Summary

An appropriate mathematical formalism for modelling systems with mixed discrete and continuous dynamics is a hybrid automaton. The hybrid automaton combine discrete control graphs with continuous dynamics defined by differential equations. In this paper, we introduce an offline learning algorithm to automatically synthesise a hybrid automaton from time series. The algorithm consists of several procedures. First, the time series is segmented using a Bottom-Up change point detection algorithm. The segments are subsequently used in a structure learning process to build an initial graph structure detailing locations and edges between them. The structure learning also associates each segment with a unique location. From the set of segments associated with a location, the dynamics governing the location are learned using the SINDy algorithm. Lastly, the guard conditions on the edges as well as the invariant conditions on the locations, are determined using one of two methodologies. The novel method computes a convex hull on the points of the segments associated with a location and is suitable for stationary data. The new time-based method introduced in this paper can determine conditions for models learned from time series exhibiting upward or downward trends. It distinguishes between seasonal time conditions where the switching conditions are determined by seasonal periods, and location time conditions which do not relate to seasonality, but where the conditions are still governed by a time interval. The evaluation shows that the algorithm can learn simple models with an accurate graph structure and appropriate dynamics from stationary time series using state-dependent conditions, as well as from non-stationary time series exhibiting trends using the timed conditions. However, the algorithm is not able to accurately synthesise complex systems with multiple locations, because the location dynamics have to be significantly distinguishable for the change point detection to segment correctly, and for the structure learning procedure to identify the locations as well as to associate the segments to locations.

Framework for Synthesis of Hybrid Automata from Time Series with Time- or State-Dependent Switching Conditions.

Alex Immerkær Kristensen
Aalborg University
Aalborg, Denmark
aikr17@student.aau.dk

Jakob Østenkjær Hansen
Aalborg University
Aalborg, Denmark
jhans17@student.aau.dk

Abstract

A hybrid automaton is an appropriate mathematical formalism for modelling systems with mixed discrete and continuous dynamics. The hybrid automaton combines discrete control graphs with continuous dynamics defined by differential equations. In this paper, we introduce an offline learning algorithm to automatically synthesise a hybrid automaton from time series. The algorithm consists of several procedures, including segmentation of time series, structure learning, and discovery of both dynamics and conditions. We present a novel method for determining the conditions of a model learned from stationary time series, as well as a new time-based method for models learned from time series that exhibit trends.

The evaluation shows that the algorithm can learn simple models with an accurate graph structure and appropriate dynamics from stationary time series using variable conditions, as well as from non-stationary time series exhibiting trends using the timed conditions. However, the results show poor performance if the change point detection algorithm is unable to accurately segment the time series, or if the location dynamics are indistinguishable. Indistinguishable dynamics between locations may cause the change point detection to misidentify segments, and it may result in the structure learning procedure being unable to identify the locations as well as to associate the segments to locations.

Keywords: Hybrid Automata, Synthesis, Framework, Time Series, Timed Conditions, Structure Learning, Continuous Dynamics

Paper Information:
Alex Immerkær Kristensen and Jakob Østenkjær Hansen. 2023. Framework for Synthesis of Hybrid Automata from Time Series with Time- or State-Dependent Switching Conditions. .

Tools and Contributions

The AI tool Git Copilot has been utilised during the development of the codebase [1]. Copilot is an AI pair programmer which helps write tedious and simple code in real-time. Furthermore, it has been utilised as an extension to IntelliSense, which can auto-complete multiple words at a time.

We would like to thank our supervisors Christian Schilling and Kim Guldstrand Larsen for their continued contribution throughout the semester.

1 Introduction

Many physical processes of today are controlled by computer-based algorithms. The controller's state moves discretely between control modes, where each mode represents a component with a unique evolution of the system state. A mode in this context is a part of the system which combines with other parts to form the whole system and is denoted as a **location** in this paper.

An appropriate mathematical formalism for modelling systems with mixed discrete and continuous dynamics is a *hybrid automaton*. A hybrid automaton combines discrete control graphs, also known as *finite state automata*, with continuously evolving variables [2]. Thus, a hybrid automaton exhibits two kinds of state changes. *Discrete jumps* which occur instantaneously, and *continuous flow* which occur while time elapses.

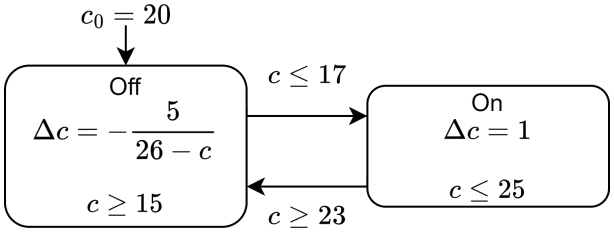


Figure 1. Automaton model of a thermostat. As to not confuse the temperature with time, it is denoted as c.

An example of a potential hybrid automaton model of a thermostat is shown in Figure 1. The model has two locations, *On* and *Off*, representing the locations of the thermostat, and a continuous variable c for the temperature. The **guards** determine under which conditions the system may switch between locations. From a **simulation** of the thermostat

using the hybrid automaton model in Figure 1, we can obtain a **trace**, which is a finite or infinite time sequence of **states**, where each data state is the value of all the variables of the system, at a time point.

$$\begin{aligned}(1, Off, 20) \rightarrow (2, Off, 18) \rightarrow (3, Off, 16.33) \rightarrow \dots \\ \rightarrow (6, On, 24.33) \rightarrow (7, Off, 23.78) \rightarrow \dots\end{aligned} \tag{1}$$

The thermostat example shows, that a hybrid automaton can represent a cyber-physical system with continuously evolving variables. The first argument of the tuple in Equation 1 denotes the current time point, while the second and third denotes the current location and the value of the variable at the time point. The main challenge is how to construct the hybrid automaton, in particular when the system dynamics are unknown. Modelling an unknown system is only possible if there is some information about the behaviour of the system. To gather information, we measure the variable evolution of a system over time, thus generating time series. A large part of the challenge when learning a hybrid automaton model is to determine the switching conditions, i.e., the relationship between the locations.

This paper is the second of two papers, which explores the synthesis of hybrid automata from time series. The first paper [3] had two primary focuses. Firstly, it details the development of a codebase to construct a hybrid automaton. Secondly, it evaluates different change point detection algorithms to find the algorithm which best segments the time series data. The change point detection algorithm with the best performance was found to be the Bottom-Up clustering algorithm. Each of the segments contains sequential data which evolves according to the dynamics of a location. The Bottom-Up algorithm can be used to effectively segment varying types of time series.

This paper aims to develop an algorithm which can synthesise hybrid automata from time series. This entails identifying the graph structure with locations and transitions, as well as the conditions and dynamics. Our contribution is the development of a unique framework for synthesising hybrid automata from time series, as well as introducing a new time-based method for determining conditions. The following list summarises the overall steps in the learning algorithm developed in this two-part project;

1. Split the time series into segments using a change point detection algorithm
2. Perform a structure learning procedure to build a graph structure and associate segments with locations
3. Determine the dynamics governing each location based on the associated segments
4. Determine the invariant conditions for each location and guard conditions for each transition
5. Construct the HA model

An overview of the entire process is shown in Figure 2.

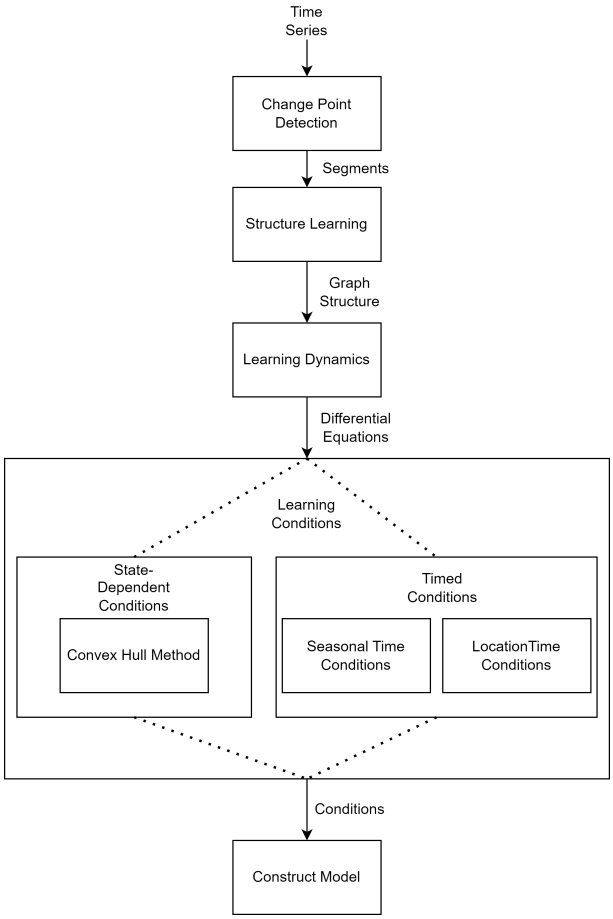


Figure 2. Flow of algorithm.

This paper is structured as follows. Section 2 discusses the related work in the field of hybrid and timed automaton synthesis. Section 3 introduces the formalism of hybrid automata and the notation used in this paper. Section 4 details how an initial graph structure is constructed from the time series. Section 5 explores how the dynamics may be determined from segments associated with a location. Section 6 discusses how invariant and guard conditions are determined, and issues regarding data exhibiting trends. The section also introduces alternative timed condition types. Section 8 evaluates the synthesised models based on distance measures on obtained traces, graph structure, dynamics, and conditions. Section 9 concludes the paper. Section 10 discusses the future work of this project.

2 Related Work

The synthesis of hybrid automata is explored in a broad range of fields. Generally, the problem is a part of the category of *model learning*, however, in the field of control theory it is often known as *system identification* [4]. Many approaches for system identification focus on input-output models, such as

the *autoregressive eXogenous* (ARX) models and in particular the piecewise (PWARX) and switched (SARX) versions.

Paoletti et al. [5] serves as an introduction to the identification problem for PWARX and SARX models. In general, PWARX and SARX models can be considered restricted linear hybrid automata with deterministic switching behaviour and a state-space partition forming the locations. As a result, the synthesis problem is essentially a parameter-optimization problem and Paoletti et al. [5] reference several procedures to solve it, such as clustering-based procedure [6], a Bayesian procedure [7], a bounded-error procedure [8] and an algebraic procedure [9]. Most of these techniques for systems in input-output form are proposed for offline identification.

In computer science, there are also several approaches detailing how to learn a hybrid automaton from time series. As for systems in state-space form, García et al. [10] proposes an online approach to synthesise hybrid automata with affine dynamics. An online approach in this context entails updating an existing automaton on new time series trajectories. They summarize the synthesis problem as constructing a hybrid automaton, that ϵ-captures the time series. This means, that from a simulation of the automaton, we must be able to obtain a trace such that the distance between the time series and the trace is below an ϵ error bound. The check of whether a hybrid automaton ϵ-captures the time series is denoted as a *membership query*. Based on the result of the membership query, the automaton is either modified based on some heuristics or left unchanged. The concept of membership-based synthesis has been explored in previous papers by the same authors, [11]. However, the approach described in [11] is restricted to linear hybrid automata, i.e., constant dynamics, whereas the approach [10] concerns affine dynamics.

The problem of synthesising linear hybrid automata from time series has also been explored in [12], where they separate the problem into two phases. In the first phase, they synthesise a discrete structure by mapping data points in the time series to a set of symbolic locations. In the second phase, they construct the parameter space, which is a polyhedron describing the fixations of continuous dynamics that ϵ-captures the time series. This technique does not support online environments, however, is more scalable in offline environments than the two aforementioned papers [10, 11]. Our paper utilises a similar methodology as in the first phase to construct a discrete graph structure from the data points in the time series.

Other works detail how dynamical systems with differential equations may be learned, without using a hybrid automaton as a model [13–15]. One paper addresses the demand for massive data analysis in real-time by using a compression method that approximates the time series stream with multiple polynomials [16]. Another relevant article details how a sparse regression algorithm can be used to discover the equations governing the time series [17]. Although this technique does not apply to hybrid systems, but to purely continuous systems, it is widely used in several steps of the overall algorithm detailed in this paper.

