A Framework for Mining Hybrid Automata from Input/Output Traces

Ramy Medhat[1], S. Ramesh[2], Borzoo Bonakdarpour[3], Sebastian Fischmeister[1]
[1]University of Waterloo, Canada. [2]GM R&D, United States. [3]McMaster University, Canada.
{rmedhat,sfichme}@uwaterloo.ca, ramesh.s@gm.com, borzoo@mcmaster.ca

ABSTRACT

Automata-based models of embedded systems are useful and attractive for many reasons: they are intuitive, precise, at a high level of abstraction, tool independent and can be simulated and analyzed. They also have the advantage of facilitating readability and system comprehension in the case of large systems. This paper proposes an approach for mining automata-based models from input/output execution traces of embedded control systems. The models mined by our approach are hybrid automata models, which capture discrete as well as continuous system behavior. Specifically this paper proposes a framework for analyzing multiple input/output traces by identifying steps like segmentation, clustering, generation of event traces, and automata inference. The framework is general enough to admit multiple techniques or future enhancements of these steps. We demonstrate the power of the framework by using some specific existing methods and tools in two case studies. Our initial results are encouraging and should spur further research in the domain.

1 Introduction

Automata-based specifications are useful for specifying behavioral descriptions of complex embedded control systems. They are intuitive and precise, abstract and high level, and more importantly they are tool-neutral. Further, automata-based models allow simulation, testing, debugging and sometimes even rigorous formal verification. In the case of legacy systems and physical systems, the automata specifications can help developers migrate to model based development. Unfortunately, legacy systems often lack such models. Further, for large systems, one could focus on specific aspects of the system and build precise and compact automata specifications capturing those aspects for better comprehension, debugging and testing.

For embedded control systems, a suitable automata-based model is the hybrid automata model [16]. Hybrid automata are extensions of traditional finite state machines capturing discrete states and transitions as well as continuous behavior using differential equations. Hybrid automata have the advantage of visualizing system states and how control transitions between them, allowing for an abstract conceptual understanding of the system.

This paper presents a framework for learning hybrid automata models from black-box implementations of embedded control systems by mining their input/output traces. For instance, consider a legacy system for which there exists no model or documentation. Our objective is to design a framework that uses input/output traces collected from executing such a system, and recovers a hybrid automaton that models the behavior present in the collected traces. To that end, we propose a general framework for mining by clustering multiple input/output training traces, translating the clustered traces to event sequences, and constructing automata based upon the input/output correlation. To demonstrate how to use the framework, we populate it with some concrete methods and tools of clustering, segmentation, automata constructions and perform some experiments.

The specific contributions of the paper include:

- A formalized framework for mining hybrid systems specifications. The formalization precisely describes the assumptions and expectations of the various components required for mining. This helps incorporate future evolutions and refinements of the components into the framework.
- A set of algorithms and heuristics for the suggested framework that have proven to work for the evaluated case studies.
- A quantitative evaluation framework for future work, so follow-up work can compare the results to this and other approaches.
- Case studies demonstrating the application of the proposed framework to different control systems.

Recently, the works in [12, 18, 23] attempted to construct models from black-box systems using only execution traces. However, that line of work requires some level of understanding of formal logic, hence it is less accessible to engineers. Whether the objective is system comprehension, simulation, or integration with other systems, it is challenging for an engineer to decipher properties in formal logic. In contrast, automata-based models are easily visualized to aid in their comprehension.

The remainder of the paper is structured as follows: Section 2 introduces a motivating example of the use of the proposed framework. Section 3 formalizes the problem and introduces some terminology. Section 4 introduces the proposed framework and discusses its steps, as well as the concrete models and tools used to build our case studies. Section 5 discusses the case studies we implement to experiment with the proposed framework. Section 6 discusses related work, Section 7 discusses the limitations of the proposed framework, and finally Section 8 presents the conclusion of the paper and future work.

2 Motivating Example

Consider the traces in Figure 1, which are extracted from an engine timing control system. The topmost trace is the output engine speed. The middle trace is the input throttle speed. The bottom trace is the input load torque. The control system is responsible for actuating the input throttle and minimizing the effect of changes in load torque.

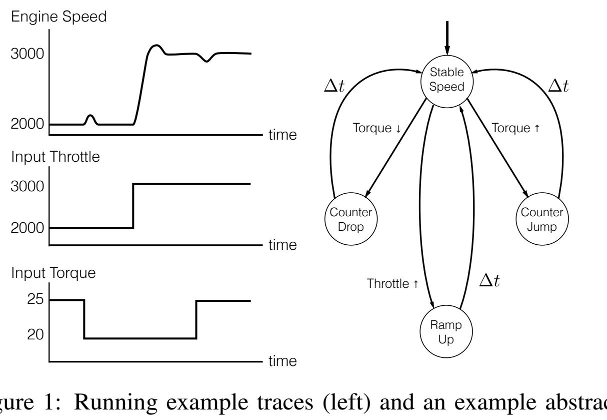


Figure 1: Running example traces (left) and an example abstract automaton (right).

The objective of the proposed framework is to construct a hybrid automaton that models and abstracts the behavior of such a control system. On the right-hand-side of Figure 1 is an example of a manually constructed automaton that models the trace on the left. The automaton realizes the following behavior:

- The system starts at a stable speed.
- If the torque drops, the transition labelled "Torque ↓" is taken. The engine speed experiences a bump while the controller is attempting to negate the effect of the torque drop. This can be seen in the engine speed trace.
- The system returns to a stable speed after some time Δt.
- If the throttle rises, the system transitions to the "Ramp Up" state where it gradually increases the engine speed.
- The system returns to a stable speed after some time Δt .
- Similar to torque drop, if the torque rises the engine speed experiences a dip, which is rectified by the controller and the speed returns to stability once again.

For the rest of the paper, we will use this example to illustrate the different stages of the proposed framework to produce a hybrid automaton model from the I/O traces of a system.

3 Problem Definition

We attempt a formal definition of the general framework which focuses on the requirements including the assumptions of the interfaces, abstracting out the details of the specific algorithms used for different steps of the mining algorithm. This helps in accommodating future versions or refinements of the different components like clustering, segmentations, etc. employed in the proposed work.

3.1 Input/Output Traces

Our objective is to construct a hybrid automaton using input/output traces of a hybrid system. Naturally, the hybrid system receives a set of input signals and produces a set of output signals. First, we define a sampled trace of any such signal.

DEFINITION 1 (SAMPLED TRACE). *A sampled trace w_S of signal S is a finite sequence of pairs of timestamps and values (t_1, v_1) $(t_2, v_2) \cdots (t_p, v_p)$ such that:*

- *$t_i \in \mathbb{R}^{\geq 0}$, $\forall i \in [1, p] : t_i < t_{i+1}$ (strict monotonicity),*
- *$\forall i \in [2, p] : t_i - t_{i-1} = c$ where c is the sampling period,*
- *$\forall i \in [1, p] : v_i \in D(S)$ where $D(S)$ is the domain of signal S sampled values.*

A sub-trace of w_S is indicated as $w_S[t_i, t_j]$ where t_i is the first timestamp in the sub-trace and t_j is the last timestamp. □

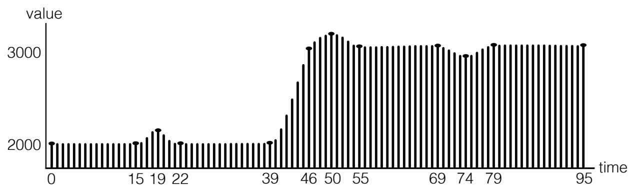


Figure 2: Visualization of a sampled trace.

Figure 2 demonstrates a visualization of an engine speed signal that has been sampled to construct a trace w. The speed is seen starting at 2000 rpm and rises to 3000 rpm as the input throttle increases.

