PRACTICAL IDENTIFICATION OF
TIMED EVENT SYSTEMS

by

Donald E. Jarvis II
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Electrical and Computer Engineering

Committee:

_____	Andrzej Z. Manitius, Dissertation Director
_____	Chun-Hung Chen, Committee Member
_____	Gerald Cook, Committee Member
_____	Brian L. Mark, Committee Member
_____	Abbas K. Zaidi, Committee Member
_____	Andrzej Z. Manitius, Department Chair
_____	Lloyd J. Griffiths, Dean, The Volgenau School of Engineering

Date: April 25, 2011

Spring Semester 2011
George Mason University
Fairfax, VA

Practical Identification of Timed Event Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Donald E. Jarvis II
Master of Science in Engineering
Catholic University of America, 1999
Bachelor of Science in Engineering
Bachelor of Philosophy
Catholic University of America, 1994

Director: Andrzej Z. Manitius, Professor
Department of Electrical and Computer Engineering

Spring Semester 2011
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to the three generations who have patiently supported me through the process of my education: to Mom and Dad, to Gin, Tres and Ruby, and most of all, to Rose.

Acknowledgments

I am extremely grateful to my director Andrzej Z. Manitius for his wisdom and encouragement throughout the course of this research, and to my committee members Chun-Hung Chen, Gerald Cook, Brian Mark, and Abbas Zaidi for generously sharing their time and insights.

Allen Goldberg, Bob Gover, Jeff Heyer, Eric Justh, Jerry Gansman, Brook Susman, Vince Tumminello, and Amen Zwa have aided me through technical discussions, advice, and encouragement over many years.

The interest and guidance of Steven Chin, Sherif El-Helaly, David Gallagher, Richard Hassing, Gustav Hensel, John McCarthy, John McCoy, George McDuffie, Ingrid Merkel, Nader Namazi, Charles Nguyen, Fr. Kurt Pritzl, and Larry Schuette has made an immeasurable difference, as has the intercession of friends too numerous to mention.

Finally, I am quite grateful for the material support of the Edison Memorial Graduate Training Program, U. S. Naval Research Laboratory.

Table of Contents

List of Figures

Abstract

PRACTICAL IDENTIFICATION OF TIMED EVENT SYSTEMS

Donald E. Jarvis II, PhD

George Mason University, 2011

Dissertation Director: Andrzej Z. Manitius

Discrete event systems (DES's) are well established models in what can now be considered their traditional applications such as manufacturing systems, air traffic and other transportation systems, and computer networks [1, 2].

The problem of discrete controller design is of increasing significance. Embedded systems, for example, commonly have both "high level" discrete digital controllers and "low level" analog systems. Other examples include monitoring and control systems for automobiles and large buildings, pacemakers, and autonomous vehicles [1, 3, 4].

Control engineering has traditionally emphasized analog systems, but is increasingly treating the discrete part as a genuine control problem and not simply an exercise in microcontroller programming. For example, the Supervisory Control Theory of DES's [5] is a continuing research area, as is the general problem of hybrid control [6], in which the discrete and analog parts must be designed jointly.

Analog control theory includes system identification as an important topic. The identification of DES in the form of finite automata also has an extensive literature. However, on the problem of timed discrete event system identification, the literature is almost silent.

The models yielded by identification are important for control design and also in their own right. Such a model can be used to analyze or simulate the behavior of a system in some context, such as its interaction with a proposed controller design, or as a component in a system-of-systems.

DES models can be arrived at by analysis (i.e., in terms of internal components), or simply by assuming a certain structural form. However, when internals are inaccessible or when analysis is relatively difficult or expensive (e.g., in the case of a legacy system that was left undocumented, or the reverse-engineering of a competitor's product), model identification could be useful.

This dissertation describes the investigation and evaluation of a novel approach to identification of timed discrete event systems. The main results are a discrete-event systems formalism amenable to identification, and the identification algorithm itself. Simulation results are given that illustrate performance under both best-case and worst-case identification conditions.