There has also been extensive research in formal modelling and analysis of timed systems, in particular using timed automata models, which are finite state machines with real-valued clocks. An et al. [18] present algorithms that learn a one-clock timed automaton. The first algorithm guesses the clock resets automatically but is of exponential complexity in the size of the learned automata. In the other algorithm, the user has to provide the clock reset information which yields a polynomial complexity. Maier et al. [19] propose an online timed automata learning algorithm, which iteratively updates the automaton based on if the new input configuration has been observed previously, similar in concept to the membership query mentioned earlier. Tapler et al. [20] present a novel method for learning timed automata that utilises genetic programming to generate a model consistent with a set of timed traces collected via testing. Similarly, Tapler et al. [21] use SMT solving to learn timed automata consistent with observations in a set of timed traces, which are gathered via active testing or passive monitoring. The test-based approaches in [20, 21] learn timed automata from black box systems, where the only information about the behaviour of the system is given by observing the output based on a test input.

In our work, we have full datasets obtained from the systems, and as such the test-based approach is not necessary to gain information about the behaviour of the system. However, the concept of defining time-based constraints could be valuable in hybrid automata as well, if the system state switches based on a timer or a particular seasonal period. Also, as we explore later in the paper, some types of non-stationary time series obtained from systems cannot be modelled when conditioning only on the variables. Furthermore, we have not found any implementation in other works concerning hybrid automata that integrates time-based constraints. As such, the utilisation of clocks in timed automata serves as a motivation for integrating time-based constraints into our hybrid automata synthesis process. Our work only considers one clock as in An et al. [18] for simplicity, and as with the rest of the synthesis process, we only consider learning the time constraints in an offline environment as opposed to Maier et al. [19].

The work in this paper concerns the synthesis of hybrid automata from time series, although it differs from the related work in the methods used to learn the dynamics, conditions and structure of the model. This work is a new framework for the entire synthesis process, however, it utilises two libraries

in the algorithms. The first library contains the SINDy algorithm which is utilised in the structure learning procedure to measure the fitting error when determining which location to associate a segment, and it is also used when learning dynamics to calculate the governing equations from the segments associated with each location. The second library contains the ODESolver which we use when simulating the hybrid automata models. Otherwise, the entire codebase is constructed by the authors of this paper and the algorithms are unique for this work. As an addition to existing work, we introduce timed conditions for the guards and invariants of the hybrid automaton. We define two types of timed conditions for different scenarios, and both can be used to model stationary time series, or non-stationary time series exhibiting trends, which is something we have not seen in related work.

3 Preliminaries

This section covers the semantics of a hybrid automaton, as well as some core definitions used throughout the paper. Note, that we use similar semantics as in the paper from last semester [3]. A hybrid automaton (HA) is a mathematical model that describes the behaviour of a system. The model may be visualized as a graph structure, as shown in Figure 1 and Figure 3. A HA consists of a finite set of discrete **locations**, each representing a component of the system. Each location is associated with a set of differential equations describing the continuous dynamics of the system when that location is active. Furthermore, a location may have an **invariant condition**, which is a condition that must be satisfied while that location is active. Invariant conditions are useful to enforce constraints on the behaviour of the system. When an invariant condition evaluates to false, the invariant is said to be *violated*.

The locations of a HA are connected by transitions, in this paper denoted as **edges**, which allow the system to transition between locations and thereby change which dynamics govern the behaviour of the system. The edges may be restricted by **guards**, which specify under which condition the transition can occur. When an expression of a guard on an edge evaluates to true, we denote the edge as *enabled*. An edge is only open for traversal when it is enabled. Additionally, each edge may have an **update function**, which is a function that is executed when the edge is traversed. It is not a differential equation and is therefore able to make a discontinuous change to the variables of the system. If the invariant condition is violated, the system must change state by traversing one of the enabled edges, or in the case that no edges are enabled, the system halts.

A HA is thus a tuple $H = (Q, E, X, Flow, Inv, Grd, Upd)$, where Q and X are the set of locations and real-valued variables respectively. We define a **data state** as a valuation of all variables in X and denote the set of all data states Σ_X. A **system state** is a pair (ℓ, σ) with location $\ell \in Q$ and a data state $\sigma \in \Sigma_X$. $E \subseteq Q \times Q \times Grd \times Upd$ is the set of edges of the system. An edge $e \in E$, where $Grd(e)$ is its guard condition, $Upd(e)$ is its update function. $Dest(e)$ is the destination location and $Source(e)$ is the source location. $OutEdg(\ell)$ is the set of edges that have ℓ as the source. An edge may be denoted $Source(e) \rightarrow Dest(e)$. $Flow(\ell)$ is defined as **flow** of a location and denotes the differential equation describing the continuous evolution of the data state on a location ℓ. $Inv(\ell)$ is the invariant condition of location ℓ.

The system state may change in two ways; by an instantaneous transition after traversing an edge, or by the elapse of time that changes only the values in Σ_X according to $Flow(\ell)$. A **simulation** ρ refers to the process of running or simulating the HA model. A **trace** τ_ρ is a finite or infinite sequence of data states describing the behaviour of the HA during a simulation ρ. S_H denotes the set of traces that corresponds to the simulations of the system H. At any time during the simulation, the configuration of the system is completely determined by the system state. For a more formal description of the semantics, refer to Alur et al.[2].

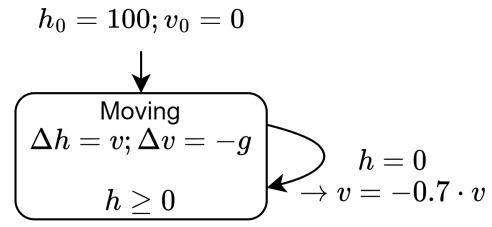
An edge may be enabled at multiple time points at a location. Likewise, multiple edges may be enabled at the same time on a location. Therefore, there must be a particular policy to determine which edge to take, and at what time to take it in the intervals at which it is enabled. Depending on the policy, this can resolve **non-determinism** in the HA. In our previous paper, we argued that to model unknown systems which may be non-deterministic, the choice of which of the enabled edges to traverse, and when to traverse it in the enabled interval, should be random. This policy allows the model to explore the different possible simulations, and as a consequence, S_H may consist of different traces, even with the same starting values in Σ_X. Although non-determinism may exist in a HA, the evolution of the data state in a location is **deterministic**, according to $Flow(\ell)$.

Synthesis of a HA refers to the process of constructing a HA model from given specifications, constraints or requirements.

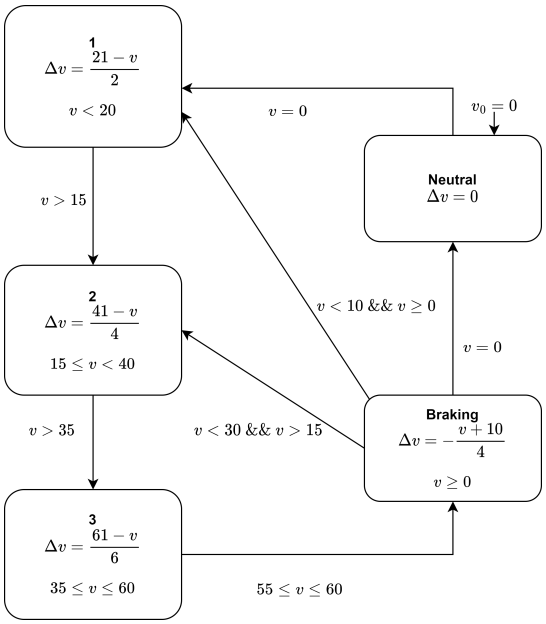
For this paper, we assume no prior knowledge of the system and as such are not given any specifications or requirements. Instead, the synthesis problem in this paper is, that specifications must be inferred from the time series of the system. That involves learning the continuous and discrete dynamics, as well as other design parameters that satisfy the underlying constraints of the system.

This paper utilises running examples throughout, which are used to illustrate the concepts and for evaluation purposes. The first example is a thermostat, the hybrid automaton of which is shown in Figure 1. It is a system with a temperature variable and a location for `On` and one for `Off`.

The temperature increases at a constant rate in the `On` location and decreases in the `Off` location. The thermostat is a basic model with simple dynamics and few locations, where the primary objective is to verify that the algorithm can learn the graph structure and the conditions. The second example is a model of a bouncing ball with one location and an update function that triggers when the ball hits the ground shown in Figure 3a. The bouncing ball only has one location, however, the dynamics are more complex than the thermostat. Thus, the bouncing ball example tests how well the algorithm can learn complex dynamics. The third example in Figure 3b is more advanced and represents a gear system with three gears, neutral, and the ability to break. This model has non-determinism in when to break, how long to break, as well as how long to stay at a gear level. The gearbox example is used to test the limits of the algorithm when facing systems with multiple locations, complex dynamics which are not significantly distinguishable, and a large degree of non-determinism.



(a) Model of the bouncing ball example.



(b) Model representing a gear system that allows the car to switch between gears.

Figure 3. Each node represents a location with a label, and the predicate within is an invariant condition. The edges' predicates are their guards. The notation "$\rightarrow v = -0.7 \cdot v$" is used to denote the update function on the bouncing ball model.[3]

4 Structure Learning

In our previous paper, we have shown that the Bottom-Up clustering algorithm may be used effectively to segment varying types of time series [3]. Each segment consists of sequential data which is assumed to be defined by the dynamics of a future location in the HA. The Bottom-Up algorithm returns a list of segments, but it does not associate any segment with a particular component nor show which segments stem from the same component dynamics.
This section focuses on a subsequent analysis of the segments, where the goal is to detect similar segments and associate them with a location, which represents a component of the system. Furthermore, when mapping a segment to a location, we keep track of the subsequent segment to later define the edges connecting the locations. The analysis makes it possible to build an initial structure of our hybrid automaton, and thus we denote this part of the synthesis as *Structure Learning*. This approach is inspired by the first phase of synthesis in García et al. [12], where they also map data points in time series to locations. We introduce the notion **Seg(ℓ)**, which denotes the set of segments of data from the time series associated with a location.

4.1 SINDy

Sparse Identification of Nonlinear Dynamical Systems (SINDy) is a sparse regression algorithm, first introduced by Brunton et al. [17].

The algorithm retrieves a time history of the data, i.e., a segment of data with time, and arranges it into two matrices denoted as X and $\dot{X}$ where $\dot{X}$ is the derivatives. Only the data in X are available, and $\dot{X}$ is approximated numerically through differentiation. SINDy utilizes a list of candidate functions of each column in X, denoted $\theta(X)$. By default, $\theta(X)$ consists of constant, linear and polynomial terms. The assumption is, that only a few of these describe the data accurately. Thus, they set up a sparse regression problem to determine the sparse vectors of coefficients $\Xi = [\xi_1, \xi_2, ..., \xi_n]$ that determine which are active. The feature library $\theta(X)$ is used to find the fewest terms needed to satisfy $\dot{X} = \theta(X)\Xi$, where the few entries of Ξ is solved by sparse regression, and denotes the terms in the right-hand side of the dynamics. In the paper, the authors show that SINDy can learn the governing dynamics of rich and chaotic systems such as the *Chaotic Lorenz System*, partial differential equations for vortex shredding behind an obstacle and equations for normal forms, bifurcations and parameterized systems. Figure 4 demonstrates the steps of the SINDy algorithm on the Lorenz

equations. In step 1 the data is obtained from the Lorenz system and the derivatives are arranged into the $dotX$ matrix and the candidate library $\theta(X)$ is constructed. From these, step 2 determines which of the candidate functions are active by using sparse regression to solve for the sparse coefficients of the dynamics. In step 3, the sparse vectors of coefficients are used to describe the actual dynamics of each variable in the Lorenz equation.

Because SINDy can learn a large variety of both simple and complex dynamics, it is used extensively in this paper. PySINDy is a Python library implementing the SINDy algorithm and it offers a suite of functions [22]. PySINDy's `fit()` function runs the SINDy algorithm and returns a model which describes the data on which it has been fitted. `simulate()` is another function, which takes a model, a starting value, and a time period to simulate data using an ODESolver and the governing equations of the model.

