then α extends naturally to mapping subsets of Q_P, and an infinite sequence $q_P[\cdot]$ to a sequence of states of A by $\alpha(q_P[\cdot]) = \alpha(q_P[0])\alpha(q_P[1])\ldots$, and similarly for β.

Lemma 3. Let A be a finite state abstraction, ϕ be an LTL formula, and B_ϕ a corresponding Büchi automaton. Also, let $P = A \otimes B_\phi$ be the Büchi product automaton as in Definition 14. Let $q_P[\cdot]$

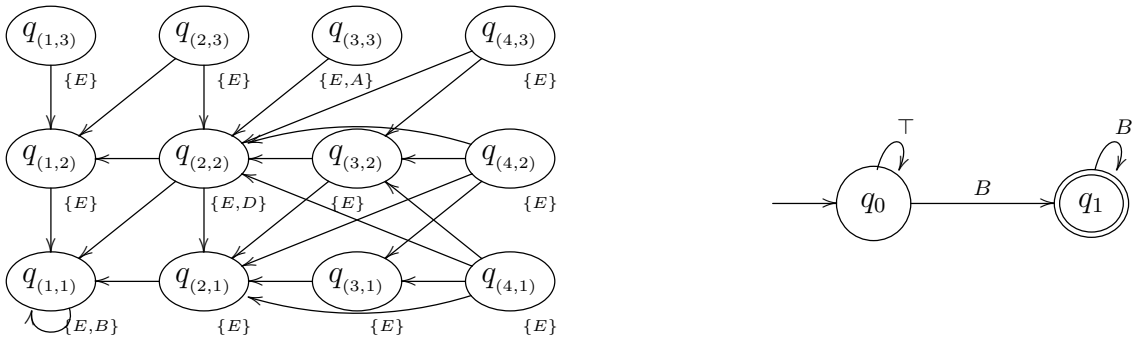

Figure 8: Finite state abstraction (a) of the robot dynamics using $F(x) = Cx$ in Example 8, with the spurious self-loops removed, and Büchi automaton (b) accepting the same language as $\phi = \Diamond\Box B$. See Example 13.

be a run of P. Then $q_P[\cdot]$ is an accepting run if and only if $\alpha(q_P[\cdot])$ is a run of A that defines an output satisfying ϕ.

Proof. For the implication from left to right, let $q_A[\cdot] = \alpha(q_P[\cdot])$ and $q_B[\cdot] = \beta(q_P[\cdot])$. By definition of P, more specifically (7), we have $q_P[k+1] \in \delta_P(q_P[k])$ implies $q_A[k+1] \in \delta_A(q_A[k])$ for each $k \geq 0$, so $q_A[\cdot]$ is indeed a run of A. Next, we show that this trajectory $q_A[\cdot]$ defines an infinite output word $o(q_A[\cdot])$ that satisfies ϕ, or equivalently, that is accepted by B_ϕ.

- First, observe that $q_B[0] \in Q_{0B}$ by construction of the set of initial states.
- By construction (7), $q_P[k+1] \in \delta_P(q_P[k])$ implies $q_B[k+1] \in \delta_B(q_B[k], o(q_A[k]))$ for all $k \geq 0$. Hence, $o(q_A[\cdot])$ is an infinite word that defines the run $q_B[\cdot]$ of B_ϕ.
- Since $q_P[\cdot]$ is an accepting run of P, by Definition 13, $\inf(q_P[\cdot]) \cap F_P \neq \emptyset$. Then also $\beta(\inf(q_P[\cdot])) \cap \beta(F_P) = \inf(q_B[\cdot]) \cap F_B \neq \emptyset$, and the infinite word $o(q_A[\cdot])$ defines an accepting run of B_ϕ.

So indeed, $\alpha(q_P[\cdot])$ is a run of A that defines an output that satisfies ϕ. The converse implication also follows readily from the definition of P in a similar way. ☺

Using Lemma 3, we can prove a lemma that is the last missing step in providing an answer to the question described in Problem 1.

Lemma 4. Let A, ϕ, B_ϕ, and $P = A \otimes B_\phi$ be as in Lemma 3. If every run of P is an accepting run, then $A \vdash \phi$.

