Monitoring Hybrid Automata

Saarland University

Department of Computer Science

Bachelor's Thesis

submitted by

Paul Bungert

Saarbrücken, September 2019

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Maximilian Schwenger
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Dr. Swen Jacobs
Submission:	11 September, 2019

Abstract

Even carefully designed systems can fail due to environmental factors not anticipated at design time. Therefore, verification at the modelling stage is insufficient and needs to be complemented with methods that detect errors while the system is deployed. One technique designed for this task is runtime monitoring. Runtime monitors are deployed alongside systems to determine when predefined properties are violated.

A common choice for the specification of monitoring properties are temporal logics, as timing is an important factor in many systems interacting with the real world. However, temporal logics only provide a coarse abstract view of underlying systems. Thus, they cannot cover all monitoring properties and need to be complemented by alternatives.

This thesis assess how hybrid automata can be used for the specification of runtime properties. Originally a modeling tool, hybrid automata are well-suited to capture the complex continuous behavior and structures of systems under scrutiny. We present a monitoring algorithm that decides whether a system trace lies within the trace set of a hybrid automaton specification. The theoretical algorithm is implemented in MATLAB and demonstrated on two case studies.

Acknowledgements

First of all, I want to thank my advisor Maximilian Schwenger for his extraordinary guidance throughout this thesis. Thank you for always making time to answer my questions and for your awesome template [1].

Furthermore, I want to thank my supervisor Prof. Bernd Finkbeiner for giving me the opportunity for writing this thesis and for reviewing it.

I also want to thank Dr. Swen Jacobs for reviewing this thesis.

Lastly, I would like to thank my family, my friends and my girlfriend for their support and encouragement.

[1] github.com/Schwenger/Thesis-Template

Saarbrücken, 11 September, 2019

Contents

Chapter 1

Introduction

Our lives rely heavily on software and digital controllers. These systems are omnipresent, not just in our modern gadgets, like smartphones and TVs, but also in many safety critical environments, like on roads and in power plants. Thus, it is crucial that these systems behave correctly. During their design phase, a multitude of measures is taken to achieve this goal. These range from testing, an informal method with weak guarantees, to formal verification, which aims to guarantee correctness of a system by proving the absence of errors. But even carefully designed systems, which are provably correct with regards to a specification, can fail due to environmental factors not anticipated when designing these specifications. Therefore, verification at the modelling stage is insufficient and needs to be complemented with methods that detect errors while the system is deployed.

One technique designed for this task is runtime monitoring. Runtime monitors are deployed alongside systems and constantly asses the correctness of the systems' behavior. During their execution, the systems under scrutiny emit traces that carry information about their internal state. The monitor checks these traces against a specification. Most of the time, these specifications are defined in a temporal logic. Temporal logics are a good choice because timing is an essential property for many real-world systems. For example, it is crucial that a train gate is closed when a train approaches, such that nobody is injured on the tracks at the time of the train's arrival. An encoding of this example property in a temporal logic specification could be the following: When the action `trainApproaches!` is triggered, within some predefined time interval the action `gateClosed!` has to be issued.

While temporal logics are well-suited for these properties, they are limited to a coarse abstract view of the system under scrutiny. It is difficult or even impossible to use them to encode the continuous train movement to increase the precision of the monitoring.

This thesis aims to bridge this gap by using hybrid automata as a specification language for monitoring properties. Hybrid automata were originally designed as a modelling tool for hybrid systems. Their strength is the ability to unify discrete and continuous dynamics in one model. This makes them especially suited for the design of controllers, where a digital, and thus discrete, component controls a continuous process of the real world. They are, however, equally well-suited for monitoring purposes: The dynamics of the train in the example could be more precisely accounted for by a hybrid automaton since it is able to take the physics of the train's movement into account. Detailed properties like changes in acceleration and the incline of the tracks can be modelled as differential equations and thus encoded in a hybrid automaton specification. Ultimately, the resulting increase in precision leads to a better monitor that can help to mitigate dangerous situations, therefore making the real world safer. Additionally, hybrid automata are well-suited to capture the structure of systems due to their nature as a modelling tool. In conclusion, they can complement existing monitoring techniques in meaningful ways.

However, the increase in expressive power comes at the price of increasing the complexity of the monitoring problem. This thesis assess if and under which conditions the runtime monitoring of hybrid automata is feasible. To this end, we present a monitoring algorithm for a subclass of hybrid automata. We implemented the algorithm in MATLAB and demonstrate the implementation through two examples.

Outline

The necessary preliminaries on hybrid automata and runtime monitoring are laid out in Chapter 2. In Chapter 3, we explain how the monitoring problem for hybrid automata is encoded and solved. To this end, we construct an algorithm that decides if an input trace of a black box system lies within the set of traces defined by a hybrid automaton specification. If the property is satisfied, all plausible paths through the automaton are returned. Further, we outline optimizations and prove the correctness of the algorithm. In Chapter 4, we explain how the theoretical algorithm is practically implemented in MATLAB. We finish the chapter by demonstrating the implementation in two case studies — a thermostat and a lonely cleaning robot. In Chapter 5, we compare our approach to other techniques for runtime monitoring and hybrid automata analysis. The results of this thesis are recapitulated in Chapter 6.

Chapter 2

Background

The goal of this thesis is to monitor a systems whose properties are expressed in a hybrid automaton. This chapter introduces hybrid automata and runtime monitoring.

2.1 Hybrid Automata

A hybrid automaton, or *HA* for short, is a model of a hybrid system [ACHH93,Hen00]. Hybrid systems owe their name to the fact that they are a hybrid of discrete and continuous systems. An example of hybrid system that can be modeled with a hybrid automaton is a thermostat as depicted in Figure 2.1

Example 2.1 (Thermostat)**.** The variable x represents the temperature. In the location *cooling* the temperature falls according to the flow condition $\dot{x} = -0.1$, meaning $-0.1°$ per time unit. In the location *heating* the temperature rises with a rate of $0.2°$ per time unit. In the beginning, the temperature is $19°$ and the thermostat is cooling. The switch labeled with the action a_1 to the location *heating* can be taken as soon as the guard condition $x < 19$ is satisfied. The invariant condition of *cooling*, $x \geq 15$, enforces that the temperature can only drop to $15°$ before a_1 has to be taken. △

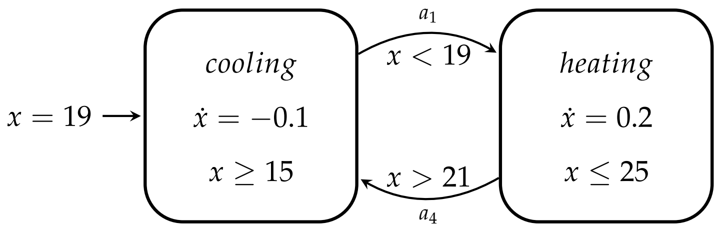

Figure 2.1: Basic Thermostat, adapted from [Hen00]

2.1.1 Syntax

This section gives definitions of the hybrid automaton model closely following the works by Henzinger and Alur [ACHH93, Hen00]. To better illustrate syntax and semantics of HA we use a simple thermostat (➔ Figure 2.2) as a leading example throughout this section.

➔ Sec. 2.1, Page 5

Def. Hybrid Automaton

Definition 1 (Hybrid Automata)**.** A *hybrid automaton* is a tuple $H = (X, V, E, flow, inv, init, action, guard)$ with the following components:

Variables A finite set $X = \{x_1, ..., x_n\}$ of real-numbered variables. We write $\dot{X}$ for the set $\{\dot{x_1}, ..., \dot{x_n}\}$ of dotted variables, which represent the first derivative during continuous change. We write X' for the set $\{x'_1, ..., x'_n\}$ of primed variables, which represent the values of the variables after a discrete change. We denote this discrete change as a *reset* of a variable x_i to a value x'_i.

Control Graph A finite directed multigraph (V, E). The vertices in V are called *locations*. The edges are called *switches*.

Flows A vertex labeling function $flow : V \mapsto (\mathbb{R}_0^+ \times \mathbb{R}^n) \mapsto \mathbb{R}^n$ which assigns a flow condition to each location $\ell \in V$. Each flow condition $flow(\ell)$ is a predicate whose free variables are from $X \cup \dot{X}$.

Invariants A vertex labeling function $inv : V \mapsto \mathbb{R}^n \mapsto \mathbb{B}$ which assigns an invariant condition to each location $\ell \in V$. Each invariant condition $inv(\ell)$ is a predicate whose free variables are from X.

Initial Conditions A vertex labeling function $init : V \mapsto \mathbb{R}^n \mapsto \mathbb{B}$ which assigns an initial condition to each location $\ell \in V$. Each initial condition $init(\ell)$ is a predicate whose free variables are from X.

Actions A finite set Σ of actions, and an edge labeling function $action : E \mapsto \Sigma$ which assigns an action $a \in \Sigma$ to each switch $e \in E$.

Guards An edge labeling function $guard : E \mapsto \mathbb{R}^n \mapsto \mathbb{B}$ that assigns a guard condition to each switch $e \in E$. Each guard condition of an edge, either denoted as $guard(e)$ or $guard(action(e))$, is a predicate whose free variables are from $X \cup X'$.

Example 2.2 (HA Components)**.** The thermostat in Figure 2.2 extends the basic thermostat in Figure 2.1 to provide a more detailed understanding of the components of HA. The additions are a variable y and a location *idle*. Since the flow predicate for y is $\dot{y} = 1$ in every location of the HA, y represents a global clock. Because y is set to

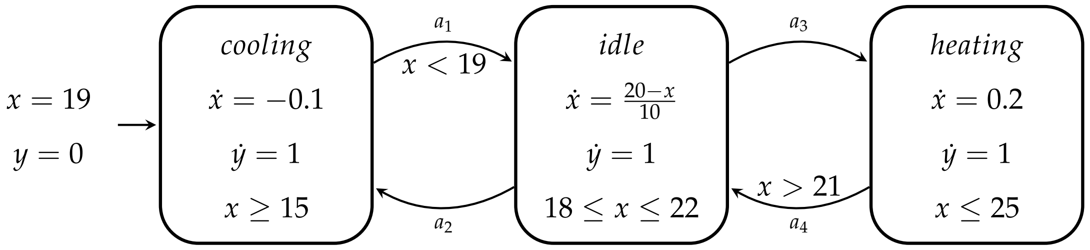

Figure 2.2: Extended Thermostat

0 in the beginning, it is a timer for how long the system has been running. In timed automata, an alternative model to HA, all clocks implicitly have the same dynamics, i.e. $\dot{x}_i = 1$. The flow of x in *idle* is not a basic linear predicate, but a more complex differential equation. Differential equations allow to model many physical processes. The flow $\dot{x} = \frac{20-x}{10}$ in *idle* means that the temperature converges towards 20°. The change of a variable can be computed by integrating its flow dynamics. For *idle*, the change from some time t_{start} to t_{end} is calculated by $\int_{t_{start}}^{t_{end}} \frac{20-x(t)}{10} dt$.