Chapter 1: Introduction

In the course of engineering practice, the problem arises of determining how to build a system (or a mathematical model of a system) with the same behavior as a given, real-world system, with the caveat that the real-world system cannot be taken apart to see how it works. The only knowledge allowed of the real-world system are records of its outputs when it is subjected to certain inputs. This is known as the system identification problem.

Many uses for identification have been conceived. The archetypical application is as a prelude to controller design, since controller design techniques assume the existence of a mathematical model of the system to be controlled. In the engineering context these mathematical models typically take the form of differential equations or difference equations, either linear or nonlinear. A mature body of techniques exists to address these cases [7, 8].

System identification becomes no less important when we turn our consideration to systems not well modeled in terms of differential or difference equations. In particular, we are interested here in variants of the "other" great dynamical systems model amenable to analysis, one complementary to the preeminent engineering model of the linear system, namely, finite state machines. We are especially concerned with finite state machine-like models that are extended to include a notion of real time. In this context, the input and output symbols of the FSM are called "events." The relationship between event-based systems and FSMs is an important theme of this research.

It is my thesis that existing efficient algorithms designed for inference of regular languages can, under appropriate circumstances, be adapted to identification of timed event systems. In this work I will present such an adaptation and analyze its applicability to problems of practical interest.

1.1 Motivation

Theoretical interest in identification of machines goes back to the beginnings of automata theory. This quotation is from the seminal 1956 paper introducing the Moore model of the finite state machine.

> "It may be instructive to consider several situations for which this sort of theory might serve as a mathematical model. The first example is one in which one or more copies of some secret device are captured or stolen from an enemy in wartime. The experimenter's job is to determine in detail what the device does and how it works." [9]

While reverse-engineering the artifacts of a military or industrial competitor are conceivable applications, many uses are less dramatic than the one Moore imagines. A surprisingly common technology problem in government and industry is to create and own a product, yet not know how it works. Such a situation can arise through poor planning or mismanagement, such as contractor requirements that fail to include system models in the deliverable, or the emergence of new kinds of questions that were not conceived of when a system was designed and documented, such as the formal verification of a legacy communication protocol. A high profile example of the former pattern is the infamous Fogbank case [10, 11]. While not involving the specific technology of system identification, the human and managerial errors made in that program can arise in any technological endeavor, making it a cautionary tale of broad relevance.

Though Moore was vague on what kind of "secret device" would be well-modeled as an automaton, contemporary technology contains many examples of finite state machine-based systems, such as embedded systems, which often use an FSM model as a basis for design [12]. Additionally, since it is the nature of embedded systems to interact with a real-world environment, this FSM basis is typically extended in one way or another to account for the flow of time.

It is observed in [13] that the typical trend of controller software is to become more complex, harder to understand, harder to predict the impact of changes, and more expensive to maintain:

> "Complex embedded software systems, such as industrial control systems or automotive systems, often consists of several million lines of code and are maintained over many years, sometimes decades, during which the software is exposed for changes continuously."

As shown in [14], legacy software, which is dominant, is invariably unmodeled; and new software often goes unmodeled too, since building and maintaining models is time-consuming and expensive. The authors state that

> "The lack of adequate, up-to-date models indeed causes misunderstandings, leads to unexpected side-effects in new releases and is the reason for many unsatisfied customers ...
>
> Modern telecommunication and IP-based applications are multi-tiered, distributed applications that typically run on heterogeneous platforms. Their correct operation depends increasingly on the interoperability of single software modules, rather than on their intrinsic algorithmic correctness."

And finally, it was incisively noted in [15] that

> "There also exist techniques for generating a model for finite-state systems by observing execution traces based on a machine learning algorithm first proposed by Angluin... However, techniques are not well-defined for real-time systems..."

This anecdotal evidence makes the case that there are potential applications for an identification technique for timed event systems.