Proof. First, we apply Lemma 3 to conclude that all runs of P map to runs of A that define an output satisfying ϕ. What remains to show is that there are no runs of A that do not satisfy ϕ. Working towards a contradiction, we assume there exists $q_A[\cdot]$ of A that defines $y[\cdot] = o(q_A[\cdot])$ such that $y[\cdot] \nvdash \phi$. Then $y[\cdot]$ is not accepted by B_ϕ, and hence, it does not define an accepting run. By construction of P there should then also exist a run that is not an accepting run, and this would contradict the assumption of the lemma. We conclude that there is no run of A that defines an output sequence that does not satisfy ϕ, and hence, $A \vdash \phi$. ☺

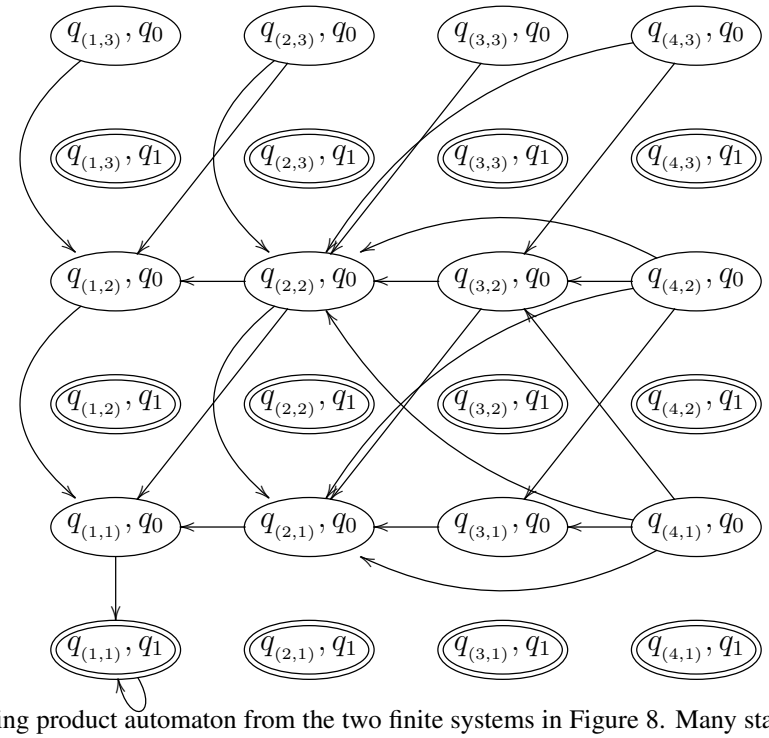

Figure 9: The resulting product automaton from the two finite systems in Figure 8. Many states are unreachable, and all runs go to $(q_{(1,1)}, q_1)$ and remain there indefinitely. See Example 13.

Example 13. Consider the system that describes the robot dynamics on $[0, 6) \times [0, 4)$ with a set of observations $O = \{A, B, D, E\}$, as in Examples 1, 7, and 8. A finite state abstraction for this is given in Figure 5b. For this abstraction, we wish to verify the formula

$$\phi = \Diamond\Box B.$$

This formula does not contain the "next" operator $\bigcirc$, so by Proposition 1, we can remove the spurious transitions that were discussed in Example 10. The resulting abstraction A' without spurious self-loops is given in Figure 8a. The Büchi automaton B_ϕ that accepts the same language as ϕ is given in Figure 8b. The resulting product automaton $P = A' \otimes B_\phi$ is shown in Figure 9. Notice that many states are unreachable. This is due to the fact that q_1 can only be reached if observation B is done. It can be seen that all runs of P are accepting, and applying Lemmas 1, 2, and 4, we can conclude that the continuous system satisfies the formula ϕ.

7 Main Results

In the following results, we let Σ be the system as in (1), with a set of observations O. The transition system T_Σ is then as in (2), and A will denote a finite state abstraction of T_Σ as in Definition 8. Also, let ϕ be a given LTL formula over the set of observations O, as in Definition 1. Recall that we write $\Sigma \vdash \phi$ if Σ satisfies ϕ as in Definition 3. As in Definition 12, the Büchi automata denoted by B_ϕ and $B_{\neg\phi}$ are given such that they accept the same language as ϕ and $\neg\phi$ respectively. Also,

accepting runs of a Büchi automaton are defined in Definition 13. Finally, P_ϕ will denote the uncontrolled Büchi product automaton $A \otimes B_\phi$ as in Definition 14, and similarly for $P_{\neg\phi}$.

The following result is realized directly from Lemmas 1, 2, and 4.

Theorem 1. If *every* run of P_ϕ is an accepting run, then $\Sigma \vdash \phi$.

Software tools can be used to check all runs of a Büchi automaton for acceptance, which makes this result very significant. For practical reasons, it is easier to check the negation of a formula. Specifically, tools allow us more easily to check the uncontrolled Büchi product automaton for *emptiness*, which means that no run is accepting. The following result, which is logically equivalent to Theorem 1, is therefore often applied in practice.