Each evaluation of a variable inside a location has to satisfy the *invariant* predicate of that location. Thus, the switch a_1 from location *cooling* to *idle* can only happen when the temperature is between 18° and 22°. Additionally, the switch can only occur when the corresponding *guard* is satisfied, i.e. when $x < 19$. This shows that discrete and continuous dynamics are closely linked: on the one hand the continuous dynamics of the HA can trigger a switch and thus a discrete change, and on the other hand discrete transitions change the continuous dynamics of the variables. △

2.1.2 Semantics

The execution of a HA consists of continuous and discrete changes. While this gives HA strong expressive power, the structure is not well suited for reasoning over their behavior. Therefore, we define a *labeled transition system* that abstracts the hybrid dynamics of HA to a fully discrete system. The discrete system is better suited for analysis. The idea behind the labeled transition systems is that the state space of the HA is unfolded. Since HA can have infinitely many states, the labeled transition system can be infinitely large. We continue following Henzinger [Hen00] for large parts of the notation.

Definition 2 (Labeled Transition Systems)**.** A *labeled transition system* is a tuple $S = (Q, Q_0, A, \rightarrow)$.

Def. Labeled Transition System

State Space Q is the (possibly infinite) set of *states*, $Q_0 \subseteq Q$ the subset of *initial states*.

Transition Relations A is the (possibly infinite) set of *labels*. $\xrightarrow{a}$ is a binary relation on the state space of Q for each label $a \in A$. Each triple $q \xrightarrow{a} q'$ is called a *transition*.

For a given HA, we define two labeled transition systems: timed transition systems and time-abstract transition systems. Both transition systems represent discrete jumps by transitions. Continuous flows are also abstracted by transitions, retaining only information about the source and the target of each flow. As the names indicate *timed* transition systems also retain information about the duration of each flow, whereas *time-abstract* transition systems neglect timing information.

Def. Timed Transition System

Definition 3 (Transition Semantics of Hybrid Automata)**.** The *timed transition system* S^t_H of the HA H is the labeled transition system with the components Q, Q_0, A and $\xrightarrow{a}$ for each $a \in A$, defined as follows.

- Define $Q, Q_0 \subseteq V \times \mathbb{R}^n$ such that
  - $(\ell, x) \in Q$ iff the closed predicate $inv(\ell)[X := x]$ is true, and
  - $(\ell, x) \in Q_0$ iff both $init(\ell)[X := x]$ and $inv(\ell)[X := x]$ are true.

  The set Q is called the *state space* of H.
- $A = \Sigma \cup \mathbb{R}_{\geq 0}$
- For each event $\sigma \in \Sigma$, define $(\ell, x) \xrightarrow{\sigma} (\ell', x')$ iff there is a control switch $e \in E$ such that
  - the source of e is ℓ and the target of e is ℓ'
  - the closed predicate $guard(e)[X, X' := x, x']$ is true, and
  - $action(e) = \sigma$.
- For each nonnegative real $\delta \in \mathbb{R}_{\geq 0}$, define $(\ell, x) \xrightarrow{\delta} (\ell', x')$ iff $\ell = \ell'$ and there is a differentiable function $f : [0, \delta] \mapsto \mathbb{R}^n$, with the first derivative $\dot{f} : (0, \delta) \mapsto \mathbb{R}^n$, such that
  - $f(0) = x$ and $f(\delta) = x'$, and
  - for all reals $\varepsilon \in (0, \delta)$, both $inv(\ell)[X := f(\varepsilon)]$ and $flow(\ell) = [X, \dot{X} := f(\varepsilon), \dot{f}(\varepsilon)]$ are true.

  The function f is called a *witness* for the transitions $(\ell, x) \xrightarrow{\delta} (\ell', x')$.

Def. Time-abstract Transition System

The *time-abstract transition system* S^a_H of H is the labeled transition system with the components Q, Q_0, B, and $\xrightarrow{b}$ for each $b \in B$, defined as follows.

- Q and Q_0 are defined as above.
- $B = \Sigma \cup \{\tau\}$, for some action $\tau \notin \Sigma$.
- For each action $\sigma \in \Sigma$, define $\xrightarrow{\sigma}$ as above.
- Define $(\ell, x) \xrightarrow{\tau} (\ell', x')$ iff there is a nonnegative real $\delta \in \mathbb{R}_{\geq 0}$ such that $(\ell, x) \xrightarrow{\delta} (\ell', x')$.

The time-abstract transition system S^a_H is called the *time abstraction* of the timed transition system S^t_H.

We are only interested in HA that diverge in time, i.e. HA which do not allow infinitely many steps in a finite amount of time. This property is widely known as *non-zenoness* in literature. We adopt the following two definitions from Henzinger [Hen00] as a formalization of non-zeno semantics of HA.

Definition 4 (Live Transition Systems)**.** Consider a labeled transition system S and a state q_0 of S. A *q_0-rooted trajectory* of S is a finite or infinite sequence of pairs $\langle a_i, q_i \rangle_{i \geq 1}$

of labels $a_i \in A$ and states $q_i \in Q$ such that $q_{i-1} \xrightarrow{a_i} q_i$ for all $i \geq 1$. If q_0 is an initial state of S, then $q_0\langle a_i, q_i\rangle_{i\geq 1}$ is an *initialized trajectory* of S. A *live transition system* (S, L) is a pair consisting of a labeled transition system S and a set L of infinite initialized trajectories of S. The set L of infinite initialized trajectories is *machine-closed* for S if every finite initialized trajectory of S is a prefix of some trajectory in L. If (S, L) is a live transition system, and $q_0\langle a_i, q_i\rangle_{i\geq 1}$ is either a finite initialized trajectory of S or a trajectory in L, then the corresponding sequence $\langle a_i\rangle_{i\geq 1}$ of labels is called a (finite or infinite) *trace* of (S, L). Trace

Definition 5 (Trace Semantics of Hybrid Automata)**.** We associate with each transition of the timed transition system S_H^t a duration in $\mathbb{R}_{\geq 0}$. For events $\sigma \in \Sigma$, the duration of $q \xrightarrow{\sigma} q'$ is 0. For reals $\sigma \in \mathbb{R}_{\geq 0}$ the duration of $q \xrightarrow{\delta} q'$ is δ. An infinite trajectory $q_0\langle a_i, q_i\rangle_{i\geq 1}$ of the timed transition system S_H^t *diverges* if the infinite sum $\Sigma_{i\geq 1}$ diverges, where each δ_i is the duration of the corresponding transition $q_{i-1} \xrightarrow{a_i} q_i$. An infinite trajectory $q_0\langle b_i, q_i\rangle_{i\geq 1}$ of the time-abstract transition system S_H^a *diverges* if there is a divergent trajectory $q_o\langle a_i, q_i\rangle_{i\geq 1}$ of S_H^t such that for all $i \geq 1$, either $a_i = b_i$ or $a_i, b_i \notin \Sigma$. Let L_H^t be the set of divergent initialized trajectories of the timed transition system S_H^t, and let L_H^a be set of divergent initialized trajectories of the time-abstract transition system S_H^a. The hybrid automaton H is *non-zeno* if L_H^t is machine-closed for S_H^t (or equivalently, L_H^a is machine-closed for S_H^a). Each trace of the live transition system (S_H^t, L_H^t) is called a *timed trace* of H, and each trace of the live transition system (S_H^t, L_H^t) is called a *time-abstract trace* of H. **Def.** Non-zeno Timed Trace Time-abstract Trace

Since trajectories and traces play an important roll in this thesis, let us look at an example.

Example 2.3 (Trajectory and Trace)**.** The following is an example trajectory for the time-abstract transition system of the thermostat in ➔Figure 2.2. ➔Sec. 2.1, Page 5

$$(cooling, \begin{pmatrix}19 & 0\end{pmatrix}^\top), \langle \tau, (cooling, \begin{pmatrix}18.75 & 2.5\end{pmatrix}^\top)\rangle,$$
$$\langle a_1, (idle, \begin{pmatrix}18.75 & 2.5\end{pmatrix}^\top)\rangle, \langle \tau, (idle, \begin{pmatrix}18.8109\ldots & 3\end{pmatrix}^\top)\rangle,$$
$$\langle \tau, (idle, \begin{pmatrix}19.2029\ldots & 7\end{pmatrix}^\top)\rangle$$

In the beginning the automaton is in the location *cooling*. The values of the variables change according to the linear dynamics $\dot{x} = -0.1$ and $\dot{y} = 1$, but the transition system only retains discrete information. The variable values in the states are computed by integrating the flow dynamics over time. For the first entry we get $19 + \int_0^{2.5} -0.1dt = 19 - 0.25 = 18.75$ for x and $0 + \int_0^{2.5} 1dt = 0 + 2.5 = 2.5$ for y. After 2.5 time units the automaton takes the transition a_1 to the location *idle*. Here the dynamics evolve according to the differential equation $\dot{x} = \frac{20-x}{10}$. The values are again generated through integration. This is the calculation for the x component: $18.75 + \int_{2.5}^{3} \frac{20-x(t)}{10}dt = 18.8109\ldots$.

The following is the same trajectory for the timed transitions system.

$$(cooling, \begin{pmatrix}19 & 0\end{pmatrix}^\top), \langle 2.5, (cooling, \begin{pmatrix}18.75 & 2.5\end{pmatrix}^\top)\rangle,$$
$$\langle a_1, (idle, \begin{pmatrix}18.75 & 2.5\end{pmatrix}^\top)\rangle, \langle 0.5, (idle, \begin{pmatrix}18.8109\ldots & 3\end{pmatrix}^\top)\rangle,$$
$$\langle 4, (idle, \begin{pmatrix}19.2029\ldots & 7\end{pmatrix}^\top)\rangle$$

The difference is that we retain information about the duration of the flows.

The following is the corresponding timed trace for (S^t_H, L^t_H).

$$\langle 2.5\rangle, \langle a_1\rangle, \langle 0.5\rangle, \langle 4\rangle$$

The trace only retains information about the transition labels and their order. △

2.1.3 Guard and Invariant Satisfaction

This subsections defines the satisfaction relation $\models$ for guards and invariants

$guard(a)$ denotes the guard of the switch a and is a function that maps variable evaluations from $\mathbb{R}^n$ to boolean verdicts.

Definition 6 (Guard Satisfaction)**.** A variable evaluation $x \in \mathbb{R}^n$ *satisfies* a guard $guard(a)$, or $x \models guard(a)$, iff $guard(a)(x) = \top$. A variable evaluation $x \in \mathbb{R}^n$ *violates* a guard $guard(a)$, or $x \not\models guard(a)$ iff $guard(a)(x) = \bot$.

$inv(\ell)$ denotes the invariant of the location ℓ and is a function that maps variable evaluations from $\mathbb{R}^n$ to boolean verdicts.

Definition 7 (Invariant Satisfaction)**.** A variable evaluation $x \in \mathbb{R}^n$ *satisfies* an invariant $inv(\ell)$, or $x \models inv(\ell)$, iff $inv(\ell)(x) = \top$. A variable evaluation $x \in \mathbb{R}^n$ *violates* an invariant $inv(\ell)$, or $x \not\models inv(\ell)$ iff $inv(\ell)(x) = \bot$.