4.2 Associating Segments with Locations

The first part of structure learning entails associating each segment with a location. The main idea is to associate a segment with a location ℓ_i if the data in $Seg(\ell_i)$ are similar to the data in the segment. The similarity is determined by a fitting error between the segment and $Seg(\ell_i)$. Consider the algorithm of the initial procedure shown in Code Listing 1, which iterates through the segments and associates each with a location.

```
1  Input: Segments of the time series, maximum error
       threshold
2  Output: A list of locations, each containing a list
       of segments
3  MapSegmentsToLocations(segments, max_error):
4    locations = []
5    for segment in segments:
6      if (locations == []):
7        locations.append(new Location(segment))
8
9      # Maps a location to an error score
10     location_errors = {}
11     for location in locations:
12       fit_error = FindFitError(segment, location)
13       location_errors.put(location, fit_error)
14
15     best_fit = min(location_errors)
16     if best_fit.value() < max_error:
17       location = best_fit.key()
18       location.AddSegment(segment)
19     else:
20       locations.append(new Location(segment))
21
22   return locations
```

Code Listing 1. Pseudo Code for mapping segments to a location.

The algorithm iterates over existing locations $(\ell_0, ..., \ell_n) \in Q$ at lines 11-13, to determine if the current segment is similar to $Seg(\ell_i)$. Such similarity implies that data in the segment may be described by the same dynamics as $Seg(\ell_i)$, where index i in ℓ_i represents the location of the current iteration. On line 9 `FindFitError()` returns an error score representing the similarity of the segment and $Seg(\ell_i)$. The error is saved with the location in a dictionary on line 13. The next section describes how the error is determined in detail. After iterating through $(\ell_0, ..., \ell_n) \in Q$, ℓ_i with the best-fit score is retrieved on line 15. Note, that `best_fit` is a key-value pair, where the key denotes the location and the value denotes the error. If the score is below a threshold `max_error` at line 16, the segment is appended to $Seg(\ell_i)$. Otherwise, the segment is mapped to a new location ℓ_{n+1} which is then added to the set of locations Q, at line 20. Note that for the first iteration where `locations` are empty, a new location is created and the segment is associated with it.

4.3 Determining the Error

The intuition behind `FindFitError()` is, that if the segment and $Seg(\ell_i)$ are similar, then the governing equations of the SINDy model should create a simulation that is close to the data in the segment, given the same starting point. On the contrary, if they are dissimilar, the values of the simulation should differ from the segment, and the error should reflect that. Code Listing 2 shows the implementation which initialises the relevant fitting data, fits a SINDy model and calculates the error between the segment and the simulated data.

```
1  Input: Current segment and ℓ_i
2  Output: Fitting error
3  FindFitError(segment, location):
4    fitting_data = []
5    for location_s in location.segments:
6      fitting_data.append(location_s)
7
8    model = PySINDy.fit(fitting_data)
9
10   segment_simulation = PySINDy.simulate(model,
       segment[0], segment.time)
11
12   error = CompareData(segment, segment_simulation)
13   return error
```

Code Listing 2. Pseudo Code for FindFitError function.

The `fitting_data` variable defined on line 4 is an array of segments of data. On lines 5-6, `fitting_data` is further populated with $Seg(\ell_i)$. On line 8, PySINDy's `fit` function is used to fit a model based on `fitting_data`. Using this model, the `simulate()` function is used on line 10 with the model to simulate data points from the starting values of the variables in the segment, given the time points of the segment. The data returned from the simulation is compared against the actual data in the segment in `CompareData()` at line 12. `CompareData` may in principle use different methods of evaluating closeness, such as `Mean Square Error`(MSE). In this case, the method utilised calculates the error as the

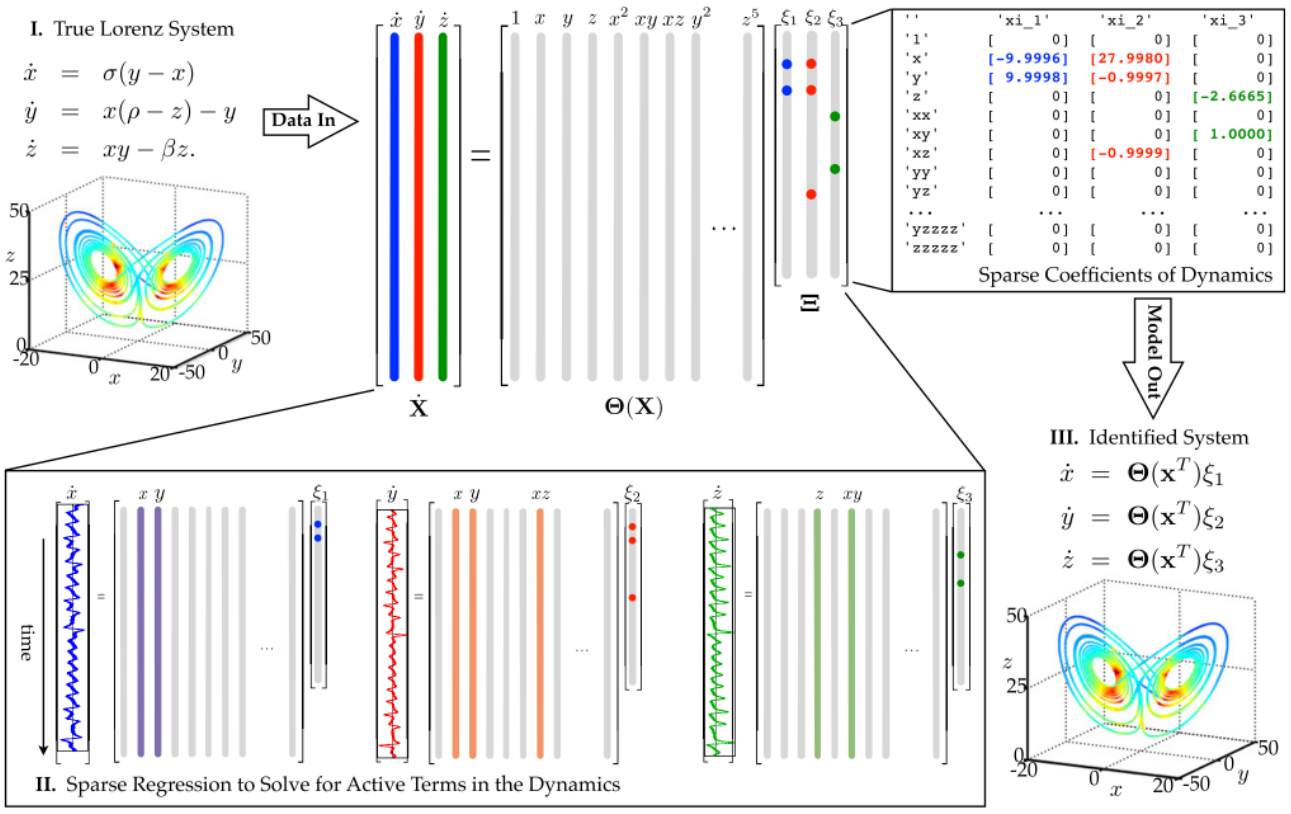


Figure 4. Schemantics of the SINDy algorithm, demonstrated on the Lorenz equations. The trajectory on the Lorenz attractor is coloured by the adaptive time step required, with red indicating a smaller time step. [17]

maximum distance between any two data points at a time point in the segment and the simulation. This works under the assumption, that the data does not contain significant outliers. If outliers are expected, MSE is likely the better option to evaluate the fit. The error is returned on line 13 and serves as the `fit_error` value in Code Listing 1. Note, that 4.2 and Code Listing 2 only produce one location for each unique set of dynamics. As such, two locations with the same dynamics cannot be modelled with this process.

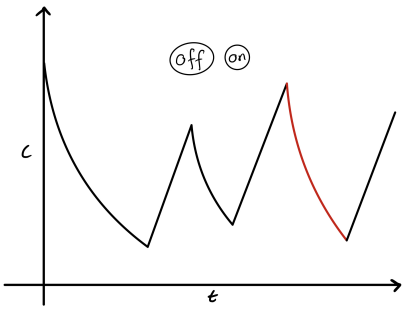
4.4 Finding Edges Between Locations

The edges are defined implicitly during structure learning. A segment contains a pointer to its neighbour segment. For segments $seg_0, \ldots, seg_{n-1}$, an edge e is created from seg_i's location ℓ_i to seg_{i+1}'s location ℓ_{i+1}. This facilitates the creation of directed edges in the graph structure.

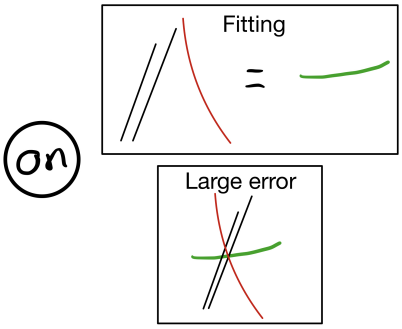
4.5 Example

Consider the thermostat example in Figure 1. For this simplified example, assume that the two distinct locations have already been identified with the algorithm, so the algorithm is four iterations in. At the current iteration, the algorithm then has to decide if the segment marked in red in Figure 5a belongs to the `Off` location, the `On` location, or to a new location.

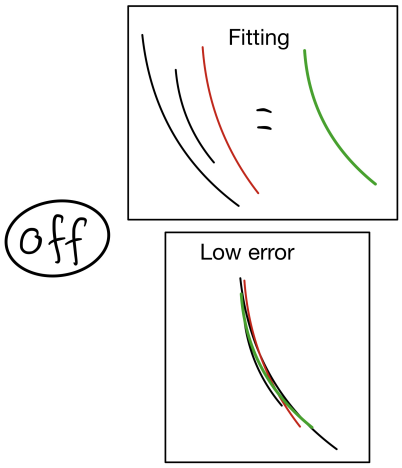
The algorithm calls `FindFitError()` on each location, measuring the error between the fit and the red segment. Figure 5b shows the pseudo-result of fitting the red segment to $Seg(On)$. It results in a large error, as denoted by the box on the bottom. In Figure 5c, the red segment is fitted with $Seg(Off)$, and as evident by the box on the bottom, this produces a low error. From this, the algorithm can infer that the red segment can not be associated with the `On` location, but it can be associated with the `Off` location. As the error is sufficiently low, the algorithm does not create a new location but instead associates the segment with the `Off` location. Note, that the smooth curves in the figures in this paper are only for simplicity. In practice, the segments contain time series and thus the curves cannot be smooth.



(a) Iteration of segments from a pseudo-dataset. `On` and `Off` locations have already been identified. The red-coloured segment is the current segment in the example. The x- and y-axis represent time and temperature, respectively.



(b) Fitting on the `Off` location with the red segment.



(c) Fitting on the `On` location with the red segment.

Figure 5. Example of how Code Listing 2 determines the fit error of a segment on a location using the thermostat model from Figure 1 as an example pseudo-dataset.

5 Learning Dynamics Governing the Locations

This section explains how the set of segments in a location is used to identify the governing dynamics.

5.1 The General Procedure

SINDy is useful for discovering governing equations from one or multiple segments. In Section 4, SINDy is used to identify governing equations of $Seg(\ell_i)$, and then apply the model to evaluate the fit of the new segment. A similar methodology is applied when identifying the dynamics of a location. All the segments of a location are used to fit a model using SINDy, from which the coefficients of the governing equations describing the data in the segments are extracted. The general procedure is described in Code Listing 3.

`CreateLocationSINDyModel` iterates over $(\ell_0, ..., \ell_n) \in Q$. The arrays `seg_times` and `seg_data` contain the time values and data of $Seg(\ell_i)$, which are populated on lines 7-9. Afterwards, the arrays are used to fit a model using the PySINDy library on line 10. Finally, `CreateCallable` builds a callable function on line 14 which is saved in the location so that it can be passed to the ODESolver as the dynamics of the location.

```
1  Input: locations
2  Output: locations
3  CreateLocationSINDyModel(locations):
4    for location in locations:
5      seg_times = []
6      seg_data = []
7      for segment in location.segments:
8        seg_times.append(segment.time)
9        seg_data.append(segment.data)
10     model = PySINDy.fit(seg_times, seg_data)
11
12     coefficients = model.coefficients
13
14     location.dynamics = CreateCallable(
   coefficients)
15   return locations
16
```

Code Listing 3. Pseudo Code for fitting SINDy model for each location.