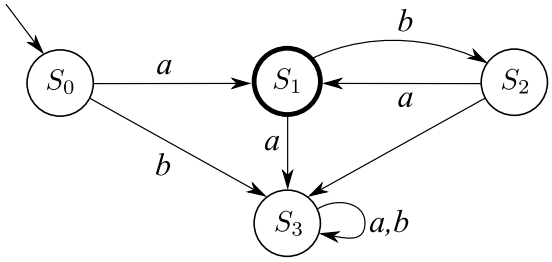

Figure 1.1: An automaton that determines a regular language L_1.

1.2 Background

This background information may serve as a brief outline of some of the concepts and techniques to be used in the main body of this work.

1.2.1 The language learning problem

Here we introduce regular languages, and some of the difficulties involved in the problem of inferring a regular language from instances of the language.

We recall some relevant definitions. An alphabet is a set of atomic symbols, for example $A = \{a, b, c, d\}$. The Kleene closure of A, denoted A^*, is the set of all possible sequences of the elements of A; for example, A^* would include the sequences a, $abbbc$, $dcba$, and a countable infinity of others, along with the sequence of length zero. A language L is a subset of such a closure, $L \subset A^*$. Often languages are described with a rule for deciding whether an element of A^* is an element of L. For example, let language L_1 in the alphabet $\{a, b\}$ be the set of all sequences alternating in a and b, that both start and end with a. The sequence $ababa$ is in L_1; the sequences bbb and $abab$ are not.

An automaton is a structure customarily depicted as in Figure 1.1. This automaton has four states, labeled S_0 to S_3, and transition arrows labeled with symbols from the alphabet. (In later figures, states are often labeled with their associated output rather than with a state identifier; which convention is being used should be clear from the context.)

From the initial state S_0 (indicated by the incoming arrow) the symbols of a sequence, taken one after another, indicate which transition to follow. For example, the sequence

abaa would be processed as follows: from the initial state S_0, the first a would take us to S_1, then the b would take us to S_2, then the a would take us back to S_1, then the final a would take us to S_3.

A regular language is one whose membership rule can be expressed as an automaton. The automaton of Figure 1.1 expresses such a rule: any sequence that ends in the "accepting" or "marked" state S_1 (indicated with a bold stroke) is in the language; any other sequence is not. (There is a variation on this structure in which the output or marking is associated with a transition arrow. That is known as a Mealy model, as opposed to the Moore model in which the output or marking is associated with a state. It is well known that the Mealy and Moore models are of equal expressive power, and therefore we will freely use whichever representation happens to be better suited to the case at hand.)

In fact, the automaton of Figure 1.1 is a formal expression of the alternating a's and b's rule described in prose above. The sequence *ababa* is in the automaton's language because the automaton's final state after processing this sequence is the marked state S_1. The sequence *bbb* is doomed after the first b, which takes us to the unmarked state S_3. In S_3, both a and b transition back into S_3 again, making it inescapable by any subsequent sequence. Therefore *bbb* is not in the language. The sequence *abab* passes through the marked state S_1 twice, but since it ends in the unmarked S_2, it is not in the language.

Note that while some may consider the prose rule ambiguous on whether the sequences a, or aa, or the sequence of length zero are in the language, there is no doubt when the rule is given in automaton form (a is in the language; aa and the sequence of length zero are not).

In the language learning problem, we are given a finite number of sequences that belong to some regular language, and we want to find the automaton that gives the rule for this language. (Engineering applications of such a capability include circuit synthesis [16] and syntactic pattern recognition [17, 18].)

The following example is adapted from [19]. Suppose we are given a single sequence *abcabdabcabd* and would like to know the automaton that produced it. While there is not

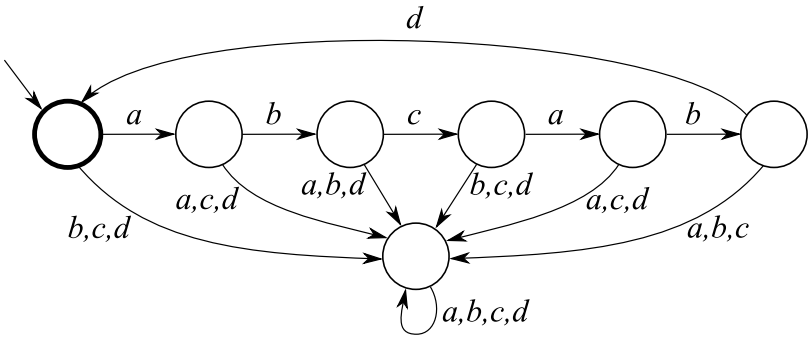

Figure 1.2: A reasonable solution to the inference problem.