3.2 Presegmented Traces

By breaking down traces into segments, we can correlate changes in the inputs to changes in the outputs in a piece-wise manner, which is essential in mining the behavior of the digital aspect of the hybrid system. Signal segmentation heavily relies on the type of signal, and multiple methods are successful within the scope of their target signal type. Hence, we assume traces are pre-segmented using some signal segmentation method that is not in the scope of this work, since we are not attempting to contribute to the science of signal processing. Therefore, a trace of an input or output signal is segmented or *split* at points where some segmentation algorithm detects abrupt change in the signal values. Ideally, these points – referred to as *change-points* – are selected by the segmentation algorithm to mirror changes in the system's state. However, the quality of the segmentation process depends on the quality of the segmentation algorithm and the inputs it receives. Different algorithms [3, 7, 34] are more accurate for specific types of signals. For instance, detecting change-points in a sinusoidal signal depends on changes in frequency, amplitude, or phase shift, whereas a boolean signal exhibits change-points only when it toggles. We refer to these various characteristics as features.

DEFINITION 2 (FEATURE). *A feature is a function*

$$\phi : W \to V$$

where W is the domain of all sampled traces, and V is the domain of feature vectors extracted from the input trace. V is also referred to as the feature space, for which the similarity relation $\sim$ is defined. □

An example of a feature is a linear fitting feature ϕ_l, which constructs a linear fit to the values of the signal in the trace. Such a feature returns a vector $\langle m, b\rangle$ where m and b are parameters of the linear fit equation ($y = mx + b$).

For instance, consider the signal in Figure 2, specifically the samples at times 0, 15, and 19. These three points form two sub-traces: $w[0, 15]$ and $w[15, 19]$. Let feature ϕ_s be a feature extracting the slope of the linear fit of the values in the segment. According to the trace in the figure, $\phi_s(w[0, 15]) = \langle 0.0\rangle$, while $\phi_s(w[15, 19]) = \langle 1.0\rangle$. Let the similarity relation for the feature space of ϕ_s be defined as follows:

$$\phi(w) \sim \phi(w') \quad \text{iff} \quad \|\phi(w) - \phi(w')\| \leq 0.1$$

According to such definition, $\phi_s(w[0, 15]) \not\sim \phi_s(w[15, 19])$. The similarity relation is normally more complicated. The work in [28, 32] introduces different methods to cluster feature vectors based on similarity.

Hence, a signal is processed by some segmentation algorithm. This algorithm uses knowledge of the feature(s) that indicate change of state in the signal to identify change-points. We formally define change-points as follows:

DEFINITION 3 (CHANGE-POINT VECTOR). *Given*

- *a sampled trace $w_S = (t_1, v_1) \cdots (t_p, v_p)$ of signal S,*
- *a set of features $\varphi = \{\phi_1, \cdots, \phi_q\}$*

a change-point vector $\pi_{w_S,\varphi}$ is a vector of time stamps that indicate a significant change in feature vectors, and is defined as follows:

$$\pi_{w_S,\varphi} = \langle \tau_1, \tau_2, \cdots, \tau_l\rangle$$

such that

- *$\forall i \in [1, l] : \tau_i \in \{t_j \mid j \in [1, |w_S|]\}$,*
- *$\tau_1 = t_1$ indicating the time of'; the first sample of the signal, and $\tau_l = t_p$ indicating the time of the last sample of the signal,*
- *each change-point τ_k, where $k \in [2, l - 1]$, indicates a significant change in the feature vector produced by at least one feature when applied to the left side sub-trace $w_s[\tau_{k-1}, \tau_k]$ versus the right side sub-trace $w_s[\tau_k, \tau_{k+1}]$. That is*

$$\forall k\ \exists j : w_s[\tau_{k-1}, \tau_k] \overset{\phi_j}{\not\sim} w_s[\tau_k, \tau_{k+1}]$$

where $j \in [1, q]$ and $\overset{\phi_j}{\sim}$ is a similarity relation defined for feature ϕ_j. □

Recall the trace in Figure 2, and feature ϕ_s which given a sub-trace returns the slope of the linear fit of all values in that sub-trace. As indicated in the figure by the samples with circles on top, these are points where the *slope* shows significant change between the left hand side segment and the right hand side segment. Thus, for this trace, $\varphi = \{\phi_s\}$, and consequently the change-point vector is

$$\pi_{w_S,\varphi} = \langle 0, 15, 19, 22, 39, 46, 50, 55, 69, 74, 79, 95\rangle \quad (1)$$

A hybrid system is a system where continuous-time dynamics interact with discrete-event dynamics. For a hybrid system S, let $\mathcal{I} = \{I_1, I_2, \cdots, I_n\}$ be a set of input signals to the system, and $\mathcal{O} = \{O_1, O_2, \cdots, O_m\}$ be a set of output signals produced by the system. We now define an input/output sampled trace as an extension to Definition 1.

DEFINITION 4 (INPUT/OUTPUT TRACE). *Given a set of input signals $\mathcal{I}$, and a set of output signals $\mathcal{O}$, an input/output trace $w_{\mathcal{I},\mathcal{O}}$ is a finite sequence of tuples:*

$$w_{\mathcal{I},\mathcal{O}} = \langle t_0, \mu_0, \nu_0\rangle\langle t_1, \mu_1, \nu_1\rangle \cdots$$

such that
- *t_i follows the same conditions as in Definition 1,*
- *μ_i is a vector of values of input signals $\mathcal{I}$ at time t_i. That is, $\mu_i = \langle u_{0,i}, u_{1,i}, \cdots, u_{n,i}\rangle$ where $u_{k,i}$ is the sampled value of input signal I_k at time t_i,*
- *ν_i is a vector of values of output signals $\mathcal{O}$ at time t_i. That is, $\nu_i = \langle v_{0,i}, v_{1,i}, \cdots, v_{n,i}\rangle$ where $v_{k,i}$ is the sampled value of input signal O_k at time t_i.* $\square$

Finally, we introduce the definition of a segmented I/O trace.

DEFINITION 5 (SEGMENTED I/O TRACE). *A segmented I/O trace is a tuple $\psi = \langle w_{\mathcal{I},\mathcal{O}}, \Phi_{\mathcal{I}}, \Pi_{\mathcal{I}}, \Phi_{\mathcal{O}}, \Pi_{\mathcal{O}}\rangle$ where*
- *$w_{\mathcal{I},\mathcal{O}}$ is an input/output trace,*
- *$\Phi_{\mathcal{I}}$ is a vector of sets of features associated with each input signal:*

$$\Phi_{\mathcal{I}} = \langle \varphi_{I_1}, \varphi_{I_2}, \cdots\rangle$$

where φ_{I_i} is the set of features associated with input signal I_i.
- *$\Pi_{\mathcal{I}}$ is a vector of change-point vectors for each input signal:*

$$\Pi_{\mathcal{I}} = \langle \pi_{w_{I_1},\varphi_{I_1}}, \pi_{w_{I_2},\varphi_{I_2}}, \cdots\rangle$$

where $\pi_{w_{I_i},\varphi_{I_i}}$ is the change-point vector of input signal I_i given its associated feature set.
- *Similarly, $\Phi_{\mathcal{O}}$ and $\Pi_{\mathcal{O}}$ are defined for output signals.* $\square$

3.3 Formal Problem

Based on the definitions introduced in the previous sections, the formal problem definition is as follows:

Given two sets of segmented traces: a training set $\Psi_r = \{\psi_{r_1}, \psi_{r_2}, \cdots\}$, and test set $\Psi_t = \{\psi_{t_1}, \psi_{t_2}, \cdots\}$

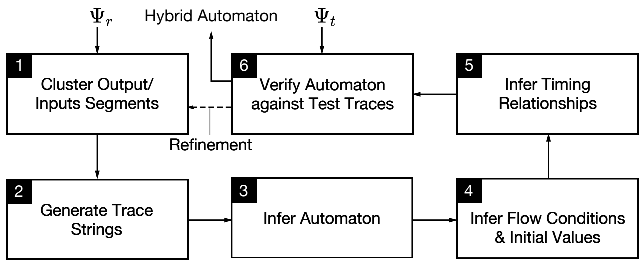
Construct a hybrid automaton A that models the behaviour of the system exhibited in the training set traces Ψ_r such that upon simulating the input test traces on A, the average error between the output test traces and the generated output from A is minimized:

$$e = \frac{1}{|\Psi_t|}\sum_{i=1}^{|\Psi_t|} \|A(\psi_{t_i}.w_{\mathcal{I}}) - \psi_{t_i}.w_{\mathcal{O}}\| \quad (2)$$

where $\psi_{t_i}.w_{\mathcal{I}}$ and $\psi_{t_i}.w_{\mathcal{O}}$ are the sets of input (output) traces in the segmented trace ψ_{t_i} (respectively), $A(\psi_{t_i}.w_{\mathcal{I}})$ is the output of automaton A given the input traces, and the subtraction operator calculates some predefined distance metric between two signals.