5.1.1 Dynamics Function.

In our previous paper [3], we covered how the ODESolver in the HA codebase uses callable functions to describe the dynamics, although a lot of details were omitted. The following details how the ODESolver uses these functions to evolve the variables over time so that it can be understood how these functions can be generated from the coefficients of the governing equations.

The HA implementation uses the ODESolver from *scipy.integrate* python library [23]. In the documentation, it states that the dynamics are expressed as a function with two arguments.

```
Input: t denoting the current time, x denoting the values of
    the variables
Output: An array of the delta changes of the variables
func(t, x):
  ẋ_0 = compute x[0] delta
  ẋ_1 = compute x[1] delta
  ...
  ẋ_n = compute x[n] delta
  return [ẋ_0, ẋ_0, ..., ẋ_n]
```

Code Listing 4. ODESolver dynamics function outline.

Code Listing 4 describes the evolution of the variables in `x` at a time point `t`. The evolution is expressed as the delta changes of the values in the variables, which is the return of the function. A more concrete example is the thermostat described in this paper. Consider the simple dynamics for the `On` location, which simply increments the temperature by 1 for every 1 t. The corresponding function can be expressed as in Code Listing 5. Note, that as the thermostat only utilises one variable, the `x` array only contains one element.

```
Input: t denoting the current time, x denoting the values of
    the variables
Output: An array of the delta changes of the variables
OnDynamics(t, x):
  ċ = 1
  return [ċ]
```

Code Listing 5. ODESolver dynamics for the thermostat.

5.1.2 Different Representation.

The above way of defining a function is cumbersome and difficult to automate with an algorithm. Instead, it would be beneficial to consider a more compact way of representing the dynamics using matrices. Equation 2 is inspired by models in *state space* form, which are often used to capture differential equations.[5]

$$\dot{x} = Ax + b \tag{2}$$

x is the continuous state and A is a matrix containing the coefficients for all the variables in x. The coefficients are obtained through the SINDy equations. The equations may contain constants as well, and those are contained in the vector b.

As an example, consider the bouncing ball model Figure 3a, which has two variables, h and v, for the height and velocity, respectively. To model the dynamics when the ball is bouncing, the equation should contain $A = [[0, 1], [0, 0]]$ and $b = [0, -G]$ where $-G$ is the gravity constant, 9.82 $\frac{m}{s^2}$. Notice that A has an entry for each variable. The first entry $[0, 1]$ relates to $\dot{h}$ and together with the first entry in the vector b the equation for $\dot{h}$ may be written as seen in Equation 3. It indicates that the coefficient for h is 0, the coefficient for v is 1, and that 0 should be added when calculating the change in h.

$$\dot{h} = 0h + 1v + 0 \tag{3}$$

Likewise, the second entry $[0, 0]$ relates to $\dot{v}$ and together with the second entry in the vector b the equation for $\dot{v}$ is written as seen in Equation 4. It indicates that the evolution of v is entirely defined by the constant G.

$$\dot{v} = 0h + 0v - G \tag{4}$$

With the above formalisation, we may parse the coefficients extracted from PySINDy to construct the appropriate A matrix and b vector. Equation 2 appears to only support linear dynamics. However, due to the flexibility of the Python programming language, an entry in the A matrix may contain a function with an expression containing the coefficient and variable. As an example $2x^2$ may be expressed as a function $lambda\ x : 2x^2$.

5.1.3 Converting Coefficients to Matrix Representations.

Code Listing 6 contains the pseudo-code for the construction of the dynamics function for the ODESolver. On line 2 the A matrix and b vector are constructed by parsing the coefficients to the `ConstructMatrices` function. On line 3 the function `ODEFunction` is returned to be used as the dynamics function. Note, how the function just applies Equation 2 to compute $\dot{x}$ and Ax is matrix-vector multiplication and x is inserted into the lambda functions of A.

```
1 Input: Current segment and ℓ_i
2 Output: Fitting error
3 CreateCallable(coefficients):
4   A,b = ConstructMatrices(coefficients)
5   return ODEFunction(t, x):
6     return Ax+b
```

Code Listing 6. Pseudo-Code for CreateCallable function.

6 Invariants and Guards

This section describes how invariant and guard conditions are determined for the locations of the HA. The invariants and guards of ℓ_i are determined by analysing $Seg(\ell_i)$, and $\{Seg(Dest(e))|e \in OutEdg(\ell_i)\}$.

6.1 Convex Hull Method

Firstly, the bounds from which the invariants and guards are later determined are found. Invariants- and guard-bounds are determined using a similar methodology as in other related papers concerning the synthesis of a HA [10–12]. In the following, **Chull**, describes a convex hull. The invariant bound of a location is described by a convex hull of all points of $Seg(\ell_i)$ in Equation 5. Similarly, guard bounds are described by a convex hull of all endpoints of $Seg(\ell_i)$ in Equation 6. Note, that for multivariate data Equation 5 and Equation 6 are applied for each variable, and the conditions are a conjunction of the results.

$$Inv\_Bounds(\ell_i) = Chull(\{points(s) | s \in Seg(\ell_i)\}) \quad (5)$$

$$Grd\_Bounds(\ell_i \rightarrow \ell_{i+1}) = Chull(\{end(s) | s \in Seg(\ell_i)\}) \quad (6)$$

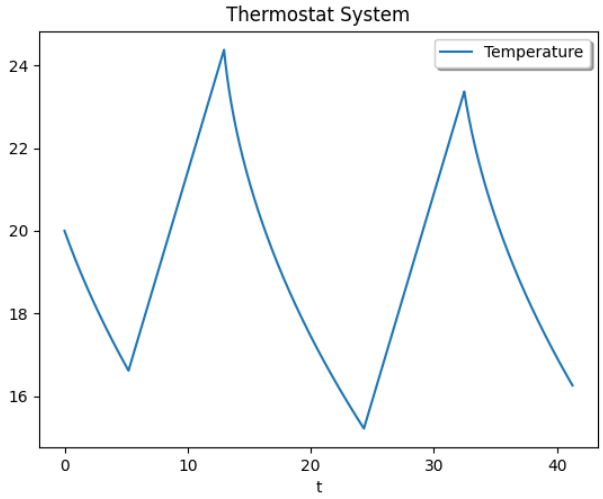


Figure 6. Example of a trace of the thermostat automaton. The segments going downwards are the Off location and the segments going upwards are the On location.

Consider the example Figure 6 describing a pseudo-dataset for a thermostat. After segmenting the data, the structure learning procedure identifies two locations, On and Off, and associates the relevant segments to each. The lower and upper bounds for the invariant of Off are seen to be 15 and 24, respectively. Similarly, the upper bound for On is 24 and the lower bound is 15.

Furthermore, the lower and upper bounds for the guard from Off to On are seen to be 15 and 17, at time 6 and 24, respectively, and the lower and upper bounds for the guard from On to Off is seen to be 23 and 24, respectively.

One important note is that only the required invariants and guards are created. If the values in the final point in at least one segment are close to either of the bounds, then an invariant condition is created using that bound. If the bounds are not captured by any of the created invariants, then a guard is created using that bound. This is done to avoid creating unnecessary conditions, which may result in a larger HA than necessary and to only have the conditions for the relevant variables in multivariate systems.

In the example of the thermostat, this entails that the Off location ends up having the invariant $Inv(Off) = x \geq 15$ because at least one segment's final value is close to the lower bound of 15, but no segment is close to the upper bound. Similarly, the guard of $Grd(Off \rightarrow On) = x \leq 17$ is created, utilising its upper bound and not the lower bound as that is captured by the invariant. Note, that conditions using the lower bound utilise the greater than or equal to operator ($\geq$), and the opposite for conditions using the upper bound. This results in the location Off having to switch location at 15 and the edge being open after 17.

6.2 Data with Trend

Some time series are non-stationary and the convex hull method is only suitable for stationary data. For a stationary time series, we expect the mean, variance and covariance to be constant with time, which is not always the case. Consider Figure 7 which describes the atmospheric CO_2 concentration from the Mauna Loa Observatory in Hawaii in the period 1958-2001 [24]. This is an example where the mean varies (increases) with time, which results in an upward trend. The following sections focus on non-stationary trending data and leave the other cases for future work.

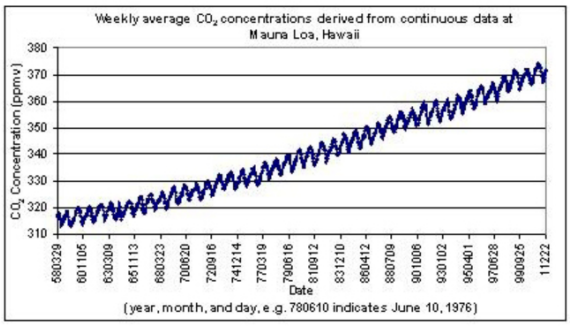


Figure 7. CO_2 concentration of the Mauna Loa Observatory in Hawaii in the period 1958-2001.[24]

Figure 7 appears to exhibit seasonality, that is, periods where the concentration decreases and periods where the concentration increases, likely defined by the seasons in a year. After segmenting the data and performing structure learning, seasonality is likely captured by two distinct locations. Using the convex hull method on the segments results in large intervals for the guards and invariants. The large intervals cause several issues. For one, the guards are always enabled and the invariants are never violated. Thus, when modelled with a HA, a transition from one location to the other can occur at any time, regardless of the season. Secondly, it is not possible to simulate further than the end of the dataset, as the bounds in the conditions are fixed. Due to these complications, the following section presents two approaches to determining the invariants and guards in time series exhibiting trends. The first method is useful when modelling systems where the time series exhibits seasonality, as shown in Figure 7. The second method is useful when modelling timer-based systems with no relation to seasonality.

6.3 Seasonal Time Conditions

For Figure 7, it appears that the fluctuation in the CO_2 concentration is defined by seasonality, which can be approximated into constant time intervals. It might be the case that the concentration decreases in the spring and summer, and increases in the autumn and winter. Under that assumption, the evolution of the concentration can be modelled with two locations. One representing the downward flow of the concentration, and one representing the upward flow of the concentration. The HA's locations may be active at different parts of the year, and thus the year may be defined as a **seasonal period**. We denote time conditions based on seasonality as *seasonal time conditions*. As an illustration of the seasonal time conditions applied on the time series in Figure 7, consider the model in Figure 8, which is a simplified and fictitious version. In Figure 8 modulo is used to associate parts of the seasonal period with a location. One location captures the spring and summer, and another captures autumn and winter.

Importantly, the entire seasonal period must be covered by the invariants of the locations to avoid the HA entering a deadlock.

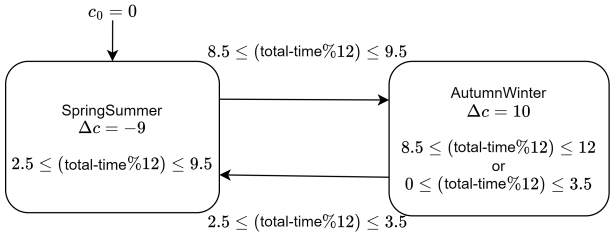


Figure 8. Example of a simplified CO_2 model with an upward trend. The dynamics are made-up for this model. The seasonality period is a year and each time unit is a month.

6.4 Location Time Conditions

For the non-seasonality-based time intervals, the modulo approach does not suffice. To enable the use of location time conditions a new concept *loc-time* is introduced. This concept grows with time but is reset when transitioning between locations. Furthermore, time will be denoted as *total-time* to avoid confusion with loc-time.

Such conditions could be $loc\text{-}time \leq 4$ and $loc\text{-}time \leq 100$, which means that the HA needs to change location after 4 and 100-time units, respectively, and we define these as *location time conditions*. Location time conditions may be useful when modelling systems that are governed by different dynamics based on a timer. As a novel example, consider Figure 9 which is a model of a thermostat exhibiting an upward trend. The thermostat stays on for 7-8 seconds before switching off for 2-3 seconds. Although such a model does not make sense in practice as the temperature increases indefinitely, it serves as an illustration of how location time conditions may be used to capture timer-based systems.

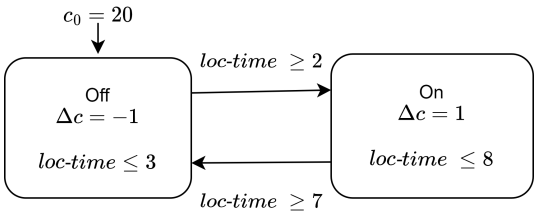


Figure 9. Example model of a thermostat with an upwards trend.