Remark 2.4 (Comparison to Timed Automata)**.** *HA can be seen as an extension of timed automata. In timed automata, all variables are clocks which are limited to a constant flow of $\dot{x} = 1$. Therefore the only continuous dynamics that timed automata can capture are time and dynamics evolving at the same rate time evolves.*

2.2 Runtime Monitoring

Our lives rely heavily on software and digital controllers. These systems are omnipresent, not just when it comes to our convenience, like in smartphones and TVs, but also in many safety critical environments, like on roads and in power plants. Thus, it is crucial that these systems behave correctly. To this end, there are several verification techniques which complement each other. They can be broadly classified into three categories.

Software testing is the most popular of the three and is routinely used in the development of almost every system [BJK$^+$05]. While it is immensely useful in practice, it has its drawbacks. Namely the wide range of methods that it covers are informal, incomplete, and error prone. They boil down to checking the execution of specific scenarios [MSB11]. Generally, software testing cannot guaranty correctness.

Formal verification, like theorem proving and model checking, *can* guarantee that a system is correct with regards to a specification. It does so by proving that every possible execution of a system adheres to the specification. The downside is that that it is usually expensive and difficult. For many properties, such as most properties of hybrid automata, it is even impossible [HKPV98].

Runtime Monitoring

Runtime monitoring (RM) is a more lightweight category than formal verification [LS09, SHL12]. Instead of arguing about the general system behavior encompassing all possible runs, it is concerned with one execution at a time. By doing so, it trades stronger formal guarantees for applicability to complex systems. More concretely, this tradeoff means RM can assure that all runs of a system up to this point are correct, even when the system is too complex to get any guarantees with formal verification.

Offline Monitoring

Online Monitoring

There are two ways to implement RM: *Offline monitoring* analyzes logged system traces with regards to a specification *after* the system has run and generated the traces. *Online monitoring* analyzes system traces *at runtime*. The latter implementation is more restricted in the resources available for two reasons: Firstly, there are only limited computation and memory resources available during runtime. This is even more of a concern when monitoring systems in the embedded domain. Secondly, if the monitoring is executed on the same hardware the system is running on, then the monitoring process can influence the behavior of the system, possibly causing errors itself. Putting these restrictions aside, online monitoring can provide valuable feedback to a running system such that it can appropriately react to mitigate potential errors.

Chapter 3

Monitoring Hybrid Automata

The preferred way of specifying properties for runtime monitoring is usually a temporal logic such as Linear Temporal Logic (LTL) [Pnu77] and derivations thereof or Signal Temporal Logic (STL) [MN04] (compare ➔Section 5). These temporal logic properties are then used to monitor a system, be it a real-life system or a model of a system such as a hybrid automaton. The idea for this thesis flips this concept around: The property itself is encoded as a hybrid automaton and system traces are checked against this automaton. ➔Page 35

This nicely complements other specification languages and thus increases the number of systems that can be monitored. HA can encode complex continuous behavior that temporal logics cannot capture. Additionally, they are well-suited for capturing the structure of systems. Therefore, they could provide an additional choice which is better suited for certain needs than other monitoring tools.

However, the approach comes with its own challenges. Usually a stronger expressiveness means that handling of a system becomes more complex . This thesis assesses this tradeoff for a subclass of hybrid automata. To this end, we designed an algorithm that takes as input a hybrid automaton and a trace of a black box system and decides if the trace represents a valid run in the input automaton. The basic idea is to keep track of all viable paths in the automaton and update these paths as system trace values become available. As long as there are viable paths left, the trace is accepted. In principle, the algorithm can be applied *online* and *offline*.

This chapter describes what kind of HA and traces are processed by the algorithm and how the algorithm is constructed.

3.1 Monitoring Traces

In section ➔Section 2.1.2 we gave a definition for trajectories, timed traces and time-abstract traces. Whereas that general definition of a trace focusses on the transition labels of the abstracted labeled transition system, the focus is now on the variable ➔Page 5

evaluations of a system. A *monitoring trace* is the output of a monitored system S. It is the only information about S that we get; we regard S as a block box system for the purpose of runtime monitoring.

Def. Reading

Def. Monitoring Trace

Definition 8 (Monitoring Trace)**.** Let $X = \{x_1, ...x_n\}$ be the set of variables of the monitored system S. Let $X_{Obs} \subseteq X$ be the set of observable variables of S. A *reading* is an evaluation of the variables in X_{Obs}. A *monitoring trace* is a sequence of readings as generated by the execution of the system S.

If not explicitly stated otherwise *trace* stands for *monitoring trace* in the rest of this thesis.

➔Sec. 2.1, Page 5

➔Sec. 2.1, Page 7

Example 3.1. The following is an example of a monitoring trace for the thermostat in ➔Figure 2.2. It represents the same execution as the trajectories and timed traces in ➔Example 2.3.

$$\begin{pmatrix}19 & 0\end{pmatrix}^\top, \begin{pmatrix}18.8 & 2\end{pmatrix}^\top, \begin{pmatrix}18.8109\ldots & 3\end{pmatrix}^\top, \begin{pmatrix}19.2029\ldots & 7\end{pmatrix}^\top, \begin{pmatrix}19.4095\ldots & 10\end{pmatrix}^\top$$

△

3.2 Restrictions for the Input

➔Sec. 2.1, Page 7

In addition to requiring HA to be *non-zeno* (➔Definition 5), we introduce other restrictions on syntax and semantics that make them more suitable for runtime monitoring. These restrictions are needed for the approach of this thesis.

3.2.1 Flows

We restrict the differential equations for flows to be ordinary differential equations (ODEs). This is a reasonable tradeoff between expressiveness and computational cost for computing reachable states. In the future it might be possible to loosen this restriction.

3.2.2 One Reading per Location

We assume that two consecutive readings in a trace cannot skip a location in the property hybrid automaton. This is a strong but necessary assumption. If any number of switches in the hybrid automaton is allowed to occur between two readings, the set of plausible paths explodes too quickly. This restriction also prohibits the automaton from rapidly switching locations, which is a desirable property, since it is closer to a stable execution of a system. One could argue that this restriction does not really fit the requirements for a monitoring algorithm because it is a requirement for the system trace and not for the monitoring tool. While that is true, the restriction represents a compromise that is not too restrictive on the behavior of the monitored system S. One way to enforce is is to set a sufficiently high sampling rate for readings of S.

3.3 Algorithm

The purpose of the algorithm is to monitor if a system satisfies or violates a property specified by a hybrid automaton. It takes as input a hybrid automaton H and a trace from a black box system S. Then it checks if the trace could be a trace of the automaton. This section starts with introducing the components of the algorithm and proceeds to explain how the components work together to form the final monitoring algorithm.

3.3.1 Paths

The algorithm maps a sequence of readings, the trace of the system S, to the hybrid automaton H. This mapping results in potential *paths* through the hybrid automaton. For each new reading, it checks if the readings match the paths. A reading matches a path if it represents a valid continuation of this path according to the semantics of the HA. The paths that match are saved and the path that do not match are deleted. If at one point the set of paths is empty, we know that the input trace does not satisfy the input HA.

Zone Graph

The representation of *paths* is inspired by *zone graphs*. *Zone graphs* are a reachability abstraction for timed automata and by extension for hybrid automata. The vertices in a zone graph are the zones consisting of a pair $\langle \ell, [\delta] \rangle$. Here ℓ denotes a location in the original automaton and $[\delta]$ is a clock zone, i.e. the maximum set of variable evaluations satisfying the constraint δ. The edges between zones correspond to the transitions we can take according to the dynamics defined in the automaton. Zone graphs are a useful reachability abstraction because they are finite in many interesting cases and can be implemented efficiently. For these reasons, they are used in popular model checking tools like UPPAAL and KRONOS. Note that since the reachability problem is generally undecidable for HA [HKPV98], it is not possible to construct a complete zone graph of a hybrid automaton.

The *paths* used in the algorithm are based on the same information that zone graphs are based on; locations and clock zones. However, we do not build a full abstraction of the system a priori, but instead construct the paths incrementally in accordance with the readings. Each reading of a trace represents an increment at which possible locations and clock zones corresponding to the reading are assessed. The specifics of this process follows in the subsequent section.

3.3.2 Encoding HA Dynamics

To asses validity of paths, the algorithm encodes the dynamics of the input HA and passes them on to a solver. The encoding depends on the chosen solver.

We considered two solver approaches: *Constraint Logic Programming* (CLP) and symbolic solvers for systems of differential equations. This subsection briefly illustrates the concepts of the two approaches. The remaining part of the chapter focusses on the latter approach, i.e. symbolic solving of systems of differential equations, as this approach is implemented in the prototype.

Constraint Logic Programming

Def. Constraint Logic Programming

Constraint Logic Programming is a combination of two paradigms: Constraint solving and logic programming [FA03]. It combines the reasoning principles of logic programming with a constraint system over a specific domain. In logic programming we want to decide if a set of rules, which can be interpreted as logical implications, is satisfiable. By generating substitutions and testing if they are consistent with the set of rules we can find a substitution which satisfies our problem. Of course, this trial-and-error approach is inefficient in most cases. In constraint logic programming we use constraints not just for checking consistency of substitutions but also for limiting the domain of our variables. This example from [FA03] illustrates the difference:

Example 3.2. $X \in \{1,2\} \land Y \in \{1,2\} \land Z \in \{1,2\} \land X = Y \land X \neq Z \land Y > Z$

Logic programming would require testing substitutions. Due to the domains of X, Y and Z, we would have to test up to 8 substitutions. Constraint logic programming contrarily does not require testing substitution, as the solution can be inferred from the constraints: $Y > Z$ implies $Y = 2$ and $Z = 1$. The constraint $X = Y$ propagates the information such that $X = 2$. The last constraint $X \neq Z$ is already satisfied. $\triangle$

Ballarin and Kauers [BK] have applied constrained logic programming to parametric linear systems. When a computer algebra system solves a parametric linear system, the general solution is not necessarily correct, since it does not account for special values of the parameters. This problem is known as the *specialization problem*. To combat the specialization problem that occurs during Gaussian elimination, the authors of the paper used constraint programming to maintain assumptions on parameters. More specifically these assumptions are stored in a constraint store during pivot selection. The pivot is the non-zero entry in the matrix which is fixed while all other entries in the corresponding column are reduced to zero through suitable row operations. Afterwards, solutions are generated for all satisfiable subsets of the parameter space.

While this approach looks promising, it is only able to solve constant differential equations since their closed from is linear. Therefore, to solve flows formulated as ODEs some modifications would be necessary. It would have to either be complemented by a pre-processing step or the solver itself would have to be extended.

Symbolic ODE Solver

Symbolic solvers that can solve systems of ODEs are a good fit for hybrid automata, since the continuous dynamics of HA, the flows, are also stated in the form of ODEs. However, it is not immediately obvious how the hybrid dynamics comprised of continuous and discrete dynamics can be encoded in such a system of equations. This subsection explains the encoding of transitions, guards, and invariants.

Encoding of Transition Times

The following illustrates how the algorithm uses parameters to handle timings of discrete transitions.