4 Proposed Framework

This section discusses a high-level overview of the proposed framework and details how different components can be replaced to allow more optimized system-centric modifications. Figure 3 presents the framework we propose to construct a hybrid automaton for a single output from segmented traces. We decompose a system with multiple outputs by building an automaton for each output signal. The process described in the figure is repeated for each output signal in the system, resulting in one hybrid automaton per output signal. The final automaton for the entire system is computed as the cross-product of all single output automatons.



Ψ_r
Hybrid Automaton
Ψ_t
1
Cluster Output/ Inputs Segments
6
Verify Automaton against Test Traces
5
Infer Timing Relationships
Refinement
2
Generate Trace Strings
3
Infer Automaton
4
Infer Flow Conditions & Initial Values

Figure 3: Overview of the proposed framework.

4.1 Clustering Segments

The main purpose of clustering segments is to identify which segments potentially represent the same internal state of a system. We have generalized clustering to be independent of what type of features are extracted from segments (see Definition 2), where it is stated that the similarity relation is assumed to be defined for the feature space. Upon clustering *similar* output segments, it is possible to hypothesize about the states of the hybrid system, since it is exhibiting similar behavior during clustered segments. Clustering of similar input segments enables us to reason about the triggers that can possibly cause transitions between output states. Note that what we can observe is correlation between an input segment and a transition to an output state. If this correlation is consistent across the training set traces, it is hypothesized as a possible cause in the output automaton.

Clustering is based on the feature vectors extracted by the features associated with signals. To this end, we compile a list of features that are indicative of state change we observe in our case studies (see Table 1). Obviously this list is incomplete, yet it should converge as more systems are studied.

Table 1: List of features and the returned feature vectors.

Feature	Feature Vector
Value	$\langle v\rangle$
Slope	$\langle m\rangle$
Range	$\langle a, b\rangle$
Linear Fit	$\langle m, b\rangle$
Higher-order Fit	$\langle c_0, c_1, \cdots\rangle$
Frequency	$\langle f, a\rangle$
Length (time)	$\langle t\rangle$

4.1.1 Input/Output Segment Clustering

Clustering groups segments in multiple traces of a signal according to similarity in the feature space associated with the signal. This enables our approach to overcome small variation between traces and identify common behavior of a specific signal across the entire training set. We use K-Means clustering [15] on the feature spaces associated with the signal, yet different clustering algorithms [19, 33] can be used within the structure of the workflow. We selected K-Means since it is widely available and simple to use. Since the optimum number of clusters is not known a priori, we use the Davies-Bouldin Index [11]. This approach is widely followed when the number of clusters is unknown. It is an iterative process of running the K-Means clustering algorithm with a preset number of clusters, calculating the Davies-Bouldin Index (DBI) for the output, incrementing the preset number of clusters then repeating the process. The final number of clusters is the number of clusters for which we observe the maximum DBI. The DBI is calculated as follows:

$$DBI = \frac{1}{N} \sum_{i=1}^{N} \max_{j,j \neq i} \frac{S_i + S_j}{d_{ij}} \tag{3}$$

where N is the number of clusters in the trial. S_i is the average distance between elements of cluster i and its centroid, and d_{ij} is the distance between the centroids of clusters i and j, calculated as follows:

$$S_i = \frac{1}{|C_i|} \sum_{x \in C_i} \|x - m_i\| \tag{4}$$

$$d_{ij} = \|m_i - m_j\| \tag{5}$$

where m_i is the centroid of cluster C_i and $|C_i|$ is the number of segments in the cluster.

To demonstrate how clustering works, consider the sampled trace in Figure 2, which is a trace of an output signal in our running example. The change-points in the figure indicate change in the slope feature ϕ_s, shown in Equation 1. Upon applying the clustering process described above to the feature vectors, namely the slopes of each segment, the result is four clusters based on the highest DBI. The cluster assignments are listed below:

$$\begin{aligned} C_1 &= \{w[0,15], w[22,39], w[55,69], w[79,95]\} \\ C_2 &= \{w[15,19], w[46,50], w[74,79]\} \\ C_3 &= \{w[19,22], w[50,55], w[69.74]\} \\ C_3 &= \{w[39,46]\} \end{aligned} \tag{6}$$

4.1.2 Input Event Detection

The purpose of input event detection is to identify significant changes in the input as potential causes for changes in the output. This is distinguishable from input segment clustering, which we perform to be able to infer whether a specific cluster of input segments is the cause for changes in the output. To clarify this, consider a discrete input signal that is set to value a for a period of time, and then changes to b. At that point of transition from a to b, the output experiences a change that we observe. The change could be attributed to

- The input becoming b, meaning that value b specifically causes the output to behave in the observed manner. That is, if the input was initially b, the output would have behaved in the observed manner from the start of the trace.
- The differential change in the input signal between a and b causes the output to behave in the observed manner. That is, $b - a$ is the reason for the output to change.

Input event detection clusters changes in feature vectors between consecutive segments instead of clustering feature vectors. For every change-point τ_k in trace w of a signal I, the change in the feature space between segment $w[k-1,k]$ and $w[k,k+1]$ is calculated as follows:

$$\delta_k = w[\tau_k, \tau_{k+1}] - w[\tau_{k-1}, \tau_k] \tag{7}$$

Similar to the clustering process in Subsection 4.1.1, clustering of input events is performed on the set of changes in feature vectors:

$$\Delta_{\psi_{\mathcal{I}}} = \{\delta_2, \delta_3, \cdots, \delta_{l-1}\} \tag{8}$$

where $\Delta_{\psi_{\mathcal{I}}}$ is the set of changes in feature vectors for signal $\mathcal{I}$ in the segmented trace ψ, and δ_k is the change in feature vectors around change-point τ_k.

To demonstrate how input event detection works, consider Figure 4, which shows a sampled trace w_A of input signal A in our running example. The samples with black circles on top are detected change-points. Assume the feature associated with this signal is ϕ_v which simply extracts the constant value of the segment. This is appropriate since this signal is a digital signal and changes are simply changes in value. The following are the resulting feature vectors for each segment:

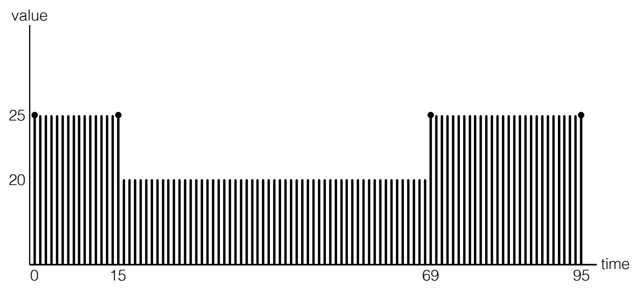


Figure 4: Sampled trace of an input signal.

$$\phi_v(w_A[0,15]) = \langle 25 \rangle$$
$$\phi_v(w_A[15,69]) = \langle 20 \rangle$$
$$\phi_v(w_A[69,95]) = \langle 25 \rangle$$

Clustering the input segments as per Subsection 4.1.1 results in two clusters: $C_1 = \{w_A[0,15], w_A[69,95]\}$ and $C_2 = \{w_A[15,69]\}$. However, the set of changes in value vectors (see Equation 8) is as follows:

$$\Delta_{w_A} = \{\langle -5 \rangle, \langle 5 \rangle\}$$

which now results in two clusters: $C'_1 = \{15\}$ and $C'_2 = \{69\}$. Thus, the next steps of the proposed algorithm will utilize multiple traces to reason about whether the change in the output at 15 and 69 (see Figure 2) is caused by C_1 and C_2 or C'_1 and C'_2.