6.5 Remarks on the Time-Based Condition Types

Location time conditions and seasonal time conditions can coexist in a model, however, such implementation rarely makes sense because they serve different purposes. Seasonal time conditions are useful for systems where each location covers a part of the seasonal period, whereas location time conditions are useful for systems which do not relate to seasonality. Thus, for principal reasons, the algorithm implementing these concepts should diagnose which of the timed conditions to use when modelling a particular system. It should be noted, that while both of the time-based condition types may be used when modelling time series with trends, they can also be used when modelling stationary time series. Thus, these new approaches solve the existing problems with the convex hull approach, and also add new ways to determine conditions from time only, for stationary time series or time series exhibiting trends.

6.6 Timed conditions algorithm

Code Listing 7 presents the general outline for the algorithm that determines the invariants and guards for a HA. Initially, the time bounds are determined and afterwards, a consistency check is performed to determine whether to use seasonal time conditions or location time conditions.

```
Input: List of locations
Output: conditions
DetermineInvariantsAndGuards(locations):
  bounds = FindBounds(locations)
  use_seasonal = IsConsistentPeriod(locations)
  if (use_seasonal):
    conditions = SetSeasonalTimeConditions(bounds,
     locations)
  else:
    conditions = SetLocationTimeConditions(bounds,
     locations)
  return conditions
```

Code Listing 7. The general outline for the algorithm that determines time-based invariants.

Code Listing 8 iterates through locations $(\ell_0, \ldots, \ell_n) \in Q$ and find the minimum and maximum time spent in their associated segments $Seg(\ell_i)$. We distinguish between location time bounds and seasonal time bounds. The location time bounds consider the time units spent in a segment, whereas the seasonal time bounds consider parts of a seasonal period. Firstly, the location time bound is determined on lines 13-14, where `segment.loc_time` denotes the amount of time in the segment. Secondly, the seasonal time bounds are determined on lines 17-19, where `segment.total_time` denotes the total time elapsed in the time series until that point. Note, line 17 which skips the first segment, as its minimum time is always equal to the start time of the time series. The `MOD_VALUE` on lines 18-19 is a constant configured by defining the seasonal period. As an example, if the seasonal period covers a day, one might choose to interpret a time unit as an hour, which requires a seasonal period of 24. In future work the seasonal period and the chosen time unit interpretation should be analysed from the time series, however, in this work, we predefine the constant according to the time series in evaluation.

```
1  Input: List of locations
2  Output: seasonal and location time bounds
3  FindBounds(locations):
4    location_bounds = []
5    seasonal_bounds = []
6    for location in locations:
7      l_min_time = MAX_VALUE
8      l_max_time = MIN_VALUE
9      s_min_time = MAX_VALUE
10     s_max_time = MIN_VALUE
11     for segment in location.segments:
12       # Location time bounds
13       l_min_time = min(segment.loc_time.first,
   l_min_time)
14       l_max_time = max(segment.loc_time.last,
   l_max_time)
15
16       # Seasonal time bounds
17       if segment != location.segments.first:
18         s_min_time = min(segment.total_time.
   first % MOD_VALUE, s_min_time)
19       s_max_time = max(segment.total_time.last %
    MOD_VALUE, s_max_time)
20
21     location_bounds.append((Null, l_max_time))
22     seasonal_bounds.append((s_min_time,
   s_max_time))
23   return seasonal_bounds, location_bounds
24
```

Code Listing 8. The FindBounds function

After the bounds have been determined, `IsConsistentPeriod` checks the segments in the locations for consistency in time. Consistency in time means that for $(\ell_0, \ldots, \ell_n) \in Q$, the time period in the segments $Seg(\ell_i)$ all start at approximately the same time, and end at approximately the same time. It does not have to be exactly the same time, as a small difference can be adjusted with a guard. If the difference in time in all $Seg(\ell_i)$ is above a threshold, location time conditions are used instead.

6.7 Determining the Condition Expressions

When the bounds have been found, the condition expressions are determined similarly to the state-dependent conditions as explained in subsection 6.1. One difference is with the seasonal time conditions where the interval spans the end of the seasonal period and the start of the next.

Consider the invariants of the `AutumnWinter` location in the model shown in Figure 8. The bounds of the system, with a period time of 12 time units, are found to be $[8.5, 3.5]$. 8.5 is the lower bound as it is found by analysing the start times of $Seg(AutumnWinter)$. Likewise, 3.5 is found to be the upper bound, by analysing the end times of $Seg(AutumnWinter)$. As a result, the invariant should be enabled when $total\_time\%12$ is above 8.5 or below 3.5. The following condition function is then generated: $Inv(AutumnWinter) = (total\_time\%12 \geq 8.5) \vee (total\_time\%12 \leq 3.5)$.

7 Complete Example

In this section, we provide a complete example of all the steps involved in learning the thermostat model in Figure 1. The time series obtained from a trace used in this example is a short snippet of the first 25 time units from a simulation on the pre-build model. An excerpt of the time series is given in Table 1.

Time	Temperature
0.00	20.00
0.12	19.87
0.18	19.74
0.24	19.61
0.30	19.47
...	...
24.76	18.96
24.82	18.83
24.88	18.66
24.94	18.54
25.00	18.38

Table 1. Example time series from the thermostat model.

7.1 Segmenting the Time Series

The first step is to segment the time series using the change point detection algorithm detailed in our previous paper [3]. The segmentation is shown in Figure 10, where each vertical dotted line represents a change point. The resulting five segments are represented by rows in Table 2 where the columns represent the value of the first point and the last point in the segments, respectively.

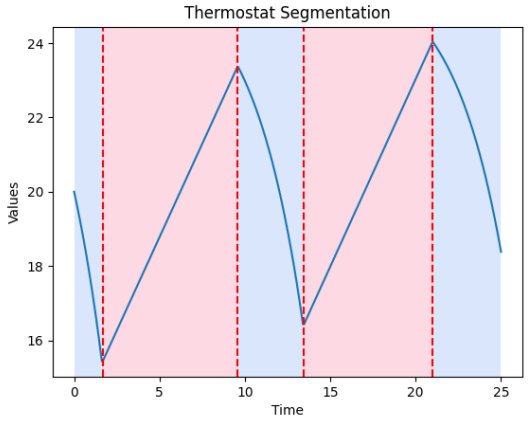


Figure 10. Segments after applying change point detection algorithm to the time series, a snippet of which is shown in Table 1.

First	Last
20.00	15.44
15.44	23.35
23.35	16.43
16.43	24.02
24.02	18.38

Table 2. Time series segments from the thermostat trace.

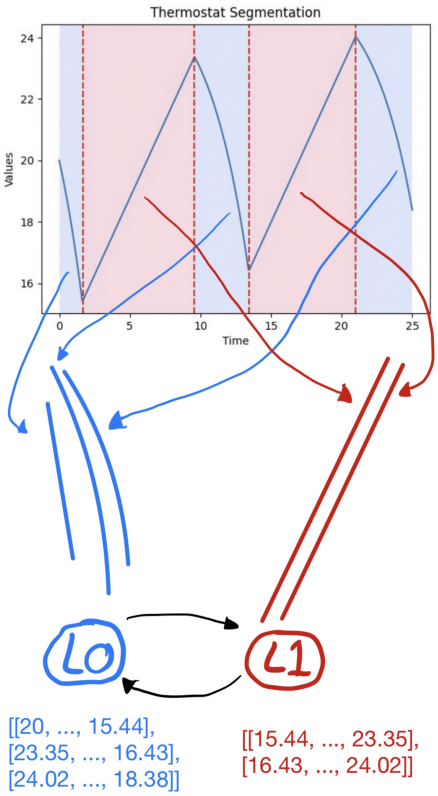


Figure 11. Shows how the segments are associated with each location.

7.2 Structure Learning

The structure learning procedure iterates through each segment and associates similar segments to a location, while keeping track of neighbouring segments to construct edges. The result of applying the structure learning procedure is shown in Figure 11.

7.3 Learning Dynamics

For each $(\ell_0, ..., \ell_n) \in Q$, $Seg(\ell_i)$ is used to fit a model with SINDy which in turn gives the governing equations for ℓ_i. The learned equations are shown in Figure 12.

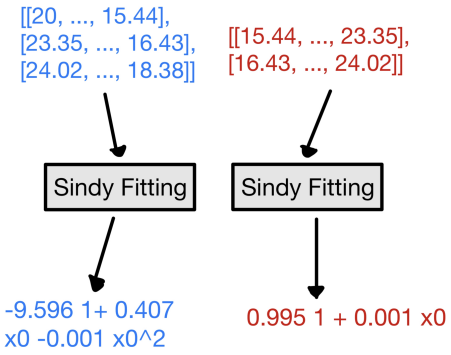


Figure 12. Dynamics learned by fitting a model on $Seg(\ell_i)$ for $(\ell_0, ..., \ell_n) \in Q$ with SINDy.

7.4 Learning State-Dependent Conditions

The thermostat uses state-dependent conditions to determine guards and invariants. Thus, the next step is to use Equation 5 and Equation 6 to determine the conditions for the HA. Note, that for the guard calculation, we do not consider the last segment as it contains no right-side neighbour segments. The calculation is shown in Equation 7 and Equation 8.

As described in subsection 6.1, only the utilised invariants and guards are created, eg. the invariant $Inv(L0) = x0 \leq 24.02$ is not created as no segment in $Seg(L0)$ ends close to 24.02. The final invariants and guards are shown in Equation 9.

$$\begin{aligned} Inv\_Bounds(L0) &= Chull([20, ..., 15.44], [23.35, ..., 16.43], \\ &\quad [24.02, ..., 18.38]) = [15.44, 24.02] \\ Inv\_Bounds(L1) &= Chull([15.44, ..., 23.35], [16.43, ..., 24.02]) \\ &= [15.44, 24.02] \end{aligned} \tag{7}$$

$$\begin{aligned} Grd\_Bounds(L0 \rightarrow L1) &= Chull([15.44, 16.43]) \\ &= [15.44, 16.43] \\ Grd\_Bounds(L1 \rightarrow L0) &= Chull([23.35, 24.02,]) \\ &= [23.35, 24.02] \end{aligned} \tag{8}$$

$$\begin{aligned} Inv(L0) &= x0 \geq 15.44 \\ Inv(L1) &= x0 \leq 24.02 \\ Grd(L0 \rightarrow L1) &= x0 \leq 16.43 \\ Grd(L1 \rightarrow L0) &= x0 \leq 23.35 \end{aligned} \tag{9}$$

7.5 Final Model

The final step is to construct the HA using the locations and edges from the graph structure, as well as the dynamics and conditions. The final synthesised model is shown in Figure 13. A simulation of the model produces the blue trace in Figure 14, which is shown with the original trace.

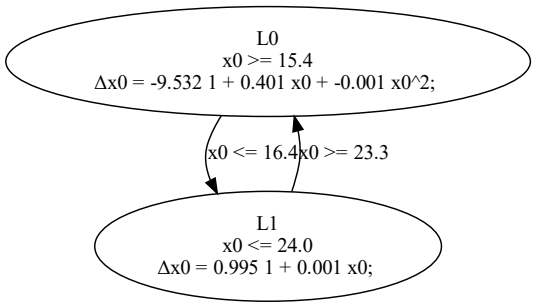


Figure 13. Final model.

8 Evaluation

The following section evaluates the final models of systems learned by our algorithm by comparing them to pre-build models of the same system. We distinguish between non-stationary time series, and stationary time series to evaluate time-based conditions and state-dependent conditions. In addition, we also synthesise the CO_2 time series in Figure 7 from the actual time series from [24]. This is done to see how the whole procedure handles a real recorded dataset with noise, outliers and trends. Note, that there is no evaluation of the change point detection algorithm, which was done in our previous paper [3].