Corollary 1. If *no* run of $P_{\neg\phi}$ is an accepting run, then $\Sigma \vdash \phi$.

Proof. For contradiction, assume $\Sigma \nvdash \phi$, then using the contrapositive of Theorem 1, there would exist a run of P_ϕ that is not an accepting run. Then, by Lemma 3, there exists a trajectory in Σ that does not satisfy ϕ. This trajectory satisfies $\neg\phi$ by Definition 2. Then, again by Lemma 3, $P_{\neg\phi}$ would have a run that is accepting, which is a contradiction. We conclude that $\Sigma \vdash \phi$. ☺

Spurious self-loops often are a hindrance when we want to apply these results. For many applications, LTL formulas without the "next" operator suffice, so the next corollary is a powerful result. We denote the abstraction A with all spurious self-loops removed by A'. Also, P'_ϕ denotes the uncontrolled Büchi product automaton $A' \otimes B_\phi$, and similarly for $P'_{\neg\phi}$.

Corollary 2. If ϕ does not contain the "next" operator $\bigcirc$, then

(i) If every run of P'_ϕ is an accepting run, then $\Sigma \vdash \phi$.

(ii) If no run of $P'_{\neg\phi}$ is an accepting run, then $\Sigma \vdash \phi$.

Proof. This is a consequence of Lemma 4 and Proposition 1, and using similar reasoning as in the proof of Corollary 1. ☺

8 Conclusions

To formulate specifications expressing non-asymptotic properties of discrete-time dynamical systems, we can use Linear-time Temporal Logic. To verify that a system Σ satisfies an LTL formula ϕ is not a trivial job. This is because in principle, the number of possible trajectories defined by Σ is infinite.

Here, we summarize the procedure of verifying a specification ϕ for a system Σ. Figure 10 illustrates the steps that were taken in the theory of this procedure. We also discuss the relevance of the results, and how they can be applied in practice.

To make the number of possible trajectories finite, a non-deterministic abstraction of Σ is constructed. This construction is done by partitioning the state space of Σ in a convenient way. Constructing the one-step reachable set of states for the abstraction can be done very efficiently by computing an overapproximation of the image of each part by using mixed monotonicity.

The abstraction simulates the original system, which means that all possible infinite output sequences generated by Σ are also generated by its abstraction. Satisfaction of a formula ϕ by an abstraction would thus imply that the system Σ also satisfies the specification.

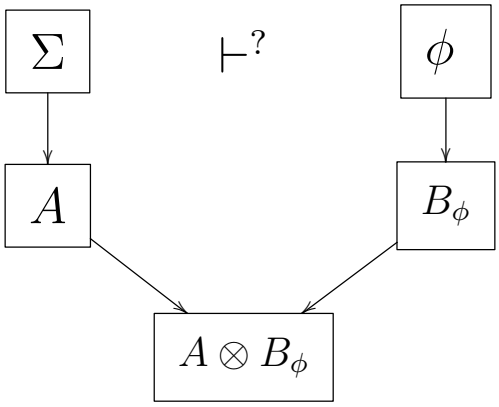

Figure 10: Illustration of the procedure to answer the question formulated in Problem 1.

Büchi automata can be constructed in such a way that they accept the same language as LTL formulas. The Büchi product automaton can capture all trajectories of the abstraction that satisfy a formula. Using this theory, we know that the original system satisfies the specification, if the runs of the product automaton of the abstraction and the Büchi are all accepting runs.

To our knowledge, proving correctness of the procedure of model checking for discrete-time dynamical systems has not yet been made as explicit as in this thesis. In most texts, *bisimulation* is used to define correspondence between a system and its abstraction. In this thesis, we have introduced simulation of states and systems directly, which significantly simplifies the theory. Also, although commonly applied, a proof of Proposition 1 was nowhere to be found in the literature.

The theory allows us to verify LTL specifications for dynamical systems using software that can check product automata for accepting runs. For example, the tool SPIN (Simple Promela Interpreter) [7] is a very popular tool for model checking. An introduction to the language Promela and the tool SPIN can be found in Appendix B. A report of a case study involving a system modeling the dynamics of a population of beetles can be found in Appendix C.

Application of the theory suffers from the "curse of dimensionality", because the number of states of the abstraction increases exponentially with the number of dimensions of the continuous system. Model checking for systems with three, four, or more dimensions can then quickly become infeasible for state-of-the-art computers, depending also on the size of the partition.