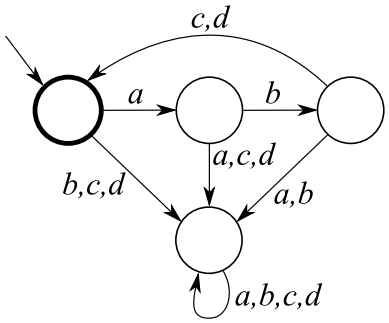

Figure 1.3: Another reasonable solution to the inference problem.

enough information in this one sequence to determine a general rule, several reasonable hypotheses may come to mind.

One might guess, since the given example is the sequence *abcabd* repeated twice, that the language membership rule allows only those sequences consisting of *abcabd* repeated some number of times. The automaton for this rule is illustrated in Figure 1.2.

Alternatively one might guess that the rule allows only those sequences in which the patterns *abc* and *abd* appear in any order; under this hypothesis, the fact that the given example alternates from the first to the second to the first to the second in an apparent pattern is merely a coincidence. The automaton for this rule is illustrated in Figure 1.3.

There are other solutions which, though satisfying the formal requirement, are not so intuitively appealing. At one extreme we have an automaton that matches exactly the one observed sequence and nothing else, shown in Figure 1.4. This hypothesis is undesirable because it makes no attempt to generalize from the example given to it. To avoid this problem

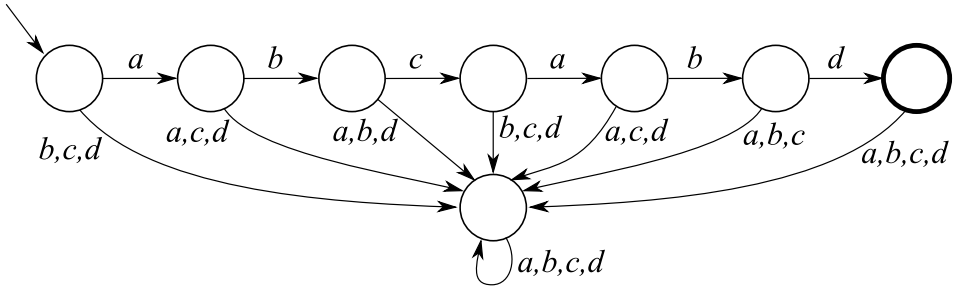

Figure 1.4: A solution that does not generalize enough.

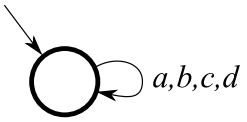

Figure 1.5: A solution that generalizes too much.

we may decide to seek the simplest automaton that can match the observed behavior. But this yields an automaton degenerate in the opposite extreme, Figure 1.5. Certainly, this automaton places *abcabdabcabd* in the language, as well as every other possible sequence. This solution is undesirable because it generalizes too much.

A famous result due to Gold [20] on the learnability of languages from examples is, roughly stated, that regular languages are not learnable from examples of the language[1]. The fundamental problem is the one just observed, namely, "that no positive example can refute a too general hypothesis" [21].

In this seminal model of learnability, formal languages are learned from an infinite series of examples. The algorithm is fed this series, and with each example it may update its current guess as to the underlying language rule. The language is identified in the limit if after some finite amount of time the guesses do not change and are equivalent to the underlying language rule. However, there is no way to know when this condition has been reached, and therefore the algorithm must continue running forever. The influence of this learning model is aptly summarized in [22]: "This deceptively simple idea has generated a

[1]Gold's results implied an interesting paradox that he noted in his paper: on the one hand, it was widely believed that children acquire language from listening to positive examples; and yet he had shown that learning any but trivial formal languages only from positive examples is unworkable.

large body of sophisticated literature".