8.1 Datasets

The evaluation includes five pre-defined models that are used to generate traces. The thermostat system Figure 1, the gearbox system Figure 3b and the bouncing ball system

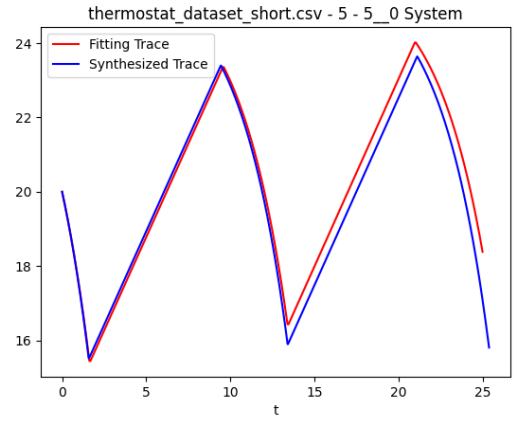


Figure 14. Example of a trace from the model in Figure 13 against the original time series. Started at $time = 0$ and ended at $time = 25$.

Figure 3a produce stationary traces, while the simplified CO_2 system **??** and the trending thermostat system Figure 9 produces traces with trends. All obtained traces contain 1000 points and at maximum 10 location switches.

8.2 Hyperparameters

The algorithm has one primary hyperparameter, which is the max-error in structure learning used to determine if a segment is associated with a location. The hyperparameter is denoted as S_e. If S_e is too high, segments may be non-similar but still associated with the same location. On the contrary, if S_e is too low, segments may be similar but associated with different locations.

8.3 Evaluation Process

The goal is to evaluate the closeness of the synthesised models, to the pre-defined models. The synthesised models are simulated, and the generated traces are compared to the generated traces of the pre-defined models. For both the synthesised model and the pre-defined models, the simulations use the same time steps. For non-deterministic models, 10 traces are produced for each pre-defined model and an arbitrary trace is used to synthesise a model, which is simulated 5 times to obtain 5 traces. The 10 pre-defined traces define one subset and the five traces obtained by the synthesised model define a second subset. For deterministic models such as the bouncing ball, only one pre-defined model trace is compared against the 5 synthesised traces. The closeness of the traces is evaluated using the Hausdorff distance measure, which measures how far two sets are from each other [25]. The Hausdorff distance is the maximum distance of a set to the nearest point in the other set, or more formally, a *maxmin* function defined as in Equation 10, where we define $d(a, b)$ as the absolute difference between a and b.

$$d_H(A,B) = \max_{a\in A}(\min_{b\in B}(d(a,b))) \tag{10}$$

For multivariate data, the Hausdorff distance is summed for each variable and divided by the number of variables. The traces are compared and evaluated, but the graph structure, dynamics and conditions are also important factors. This is because non-determinism in the models can make the trace significantly different from the pre-defined model traces, even though the model is correctly synthesised. Thus, a figure of the synthesised model is saved for manual review.

The evaluation process is a grid search on S_e with the values $[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]$ inspired by parts of the fibonnachi sequence. The process is shown in Code Listing 9.

```
Evaluation():
  for S_e in [1,2,3,5,8,13,21,34,55,89,144]:
    # Stationary datasets
    for s_traces in [thermostat, gearbox, b-ball]:
      model = train(s_traces[0], S_e)
      result = eval(s_traces, model)
      save(model, result)
    # Non-stationary datasets (with trend)
    for n_s_traces in [trend-thermostat,
    simple_CO2, CO2]:
      model = train(n_s_traces[0], S_e)
      result = eval(n_s_traces, model)
      save(model, result)

eval(s_traces, model):
  test_traces = generateTestTraces(model)
  return calculateHausdorff(s_traces, test_traces)
```

Code Listing 9. The general outline for the evaluation.

8.4 Results

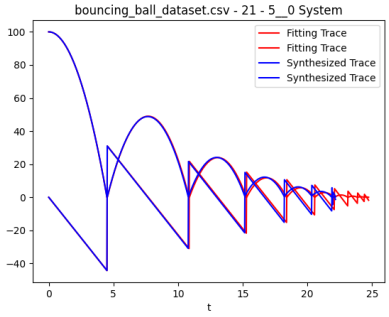
In the following, the best results from the grid search evaluation as well as the results from the CO_2 dataset are detailed.

8.4.1 State-Dependent Condition Models.

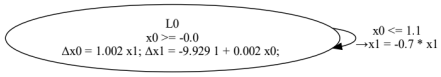
The stationary time series are used to evaluate state-dependent conditions. The traces are obtained using the thermostat, the bouncing ball and the gearbox pre-build models.

Bouncing Ball Evaluation. Figure 15a shows the trace comparison for the bouncing ball example. The 21 in the title signifies the error threshold for the change point detection algorithm, the 5 signifies S_e and the _0 means that the pre-build model trace is shown with the 0th synthesised trace, out of the 5. Initially, the traces are similar, however, towards the end of the synthesised trace, it appears to deviate from the pre-build model trace which is evident by the Hausdorff distance of 35.42. Looking at the model structure in Figure 15b, the issue may be identified as the guard condition. The bouncing ball model should only jump when it touches the ground at height zero, however, the synthesised model guard condition indicates that it can jump between zero and 1.1. This is even more evident when evaluating multiple synthesised traces, as seen in Figure 15c. This is a byproduct of the structure learning, as the change point detection may be slightly imprecise and may miss by a point, this is further explained in our last paper [3]. Consequently, the missed point may appear in the previous segment and thus it is required to cut off one point from the ends of all segments to ensure good results. However, this causes some of the condition bounds to be slightly offset from the correct value.

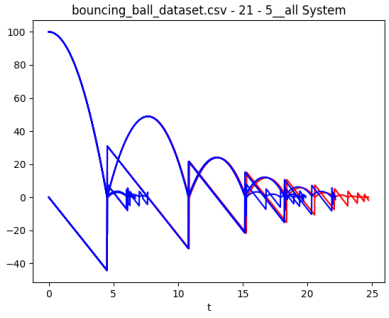
Note, that our algorithm cannot create update functions, therefore the bouncing ball's update function has been manually inserted. In Figure 15b the constant -0.7 on the update function is a dampening factor on the velocity.



(a) Bouncing ball result.



(b) Bouncing ball model structure. No conditions utilise the x1 variable, which represents the velocity, as no segments end close the bounds of the velocity, as explained in subsection 6.1.

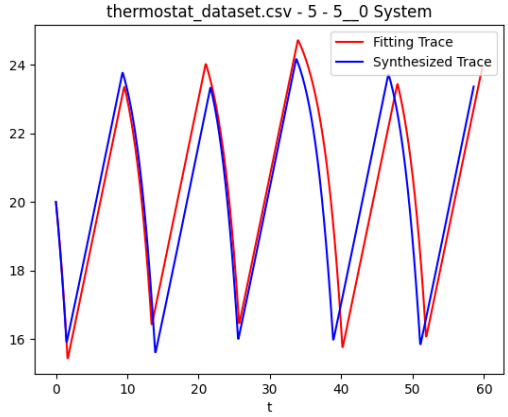


(c) All bouncing ball evaluation traces.

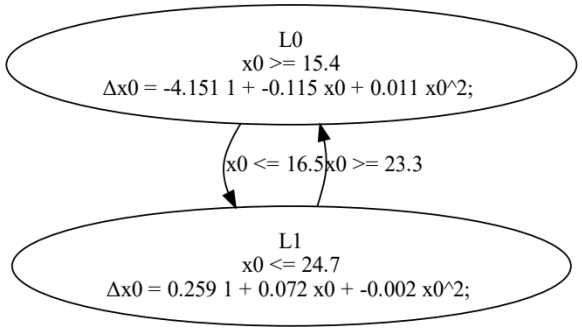
Figure 15. Bouncing ball trace evaluation and learned model.

Thermostat Evaluation. The thermostat result is shown in Figure 16a and the overall Hausdorff score is 0.26. There is non-determinism in when to switch between locations, which is more evident when looking at all the comparisons in Figure 16c. However, the model structure in Figure 16b is

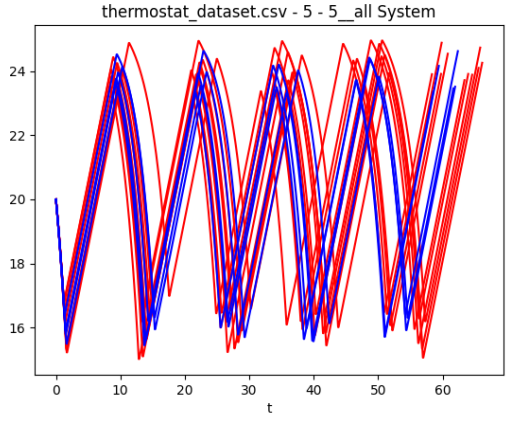
similar to the pre-build model which means that the locations, edges and conditions were learned correctly. The dynamics appear to be close as well.



(a) Thermostat result.



(b) Thermostat model structure.



(c) All thermostat evaluation traces.

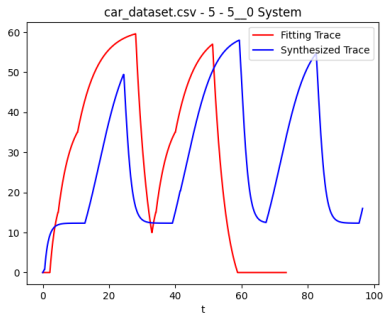
Figure 16. Thermostat trace evaluation and learned model.

Gearbox Evaluation. The synthesised gearbox model is seen in Figure 17b and it does not resemble the original model in Figure 3b. It only has three locations compared to 5, and location L0 may transition into itself. The dynamics of L0 appear to represent the three gears in one location, which is likely due to the similar dynamics. L2 appears to represent the neutral location, and L1 appears to represent the breaking location. The model may transition from the breaking location to the gear location, denoted by the guards on the edges from L0 to L1. Thus, the general outline of the model is learned with the gear, breaking and neutral state, but the edges identified during structure learning are not accurate. As an example, it is not possible to leave the neutral location.

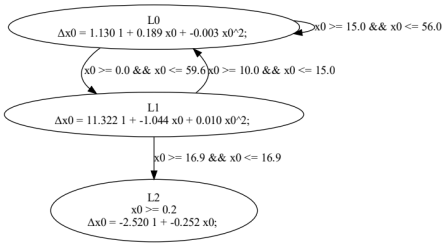
When investigating the problem, we found that the change point detection is generally able to divide the trace into segments for the neutral location and segments for the breaking locations. This is expected as both the neutral location and the breaking location have distinct dynamics. However, the change point detection was generally not able to distinguish between the three gear levels, and at most times would create one, sometimes two and rarely three segments to represent the trace throughout the gear shifts. The issue trickles down into the structure learning procedure, where the segments are associated with a location based on similarity. Less segments, as well as indistinguishable dynamics, creates more similarity in the segments which reduces the number of locations identified in the structure learning procedure. Furthermore, an edge is only created from a location to the location containing the right-side neighbour segments in the segmented trace. When L0 have an edge going to itself, it denotes that two consecutive segments are associated with the same location, which should not be possible because the change point detection denoted them as stemming from separate dynamics. In this case, the change point detection algorithm can distinguish between the dynamics, however, the structure learning procedure can not.

The problem with large models is two-fold. Firstly, it is required that the change point detection can segment the data accurately. Secondly, the structure learning procedure must be able to associate the segments to the appropriate locations. The dynamics and conditions are determined based on the segments associated with each location. If the first two procedures provide inaccurate results, the rest of the procedures are prone to error. Because the dynamics are based on all the segments in a location, a few misplacements in the structure learning procedure may still provide somewhat accurate dynamics. However, one misplacement creates an edge, and thus misplaces the edge, which explains the edges going from and to the same location, and why it is not possible to transition from the neutral location to the gear location. This is likely to be an issue in most complex systems with multiple locations without significant distinction in dynamics. Also, the original trace in Figure 17a is likely too short for the algorithm to synthesise the model correctly.

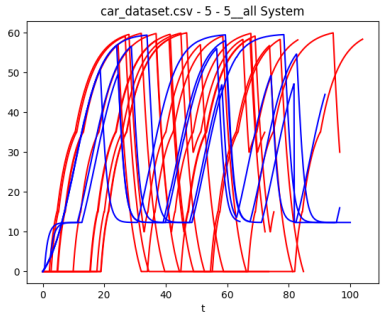
The obtained trace is compared in Figure 17a and Figure 17c, and the Hausdorff distance is 3.84.



(a) Gearbox result.