The case study in the appendix failed to prove a specification for the entire domain. However, restricting the initial values to a certain region, we were able to prove the that the specification was satisfied. An interesting problem for future work is to explore how we can find the *greatest satisfying region* as a subset of the domain, where any initial value chosen from this region will define trajectories satisfying the specification. This theory is discussed more elaborately in [1].

For systems modeled with input, it would be interesting to synthesize a control strategy such that a given specification holds. Research into control synthesis has been done in for example [1, 3]. The results are for finite-state transition models, and it would be exciting to explore the possibilities of translating the control strategy for an abstraction back to the continuous domain.

An acknowledgement of great gratitude goes out to the supervisors of this thesis for the continuous guidance and a repeated provision of constructive feedback during the process of the research. The author would also like to thank Samuel Coogan for providing help with the case study in the appendix.

References

[1] C. Belta, B. Yordanov, and E. A. Gol. *Formal Methods for Discrete-Time Dynamical Systems*, volume 89. Springer, 2017.

[2] S. Coogan and M. Arcak. Efficient finite abstraction of mixed monotone systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, pages 58–67, New York, NY, USA, 2015. ACM.

[3] S. Coogan, M. Arcak, and C. Belta. Formal methods for control of traffic flow. 2016.

[4] R. Costantino, J. Cushing, B. Dennis, and R. A. Desharnais. Experimentally induced transitions in the dynamic behaviour of insect populations. *Nature*, 375(6528):227, 1995.

[5] P. Gastin and D. Oddoux. Fast ltl to büchi automata translation. In *International Conference on Computer Aided Verification*, pages 53–65. Springer, 2001.

[6] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification XV*, pages 3–18. Springer, 1995.

[7] G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[8] S. K. Khaitan and J. D. McCalley. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, 9(2):350–365, 2015.

[9] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.

Appendices

A Proof of Proposition 1

This section contains a proof for Proposition 1, which is stated again here for convenience.

Proposition 1. Let T_Σ be as in (1) and A be its finite state abstraction as in Definition 8. If A' is the abstraction A with all spurious self-loops removed, and ϕ is an LTL formula without the "next" operator $\bigcirc$, then

$$A' \vdash \phi \implies T_\Sigma \vdash \phi$$

For the given O, let $W = (2^O)^\omega$ denote the collection of all infinite words $\xi = y_0 y_1 \dots$ where each $y_k \in 2^O$ for $0 \leq k \in \mathbb{Z}$. Next, let $\Omega(T) \subset W$ denote the collection of all possible infinite output words generated by a transition system T with observation set O. Note that by Lemma 1 and 2, $\Omega(T_\Sigma) \subset \Omega(A)$.

We first coinductively define a relation $\mathcal{R}$ on infinite words. For any possible infinite output word $\xi \in \Omega(T_\Sigma)$ then, there exists a word $\xi' \in \Omega(A')$ such that $(\xi, \xi') \in \mathcal{R}$. For the proof then, we will use induction on the structure of the formula ϕ to show that $\xi \models \phi \iff \xi' \models \phi$ for all $(\xi, \xi') \in \mathcal{R}$.

Definition 15. The relation $\mathcal{R} \subset W \times W$ is coinductively defined as follows: whenever $(\mu, \xi) \in \mathcal{R}$, then

(i) If $\mu = y\mu'$ for some $y \in 2^O$ and $\mu' \in W$, then there exists $\xi' \in W$ such that $\xi = y\xi'$ and $(\mu', \xi') \in \mathcal{R}$,

(ii) If $\mu = yy\mu'$ for some $y \in 2^O$ and $\mu' \in W$, then there exists $\xi' \in W$ such that $\xi = y\xi'$ and $(y\mu', y\xi') \in \mathcal{R}$.

The first property says that for any $\xi \in W$, we have $(\xi, \xi) \in \mathcal{R}$. The second property intuitively allows repetitions in the first word to be removed in the second. This reflects the fact that spurious transitions are removed in A'. Indeed, the reader should now be convinced that for every $\xi \in \Omega(T_\Sigma)$, there exists $\xi' \in \Omega(A')$ such that $(\xi, \xi') \in \mathcal{R}$.

Consider the following examples with $a, b, c \in 2^O$. By property (i), $(bc^\omega, bc^\omega) \in \mathcal{R}$ and $(bbc^\omega, bbc^\omega) \in \mathcal{R}$. Then, considering property (ii), also (bbc^ω, bc^ω) and $(aabbbc^\omega, abbc^\omega)$.