Let ℓ_1 and ℓ_2 be two consecutive locations of an HA H connected by a switch a. The continuous dynamics of the locations are given by the flow conditions $flow(\ell_1)(t)$ and $flow(\ell_2)(t)$.

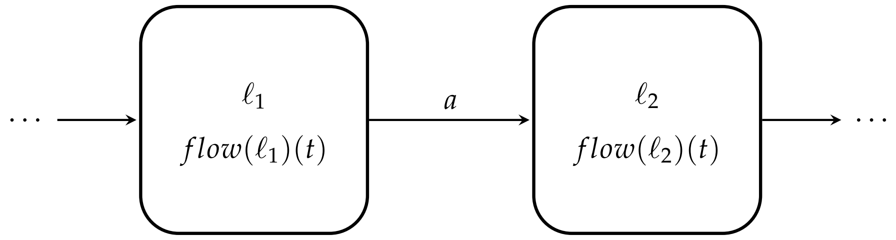

Figure 3.1: Two Locations Connected by a Switch

Additionally, there are two trace readings r_1 and r_2. We want to encode the assumption that reading r_1 corresponds to location ℓ_1 and reading r_2 to location ℓ_2. We introduce the parameter t_{start} for the time of the reading r_1, t_1 for the time of the transition a and t_2 for the time of the reading r_2.

The following parameterized equation models the behavior of the HA:

$$r_1 + \int_{t_{start}}^{t_1} flow(\ell_1)(t, x(t))dt + \int_{t_1}^{t_2} flow(\ell_2)(t, x(t))dt = r_2$$

In the same way every transition of a path is encoded. For the initial equation the time t_{start} of the first reading is known. This simplifies the first equation and generates information about t_2. As the parameter t_2 corresponds to the parameter t_{start} of the next equation this next equation can be simplified as well. The whole system is incrementally simplified in this way.

Encoding of Guard Conditions

Guards are represented through additional equations. In the following the approach from above is extended with a guard condition. We illustrates how the system of equations is expanded with the encoding of the $guard(a)$.

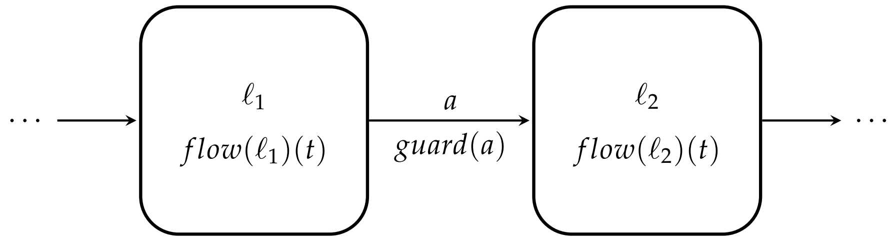

Figure 3.2: Guard Condition

All assumptions from before still hold. In addition, the guard condition has to hold during the transition a at time t_1. That results in the following system of equations:

$$r_1 + \int_{t_{start}}^{t_1} flow(\ell_1)(t, x(t))dt + \int_{t_1}^{t_2} flow(\ell_2)(t, x(t))dt = r_2$$

$$r_1 + \int_{t_{start}}^{t_1} flow(\ell_1)(t, x(t))dt \vDash guard(a)$$

→ Page 8

The relation $\vDash$ can be translated to equations according to → Section 2.1.3. This guard encoding is applied to all guard conditions of a path.

Encoding of Invariant Conditions

Invariants are also represented through additional equations. However, their encoding requires and additional step. The following extends the approach for the timing of transitions from Figure 3.1 with the invariant conditions $inv(\ell_1)$ and $inv(\ell_2)$. All assumptions from that approach continue to hold, including that parameter t_{start} is the time of the reading r_1, t_1 the time of the transition a and t_2 the time of the reading r_2.

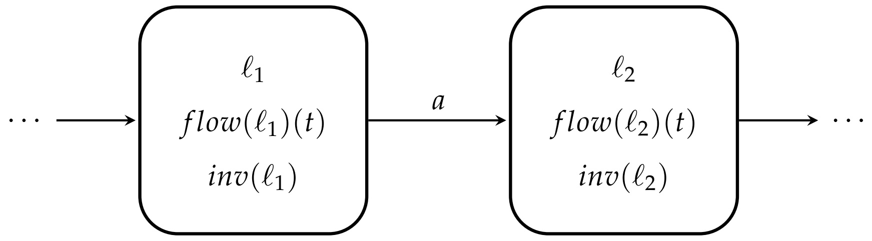

Figure 3.3: Invariant Conditions

In addition to the assumptions from before, the invariant condition $inv(\ell_1)$ has to hold in location ℓ_1, i.e. from t_{start} to t_1, and the invariant $inv(\ell_2)$ has to hold in location ℓ_2, i.e. from t_1 to t_2.

$$\forall t' \in [t_{start}, t_1] : r_1 + \int_{t_{start}}^{t'} flow(\ell_1)(t, x(t))dt \vDash inv(\ell_1) \wedge$$

$$\forall t' \in [t_{t_1}, t_2] : r_1 + \int_{t_{start}}^{t_1} flow(\ell_1)(t, x(t))dt + \int_{t_1}^{t'} flow(\ell_2)(t, x(t))dt \vDash inv(\ell_2)$$

$$\equiv \neg\Big(\exists t' \in [t_{start}, t_1] : r_1 + \int_{t_{start}}^{t'} flow(\ell_1)(t, x(t))dt \nvDash inv(\ell_1) \vee$$

$$\exists t' \in [t_1, t_2] : r_1 + \int_{t_{start}}^{t_1} flow(\ell_1)(t, x(t))dt + \int_{t_1}^{t'} flow(\ell_2)(t, x(t))dt \nvDash inv(\ell_2)\Big)$$

The latter form with the existential quantification can be translated to equations that extend the system of equations.This invariant encoding is applied to the whole path.

3.3.3 Combining the Components

This section describes how the components described above are combined to form the monitoring algorithm. There are three levels of nesting.

Algorithm 1 - `checkDynamics`

The lowest level is the algorithm `checkDynamics` (➔Algorithm 1). It takes four inputs: The HA specification, a path from the set of plausible paths, a new location and a new reading. `checkDynamics` checks if the new reading in the given location is a viable continuation of the path according to the dynamics defined in the HA. It starts with comparing the end of the path, i.e. its tail, with the new input. More concretely, only the the last reading and location of the path are at first compared with the input location and input reading. Only when these dynamics have been successfully tested, the dynamics of the complete path are checked. ➔Sec. 3.3, Page 18

The actual checks are performed in order of computational cost: The easiest checks are performed first and the most expensive checks in the end. If one of the checks fails, the function returns the empty set. Additionally, the partial results of each check are used in the subsequent steps. In the first five steps, the HA constraints are encoded in a new system of parameterized differential equation by the function `createEquation`. The specifics of this encoding are explained in ➔Section 3.3.2. Then this system of equations is passed to a solver, denoted by the function `solve`. If the result set is empty, there is no solution to the problem. In this case `checkDynamics` returns $\emptyset$. If the check passed, the result set is a simplified system of equations. Often this simplification is a constraint set for the parameters, or even a unique solution to the problem. In other situations the information in the system of equations does not suffice for any simplifications and it remains unchanged. In either case the result is passed on to the next check and used in the construction of the next system of differential equations. ➔Page 13

The following describes the purpose of each check. The first merely tests if the new reading satisfies the invariant condition of the input location. The computation for checks two through five is based on the the last reading in the path, the input reading and the dynamics of last path location and input location. The second check computes plausible transition times between the two locations. The third checks satisfaction for the guard condition at the plausible transition times. The fourth is responsible that the invariant condition holds in the first location, as long as the system has not transitioned yet. The fifth does the same for the invariant condition of the input location.

The sixth check is different from the prior checks. While the others only took the last two locations and readings into account, the final one checks the complete path dynamics. To do so it creates a new path based on the old path and the solved system of equations from the steps before. Note that the responsible function `createPath` does not construct a completely new system of equations. It merely extends the system of equations saved in the old path with the system of equations constructed in the other five steps. This system of equations is then passed to the solver. At last, when all check are satisfied, `checkDynamics` returns the newly constructed path.

Algorithm 1: checkDynamics

```
1  ht
   Input: HA, path, location, newReading
   Output: newPath bases on continuation of path with newReading in location
           according to dynamics of HA
   /* 1st check - invariant of new location for newReading                */
2  if newReading $\not\models$ location.*invariant* then
3      return $\emptyset$
   /* 2nd check - determine possible transition times                     */
4  equationTransition $\longleftarrow$ createEquation(HA, path.tail, location, newReading,
     *transition*)
5  equationSystem $\longleftarrow$ solve(equationTransition)
6  if equationSystem $= \emptyset$ then
7      return $\emptyset$
   /* 3rd check - guard to new location                                   */
8  equationGuard $\longleftarrow$ createEquation(HA, path.tail, location, newReading, *guard*)
9  equationSystem $\longleftarrow$ solve(equationSystem $\wedge$ equationGuard)
10 if equationSystem $= \emptyset$ then
11     return $\emptyset$
   /* 4th check - invariant of path.tail                                  */
12 equationInvariant1 $\longleftarrow$ createEquation(HA, path.tail, location, newReading,
     *invariant1*)
13 equationSystem $\longleftarrow$ solve(equationSystem $\wedge$ equationInvariant1)
14 if equationSystem $= \emptyset$ then
15     return $\emptyset$
   /* 5th check - invariant of location                                   */
16 equationInvariant2 $\longleftarrow$ createEquation(HA, path.tail, location, newReading,
     *invariant2*)
17 equationSystem $\longleftarrow$ solve(equationSystem $\wedge$ equationInvariant2)
18 if equationSystem $= \emptyset$ then
19     return $\emptyset$
   /* 6th check - path dynamics                                           */
20 newPath $\longleftarrow$ createPath(path, equationSystem)
21 newPath.equationSystem $\longleftarrow$ solve(newPath.equationSystem)
22 if newPath.equationSystem $= \emptyset$ then
23     return $\emptyset$
   /* passed all checks - return newPath                                  */
24 return newPath
```

Algorithm 2 - `updatePaths`

The second level of the monitoring algorithm consits of the function `updatePaths` (➔ Algorithm 2). It takes three inputs: The HA specification, the set of plausible paths for the trace of the system S and a new reading of that trace. `updatePaths` updates the given set of plausible paths with the new reading. To this end it checks how each path could be continued with regards to the reading and the HA dynamics. The checks are performed by the function `checkDynamics` (Section 3.3.3). First the successor locations of the last location in the path, the tail location, are checked. For each viable successor continuation a new path is added to the set `paths`. After the successor locations the HA dynamics for remaining in the same location are checked. If they match, the path in the paths set is updated. Otherwise the paths is deleted from `paths`.