(b) Gearbox model structure.

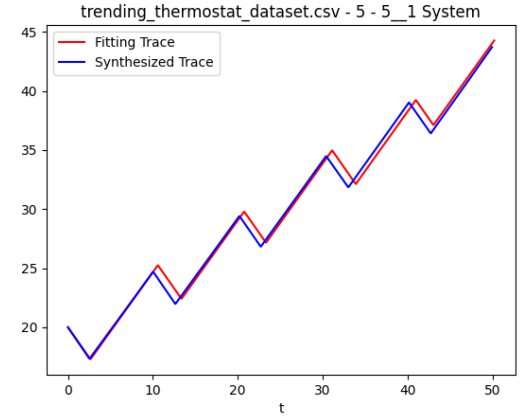


(c) All gearbox evaluation traces.

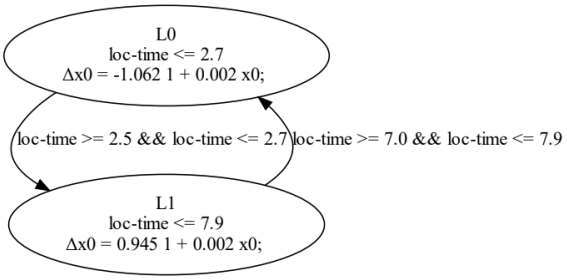
Figure 17. Gearbox trace evaluation and learned model.

8.4.2 Timed Condition Models. We evaluate the trending thermostat, which contains location time conditions, and the simplified CO_2 model which contains seasonal time conditions.

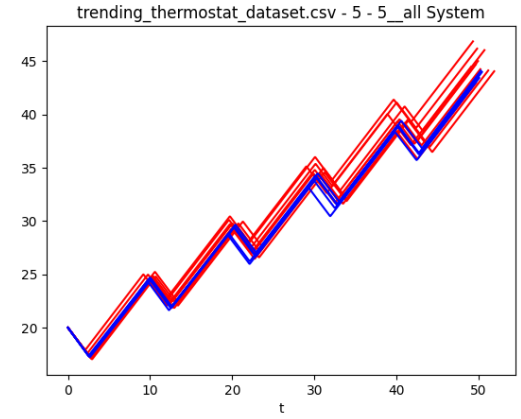
Trending Thermostat Evaluation. The trending thermostat result is shown in Figure 18a. Like the thermostat example, the trending thermostat has a degree of non-determinism as evident by the difference in traces in Figure 18c. However, the Hausdorff distance is 0.13 so the synthesised model can capture the non-determinism. The model structure in Figure 18b contains appropriate location time conditions, so the switching conditions are entirely dependent on time, as intended.



(a) Trending thermostat result.



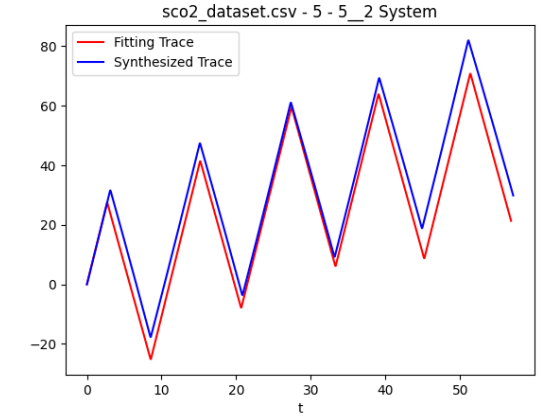
(b) Trending thermostat model structure.



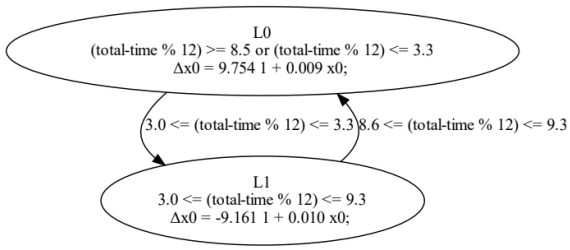
(c) All trending thermostat evaluation traces.

Figure 18. Trending thermostat trace evaluation and learned model.

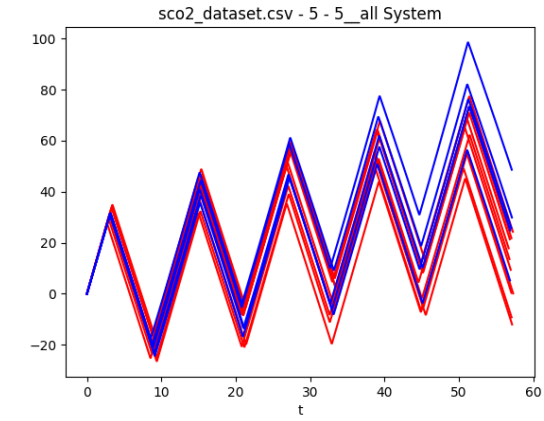
Simplified CO_2 Evaluation. The simplified CO_2 result is shown in Figure 19a with a Hausdorff distance of 1.36. As with the thermostat, there is a degree of non-determinism in the model which is evident in Figure 19c. The learned model structure in Figure 19b contains the correct dynamics and the model learned to use appropriate seasonal time conditions, as intended.



(a) Simplified CO_2 result.



(b) Simplified CO_2 model structure.



(c) All simplified CO_2 evaluation traces.

Figure 19. Simplified CO_2 trace evaluation and learned model.

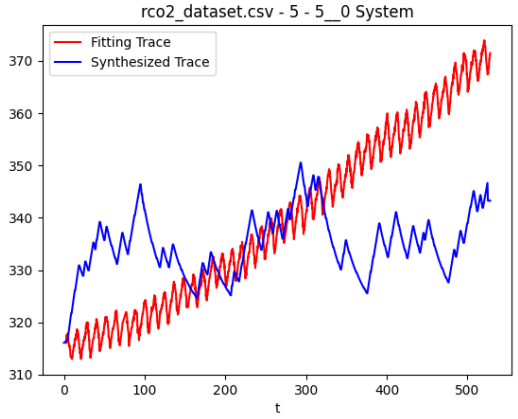
8.4.3 Pre-Build Model Conclusion. In general, the learning algorithm can synthesise simple stationary models with state-dependent conditions, however, it has difficulty synthesising complex models such as the gearbox example. This is due to the many locations in those models, which require the change point detection to segment correctly, and the structure learning procedure to detect the number of locations as well as associate the segments to the locations. In order for the change point detection and the structure learning procedures to be successful on complex models, the dynamics of each location have to be significantly distinguishable, otherwise, several locations are likely to be viewed as one.

The evaluation shows that the algorithm can synthesise simple systems based on location time conditions or seasonal time conditions. Thus, the algorithm also supports simple systems which exhibit trends.

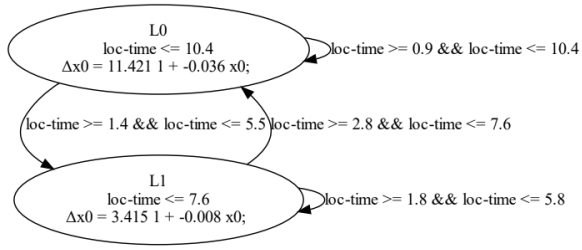
8.4.4 Real CO_2 Dataset. The CO_2 model learned from the dataset in Figure 7 is seen in Figure 20b. There are no seasonal time conditions on the model, and the dynamics do not capture any long-term upward trends. As in the gearbox example, this issue is in the first steps of the algorithm. In the CO_2 model, the greatest issue is the segmentation, which is seen in Figure 21. To illustrate the point, consider Figure 22 which is a small excerpt of the trace in Figure 21. The CO_2 concentration appears to be noisy and while the oscillation seems to be consistent throughout the trace, the data in each variation is not. Unlike the thermostat which increases linearly until its peak, the CO_2 trace has several smaller up and down periods before reaching the peak or the bottom. Due to these complications, the change point detection cannot perform an accurate segmentation of the trace. As a result, the structure learning procedure associates contradicting parts of the trace to the same location. As in the gearbox example, such misplacements may cause locations to have edges going to themselves such as for L0 and L1 in Figure 20b.

The reason there are no seasonal time conditions on the model is because of the consistency check, which determines whether seasonal time conditions are chosen. If the segments in the locations are not in approximately the same time period of the season, then it is not possible to approximate an accurate seasonal period, and thus, the seasonal time conditions are omitted. When it is decided to omit the seasonal time conditions, the algorithm uses location time conditions instead as seen in Figure 20b. Consider $Grd(L0 \longrightarrow L1)$ which describes the minimum time of a segment on L0 to be 1.4, and the maximum time to be 5.5. All the guards have similar large time intervals, which signifies that there is no consistency in the time periods in each segment. To synthesise the CO_2 dataset using seasonal time conditions, the inconsistency should be solved by further tuning the change point detection for better segmentation on that particular dataset.

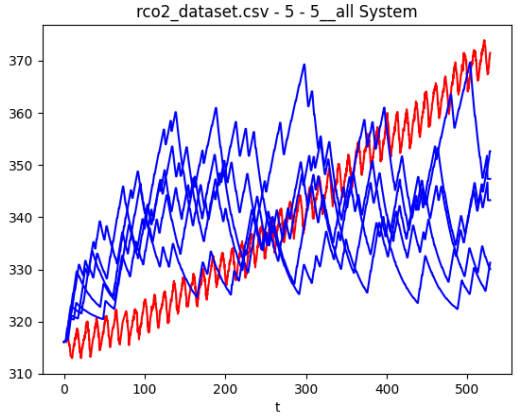
The traces obtained from Figure 20b are compared to the original trace in Figure 20a and Figure 20c with a Hausdorff distance is 18.53.



(a) Real CO_2 model structure using location time conditions.



(b) Simplified CO_2 model structure.



(c) All simplified CO_2 evaluation traces.

Figure 20. Simplified CO_2 trace evaluation and learned model.

9 Conclusion

This paper details a learning algorithm for modelling hybrid systems from time series using hybrid automata.

The first step of the algorithm consists of segmenting the time series using the Bottom-Up change point detection algorithm, which was detailed and evaluated in our previous paper [3]. Afterwards, a structure learning procedure groups similar segments into unique locations and also keeps track of the neighbouring segments to determine the edges connecting the locations. The structure learning procedure only produces locations with unique dynamics. As a consequence, it is not possible to learn a model with multiple locations with the same dynamics. From the graph structure and the segments of data associated with each location, the differential equations describing the dynamics governing a location can be learned using the SINDy algorithm.

The paper details two implementation strategies for determining the guard conditions on the edges, as well as the invariant conditions on the locations. The first strategy uses a convex hull to compute the state-dependent conditions and works well for stationary data. The second strategy is a new time-based methodology introduced in this paper and revolves around creating conditions for time series exhibiting upward or downward trends. It distinguishes between seasonal time conditions and location time conditions. The seasonal time conditions are suitable for models where a location is always active at a certain time interval in the seasonal period. The location time conditions do not relate to seasonality but instead denote an upper limit on the time a location can stay active and is useful when modelling timer-based systems. Both timed-type conditions may be used for stationary or non-stationary data.

The evaluation shows that the algorithm can synthesise simple models with an accurate graph structure and dynamics as well as state-dependent conditions and both types of timed conditions. The algorithm is not able to accurately synthesise complex models with multiple locations, as the location dynamics have to be significantly distinguishable for the change point detection to segment correctly, and for the structure learning to identify the appropriate locations. When trying to learn the real CO_2 dataset, we also find that the change point detection does not segment correctly, possibly due to the amount of noise in the dataset. In conclusion, the overall algorithm is conceptually sound and produces good results for simple models, but further work is required for the change point detection and structure learning procedure to support more complex datasets and systems.