Proof. The proof is to show that the biconditional $\xi \models \phi \iff \xi' \models \phi$ holds for all $(\xi, \xi') \in \mathcal{R}$. The proof for this is done by induction over the structure of ϕ and applying the rules of the semantics, where we can omit the rule for $\bigcirc$. For $(\xi, \xi') \in \mathcal{R}$, we write $\xi = y_0 y_1 \dots$ and $\xi' = y'_0 y'_1 \dots$.

- The base case for $\phi = \top$ is immediately satisfied.
- For $\phi = p$ with $p \in O$, it follows that $\xi \models p \iff \xi' \models p$ by properties (i) or (ii) of $\mathcal{R}$ regardless.
- The classical logical operators for negation, conjunction, and disjunction are all done in similar ways, so we only show negation here. If $\phi = \neg\psi$, we may assume using the induction

hypothesis that for all $(\xi, \xi') \in \mathcal{R}$, indeed $\xi \models \psi \iff \xi' \models \psi$. We then use this and the semantics to show that

$$\xi \models \neg\psi \iff \xi \not\models \psi \iff \xi' \not\models \psi \iff \xi' \models \neg\psi,$$

and hence, $\xi \models \phi \iff \xi' \models \phi$.

- The important and only remaining case is when $\phi = \psi \; U \; \eta$. Here, the induction hypothesis is that for every $(\xi, \xi') \in \mathcal{R}$, $\xi \models \psi \iff \xi' \models \psi$ and $\xi \models \eta \iff \xi' \models \eta$.

  First, assume $\xi' \models \psi \; U \; \eta$. Then there exists j' such that $\xi'_{j'} \models \eta$ and $\xi'_{i'} \models \psi$ for $0 \leq i' < j'$. Since $(\xi, \xi') \in \mathcal{R}$, we can repeatedly apply one of the coinductive steps (i) or (ii) to arrive at some $j \geq j'$ such that $(\xi_j, \xi'_{j'}) \in \mathcal{R}$. Since $\xi'_{j'} \models \eta$, by the induction hypothesis then also $\xi_j \models \eta$. Moreover, by definition of $\mathcal{R}$, $(\xi_i, \xi'_{i'}) \in \mathcal{R}$ for every $0 \leq i < j$ and some $0 \leq i' < j'$. Then, applying the induction hypothesis will give us $\xi'_{i'} \models \psi$ for some i' implies that $\xi_i \models \psi$ for all $0 \leq i < j$. We have now shown that $\xi' \models \psi \; U \; \eta \implies \xi \models \psi \; U \; \eta$ for arbitrary $(\xi, \xi') \in \mathcal{R}$ by finding $j \geq 0$, more specifically, $j \geq j'$, such that $\xi_j \models \eta$ and $\xi_i \models \psi$ for $0 \leq i < j$.

  The converse is shown analogously by finding a $j' \leq j$ that shows that $\xi \models \phi \implies \xi' \models \phi$.

This concludes the proof of the claim that $\xi \models \phi \iff \xi' \models \phi$ for all $(\xi, \xi') \in \mathcal{R}$.

If now $T_\Sigma \not\vdash \phi$, then by Definition 3, there is a trajectory of T_Σ that defines an output ξ such that $\xi \not\models \phi$. Then A' has an output ξ' such that $(\xi, \xi') \in \mathcal{R}$, and by the claim that was just shown it must be that $\xi' \not\models \phi$. Then $A \not\vdash \phi$. This shows the contrapositive of the proposition, and the proof is thus finished. ☺