4.2 Generating Trace Strings

The second step in the proposed framework (see Figure 3) is to generate trace strings. A trace string is defined as follows:

DEFINITION 6 (TRACE STRING). *Given a segmented trace w, its associated change-point vector π_w, and its cluster mapping function $\mathbb{C}$ which maps a segment to a cluster symbol. A trace string is a sequence of symbols that indicate the cluster to which each segment of a trace belongs. That is $s = \alpha_1\alpha_2 \cdots \alpha_i$ where s is a trace string and α_i is the symbol associated with the segment $w[\tau_i, \tau_{i+1}]$. That is $\alpha_i = \mathbb{C}(w[\tau_i, \tau_{i+1}])$.* □

As mentioned earlier, we focus on segmented and clustered traces for which there is one output and possibly multiple inputs. Consider the examples in Figures 5 and 6, where all signal traces are segmented and clustered. The engine speed signal is an output signal, while the throttle and torque are inputs. The engine speed signal is clustered as per Equation 6. The symbol assignment proceeds as follows:

1. We assign a unique symbol to each cluster in the output signal (a, b, c, d).
2. We assign a unique symbol to each cluster in the input signals (f, g, h, i).
3. We assign a unique symbol to each transition from one input segment to another ($f \to g = j$, $h \to i = k$, and $i \to h = l$).
4. We assign a unique symbol to each cluster of input events (see Subsection 4.1.2).
5. Finally, we assign a unique symbol to each unique combination of concurrently changing inputs or concurrent input events. For instance, if the throttle in Figure 5 transitions from f to g at the same time the torque transitions from h to i, this event is assigned a unique symbol. This scenario occurs at the initial point in the traces in Figures 5 and 6, and is assigned the symbols m for when f and h occur concurrently, and n for when f and i occur concurrently.

Note that step 3 and 4 cannot be used concurrently, in the sense that a transition in the input trace can either be assigned a symbol that represents a change from the previous input segment to the next input segment (step 3), or assigned a symbol that represents the cluster to which the change belongs (step 4). The symbol assignment we make in step 5 relies on whether we select the symbol from step 3 or step 4. Initially we select the choice that results in a lower number of symbols, however this is later modified in the feedback loop of our framework, where we refine specific states based on their error values. This is detailed in Subsection 4.6.1.

As mentioned earlier, we assume that the effect of a change in the input will appear instantaneously in the output trace. This can be achieved by assuming that the sampling period is larger than the propagation delay of the input signal through the system. This assumption simplifies the problem of determining causality, and results in perfectly synchronized traces as seen in Figures 5 and 6.

The result is an I/O trace string pair, which is defined as follows:

DEFINITION 7 (I/O TRACE STRING PAIR). *An I/O trace string pair is a pair of trace strings $s_I = \alpha_1\alpha_2 \cdots \alpha_i$ and $s_O = \beta_1\beta_2 \cdots \beta_i$ such that the segment α_i occurs concurrently with the occurrence of the input event combination β_i. For any β_i and β_{i+1}, if $\alpha_i = \alpha_{i+1}$ and $\alpha_i \neq \Delta$, $\alpha_{i+1} = \Delta$.* □

where Δ indicates that while there is detectable change in the output, there is no change in the input. This can be demonstrated by observing the trace in Figure 5, for which the I/O trace string pair is as follows:

Table 2: I/O Trace string for the trace in Figure 5.

s_O	a	b	c	a	d	b	c	a	c	b	a
s_I	m	k	Δ	Δ	j	Δ	Δ	Δ	l	Δ	Δ

In case of the trace in Figure 6, the I/O trace string pair is:

Table 3: I/O Trace string for the trace in Figure 6.

s_O	a	c	b	a	d	b	c	a	b	c	a
s_I	n	l	Δ	Δ	j	Δ	Δ	Δ	k	Δ	Δ

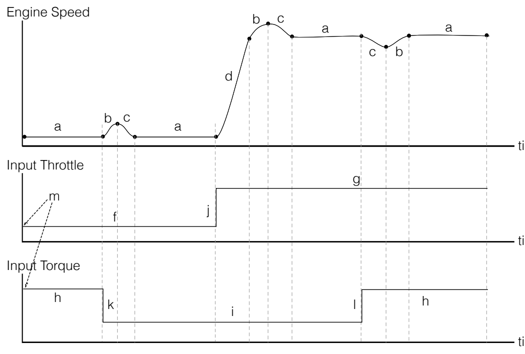


Figure 5: The first example of a segmented, clustered Input/Output trace in our running example.

4.3 Automaton Inference

The automaton inference step in our framework receives a set of input/output trace string pairs as shown in the previous section, and produces an automaton that models these traces. Since we are dealing with input/output traces, the produced automaton is a Mealy automaton. Mealy automata inference has been rigorously studied in various work [35,38]. The algorithms in most of the work on Mealy inference are based on Angluin's L^* algorithm [2]. Note that as mentioned earlier, our framework supports plug-and-play components to replace the inference step; as long as the component receives input/output trace string pairs and produces Mealy automata.

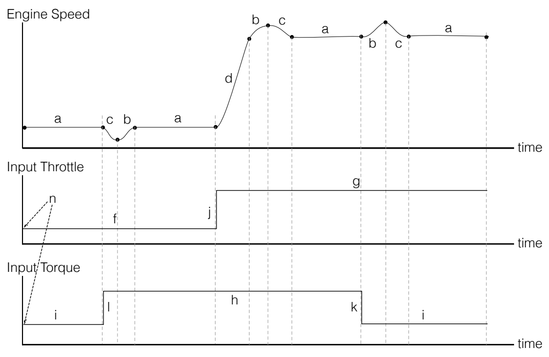


Figure 6: The second example of a segmented, clustered Input/Output trace in our running example.

4.3.1 *Modification of Mealy Inference*

Mealy inference algorithms based on L^* assume there is a system for which we want to produce a Mealy automaton. To produce such automaton, these algorithms rely on two oracles: a *membership oracle* and an *equivalence oracle*:

- **Membership Oracle.** This oracle receives an input string and returns the output string produced by the system.
- **Equivalence Oracle.** This oracle checks whether the automaton and the system are equivalent. If not, the oracle returns a counterexample in which the output of the system does not match the output of the automaton.

Unfortunately a membership oracle is often unavailable. This is specially true, if the traces are recorded live and not through simulation, requiring an engineer to rerun the system every time the Mealy inference algorithm requests a membership query. An equivalence oracle is even harder to achieve, since it requires a formal model of the original system, which is not available in the first place. Various work has tackled this problem using different approaches [17,37]. We propose a similar approach that fits within our framework: namely modified membership oracles and equivalence oracles.