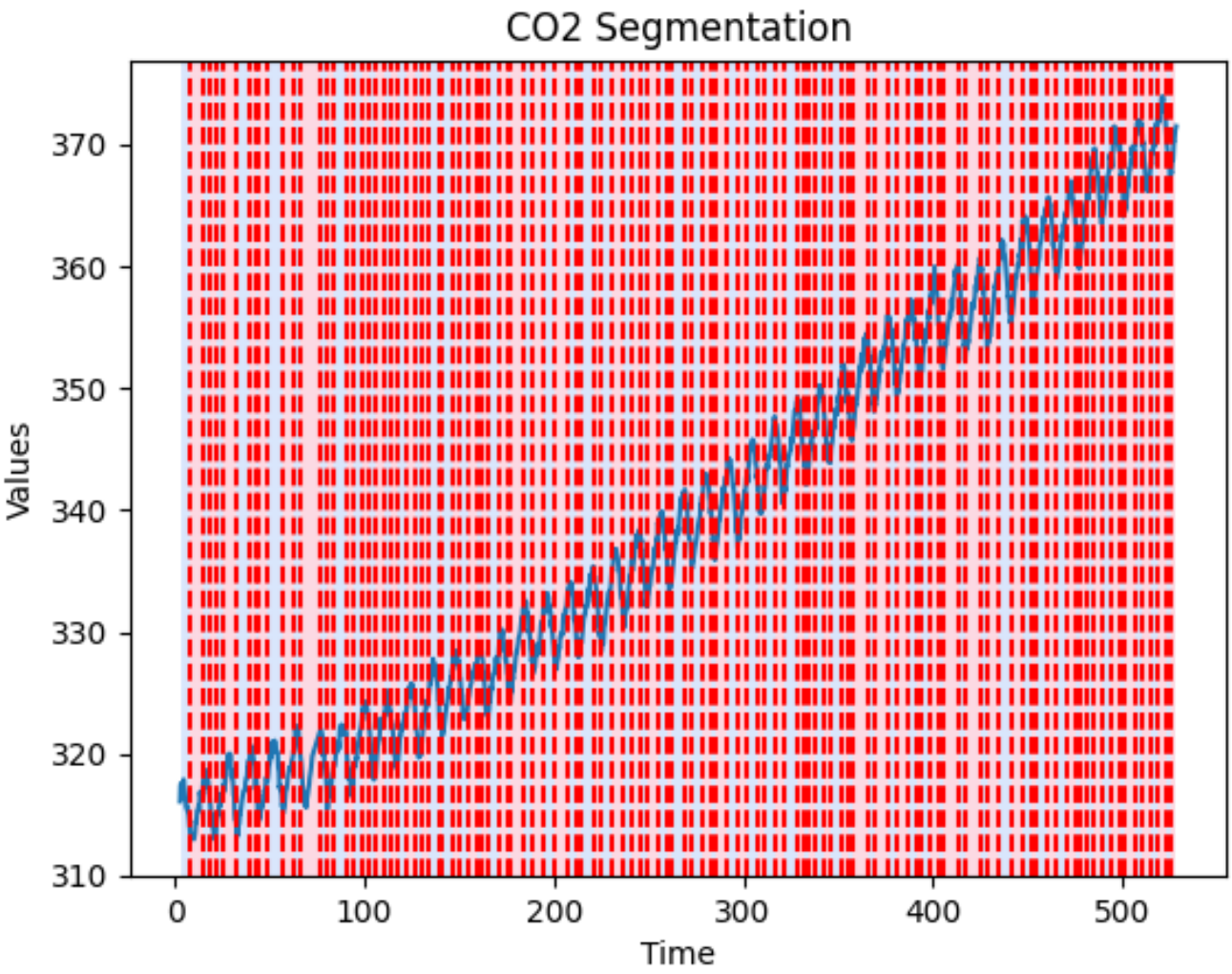


Figure 21. Real CO_2 segmentation.

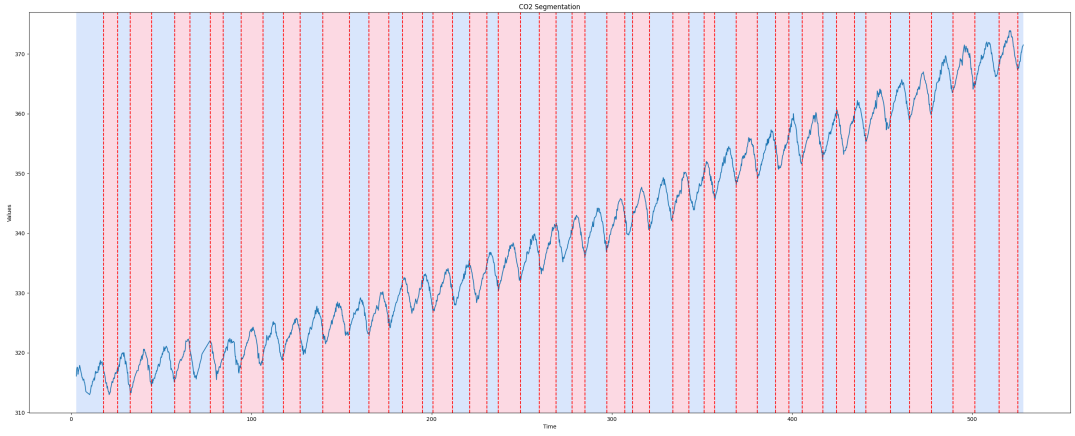


Figure 22. Excerpt of the CO_2 segmentation.

10 Future Work

The following details proposals for improvements and additions to the codebase and algorithms.

10.1 Automatically Detect if a Time Series is Stationary

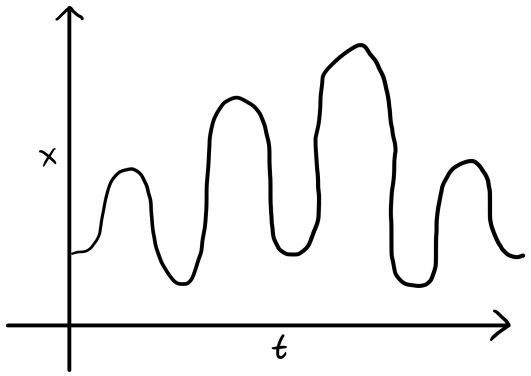
We distinguish between stationary time series and time series with trends for the synthesis process. As of right now, each time series is manually labelled by the authors as one of the two. In future work, an analysis should be performed on the time series to automatically label it before the synthesis process begins.

10.2 Determining the Seasonal Period for Timed Conditions

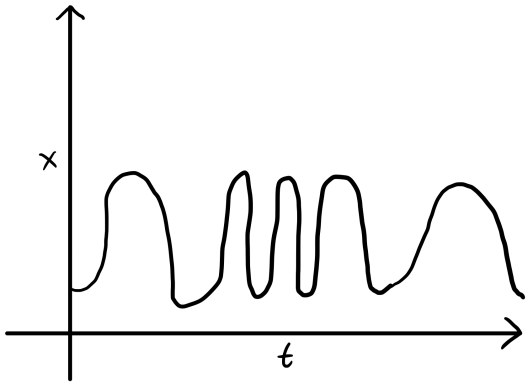
Currently, the seasonal period and a single time unit in the period, are predetermined based on the authors' knowledge of the time series. Ideally, those variables would be automatically configured by performing an analysis of the data. The structure learning procedure builds the graph structure and denotes the locations as well as their associated segments of data. From this, a subsequent analysis could be performed, which approximates the seasonal period accordingly.

10.3 Support for Other Types of Non-Stationary Data

Only non-stationary data which exhibits trend are supported in the current synthesis process. Future work includes support for other types of non-stationary data, such as when the variance of a series is a function of time as seen in Figure 23a, or when the covariance is a function of time as in Figure 23b.



(a) Variance of the series is a function of time.



(b) Covariance is a function of time.

Figure 23. Describes two forms of non-stationary data.

10.4 Update Functions

The current synthesis process cannot learn update functions, which limits the number of systems that can be modelled. In the future, detecting and learning update functions should become a part of the synthesis of the HA model.

One possibility is to use the SINDy algorithm to learn the update functions, by fitting a model on the points before and after the update. This has been shortly tested which gave promising results, but was not developed further.

10.5 Experimenting With Other SINDy Candidate Functions

SINDy offers a suite of libraries with different candidate functions to use during the sparse regression. It is also possible to define custom candidate functions. Other candidate functions could have been integrated into our codebase to support a larger variety of dynamics. The current implementation utilises the default library of constants, linear, and polynomial functions. [26]

References

[1] Github. Git copilot website, 2023. [Online; accessed 08. June. 2023].

[2] 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 Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1992.

[3] Alex Immerkær Kristensen Jakob Østenkjær Hansen. Hybrid system simulation using hybrid automaton and change point detection for segmentation of time series data. Aalborg University, 2023.

[4] L. Ljung and T. Glad. *Modeling of Dynamic Systems*. Prentice-Hall information and system sciences series. PTR Prentice Hall, 1994.

[5] Simone Paoletti, Aleksandar Lj. Juloski, Giancarlo Ferrari-Trecate, and René Vidal. Identification of hybrid systems a tutorial. *European Journal of Control*, 13(2):242–260, 2007.

[6] Giancarlo Ferrari-Trecate, Marco Muselli, Diego Liberati, and Manfred Morari. A clustering technique for the identification of piecewise affine systems. *Automatica*, 39(2):205–217, 2003.

[7] Aleksandar Juloski, S. Weiland, and W.P.M.H. (Maurice) Heemels. A bayesian approach to identification of hybrid systems. *Automatic Control, IEEE Transactions on*, 50:1520 – 1533, 11 2005.

[8] A. Bemporad, A. Garulli, S. Paoletti, and A. Vicino. A bounded-error approach to piecewise affine system identification. *IEEE Transactions on Automatic Control*, 50(10):1567–1580, 2005.

[9] Yi Ma and René Vidal. Identification of deterministic switched arx systems via identification of algebraic varieties. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*,

pages 449–465, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] Miriam García Soto, Thomas A. Henzinger, and Christian Schilling. Synthesis of hybrid automata with affine dynamics from time-series data. In *HSCC*, United States, 2021. Association for Computing Machinery.

[11] Miriam García Soto, Thomas A. Henzinger, Christian Schilling, and Luka Zeleznik. Membership-based synthesis of linear hybrid automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 297–314, Cham, 2019. Springer International Publishing.

[12] Miriam García Soto, Thomas A. Henzinger, and Christian Schilling. Synthesis of parametric hybrid automata from time series. In Ahmed Bouajjani, Lukás Holík, and Zhilin Wu, editors, *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings*, volume 13505 of *Lecture Notes in Computer Science*, pages 337–353. Springer, 2022.

[13] P Rajendra and V Brahmajirao. Modeling of dynamical systems through deep learning. pages 1311–1320, 2020.

[14] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, and Ali Jasim Ramadhan. Universal differential equations for scientific machine learning. *CoRR*, abs/2001.04385, 2020.

[15] Michael Lutter and Jan Peters. Combining physics and deep learning to learn continuous-time dynamics models. *CoRR*, abs/2110.01894, 2021.

[16] Jianhua Gao, Weixing Ji, Lulu Zhang, Senhao Shao, Yizhuo Wang, and Feng Shi. Fast piecewise polynomial fitting of time-series data for streaming computing. *IEEE Access*, 8:43764–43775, 2020.

[17] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, mar 2016.

[18] Jie An, Mingshuai Chen, Bohua Zhan, Naijun Zhan, and Miaomiao Zhang. Learning one-clock timed automata, 2020.

[19] Alexander Maier. Online passive learning of timed automata for cyber-physical production systems. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 60–66, 2014.

[20] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. Time to learn – learning timed automata from tests. In Étienne André and Mariëlle Stoelinga, editors, *Formal Modeling and Analysis of Timed Systems*, pages 216–235, Cham, 2019. Springer International Publishing.

[21] Martin Tappler, Bernhard K. Aichernig, and Florian Lorber. Timed automata learning via smt solving. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, pages 489–507, Cham, 2022. Springer International Publishing.

[22] Markus Quade Dynamicslab. Pysindy documentation, 2023. [Online; accessed 26. March. 2023].

[23] docs.scipy. solve ivp ode solver documentation, December 2022. [Online; accessed 11. March. 2021].

[24] C.D. Keeling and T.P. Whorf. Atmospheric co2 concentrations derived from flask air samples at sites in the sio network. in trends: A compendium of data on global change., 2004. [Online; accessed 06. April. 2023].

[25] C.F. Olson. A probabilistic formulation for hausdorff matching. In *Proceedings. 1998 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No.98CB36231)*, pages 150–156, 1998.

[26] Markus Quade Dynamicslab. Pysindy library documentation, 2023. [Online; accessed 8. June. 2023].