When the series of examples is broadened to include not only sequences in the language, but also sequences not in the language and labeled as such, regular languages do become learnable in the limit. However, in later work Gold [23] showed that the problem of inferring a minimal automaton from a given finite set of labeled example sequences is intractable.

The publication of [24] represented a considerable breakthrough in the language learning problem. This result and its significance for system identification are the subject of Chapter 3.

1.2.2 Timed models

Engineered systems with elements of both discrete states and timing are nothing new. However, the models supporting the analysis and design of such systems do not have the coherent and unified presence of linear systems theory. Rather, they appear to have emerged in different sub-disciplines in response to engineering needs.

The deployment of programmable logic controllers (PLC's) relies on languages for the programming of complex industrial automation. The IEC 61131-3 standard can be taken as a codification of generally accepted models [25, 26]. The standard includes the ladder diagram, which conveys control concepts in terms close to relay logic — a notation judged expressive enough to be worth keeping even as its motivating implementation technology was going by the wayside [27].

Discrete event simulation has been widely applied to industry, operations, and business processes. A viewpoint emphasizing the relationship between traditional engineering systems and controls and discrete event systems includes figures such as Yu-Chi Ho, and is carried forward in textbooks such as [1].

In the field of formal verification a need arose for timed models with precise semantics. The seminal work [28] for this family of models is the timed automaton of Alur and Dill [29].

The following selection of models provides a context for the modeling approach taken

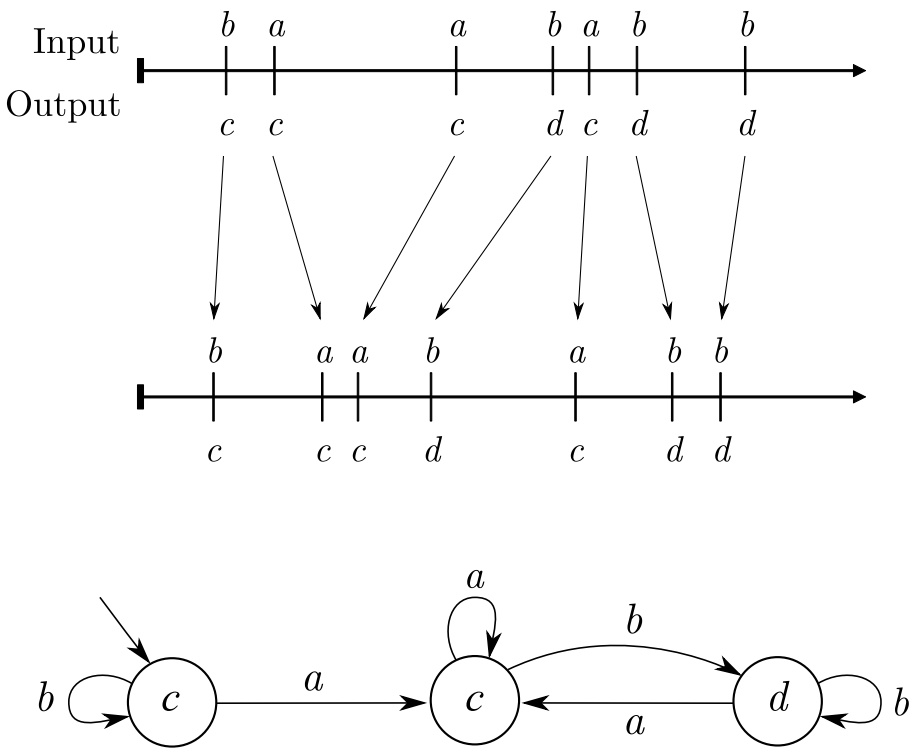

Figure 1.6: An untimed finite state machine.

in this work. (Note that some are presented with minor simplifications or adaptations for expository purposes. Specifically, all models are deterministic, and automata have been liberalized to allow a finite output alphabet.)