- **Modified Membership Oracle.** This membership oracle searches through the training set of traces and finds the longest match. It returns the output corresponding to that match. If the input is longer than the longest match, the rest of the output is set to an error state. For example, consider the traces in Tables 2 and 3. Assume the input is `mk`, the output will be `ab` as per the trace in Table 2. Now assume the input is `mkl`, the output will be `ab`$\otimes$, indicating an error occurring on passing in the third input symbol (`m`).
- **Modified Equivalence Oracle.** This equivalence oracle simply searches for a counterexample using the training set traces. That is, it runs every input trace in the training set through the produced Mealy automaton, and checks whether the output matches the output in the training set. Granted, this does not prove formal equivalence, yet since our objective is to produce an abstract model of the system, strict equivalence is unnecessary.

We have used Learnlib [31] as the basis of our implementation. We have extended the built-in Mealy membership and equivalence oracles to rely on the training set of traces instead of an actual automaton.

4.3.2 *Automaton Post Processing*

After the automaton is inferred using LearnLib and our modified oracles, it is processed to ensure that no state has multiple incoming edges that produce different outputs. This is to prevent confusion when generating the flow condition for such state.

4.4 Infer Flow Conditions and Initial Values

Initial values are functions that define the initial value of a variable when the system enters a specific state in the hybrid automaton. Flow conditions are functions associated with a state in the hybrid automaton that describe how a variable changes while the system is in that state. The purpose of inferring initial values and flow conditions is to produce functions that given input values return accurate output values as per the training set. Since our framework produces one automaton per output signal, each state is associated with a function that returns the initial value of the output signal upon entering the state, and a function that describes the change of the output signal when the system resides in that state.

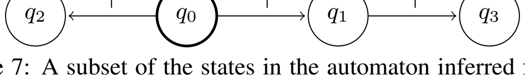
To this end, we propose a heuristic to infer flow conditions and initial values that is based on simple statistical methods such as standard correlation and linear regression. As mentioned earlier, the plug-and-play design of the framework allows for replacing the proposed heuristic with different methods in the future. An overview of the heuristic is presented in Algorithm 1. The heuristic receives as input the following:

- Automaton A, which is the automaton produced by the previous step in the framework (see Figure 3).
- Training set of traces Ψ_r,
- Cluster mappings $\mathbb{C}$ which maps each segment in the trace to a symbol. This mapping should be readily available after trace strings are generated (see Figure 3).

The heuristic proceeds as follows:

- Line 2 declares a map between states in the automaton and traces.
- The loop at Line 3 iterates over the training set, and foreach segmented trace, calls `SPLIT` which breaks down the traces into input/output segments defined by the change-points in the trace. `SPLIT` then returns a vector of all the segments in the trace.
- The automaton is reset to its initial state, and then the loop at Line 6 iterates over each input/output segment in the trace.
- For each segment, the symbol of the associated input cluster is retrieved (Line 7). The automaton then advances to the next state given the input symbol.
- The segment is then associated to the current state of the automaton (Line 9).
- Next, for each state in the automaton, a standard correlation matrix is built using a normalized version of all the segments associated to that state. The matrix expresses the correlation between the output signal and the input signals. Note that time is also considered an input. It is readily available as the timestamp of each sample in the segment.
- The matrix is then filtered for significant relationships only (relationships for which there is strong positive or negative correlation between the output and input). The filter threshold can be configured, for instance to allow only relationships with correlation coefficient $|c| > 0.5$.
- A linear regression is then constructed for the output versus significant inputs, and the resulting coefficients are assigned to variable c.
- The initial condition assigned to the state is a linear function in the input variables and the coefficients c in the form $c_1x_1 + c_2x_2 \ldots$ (Line 16).
- Finally, the flow condition is set to the derivative of the above linear function with respect to time.

Consider the automaton in Figure 7, which a subset of the automaton we achieve through learning from traces of our running example. As can be seen, the input symbols m and n direct control to states q_1 and q_2 respectively, while both producing the output a. This directly maps to the two traces in Figures 5 and 6.



$$\begin{tikzcd} q_2 & q_0 \arrow[l, "n \mid a"'] \arrow[r, "m \mid a"] & q_1 \arrow[r, "k \mid b"] & q_3 \end{tikzcd}$$

Figure 7: A subset of the states in the automaton inferred for our running example. q_0 is the initial state.

Now let us focus on state q_1. Upon executing the first for loop in

Algorithm 1 Infer Flow Conditions.

```
 1: INPUT: Automaton $A$, Training set traces $\Psi_r$, Cluster mappings $\mathbb{C}$
 2: declare traces mapping $\mathbb{T}$
 3: for $\psi \in \Psi_r$ do
 4:     $\mathcal{W} \leftarrow$ SPLIT($\psi.w_{\mathcal{I},\mathcal{O}}$,$\psi.\Pi_{\mathcal{I}}$,$\psi.\Pi_{\mathcal{O}}$)
 5:     RESET($A$)
 6:     for $\omega \in \mathcal{W}$ do
 7:         $s \leftarrow \mathbb{C}[\omega_{\mathcal{I}}]$
 8:         $v \leftarrow$ PROCESSSTRING($A$,$s$)
 9:         APPEND($\mathbb{T}[v]$,$\omega$)
10:     end for
11: end for
12: for State $v \in A$ do
13:     $r \leftarrow$ CORRELATION($\mathbb{T}[v]$)
14:     $I \leftarrow$ FILTER($R$)
15:     $c \leftarrow$ REGRESSION($I$)
16:     $v$.init $\leftarrow c$
17:     $v$.flow $\leftarrow \frac{dc}{dt}$
18: end for
```

Algorithm 1, the state will be associated with all segments where initially the input throttle is low and the torque is high, which constitutes input event m (refer to Figure 5). To clarify this, Figure 8 demonstrates seven instances of the output signal taken from seven different traces while the system is in state q_1. We concatenate these seven instances to form one vector for the purpose of studying correlation.

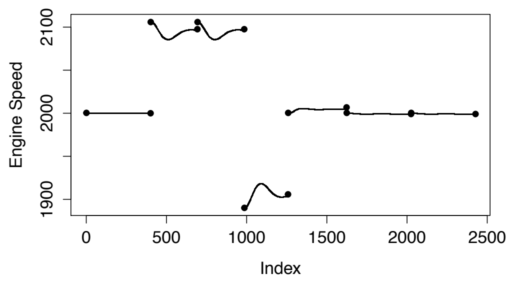


Figure 8: Concatenated output traces for segment a. The x-axis represents the index of the concatenated data frame.

Next, we study the correlation of the output signal with the inputs: namely time, input throttle and input torque. The following are the resulting correlation coefficients:

Time	Input Torque	Input Throttle
0.079	0.014	$0.997 \approx 1.0$

Time and input torque are insignificant with respect to the output, as observed by their low correlation coefficient. Input throttle however is strongly correlated with the output. This results in $y = x_1$ where y is the output and x_1 is the input throttle. Since time is insignificant to the output at state q_1, the flow condition of q_1 is $\dot{y} = \frac{dx_1}{dt} = 0$, resulting in no change in output since the input when the system is in that state does not change with time. The updated automaton is shown in Figure 9.

To further clarify flow conditions, let q_3 be the state that represents the system when segment b first occurs after segment a (see Figure 5). Upon studying the correlation of the output in that state against the inputs, it is obvious that time is contributing factor, since the output exhibits a steady rise as time passes. In that case, the derivative of the initial condition (based on the regression coefficients) will be as follows:

$$\dot{y} = c_t + c_1\frac{dx_1}{dt} + c_2\frac{dx_2}{dt}$$

where c_t is the regression coefficient of the input time variable.

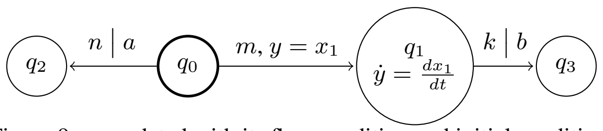


Figure 9: q_1 updated with its flow condition and initial condition.