➔Sec. 3.3, Page 19

Algorithm 2: updatePaths

```
Input: HA, paths, newReading
Output: paths updated with dynamics according to newReading
1  foreach path ∈ paths do
2      if path ≠ initialPath then
3          foreach successorLocation of path.tailLocation do
               /* check dynamics for successor locations              */
4              newPath ⟵ checkDynamics(HA, path, successorLocation, newReading)
5              if newPath ≠ ∅ then
6                  paths ⟵ paths ∪ newPath

       /* check dynamics for remaining in location                    */
7      newPath ⟵ checkDynamics(HA, path, path.tailLocation, newReading)
8      if newPath ≠ ∅ then
9          path ⟵ newPath
10     else
11         paths ⟵ paths \ { path }
```

Algorithm 3 - Main Loop

The main monitoring loop (➔ Algorithm 3) represents the top level of the algorithm. The input to the algorithm are the hybrid automaton specification `HA` and the system trace `trace`. It is important to note that the system trace is not fully available in the beginning, since the algorithm is designed for both *online* and *offline* monitoring. Instead, a call to the function `checkTrace` returns a new trace value if one is available. `checkTrace` also provides feedback whether the monitored system S and with it the monitoring process are active. The algorithm starts by creating the initial path based on the initial location and if available the initial values of the variables. These initial

➔Sec. 3.3, Page 20

values are derived from the *init* condition of the initial location. While the monitoring of the system S is active and there are plausible paths in the set `paths`, the algorithm waits for new trace readings. For each new reading the set of paths is updated by the function `updatePaths`. In the end, the set of plausible paths is returned. If the empty set is returned, `trace` violates the property given by `HA`.

Algorithm 3: Main Monitoring Loop

```
Input: Hybrid Automaton HA, System Trace trace
Output: paths in HA according to trace
1  monitoringActive = $\top$
2  paths $\longleftarrow$ createInitialPath(HA)
3  while paths $\neq \emptyset$ and monitoringActive do
4      (newReading, monitoringActive) $\longleftarrow$ checkTrace(trace)
5      if newReading $\neq \emptyset$ then
6          paths $\longleftarrow$ updatePaths(HA, paths, newReading)
7  if paths $= \emptyset$ then
8      print Property violated
9  else
10     print Property satisfied
11 return paths
```

3.3.4 Optimizations

This subsection describes how some aspects of the basic algorithm above can be improved.

Reachable States

The algorithm incrementally processes the trace reading-by-reading. This can result in a backlog of unprocessed readings when the processing takes too long and the sampling rate of the trace is high. A big backlog can mean that the monitoring algorithm is only able to report a property violation when it is too late. The problem is even more significant when the hardware the monitoring is running on is low powered, as is often the case with online monitoring. Since online monitoring is often used in embedded systems, and since these systems tend to be constrained in their form factor and their energy resources, these systems cannot be equipped with a high processing power.

To mitigate the problem of a backlog, we introduce two approaches that supplement the monitoring with a reachability analysis. The reachability analysis simplifies the processing of trace readings, thereby speeding it up resulting in a reduced backlog. The first component is a static reachability abstraction of the specification HA. A complete reachability analysis is generally not possible because the reachability problem for hybrid automata is not decidable [HKPV98]. However, an approximation of parts of the reachable state space can be computed by tools like FLOW* [CAS12]. The precision and the depth of the reachability analysis depend on the complexity of the HA and the resources available. There are more computational resources available in this step as it is performed before the actual system runs. Therefore, it is not limited to the hardware of the system and does not produce a backlog.

The other way a precomputation of reachable states can reduce processing time, and thus backlog, is during monitoring. It is reasonable to assume that the required processing of trace values is not evenly distributed across the run of the system. For one, trace values might not become available at a constant sampling rate, but instead in bursts. Another reason for an uneven distribution is that the processing itself depends on the dynamics of the HA model, which varies throughout the automaton. Another big factor is how many viable paths have to be assessed for continuation with each reading. This number can fluctuates significantly during monitoring. For these reasons, there are sections of a run where more processing resources are available for monitoring. In these sections, the algorithm can compute the states that are reachable from the end of the paths. Since the end of the paths consist of the last trace reading and the last location of the path, they provide a precise starting point for the reachability computation. When computed, the reachable states can be encoded as constraints in the systems of equations of the paths. These additional constraints reduce the time needed for solving the systems of equations.

The main issue when computing reachable states is a state explosion similar to the path explosion problem outlined in the next subsection. Exploring a greater number of steps can quickly become infeasible. However, in our scenario, the reachability analysis only needs to be performed for a shallow depth to be useful for the analysis of future trace values. Instead of one deep exploration of reachable states, which is a computationally intensive task, the structure of our problem means that many shallow explorations, which are easy to compute, suffice.

Path Explosion

As we monitor a system, the number of possible paths can explode quickly. This subsection investigates this problem of path explosion and possible remedies that keep the problem contained. The cause for the problem is similar to the cause of the state explosion problem when exploring reachable states in a transition system or a graph. As the monitoring progresses the set of plausible paths increases. Each path can spawn more potential paths and thus the number of path can quickly explode. The problem is especially pronounced when different sections of the automaton have similar dynamics and when the set of observable variables is small. A large number of paths has two problematic consequences: If there are a lot of viable paths that all keep on getting longer, more memory is consumed to save these paths. Additionally, each processing of a trace reading takes longer because every continuation of every path needs to be evaluated.

Checkpoints

Thankfully, some regions of the HA drastically reduce the set of paths. We call these occurrences *checkpoints*. They are locations or switches that enable the algorithm to rule out other behavior, thus greatly reducing the number of possible paths. An example of a checkpoint would be an invariant condition that is disjoint to every other invariant condition or a change in the dynamics of traced variables that can only occur with a certain transition. The more constraints are specified on the dynamics of the system, the easier it is to rule out other behavior. Therefore, it is import that all implicit assumptions about the behavior of the monitored system are formulated as explicit constraints in the HA specification.

The algorithm can also be optimized to better handle the path explosion. For the following optimizations we will change the requirements for the monitoring algorithm. This subsection assumes that the result set of the monitoring algorithm is binary, either 'satisfaction' or 'violation'. This means the algorithm does *not* output the plausible path through the automaton if the trace lies within the HA.

Path Trimming

One of the consequences of this assumption is that we do not need to keep the complete paths in memory. This subsection investigates what path information needs to be kept to correctly decide the monitoring problem.

The crux of this problem are the unobservable variables that are not part of the trace readings. Assume all variables are observable and thus part of every reading. Then a reading and the corresponding location would fully specify the state of the automaton at the time of the reading. The full state is enough to decide if the next trace reading matches a valid continuation of the HA in either the same location or after taking one transition to a successor location. Therefore, when all variables are observable, it is enough to keep the last location and the last trace reading of each path in memory to correctly decide the monitoring problem.

However, this is not the case if there are also unobservable variables. For an intuition on this, consider the following example:

Example 3.3. Assume $flow(\ell_2) = flow(\ell_3)$. Further assume the monitoring algorithm has determined that there are two plausible paths, one through ℓ_1, ℓ_2 and ℓ_4 and the other through ℓ_1, ℓ_3 and ℓ_4. The only difference in the dynamics of the paths is the respective reset of the variable x at transition a_1 and a_2. x is an unobservable variable and thus not part of the trace readings. Only the path through ℓ_1, ℓ_2 and ℓ_4 satisfies the transition guard $x = 1$ to ℓ_5. But the last trace reading in ℓ_4 and the information that we are in ℓ_4 are not enough to make this distinction as they provide no information about the unobservable variable x. △

The example illustrates that we need to save information about unobservable variables in the paths to correctly decide the monitoring problem. We denote these variables, which are not part of the trace readings but included in the HA, as *indirectly*

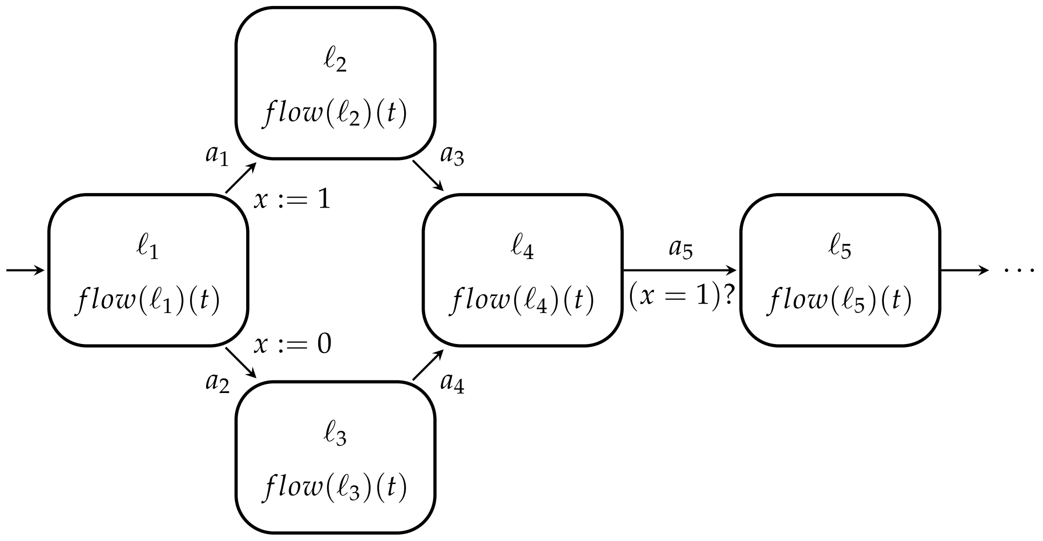

Figure 3.4: x is an unobservable variable

observable. In addition to the last reading and the last location of a path, all information that might lead to insights about indirectly observable variables needs to be kept in memory to correctly decide property satisfaction of a trace. Exactly what kind of information can provide insights on indirectly observable variables could be explored in future work. This regards information about the dynamics of the automaton saved in the system of equations of the path. The sequence of locations does not hold any additional information for deciding the monitoring problem and is not saved.

Moreover, it is sometimes possible to combine paths to reduce their number. Two trimmed paths can be combined when they are indistinguishable with regards to their information. When the current location and the constraints on variables and parameters encoded in the system of equation are the same, they are indistinguishable and one of the paths is deleted. This eliminates redundant information and thus memory consumption. Additionally, since the number of paths is reduced, each new reading can be processed faster.

3.4 Complexity

At the core of the algorithm is the solving of systems of differential equations. As the readings provide an initial value for each differential equation, they can be solved. The system of equations can be solved by quantifier elimination over the reals using cylindrical algebraic decomposition [ACM84]. This process is doubly exponential in the number of quantified variables. For each reading $r_i \in \mathbb{R}^n$ in each path, two quantifiers are used to encode the invariant conditions. Each time a new reading is added to the path, the system of equations has to be reevaluated. The complexity of the evaluation of a path is doubly exponential in the length of the path. This means the approach is feasible when the number of readings and paths is small and quickly becomes infeasible when their number becomes large. The main potential for optimization is reducing the set of paths, as we cannot influence the number of readings. Accounting for intermediate results, that are used when evaluating paths, could also improve the complexity analysis.

➔ Page 30

The results are validated by the two test cases of the implementation (➔ Section 4.2).

3.5 Correctness

In the following we prove the correctness of the algorithm. We start by formalizing a notion of paths of HA.