B Introduction to SPIN

The software tool Simple Promela Interpreter (SPIN) was originally developed for formal verification of multi-threaded software applications. For us, this is useful, because the language Promela (Process Meta Language) allows us to encode the finite state abstraction as an active process. This means that a software tool will be able to go through every possible trajectory of the abstraction. The following lines encode the abstraction given in Figure 8 in the Promela language.

```
/* Found 4 candidate stutter states
 * and removed 3 actual spurious self-loops.
 */
bool a = 0;
bool b = 0;
bool d = 0;
bool e = 0;

active proctype FSA() {
q_init:
  if
  :: (true) -> goto q_0;
  :: (true) -> goto q_1;
  :: (true) -> goto q_2;
  :: (true) -> goto q_3;
  :: (true) -> goto q_4;
  :: (true) -> goto q_5;
  :: (true) -> goto q_6;
  :: (true) -> goto q_7;
  :: (true) -> goto q_8;
  :: (true) -> goto q_9;
  :: (true) -> goto q_10;
  :: (true) -> goto q_11;
  fi
q_0:    /* I_q = [ (0,0) , (1,1) ] */
  atomic { a=0; b=1; d=0; e=1;  }
  if
  :: true -> goto q_0;
  fi
q_1:    /* I_q = [ (1,0) , (3,1) ] */
  atomic { a=0; b=0; d=0; e=1;  }
  if
  :: true -> goto q_0;
  fi
q_2:    /* I_q = [ (3,0) , (4,1) ] */
  atomic { a=0; b=0; d=0; e=1;  }
  if
  :: true -> goto q_1;
  fi
q_3:    /* I_q = [ (4,0) , (6,1) ] */
```

```
  atomic { a=0; b=0; d=0; e=1;  }
  if
  :: true -> goto q_1;
  :: true -> goto q_2;
  :: true -> goto q_5;
  :: true -> goto q_6;
  fi
q_4:    /* I_q = [ (0,1) , (1,3) ] */
  atomic { a=0; b=0; d=0; e=1;  }
  if
  :: true -> goto q_0;
  fi
q_5:    /* I_q = [ (1,1) , (3,3) ] */
  atomic { a=0; b=0; d=1; e=1;  }
  if
  :: true -> goto q_0;
  :: true -> goto q_1;
  :: true -> goto q_4;
  fi
q_6:    /* I_q = [ (3,1) , (4,3) ] */
  atomic { a=0; b=0; d=0; e=1;  }
  if
  :: true -> goto q_1;
  :: true -> goto q_5;
  fi
q_7:    /* I_q = [ (4,1) , (6,3) ] */
  atomic { a=0; b=0; d=0; e=1;  }
  if
  :: true -> goto q_1;
  :: true -> goto q_2;
  :: true -> goto q_5;
  :: true -> goto q_6;
  fi
q_8:    /* I_q = [ (0,3) , (1,4) ] */
  atomic { a=0; b=0; d=0; e=1;  }
  if
  :: true -> goto q_4;
  fi
q_9:    /* I_q = [ (1,3) , (3,4) ] */
  atomic { a=0; b=0; d=0; e=1;  }
  if
  :: true -> goto q_4;
  :: true -> goto q_5;
  fi
q_10:   /* I_q = [ (3,3) , (4,4) ] */
  atomic { a=1; b=0; d=0; e=1;  }
  if
  :: true -> goto q_5;
```

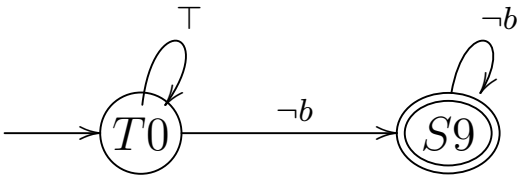

Figure 11: Büchi automaton accepting the same language as $\neg\Diamond\Box b$

```
  fi
q_11:   /* I_q = [ (4,3) , (6,4) ] */
  atomic { a=0; b=0; d=0; e=1;  }
  if
  :: true -> goto q_5;
  :: true -> goto q_6;
  fi
}
```

The assignments within the atomic statement update the set of observations for each state accordingly. Note that the booleans are initialized as false, which is a minor unsatisfactory feature of this encoding. For the LTL formula that we will work with, this is however not an issue.

For this abstraction, we wish to verify $\phi = \Diamond\Box B$. For this, a Büchi automaton can also be encoded in Promela. In fact, SPIN itself provides the means to translate an LTL formula into an automaton. To acquire the automaton corresponding to $\neg\phi$ in Promela code, one enters `spin -f '!<>[]b'` in a command-line interface. In SPIN, we use lower-case letters for propositions. This results in the automaton encoded in Promela code, given as

```
never  {    /* ! <> [] b */
T0_init:
  do
  :: (! ((b))) -> goto accept_S9
  :: (1) -> goto T0_init
  od;
accept_S9:
  do
  :: (1) -> goto T0_init
  od;
}
```

This Büchi automaton within the never-claim is also shown graphically in Figure 11. The never-claim makes sure that the automaton modeled within it has no accepting runs. After putting both the process and the automaton together in one file named `model-example.pml`, and running the command `spin -run model-example.pml`, the tool will start checking the cycles of the product automaton. Within a second, zero errors are detected, which means that the never-claim is satisfied. We conclude that the Büchi product automaton with the negation of ϕ has no accepting runs, and by Corollary 2, the continuous system in Example 8 satisfies $\phi = \Diamond\Box B$.

C Case Study

In this appendix we make an attempt at applying Corollary 2 to a system that models the population dynamics of a cannibalistic beetle, similarly to the case study in [2], based on [4]. For this, the software tool SPIN is used. See also Appendix B.