4.5 Inferring Timing Relationships

A time-based condition (*time guard*) is added to a transition in a hybrid automaton to illustrate that the transition may be taken when the condition is satisfied. Recall that in our I/O trace strings we used the notation Δ to signify that the output changes while no change occurs in the input. Naturally this can be caused due to a change in a hidden internal state of the system, or due to some time limit elapsing. Since we cannot postulate about a hidden unobservable system state, we attempt to model all such transitions as time-based transitions.

The amount of time spent in a specific output segment is measurable through trace timestamps. By analyzing multiple traces we can attempt to infer an average time which must pass before the transition occurs, or try to deduce a relationship between the previous change in the input and timed change in the output. Consider the following subset of the automaton inferred from our running example.

$$\begin{tikzcd} q_1 \arrow[r, "k \mid b"] & q_3 \arrow[r, "\Delta \mid c"] & q_4 \arrow[r, "\Delta \mid a"] & q_5 \end{tikzcd}$$

The states in this example map to states observed in the trace in Figure 5. The two timed transitions are $q_3 \to q_4$ and $q_4 \to q_5$. We introduce the notion of a *closest input events*, which are the latest input events to occur before a time transition is taken. In the case of both transitions in the above automaton, the closest input event is k.

We now follow a process similar to the inference of flow conditions and initial values in Subsection 4.4. To that end, we distinguish between two different scenarios:

1. The time between the two states surrounding a timed transition across all traces is normally distributed. In such a case, we assign the average time to the transition time guard. Note that other statistical techniques might yield more accurate results, such as survival analysis. However, a normality test is sufficient for an initial heuristic and can be replaced later within the framework.
2. The time between two states surrounding a timed transition has a high correlation to one or more inputs at the closest input event, and we hence build a linear regression and assign the expression $c_1x_1 + c_2x_2 \ldots$ to the transition guard, where c_i are regression coefficients.

To test for normality, we use D'Agostino and Pearson's tests of normality that combine skewness and kurtosis [6,9].

To clarify how we infer timing relationships, let us apply the proposed technique to our running example. Assume we have collected the time it takes the system to transition from q_3 to q_4 across eight traces. If the Δ dataset passes the normality test, then the transition is assigned the average time as a time-guard.

By assigning time guards to the appropriate transitions, we complete the final step in preparing the hybrid automaton. The automaton can now be verified against test traces to quantify its accuracy, as detailed in the next section.

4.6 Verify Automaton Against Test Traces

The final step in our framework is to verify the produced hybrid automaton. Our evaluation technique proceeds as follows:

1. Given a segmented input/output trace ψ, for each input assign each segment to a cluster from the set of precomputed clusters for that input. This is done using a K-Means predictor.
2. Generate an input trace string using the same methodology as that used in Subsection 4.2.
3. Run the input string and the accompanying input traces through the hybrid automaton and record the output.
4. Compare the output of the automaton to the original output in the segmented input/output trace and return the total distance. The total distance is calculated as follows:

$$\gamma_{A,\psi} = \frac{1}{L} \sum_{i=1}^{L} |w_A[i] - \psi.w_{\mathcal{O}}[i]| \tag{9}$$

where w_A is the sampled output from the automaton, $\psi.w_{\mathcal{O}}$ is the original output in the input/output trace, and $L = \max\{|w_A|, |\psi.w_{\mathcal{O}}|\}$. If $w[i]$ does not exist for either trace, it is assumed to be zero.

The final error metric for the entire test set is the average γ for each individual trace in the test set:

$$e = \frac{1}{\Psi_t} \sum_{i}^{|\Psi_t|} \gamma_{A,\psi_i} \tag{10}$$

which is the error metric explained in Eq. 2. The objective of the framework is to reduce the error through multiple iterations of automaton refinement. This is further explained in Subsection 4.6.1.

4.6.1 Automaton Refinement

Automaton refinement utilizes data collected from the test runs to improve the accuracy of the automaton. Since test input traces are run through the automaton, and the output of the automaton is compared to the output traces point by point, we can breakdown the error to every state and transition in the automaton. We plan to elaborate on refinement in our future work (Section 8).

5 Case Studies

This section presents the case studies we use to evaluate the applicability of our framework to different systems. Obviously these case studies are not representative of the entire spectrum of hybrid systems, yet this paper proposes a framework that can be further enhanced in the future to support different types of systems and evolve into a robust abstraction engine.

The case studies are based on systems for which there already exists a Simulink model. Our framework is designed to support any input/output traces, whether their origin is simulation or live execution. However, having the models helped us in producing noiseless input/output traces which is beyond the scope of the current work. Moreover, the Simulink models helped us in validating the proposed approach by generating test traces.

We implemented the framework using a mixture of Python, Java, and R with various libraries associated with it. R was used for segmentation, Java was used for automata inference based on LearnLib, and Python was used for clustering, flow condition inference and timing condition inference.

5.1 Engine Timing Model

This case study is based on the engine timing model in the Simulink toolbox examples (www.mathworks.com), which is also the basis for our running example in Figure 1. The system has two inputs: (1) input throttle, which is the desired speed of the engine in rounds per minute (RPM), and (2) load torque in joules per radian. The output of the system is the engine speed in RPM.

To test our approach, we collect eight traces of the model in Simulink, which cover permutations of inputs as well as basic states. We then segment them using the `changepoint` library in R [22]. At this point, we have the training set Ψ_r. Next, we feed them through the framework outlined in Figure 3. The final output of the framework is a hybrid automaton that models the behavior of the system as observed in the training traces. Figure 10 shows the produced hybrid automaton.

To verify the output automaton, we use a test set of traces to ensure that the automaton can produce correct output to traces other than its training traces. The test traces have different values for inputs and transition at different times. To that end, we run a set of these

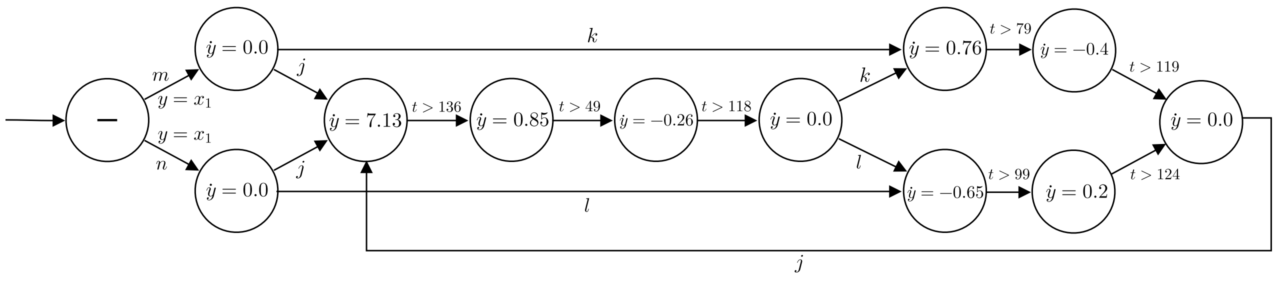


Figure 10: Hybrid automaton for the engine timing control model.

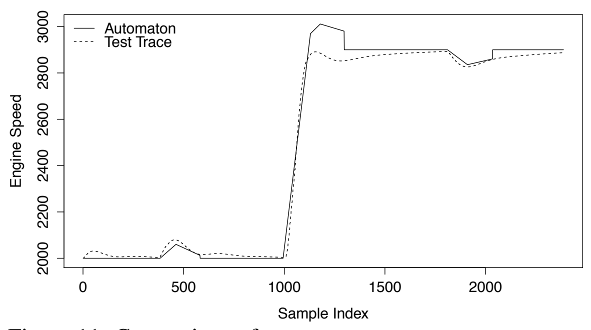


Figure 11: Comparison of automaton output versus test output.

test input traces through the automaton and compare the outputs to the respective output test traces. Figure 11 shows the test output versus the automaton output. The dashed line shows the output in the test trace recorded from Simulink, and the solid line shows the output of the hybrid automaton. As can be seen in the figure, the automaton output closely follows the test trace output, except when the input throttle increases, where the automaton overshoots beyond the actual system. This is caused by inaccurate assessment of the slope of the overshoot, which is in fact a function of the change in the throttle input. We plan to rectify this in future work. Based on our experiments, increasing the number of traces that cover the same use case improves the fidelity of the model. Using less than eight traces in this case study results in an inaccurate model, simply because the traces do not cover the use cases shown in the automaton in Figure 1.