Definition 9 (Hybrid Automaton Paths)**.** The set of paths of a hybrid automaton H is defined as the infinite sequences of states and transitions as defined by the semantics of H: $Paths(H) = \{q_1\delta_1 q_2\delta_2 \cdots \mid \forall i : q_i = (\ell_i, val_i) \in (V \times \mathbb{R}^n), \delta_i = (\hat{t}_i, \sigma_i) \in (\mathbb{R}_0^+ \times \Sigma)$ as defined by the semantics of H$\}$

➔ Page 12

The following formalizes the requirement that two consecutive readings in a trace cannot skip a location in the property hybrid automaton (➔ Section 3.2.2) and defines the monitoring problem. Note that the first line encodes the requirement.

Definition 10 (Monitoring Problem)**.** Let the trace of a system S be a finite sequence of readings $r_i \in \mathbb{R}^n$: $\rho = r_1, \ldots, r_m$ and let $Paths(H)$ be defined as above.

The trace ρ generated by the system S *satisfies* the hybrid automaton specification H, or $\rho \vDash H$, iff

$$\forall i \in \mathbb{N}, 1 \leq i \leq m : \exists j \leq i : \delta_j = (\hat{t}_j, \sigma) \wedge$$
$$\exists t \leq \hat{t}_j : r_i = val_j + \int_{\Sigma_{k=1}^{j-1}\hat{t}_k}^{t+\Sigma_{k=1}^{j-1}\hat{t}_k} flow(\ell_j)(\tau, x(\tau)))d\tau$$

Theorem 3.4. *Algorithm 3(H,tr) implements the monitoring problem defined in Definition 10 for an input hybrid automaton H and a corresponding input trace ρ of a system S.*

Proof. Proof sketch by induction over the trace readings

Induction Base $m = 1$

$$\forall i \in \mathbb{N}, 1 \leq i \leq 1 : \exists j \leq i : \delta_j = (\hat{t_j}, \sigma) \wedge$$

$$\exists t \leq \hat{t_j} : r_1 = val_j + \int_{\Sigma_{k=1}^{j-1} \hat{t_k}}^{t+\Sigma_{k=1}^{j-1} \hat{t_k}} flow(\ell_j)(\tau, x(\tau))d\tau$$

$$\equiv \exists j \leq 1 : \delta_j = (\hat{t_j}, \sigma) \wedge \exists t \leq \hat{t_j} : r_1 = val_j + \int_{\Sigma_{k=1}^{j-1} \hat{t_k}}^{t+\Sigma_{k=1}^{j-1} \hat{t_k}} flow(\ell_j)(\tau, x(\tau))d\tau$$

$$\equiv \delta_1 = (\hat{t_1}, \sigma) \wedge \exists t \leq \hat{t_1} : r_1 = val_1 + \int_{\Sigma_{k=1}^{0} \hat{t_k}}^{t+\Sigma_{k=1}^{0} \hat{t_k}} flow(\ell_1)(\tau, x(\tau))d\tau$$

$$\equiv \delta_1 = (\hat{t_1}, \sigma) \wedge \exists t \leq \hat{t_1} : r_1 = val_1 + \int_{0}^{t} flow(\ell_1)(\tau, x(\tau))d\tau$$

Algorithm 3(H, r1) implements this behavior: In line 2 the initial path is created based on the start location and the variable valuation encoded in the initial conditions. This first variable valuation corresponds to val_1. Then, the while-condition is satisfied and checkTrace returns $(r_1, \top)$. Consequently, the if-condition is evaluated to $\top$ and updatePaths is called in line 6. Since only the initial path is in the set of paths, only one call to checkDynamics is issued in line 7. checkDynamics implements the definition of the HA dynamics, thus producing a new path based on the inital path. In this new path a systems of equations is encoded corresponding to $r_1 = val_1 + \int_0^t flow(\ell_1)(\tau, x(\tau))d\tau$ as it is precisely the definition of the HA dynamics in the first location, taking val_1 as r_1 and r_1 as r_2 (compare Section 3.3.3). In line 8 the if-condition is satisfied and the initial path in the path set is updated to account for the first reading. The algorithm returns to Algorithm 3. In the second iteration of the while loop checkTrace returns $(\emptyset, \bot)$. Thus the if-condition and the while-condition are violated. Since paths is non-empty, 'Property satisfied' is printed in line 10. In line 11 the path corresponding to the equation is returned, which is equivalent to property satisfaction.

Induction Hypothesis

We assume for some arbitrary $m \in \mathbb{N}$, that the Algorithm 3 correctly implements the monitoring problem defined in Definition 10 for the first m trace readings.

Induction Step

We sketch that the induction hypothesis implicates that the algorithm is correct for the first $m + 1$ readings. The intuition behind the correctness is that the definition of the monitoring problem and the algorithm are both constructed based on the HA dynamics presented in this thesis, albeit with a different encoding.

From the induction hypothesis follows that the first m iterations of the while loop are correct. In the $m + 1^{st}$ iteration of the loop, the $m + 1^{st}$ trace reading is checked. In line 6, updatePaths is called with the HA definition, the correct set of paths for the first m readings, and the $m + 1^{th}$ reading.

The induction hypothesis also implies that for the first m readings, the following holds:

$$\forall i \in \mathbb{N}, 1 \leq i \leq m : \exists j \leq i : \delta_j = (\hat{t_j}, \sigma) \wedge$$
$$\exists t \leq \hat{t_j} : r_i = val_j + \int_{\Sigma_{k=1}^{j-1} \hat{t_k}}^{t+\Sigma_{k=1}^{j-1} \hat{t_k}} flow(\ell_j)(\tau, x(\tau))d\tau$$

For the theorem to hold, it has to follow from the equation above and the algorithm, that

$$\exists \eta \leq m+1 : \delta_\eta = (\hat{t_\eta}, \sigma) \wedge \exists t \leq \hat{t_\eta} : r_{m+1} = val_\eta + \int_{\Sigma_{k=1}^{\eta-1} \hat{t_k}}^{t+\Sigma_{k=1}^{\eta-1} \hat{t_k}} flow(\ell_\eta)(\tau, x(\tau))d\tau$$

On an intuitive level, the above means that the reading r_{m+1} is between $\eta - 1^{st}$ and the η^{th} location of a path $\pi \in Path(H)$ for some $\eta \leq m+1$. Since we know that the algorithm is correct for the first m readings, there exists an implementation path p' corresponding to π for the first m readings in the set of plausible paths. updateChecks checks the continuation of the path either for remaining in ℓ_j or for a successor location ℓ_{j+1}. Thus, $\eta \leq j+1$ and since $j \leq m$ it follows that $\eta \leq m+1$.

In the prior m steps, checkDynamics has created a system of equations encoding val_j. If the path is updated because the system remains in the j^{th} location, then $val_\eta = val_j$. When a new path is created based on a successor location, then checkDynamics encodes $val_\eta = val_j + \int_{\Sigma_{k=1}^{j-1} \hat{t_k}}^{t+\Sigma_{k=1}^{j-1} \hat{t_k}} flow(\ell_j)(\tau, x(\tau))d\tau$ as defined by the encoding of the semantics of the HA H.

For the first case, checkDynamics encodes r_{m+1} as $r_{m+1} = val_\eta + \int_{\Sigma_{k=1}^{j-1} \hat{t_k}}^{t+\Sigma_{k=1}^{j-1} \hat{t_k}} flow(\ell_j)(\tau, x(\tau))d\tau$. For the second case, r_{m+1} is encodes as $val_\eta + \int_{\Sigma_{k=1}^{j} \hat{t_k}}^{t+\Sigma_{k=1}^{j} \hat{t_k}} flow(\ell_j)(\tau, x(\tau))d\tau$. One of these paths corresponds to the path π. The other one corresponds to a path π' that is an alternative path through the automaton.

$\square$

Chapter 4

Experiments

This chapter discusses the implementation of the algorithm introduced in ➔Section 3.3, highlighting the differences between the theoretical algorithm and the practical implementation. Additionally, we present how the implementation handles two test cases and illustrate how the complexity changes. The code of the implementation can be found on github [1]. ➔Page 13

4.1 Implementation

The algorithm was implemented in MATLAB R2019a. We chose MATLAB because of its support for symbolic and numerical solving of systems of differential equations. Additionally, it allows for fast development of a prototype that demonstrates feasibility of a practical implementation.

While the theoretical algorithm is correct, the implementation introduces potential approximation error sources. An error may happen because numerically solving the equations is implemented as the fallback option for situations where the symbolic solver fails to find a solution. Additionally, some intermediate results are saved numerically because the symbolic expressions become to complex for the solver. We go into more detail when describing how trace values are processed.

4.1.1 Inputs

The algorithm decides whether a system trace of a black box system satisfies a property specified as a hybrid automaton. A system trace satisfies a property HA when the trace lies within the trace set of the automaton. We assume the trace and the automaton fulfill the requirements described in ➔Section 3.1 and ➔Section 3.2. ➔Page 11 ➔Page 12

The input automata are encoded in the file `initHa.mlx`. The definitions of the HA presented in the test cases in Section 4.2 can be found in this file. HA are represented

[1]github.com/pbungert/MonitoringHA

as structs of locations and guards. Locations are implemented as struct arrays of unique identifiers, flows, invariants and successor locations. Flows are specified as differential equations. Thus, they can fully model ODEs as specified by the theoretical algorithm. Invariant conditions are simplified to intervals of the form $\{x \in \mathbb{R}^n \mid l \leq x \leq u \wedge l, u \in \mathbb{Q}^n\}$. We opted for this simplification as it demonstrates how invariant checking integrates into the algorithm and lets us focus on other areas of the implementation. Invariants could be extended with reasonable effort since it would only require changes to few parts of the code. Switches of the HA are encoded by including a set of successor locations in each location. This simple encoding proofed beneficial for the implementation. The transition guards are also simplified to intervals of the form $\{x \in \mathbb{R}^n \mid l \leq x \leq u \wedge l, u \in \mathbb{Q}^n\}$, for the same reason as the integrals. The *guard* function, i.e. the mapping from transitions to guards, is realized by a four dimensional array. One dimension encodes the origin location of the transition, one the destination location, one which variable is concerned, and one whether the lower or the upper bound is specified.

The system trace input is realized as a table of trace readings. Each call to the function `checkTrace`, which is also defined in the theoretical definition of the algorithm, retrieves a reading until all readings have been processed. While this means that we have implemented offline monitoring, the interface could be extended to inputs of a runnings system, making it an online monitoring algorithm. The main code would be unaffected by this change.

4.1.2 Processing of Trace Readings

The overall structure of the implementation is very similar to the structure outlined in ➔ Algorithm 2 and ➔ Algorithm 3. The most important difference between the theoretical algorithm and the implementation is that the implementation assumes the set of observable variables is the same for each reading and that there are no additional variables in the automaton. Consequently, the implementation can correctly construct all plausible paths solely based on incremental checks between two locations corresponding to two readings (compare ➔ Section 3.3.4). Therefore, only checks one through five are sufficient to correctly decide this variant of the monitoring problem. The sixth check and the extension of the system of equations is not necessary. The implementation does not store a system of equations for the dynamics of the overall path. Only the sequence of locations, the trace readings and the possible timing of the trace readings is kept in memory. For the remainder of the section we assume that the possible timings of the readings is saved in the readings.