Dynamical System

The discrete-time dynamical system Σ is given by $x[t+1] = F(x[t])$, where each $x[t] \in \mathbb{R}^3$, and $o : \mathbb{R}^3 \to O$, where $O = \{p, q, r\}$. The functions F and o are given by

$$F(x) = \begin{pmatrix} 0 & 0 & b \cdot \exp(-c_{el}x^1 - c_{ea}x^3) \\ \mu_l & 0 & 0 \\ 0 & \exp(-c_{pa}x^3) & \mu_a \end{pmatrix} \begin{pmatrix} x^1 \\ x^2 \\ x^3 \end{pmatrix},$$
$$o(x) : \begin{cases} p \in o(x) & \text{iff } x_1 < 10, \\ q \in o(x) & \text{iff } x_3 \geq 40, \\ r \in o(x) & \text{iff } x_1 \geq 150, \end{cases} \tag{8}$$

where $x = (x^1, x^2, x^3) \in \mathbb{R}^3$. Here, x^1 represents the size of the population of larvae, x^2 that of the pupae, and x^3 is the number of adult beetles. The parameters $\mu_l, \mu_a \in (0, 1]$ are survival probabilities of larvae and adults. The quantity b is the number of new larvae that are born in one time step in the absence of cannibalism. The parameters c_{el} and c_{ea} model the probability of eggs being eaten by larvae and adults. The parameter c_{pa} models the probability of a pupa being eaten by an adult.

The system is known to exhibit oscillating behaviour, which is a non-asymptotic property. We will try to verify a specification that says that it is always true that if there are very few larvae (p), but sufficient adult beetles (q), then at some point, the number of larvae will be high again (r). The following LTL formula specifies this behaviour:

$$\phi = \Box\big((p \wedge q) \to \Diamond r\big). \tag{9}$$

Notice that the formula is free of the "next" operator.

Abstraction

As shown in [2], the system is domain invariant on $\mathcal{X} = [0, (265, 225, 450))$. Using this, we may create a gridded partition as in Definition 9, where we pick $N_1 = 12$, $N_2 = 11$, and $N_3 = 18$, and

$$\begin{aligned} \gamma^1 &= (0, \mathbf{10}, 20, 40, 50, 60, 80, 100, 125, \mathbf{150}, 175, 200, 265) \\ \gamma^2 &= (0, 20, 40, 50, 60, 80, 100, 125, 150, 175, 200, 225) \\ \gamma^3 &= (0, 10, 20, \mathbf{40}, 50, 60, 80, 100, 125, 150, 175, 200, 225, 250, 275, 300, 325, 350, 450). \end{aligned}$$

Notice how this gridded partition respects the observation map. The partition gives us a finite state abstraction with exactly $12 \times 11 \times 18 = 2376$ states. For constructing δ using (6), we require that the

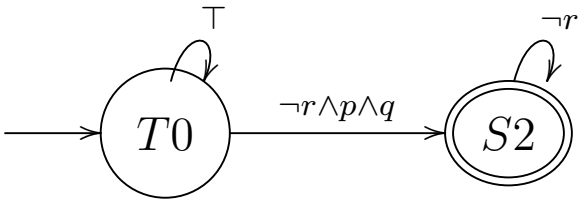

Figure 12: Büchi automaton accepting the same language as the negation of ϕ in (9)

system is mixed monotone, as in Definition 10. This can be shown by providing a decomposition function $f(x, y)$ such that $F(x) = f(x, x)$. Let $x = (x^1, x^2, x^3)$ and $y = (y^1, y^2, y^3)$, then

$$f(x,y) = \begin{pmatrix} 0 & 0 & b \cdot \exp(-c_{el}y^1 - c_{ea}y^3) \\ \mu_l & 0 & 0 \\ 0 & \exp(-c_{pa}y^3) & \mu_a \end{pmatrix} \begin{pmatrix} x^1 \\ x^2 \\ x^3 \end{pmatrix}$$

is non-decreasing in x, and non-increasing in y. Also, $F(x) = f(x, x)$, and hence, the system is mixed-monotone.

With a small C program that implements the function f, we compute the one-step reachable state set for each state. An implementation of Algorithm 1 detects that 14 out of 16 candidate self-loops are spurious. Corollary 2 says that we can safely remove these in the process of verification. The C program outputs the abstraction in the Promela language. The abstraction is modeled in such a way that entering a new state will update the set of observations accordingly.

Model Checking

A Büchi automaton that accepts the same language as the negation of ϕ is given as the following snippet of Promela code:

```
never { /* ! G ( ( p && q) -> F r) */
T0_init :    /* init */
  if
  :: (1) -> goto T0_init
  :: (!r && p && q) -> goto accept_S2
  fi;
accept_S2 :    /* 1 */
  if
  :: (!r) -> goto accept_S2
  fi;
}
```

The Büchi automaton in the never-claim is also shown graphically in Figure 12.