The error γ, which is the average distance between the automaton trace and the test trace in Figure 11, is 26.37 RPM per sample. Knowing that the range of the output in the trace is between 2000 RPM and 3000 RPM, this error amounts to 2.6%.

5.2 Fault-Tolerant Fuel Control System

The second case study we conduct is based on the Simulink fault-tolerant fuel control system (www.mathworks.com). The model contains both Simulink and StateFlow components. The controller should regulate the fuel rate without interruption in case of individual sensor failure. Four sensors are connected to the controller: throttle, speed, EGO, and MAP. Failure in at most one sensor should not interrupt fuel rate control.

The training set traces record the output fuel rate control signal in different combinations of sensor failures. Each input/output trace consists of four inputs (one for each sensor) and one output. Using Simulink, any sensor can be toggled to simulate failure. The model simulates a failed sensor by producing readings that are outside the valid range for that sensor.

Although we require presegmented traces, the traces for this case study were easy to segment automatically. We applied k-means clustering to the values of the input traces (which are sensor readings), and we were able to identify the periods in the traces where a specific sensor is turned off. For instance, as per the Simulink model, the speed sensor normally provides readings between 300 and 700, whereas when it is off it reads 0 (see Figure 12). Similarly for the remaining three sensors, faulty readings are easily identifiable. The output trace is segmented according to the input traces, meaning that the change-points in the output trace are the union of all change-points in the input trace. Figure 13 shows a concatenated output trace of the system when different sensors fail, where change-points are black circles.

Upon applying our framework, we first cluster the output segments into four clusters based on the range feature (see Table 1). Each cluster is assigned a symbol from a to d. For example, the trace in Figure 13 is translated to the output trace string $abacabababdba$. The input traces in Figure 12 are clustered using the same technique used for our running example. Throttle is assigned the symbols f and g for when it is on (value above 10) and off (value is zero) respectively. Similarly, Speed is assigned h and i, EGO is assigned j and k, and MAP is assigned l and m. Note that although we refer to a sensor as on or off, the framework is unaware of that notion and only sees two clusters per input signal that are assigned two unique symbols. Since it never occurs in the training set traces that two sensors fail at the same time, there are no symbols assigned to two or more sensors failing concurrently. However, initially all sensors are on in all traces, and thus this is assigned the symbol n.

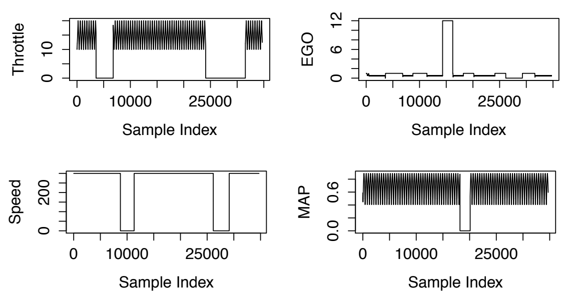


Figure 12: The four input traces of the fuel rate controller.

Using 17 training traces, the inferred hybrid automaton for the system is shown in Figure 14. Note that we assume once a sensor is faulty it cannot recover. The input events g, i, k, and m indicate failures in the throttle, speed, EGO, and MAP sensors respectively. A sample run of the hybrid automaton against a test trace (a trace not in the training set) is plotted in Figure 15. While the automaton mimics the oscillations of the original output, it fails to replicate the amplitude of these oscillations. This can be improved by using other regression models, namely for this case study a sinosoidal regression model would produce more accurate results. Since our framework allows for plug-and-play components, one can replace linear regression with sinosoidal regression to reduce error. Even with the use of simple linear regression models, the error percentage is 3.7%.

6 Related Work

Many researchers tackled the problem of specification mining of software systems [26], the work in [5, 10, 25, 36] being the prominent advances in the field. But none of these works deal with the problem of mining specifications for control systems, where analog and digital

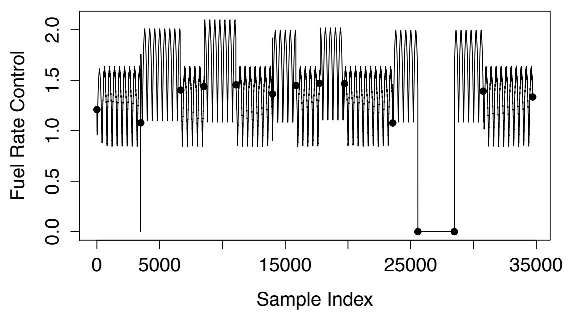
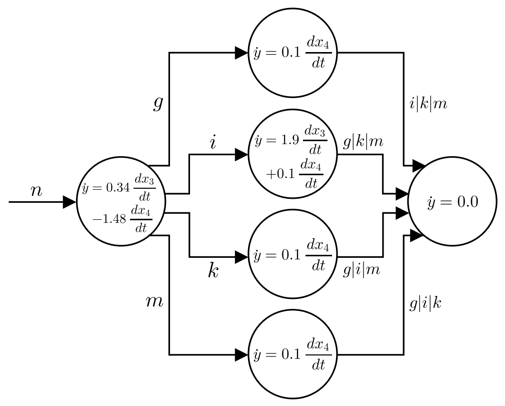


Figure 13: Fuel rate control signal trace with change-points.



n
$\dot{y} = 0.34\frac{dx_3}{dt} - 1.48\frac{dx_4}{dt}$
g
i
k
m
$\dot{y} = 0.1\frac{dx_4}{dt}$
$\dot{y} = 1.9\frac{dx_3}{dt} + 0.1\frac{dx_4}{dt}$
$\dot{y} = 0.1\frac{dx_4}{dt}$
$\dot{y} = 0.1\frac{dx_4}{dt}$
$i|k|m$
$g|k|m$
$g|i|m$
$g|i|k$
$\dot{y} = 0.0$

Figure 14: The inferred hybrid automaton of the fuel rate controller.

signals coexist. There are some works on generating API specifications [1, 39], yet control systems offer a new challenge due to the need to mine their input output relationship.

The work on invariant generation from traces has yielded favourable results and has proven to be versatile in mining basic invariants on variables in traces [8, 13]. Our work is capable of breaking down traces and constructing an automaton in which different states can hold different invariants. Thus, an invariant generation module can be a pluggable component within our framework to produce a more accurate hybrid automaton. Invariant generation techniques can also be used to infer transition conditions along with input event detection.

The work on behavioral model inference is close to the work in this paper since it targets inference of behavior from execution traces [14, 24, 27, 29]. The objective is to produce a model that represents the system behavior. However, these works target discrete system behavior and can not be used for inferring hybrid models.

The work on system identification is well established and various robust tools exist to aid in the process [4, 20, 21, 30]. System identification approaches produce complete models which would be large and complex. In contrast, our objective is to to focus on certain aspects of the system and generate small and comprehensible models, thus aiding in the comprehension of such system, its integration with other components, and its simulation. Further, our framework is general enough to accommodate system identification techniques and combine them with other techniques of specification mining to infer abstract accessible hybrid automata.