➔ Sec. 3.3, Page 19
➔ Sec. 3.3, Page 20
➔ Page 21

The following describes how checks one through five of `checkDynamics` (➔ Algorithm 1) are implemented. These checks test if the dynamics of the property HA allow the continuation of an input path with an input reading. They are based on the last location and reading in the input path and a new location and reading, which are potentially the next element of the path.

➔ Sec. 3.3, Page 18

The first simply checks if the new reading is within the interval of the invariant of the new location. For checks two through five a system of equations is created and then solved. The following exemplifies that process by focussing on the second check. This check computes possible values for the time t_1 when the transition happens and for the time t_2 of the new input reading, denoted by `r2` in the code. t_{start} is a parameter for the time of the first reading, denoted as `r1`. The possible values for t_{start} have been computed in the step before.

The following code starts with encoding the last reading of the path, `r1`, as an assumption. This assumption is then used to simplify the system of *flow* equations of the first location. This step uses the symbolic solver `dsolve` that produces symbolic output. Thus no approximation error is introduced in this step. The same is done for the second reading and the flow of the second location.

```
%% simplify flow of l1 with r1
% create assumption
assume = ha.variables(t_start) == r1;
[...]
% solve
[flow_l1{:}] = dsolve(l1.flow, assume);
[...]
%% simplify flows of l2 with r2
% create assumption
assume = ha.variables(t_2) == r2;
[...]
% solve
[flow_l2{:}] = dsolve(l2.flow, assume);
```

In an intermediate step not shown here, the simplified equations are converted to the function handles `eval_flow_l1` and `eval_flow_l2`. These function handles are then used to formulate the equation system including the integrals. Compare → Section 3.3.2 for details about the equation. → Page 14

```
%% formulate equation with simplified flows and readings
equation = r1 + eval_flow_l1(t_1) - eval_flow_l1(t_start) +
    eval_flow_l2(t_2) - eval_flow_l2(t_1)  == r2;

%% solve equation
solution = solve(equation);
solution = (structfun(@double,solution, 'UniformOutput', false));
```

There are two steps in the listing above that can introduce numerical errors. The first one is the function `solve`. When `solve` cannot find a solution symbolically, it tries a numerical solver instead. Afterwards, if the solution set is not already numerical, it is converted from a symbolic to a numerical solution. This is different from the theoretical algorithm that only uses exact symbolic computations.

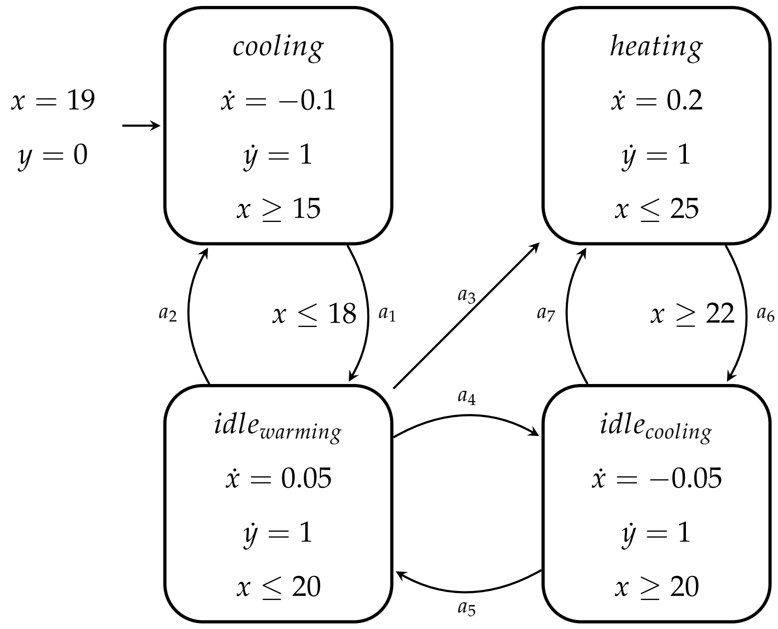

Figure 4.1: Linear Thermostat

The method for the other checks in `checkDynamics` is very similar as only the equation itself is different.

4.2 Test Cases

This section presents two test cases for the implementation.

Thermostat

The first test case is similar to the thermostat used throughout Chapter 2. However, we simplified the location *idle* with ODE flow dynamics by substituting it with the two locations $idle_{cooling}$ and $idle_{warming}$ with linear dynamics. The resulting property HA is depicted in Figure 4.1.

The corresponding trace is depicted in Table 4.2 below. The monitoring algorithm only uses the columns **Temperature (x)** and **Clock (y)** as they present the variable evaluations. The columns **Time** and **Note** columns are only included for clarity.

The algorithm correctly determines that the trace satisfies the property and outputs two plausible paths:

- *cooling, cooling, cooling,* $idle_{warming}$*, heating, heating, heating,* $idle_{cooling}$
- *cooling, cooling, cooling,* $idle_{warming}$*, heating, heating,* $idle_{cooling}$*, heating*

The first path is the path used for precomputing the trace as described in the **Note** column. Due to how different the $flow$ conditions in the locations are, the maximum

Time	**Note**	**Temperature (x)**	**Clock (y)**
2019-09-11 00:00:02	*cooling*	18,80	2
2019-09-11 00:00:08	*cooling*	18,20	8
2019-09-11 00:00:13	3s in *cooling* $\rightarrow$ 2s in $idle_{warming}$	18,00	13
2019-09-11 00:00:17	3s in $idle_{warming}$ $\rightarrow$ 1s in *heating*	18,35	17
2019-09-11 00:00:25	*heating*	19,95	25
2019-09-11 00:00:46	*heating*	24,15	46
2019-09-11 00:00:50	0.75 s in *heating* $\rightarrow$ 4s in $idle_{cooling}$	24,10	50,75

Table 4.2: Thermostat Trace

number of potential paths during the execution is two.

WALL·E[2]

This is different for the other test case, WALL·E, since here the *flow* dynamics are more similar. The HA specifying the correct behavior for WALL·E is depicted in Figure 4.3. The following table is an execution trace that is checked against the specification.

Time	**Note**	**Left to Clean (c)**	**Energy (e)**
2519-09-11 00:00:03	3s in *recharge*	1000000003	103
2519-09-11 00:00:06	2s in *recharge* $\rightarrow$ 1s in *eager*	999999996	102
2519-09-11 00:00:10	1s in *eager* $\rightarrow$ 3s in *normal*	999999969	93
2519-09-11 00:00:14	2s in *normal* $\rightarrow$ 2s in *lazy*	999999951	87
2519-09-11 00:00:19	3s in *lazy* $\rightarrow$ 1s in *eager*	999999933	81

Table 4.4: WALL·E Trace

- *recharge, recharge, eager, normal, lazy, eager*
- *recharge, recharge, eager, eager, eager, eager*
- *recharge, recharge, eager, normal, normal, normal*
- *recharge, recharge, eager, eager, normal, normal*
- *recharge, recharge, eager, normal, eager, eager*
- *recharge, recharge, eager, lazy, eager, eager*
- *recharge, recharge, eager, lazy, normal, normal*
- *recharge, recharge, eager, eager, eager, normal*
- *recharge, recharge, eager, eager, eager, lazy*
- *recharge, recharge, eager, normal, normal, eager*

[2]All rights to the name belong to The Walt Disney Company and its subsidiary Pixar Animation Studios

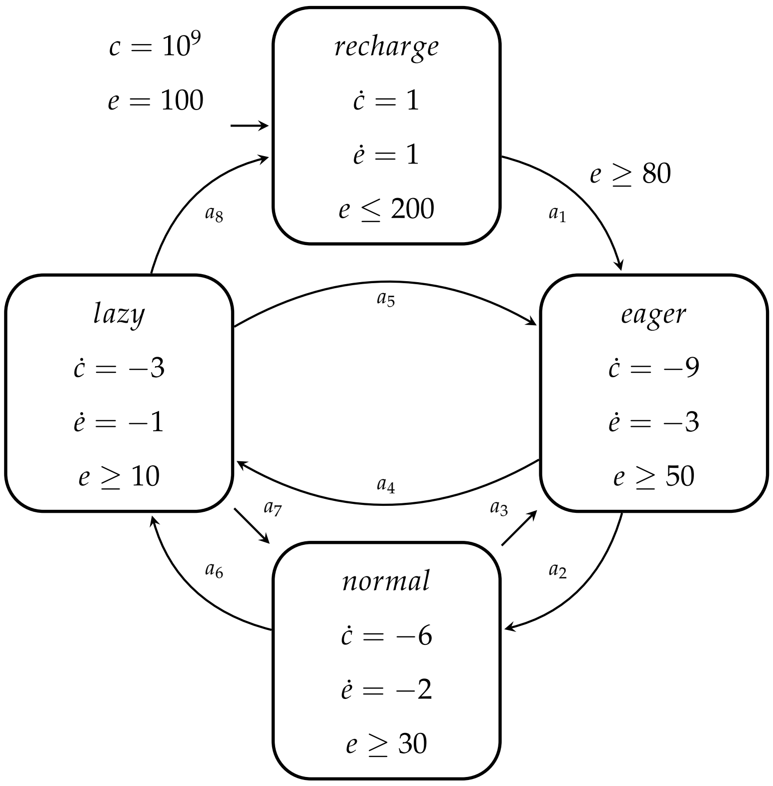

Figure 4.3: WALL•E

WALL•E was left behind to clean up after humans have left earth. He has 10^9 areas left to clean. Cleaning up requires energy that has to be recharged when it is almost depleted. WALL•E is a moody robot that can decide how fast he cleans up. His moods are modeled by the locations *eager*, *normal* and *lazy*. With depleting energy levels he becomes lazier. Earth deteriorates with a rate of one area per time unit while he is charging.

- *recharge, recharge, eager, normal, normal, lazy*
- *recharge, recharge, eager, eager, normal, eager*
- *recharge, recharge, eager, eager, normal, lazy*
- *recharge, recharge, eager, eager, lazy, eager*
- *recharge, recharge, eager, eager, lazy, normal*
- *recharge, recharge, eager, normal, eager, normal*
- *recharge, recharge, eager, normal, eager, lazy*
- *recharge, recharge, eager, normal, lazy, normal*
- *recharge, recharge, eager, lazy, eager, normal*
- *recharge, recharge, eager, lazy, eager, lazy*
- *recharge, recharge, eager, lazy, normal, eager*
- *recharge, recharge, eager, lazy, normal, lazy*

Even though there are only 5 reading in the trace, the implementation returns 22 possible paths. The first path is the one that was used for the computation of the trace values. This path explosion occurs because the continuous dynamics in the locations are very similar. Since the global clock is only implicit and not explicitly given like in the thermostat test case, lots of paths are possible in each step. When WALL•E starts to recharge the possible paths are drastically reduced because it is the only location where the energy level increases.

4.3 Complexity of the Implementation

Compared to the theoretical algorithm, adding a reading to a path has a lower complexity. This is because the introduced restrictions mean, that the computationally most expensive step of evaluating the whole path is skipped. An exact complexity analysis is not possible since there is no information on the complexity of the numerical MATLAB solvers we used. However, the main result of the complexity analysis of the theoretical algorithm remains true for the implementation: an high number of readings or paths makes the monitoring algorithm infeasible. This result is verified by the two case studies.