With the never-claim and the finite state abstraction running as an active process, SPIN runs every possible trajectory of the synchronous product automaton, and checks for acceptance. The tool returns an error, and we can conclude that the abstraction does *not* satisfy ϕ in (9). We are therefore unable to conclude that Σ in (8) satisfies ϕ.

If we restrict ourselves to the trajectories of Σ starting in $\mathcal{X}_r \subset \mathcal{X}$ defined as

$$\mathcal{X}_r = [(80, 80, 80), (125, 125, 125)),$$

then the abstraction has exactly eight designated initial states. The model for this abstraction with the never-claim is partly shown below.

Executing the command `spin -run model-insect.pml` takes almost four hours, and requires a computer with more than 4GB of memory. For this small initial region $\mathcal{X}_r$, the tool returns zero errors. By Corollary 2, we have thus verified that Σ in (8) satisfies ϕ in (9) for the initial region $\mathcal{X}_r$.

Many attempts of verification have failed, because of a poor choice of a partition. Understanding the dynamics of the original system is of utmost importance when choosing a partition for the abstraction. For this case study, in the lower regions, finer partitioning is necessary because there are the parts that are popular destinations, and spurious trajectories are likely to appear. On the other hand, coarser partitioning suffices for the higher regions, because of the exponentials, and because they are less often reached by other states.

```
/* Found 16 candidate stutter states
 * and removed 14 actual spurious self-loops.
 */
bool p = 0;
bool q = 0;
bool r = 0;

active proctype FSA() {
q_init:
  if
  :: (true) -> goto q_858;
  :: (true) -> goto q_859;
  :: (true) -> goto q_870;
  :: (true) -> goto q_871;
  :: (true) -> goto q_990;
  :: (true) -> goto q_991;
  :: (true) -> goto q_1002;
  :: (true) -> goto q_1003;
  fi
q_0:    /* I_q = [ (0,0,0) , (10,20,10) ] */
  atomic { p = 1; q = 0; r = 0;  }
  if
  :: (true) -> goto q_0;
  :: (true) -> goto q_1;
  :: (true) -> goto q_2;
  :: (true) -> goto q_3;
  :: (true) -> goto q_4;
  :: (true) -> goto q_5;
  :: (true) -> goto q_132;
  :: (true) -> goto q_133;
  :: (true) -> goto q_134;
  :: (true) -> goto q_135;
  :: (true) -> goto q_136;
  :: (true) -> goto q_137;
  :: (true) -> goto q_264;
  :: (true) -> goto q_265;
  :: (true) -> goto q_266;
  :: (true) -> goto q_267;
  :: (true) -> goto q_268;
  :: (true) -> goto q_269;
  fi
q_1:    /* I_q = [ (10,0,0) , (20,20,10) ] */
  atomic { p = 0; q = 0; r = 0;  }
  if
  :: (true) -> goto q_0;
  :: (true) -> goto q_2;

          ...

q_2374:   /* I_q = [ (175,200,350) , (200,225,450) ] */
```

```
  atomic { p = 0; q = 1; r = 1;  }
  if
  :: (true) -> goto q_1404;
  :: (true) -> goto q_1416;
  :: (true) -> goto q_1536;
  :: (true) -> goto q_1548;
  :: (true) -> goto q_1668;
  :: (true) -> goto q_1680;
  :: (true) -> goto q_1800;
  :: (true) -> goto q_1812;
  fi
q_2375:   /* I_q = [ (200,200,350) , (265,225,450) ] */
  atomic { p = 0; q = 1; r = 1;  }
  if
  :: (true) -> goto q_1416;
  :: (true) -> goto q_1428;
  :: (true) -> goto q_1440;
  :: (true) -> goto q_1548;
  :: (true) -> goto q_1560;
  :: (true) -> goto q_1572;
  :: (true) -> goto q_1680;
  :: (true) -> goto q_1692;
  :: (true) -> goto q_1704;
  :: (true) -> goto q_1812;
  :: (true) -> goto q_1824;
  :: (true) -> goto q_1836;
  fi
}

never { /* ! G ( ( p && q) -> F r) */
T0_init :    /* init */
  if
  :: (1) -> goto T0_init
  :: (!r && p && q) -> goto accept_S2
  fi;
accept_S2 :    /* 1 */
  if
  :: (!r) -> goto accept_S2
  fi;
}
```