7 Discussion

Mining hybrid automata from traces is a complex problem with numerous challenges. Here we identify the limitations of our framework: (1) Although we were successful with automatically segmenting the signals in our case studies, the methods used may not be applicable to all types of signals. The signals collected from live systems pose some challenges. They are noisy and need to be filtered and processed before being fed to our framework. (2) Any hidden state that cannot be observed in the traces cannot be modelled. The modelling technique we use relies on *correlating* inputs and outputs as observed

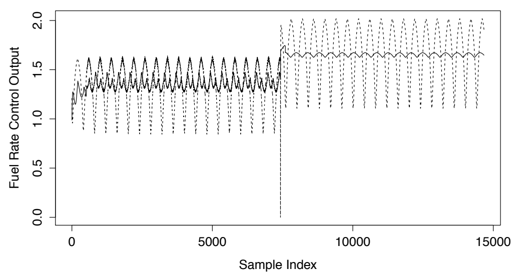


Figure 15: A sample of the hybrid automaton output (solid) versus the original system output (dashed).

in the training set traces. If certain behavior is triggered by an internal state that is not observable through inputs and outputs, the behavior will not be modelled, or the framework will model it with faulty assumptions. (3) Some transition guards cannot be modelled such as guards on output values, which will be addressed in the future. (4) We assume that the propagation delay of a change in the input to an observable change in the output is instantaneous. This allows consistent segmentation and synchronized trace strings. In future work, this assumption needs to be relaxed to infer time-delayed responses.

8 Conclusion

This paper has presented a general framework for inference of hybrid automata models from black box system implementations by mining their input/output traces. The framework outlines an iterative process for the inference and refinement of hybrid automata. We presented specific techniques in our case studies for the purpose of demonstration but the framework is general enough to allow replacement/improvements of these techniques. We introduced two case studies that demonstrate the applicability of the approach (and also some limitations). The results are highly encouraging and we believe would spur more work on hybrid automata inference.

The framework allows for multiple directions of improvement from within. As future work, we plan to explore enhancements such as better preprocessing of data, improved segmentation, more efficient algorithms, support for more causality rules, and improved automata inference. In parallel with these enhancements, we plan to elaborate on automata refinement by experimenting with case studies where automata refinement is necessary. We plan to explore various feedback techniques to reduce the error of the produced automaton iteratively and apply these techniques to case studies. Finally, we plan to explore modifying the framework to produce constrained hybrid automata on which we can verify properties formally.

9 Acknowledgements

This research was sponsored as part of the NECSIS project by the Ontario Research Fund, the Natural Sciences and Engineering Research Council of Canada, and associated industry partners. The authors would like to thank NECSIS and General Motors R&D for their generous support, without which this work would not have taken shape. The authors would also like to thank Oleg Sokolsky and the anonymous reviewers for their valuable feedback.

10 References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 25–34. ACM, 2007.

[2] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[3] I. E. Auger and C. E. Lawrence. Algorithms for the Optimal Identification of Segment Neighborhoods. *Bulletin of*

Mathematical Biology, 51(1):39–54, 1989.

[4] 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.

[5] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 267–277. ACM, 2011.

[6] K. Bowman and L. Shenton. Omnibus Test Contours for Departures from Normality Based on b1 and b2. *Biometrika*, 62(2):243–250, 1975.

[7] J. Chen and A. K. Gupta. *Parametric Statistical Change Point Analysis: With Applications to Genetics, Medicine, and Finance*. Springer Science & Business Media, 2011.

[8] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Recent Advances in Intrusion Detection*, pages 63–86. Springer, 2007.

[9] R. B. d'Agostino. An Omnibus Test of Normality for Moderate and Large Size Samples. *Biometrika*, 58(2):341–348, 1971.

[10] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating Test Cases for Specification Mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 85–96. ACM, 2010.

[11] D. L. Davies and D. W. Bouldin. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, April 1979.

[12] A. Donzé, T. Ferrère, and O. Maler. Efficient Robust Monitoring for STL. In *Proceedings of the 25th International Conference on Computer Aided Verification*, pages 264–279, Berlin, Heidelberg, 2013. Springer-Verlag.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[14] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining Behavior Models from User-intensive Web Applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 277–287. ACM, 2014.

[15] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):pp. 100–108, 1979.

[16] T. A. Henzinger. The Theory of Hybrid Automata. In *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series*, pages 265–292. Springer, 2000.

[17] H. Hungar, O. Niese, and B. Steffen. Domain-specific Optimization in Automata Learning. In *Computer Aided Verification*, pages 315–327. Springer, 2003.

[18] X. Jin, A. Donze, J. V. Deshmukh, and S. A. Seshia. Mining Requirements from Closed-Loop Control Models. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, pages 43–52. ACM, 2013.

[19] S. C. Johnson. Hierarchical Clustering Schemes. *Psychometrika*, 32(3):241–254, 1967.

[20] A. L. Juloski, W. Heemels, G. Ferrari-Trecate, R. Vidal, S. Paoletti, and J. Niessen. Comparison of Four Procedures for the Identification of Hybrid Systems. In *Hybrid Systems: Computation and Control*, pages 354–369. Springer, 2005.

[21] A. L. Juloski, S. Weiland, and W. Heemels. A Bayesian Approach to Identification of Hybrid Systems. *IEEE Transactions on Automatic Control*, 50(10):1520–1533, 2005.

[22] R. Killick and I. Eckley. changepoint: An R Package for Changepoint Analysis. *Journal of Statistical Software*, 58(3):1–19, 2014.

[23] Z. Kong, A. Jones, A. Medina Ayala, E. Aydin Gol, and C. Belta. Temporal Logic Inference for Classification and Prediction from Data. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, pages 273–282. ACM, 2014.

[24] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using Dynamic Execution Traces and Program Invariants to Enhance Behavioral Model Inference. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, pages 179–182. ACM, 2010.

[25] W. Li, A. Forin, and S. A. Seshia. Scalable Specification Mining for Verification and Diagnosis. In *Proceedings of the 47th Design Automation Conference*, pages 755–760, 2010.

[26] D. Lo, S.-C. Khoo, J. Han, and C. Liu. *Mining Software Specifications: Methodologies and Applications*. CRC Press, 2011.

[27] D. Lo, L. Mariani, and M. Pezzè. Automatic Steering of Behavioral Model Inference. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 345–354. ACM, 2009.

[28] P. Mitra, C. Murthy, and S. Pal. Unsupervised Feature Selection Using Feature Similarity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(3):301–312, Mar 2002.

[29] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral Resource-aware Model Inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 19–30. ACM, 2014.

[30] S. Paoletti, A. L. Juloski, G. Ferrari-Trecate, and R. Vidal. Identification of Hybrid Systems — a Tutorial. *European Journal of Control*, 13(2):242–260, 2007.

[31] H. Raffelt, B. Steffen, and T. Berg. Learnlib: A Library for Automata Learning and Experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, pages 62–71, 2005.

[32] T. Räsänen and M. Kolehmainen. Feature-Based Clustering for Electricity Use Time Series Data. In M. Kolehmainen, P. Toivanen, and B. Beliczynski, editors, *Adaptive and Natural Computing Algorithms*, volume 5495 of *Lecture Notes in Computer Science*, pages 401–412. Springer, 2009.

[33] J. Sander, M. Ester, H.-P. Kriegel, and X. Xu. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. *Data Mining and Knowledge Discovery*, 2(2):169–194, June 1998.

[34] A. Scott and M. Knott. A Cluster Analysis Method for Grouping Means in the Analysis of Variance. *Biometrics*, pages 507–512, 1974.

[35] M. Shahbaz and R. Groz. Inferring Mealy Machines. In *FM 2009: Formal Methods*, pages 207–222. Springer, 2009.

[36] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static Specification Mining Using Automata-based Abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

[37] B. Steffen and H. Hungar. Behavior-based Model Construction. In *Verification, Model Checking, and Abstract Interpretation*, pages 5–19. Springer, 2003.

[38] G. van Heerdt. Efficient Inference of Mealy Machines. Bachelor thesis, Radboud University Nijmegen, June 2014.

[39] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of the 28th International Conference on Software Engineering*, pages 282–291, 2006.