Chapter 5

Related Work

The topic of the thesis is validating if a trace is modelled by a hybrid automaton (HA). An interpretation of this is seeing the HA as a property specification, and then monitoring this specification in much the same way as a specification expressed in a temporal logic. The logic of choice is often linear temporal logic (LTL) or an extension thereof. As RM does not reason over infinite traces, which are usually associated with LTL properties, but over finite prefixes of infinite traces, Bauer et al. have lifted LTL from a two-valued domain to a three-valued domain [BLS06, BLS11]. The resulting logic LTL_3 adds the verdict *inconclusive* to the usual *true* and *false*. This verdict is reached when it is not possible to yield either *true* or *false* because the necessary information is not contained in the finite prefix, but only in the infinite continuation of the trace.

Other monitoring approaches avoid these problems by reasoning over temporal logics with bounded time. MITL (metric interval temporal logic), for example, is a logic which adds time bounds to temporal operators. Bounded operators are easier to handle than their unbounded counterpart since the evaluation of a property is clear once the time specified by the bound has passed. STL (signal temporal logic) by Maler et al. in turn extends MITL to allow monitoring of real-valued continuous signals instead of boolean-valued atomic propositions [MN04]. Maler et al.'s offline monitoring algorithm for STL formulae uses backtracking to assess these continuous signals. Because backtracking requires the trace to be fully available, this approach does not work for online monitoring. Thus other methods for online monitoring have been developed [DDG+17, NM07]. These techniques resort to an incremental evaluation of the trace. Since STL is expressive and has been show to work well with hybrid systems, we want to compare it to our problem of monitoring HA online. In particular, our aim is to show that HA monitoring is more expressive than STL monitoring.

Another field related to our problem of runtime monitoring is system falsification. Just as in our approach, the problem Annapureddy et al. [ALFS11] tried to solve boils down to finding a particular path in a hybrid automaton. However, there is an important difference. We aim to check if there is a path in a property HA which corresponds to the output trace of a system. Annapureddy et al., on the other hand, take the system modelled as an HA as an input and then try to match an input signal to this HA such that an MTL specification φ is *not* satisfied — hence the name system falsification. Their tool S-TaLiRo simulates several runs for the input signal and uses stochastic optimization techniques to come closer to a falsifying trace with each iteration. However, the stochastic optimization is not efficient when the result set of an MTL- or STL-specification evaluation is binary. Instead of just satisfaction or violation it would be better to receive feedback on how close the path was to satisfying or violating the property to adjust for the next iteration. This is where robustness metrics come into play. Space robustness metrics [DM10, DFM13] provide feedback by assigning a value close to zero if a variable evaluation came close to a required value and a value further away from zero if the margin was greater. Time robustness metrics, on the other hand, provide feedback whether a goal was reached/violated just-in-time as opposed to within a larger time margin. Akazaki et al. [AH15] have combined space- and time robustness in a logic called AvSTL to improve the performance of the falsification solver S-TaLiRo. While the falsification problem is fundamentally different than runtime monitoring in some aspects, it shares other key aspects with our problem. Both approaches are concerned with finding traces in HA at their core, albeit with a different background.

Similar to our approach, Sistla et al. [SŽF12] use probabilistic hybrid automata (PHA) to specify properties for monitoring. However, their focus is on a different problem, namely monitorability of probabilistic hybrid automata. Hence, they only explore the idea of HA as property models on a superficial level driven by the motivation to approximate liveness properties. When monitoring safety properties they default to Büchi automata. The goal of this thesis, on the other hand, is to model valid system behavior in a HA and then check whether the output of a real system corresponds to this model specification.

An important part of our algorithm is estimating the current location of the HA based on the trace information from the dynamical system. Diagnosability is also concerned with location estimation, though from a different angle. A system is defined as diagnosable if it is possible to determine within a fixed time bound if the current location is element of a subset of all locations, called critical locations. Benedetto et al. have tried to answer that for hybrid systems [DBDGD09] and HA [DBDGD11] in particular. However, their approach is based on a different notion of a trace than ours. While we want to use all trace information consisting of transitions and variable evaluations, they primarily use the finite set of discrete transition labels and the delay between these transitions. Only when this information does not suffice to show that the system, modelled by a hybrid system called switching system, is diagnosable, their algorithm performs an intermediate step. In this step the continuous input and the observable continuous output are transformed into discrete output labels.

Other methods for the analysis of HA approach the problem from the opposite direction. Instead of monitoring a trace of a dynamical system to match it to a HA like we do, they simulate the execution of the HA and monitor this simulated trace. HySIA

[IG17] is a tool designed for this purpose. It computes an approximation of a trace of a nonlinear deterministic HA. Deterministic in this context means that a transition is taken as soon as it is enabled. While this is an interesting problem on its own, we will consider the more general case of nondeterministic models in our work.

The tool FLOW* allows us to analyze nondeterministic HA with nonlinear flow dynamics [CAS12, CÁS13]. It computes flowpipes, which represent the set of states that are reachable within a given time interval according to the dynamics of the system under scrutiny. The main contribution of the authors is extending the use of Taylor models to compute flowpipes from purely continuous models to HA, which combine continuous dynamics with guarded discrete location changes. Our work complements this general reachability problem for HA by updating the approximation with trace values from the running system. This substantially reduces the set of reachable states thus making the computation more efficient and accurate.

Chapter 6

Conclusion

This thesis is based on the idea to encode runtime monitoring properties in hybrid automata. Due to the ability of hybrid automata to capture both discrete and continuous behavior, they are well-suited as a specification tool for many systems. However, since they are so different from other means of specification, it is not obvious that this complex model can be used for monitoring.

To assess the feasibility of the idea, we constructed an algorithm that takes as input a property hybrid automaton and an input trace from a black box system. The algorithm decides if the input trace lies within the set of traces defined by the property automaton. To this end the trace readings are mapped into possible paths through the HA. If a trace satisfies the specification, the set of all plausible paths is returned.

We soon realized that the input to the algorithm needs to fulfill certain requirements and outlined these restrictions. For one, the continuous dynamics of the hybrid automaton are restricted to ordinary differential equations. Additionally, we require a trace reading for every visit of a location. The latter restriction is severe as it is a requirement on the system trace. However, this compromise is necessary to limit path explosion to a manageable level. Further, we outlined potential problems of the algorithm and how they can be mitigated with optimizations.

In a second step, we implemented a prototype of the theoretical algorithm in MATLAB. The implementation encodes the monitoring problem in a system of differential equations. We used the symbolic engine of MATLAB to solve this system of equations. When the symbolic solver cannot find a solution, a numerical solver is used. This numerical solver and the conversion of intermediate results from the symbolic to the numerical domain can introduce approximation errors. We presented two test cases for the implementation which illustrate that monitoring properties encoded in hybrid automata is possible. The examples also show that our approach has to cope with path explosion problems similar to the ones occurring in the reachability analysis of hybrid systems, albeit to a lesser degree.

In conclusion, this thesis showed that monitoring of hybrid automata is possible when certain requirements are met.

In future work, the outlined optimizations to better handle paths could be further explored. The MATLAB implementation fulfills its purpose of showing the feasibility of a practical implementation, but could certainly be improved with regards to performance. Other solvers based on Constraint Logic Programming or Satisfiability Modulo Theories might offer better performance.

A comparison to other specification languages, such as temporal logic, would also be interesting.

Bibliography

[ACHH93] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229. Springer, 1993.

[ACM84] Dennis S. Arnon, George E. Collins, and Scott McCallum. Cylindrical algebraic decomposition I: The basic algorithm. *SIAM Journal on Computing*, 13(4):865–877, 1984.

[AH15] Takumi Akazaki and Ichiro Hasuo. Time robustness in MTL and expressivity in hybrid system falsification. In *International Conference on Computer Aided Verification*, pages 356–374. Springer, 2015.

[ALFS11] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605, pages 254–257. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[BJK$^+$05] Manfred Broy, Bengt Jonsson, J.-P. Katoen, Martin Leucker, and Alexander Pretschner. Model-based testing of reactive systems. In *Volume 3472 of Springer LNCS*. Springer, 2005.

[BK] Clemens Ballarin and Manuel Kauers. Solving Parametric Linear Systems: An Experiment with Constraint Algebraic Programming. page 14.

[BLS06] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 260–272. Springer, 2006.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[CAS12] Xin Chen, Erika Abraham, and Sriram Sankaranarayanan. Taylor Model Flowpipe Construction for Non-linear Hybrid Systems. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 183–192, San Juan, PR, USA, December 2012. IEEE.

[CÁS13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An Analyzer for Non-linear Hybrid Systems. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Natasha Sharygina, and Helmut Veith, editors, *Computer Aided Verification*, volume 8044, pages 258–263. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[DBDGD09] Maria D. Di Benedetto, Stefano Di Gennaro, and Alessandro D'Innocenzo. Discrete state observability of hybrid systems. *International Journal of Robust and Nonlinear Control: IFAC-Affiliated Journal*, 19(14):1564–1580, 2009.

[DBDGD11] Maria D. Di Benedetto, Stefano Di Gennaro, and Alessandro D'Innocenzo. Verification of Hybrid Automata Diagnosability by Abstraction. *IEEE Transactions on Automatic Control*, 56(9):2050–2061, September 2011.

[DDG$^+$17] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51(1):5–30, 2017.

[DFM13] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient Robust Monitoring for STL. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Natasha Sharygina, and Helmut Veith, editors, *Computer Aided Verification*, volume 8044, pages 264–279. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[DM10] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 92–106. Springer, 2010.

[FA03] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Cognitive Technologies. Springer, Berlin ; New York, 2003.

[Hen00] Thomas A. Henzinger. The theory of hybrid automata. In *Verification of Digital and Hybrid Systems*, pages 265–292. Springer, 2000.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of computer and system sciences*, 57(1):94–124, 1998.

[IG17] Daisuke Ishii and Alexandre Goldsztejn. HySIA: Tool for Simulating and Monitoring Hybrid Automata Based on Interval Analysis. *arXiv:1712.00570 [cs]*, 10548:370–379, 2017.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, May 2009.

[MN04] Oded Maler and Dejan Nickovic. Monitoring Temporal Properties of Continuous Signals. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Yassine Lakhnech, and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253, pages 152–166. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[NM07] Dejan Nickovic and Oded Maler. AMT: A property-based monitoring tool for analog systems. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 304–319. Springer, 2007.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*, pages 46–57. IEEE, 1977.

[SHL12] Oleg Sokolsky, Klaus Havelund, and Insup Lee. Introduction to the special section on runtime verification. *International Journal on Software Tools for Technology Transfer*, 14(3):243–247, June 2012.

[SŽF12] A. Prasad Sistla, Miloš Žefran, and Yao Feng. Runtime Monitoring of Stochastic Cyber-Physical Systems with Hybrid State. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Sarfraz Khurshid, and Koushik Sen, editors, *Runtime Verification*, volume 7186, pages 276–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.