Untimed Finite State Machine

The classic finite state machine (FSM), Figure 1.6, does not come with any notion of time, apart from the fact that the inputs and outputs have a certain ordering. What makes this model untimed is not that the input and output symbols do not have associated times of occurrence — indeed they may — but that timing does not affect its behavior. The outputs depend only on the sequence of input events, regardless of any timing that may be associated with them.

The interpretation of the finite state machine diagram is very close to that of the automaton of Figure 1.1. (The key difference is that the automaton is restricted to a binary

output — the final state is either marked or unmarked — while each state of an FSM can have any of a finite number of output symbols.) The initial state is indicated with the incoming arrow. An input b causes a re-entry to the initial state, with output c. An input a transitions to the middle state, whose output is also c. A second a re-enters the middle state, for another output c. An input b transitions to the rightmost state, whose output is d, etc.

The Figure also shows two timelines. Each timeline represents a trace or execution of the system. The inputs are depicted above the timeline and the outputs below it. The time associated with these events is encoded in their distance from the origin, as on any ordinary axis, so that the timeline may be read left-to-right. The second timeline has the same input sequence as the first, but with perturbed timing. Since the two input sequences differ only in their timing, their output sequences are identical (except for their timing), i.e., in either case, when the input is *baababb*, the output is *cccdcdd*, with output taking the timing of the input. The addition of time here is superficial.

FSM With Tick Event

One approach to adding time to a finite state machine that must be mentioned at the outset is the use of a special "tick" input symbol that represents the passage of a fixed quantum of time. Researchers have looked upon this approach with disfavor for a variety of reasons [28, 29].

Significantly, this approach immediately reduces the timed system identification problem to the problem of finite state machine identification.

Delay Automaton

A delay automaton (DA), introduced in [30], is an automaton augmented with a single analog clock. This clock is reset every time the automaton makes a state transition; therefore the clock indicates the amount of time the system has been in its current state. Significantly, this clock value can affect the DA behavior. The guard conditions on a transition denote

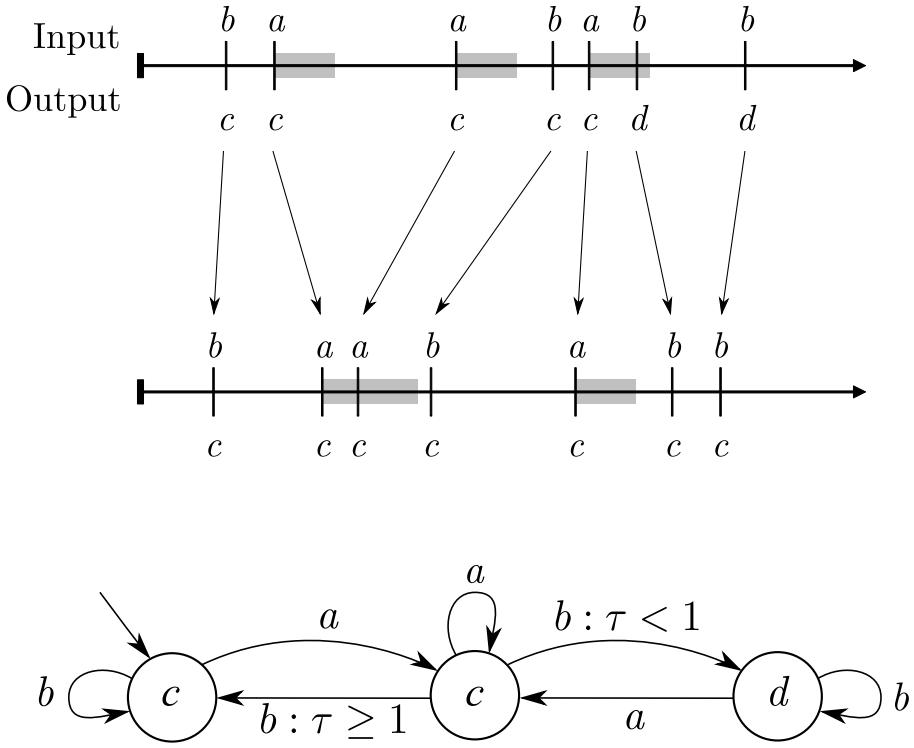

Figure 1.7: A delay automaton.

that this transition can be taken only if the clock state satisfies the expression.

Consider the machine depicted in Figure 1.7. The transition from the middle to the rightmost state occurs on input event b as it did in Figure 1.6, but the transition now also has an expression associated with it called a guard. The symbol τ represents the value of the clock at the time of the input event. From the middle state upon input b, the transition to the rightmost state will only occur if the system has been in the middle state for less than one second, $\tau < 1$. Otherwise, if $\tau \geq 1$, then an input b will take the system back to the initial state.

In the timelines, the gray bar indicates a one-second interval following entry of the middle state. If a b event occurs before one second has elapsed, the transition is to the rightmost state with an output d.

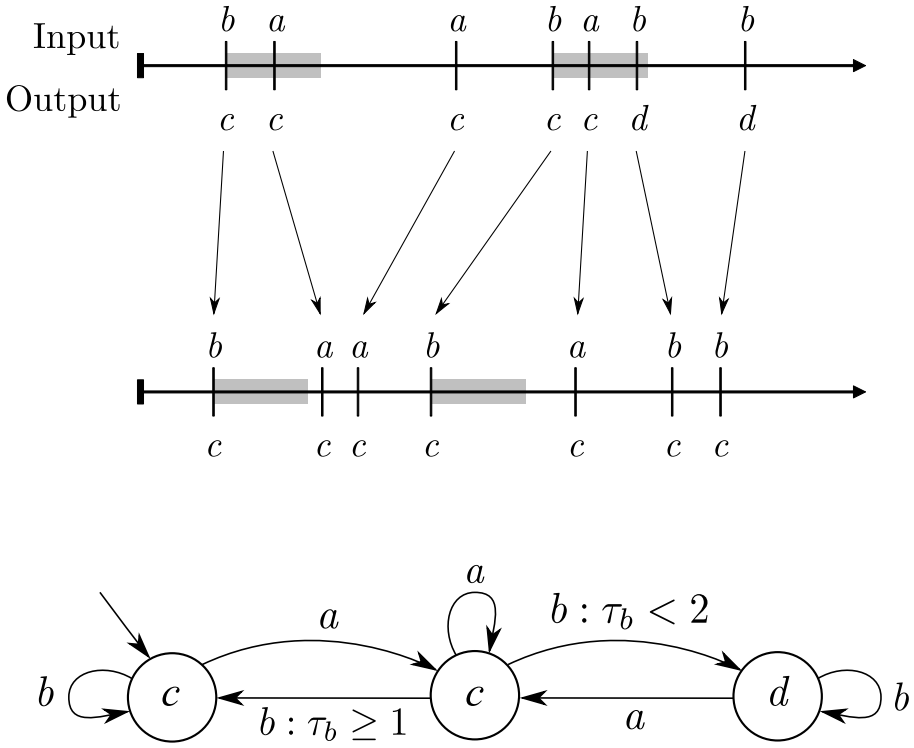

Figure 1.8: An event-recording automaton.

Event-Recording Automaton

In the event-recording automaton (ERA) model, introduced in [31], each input symbol has an associated clock. Whenever a input occurs, its associated clock is reset. So at any given time, the clocks either give the time elapsed since the last occurrence of each corresponding input symbol, or take a special value indicating that the corresponding symbol has not occurred yet in the run.

Consider the machine depicted in Figure 1.8. Since this machine has two input symbols a and b, it also has two clocks τ_a and τ_b. (The τ_a clock happens to go unused in this example.) The guards on transitions out of the middle state now refer to the time elapsed since the previous input b. From this state, an input b occurring less than two seconds after the previous b transitions to the rightmost state, with output d. If it has been more than two seconds since the last input b, a b takes us back to the initial state.

In the timelines, the gray bar indicates a two-second interval following an input b.

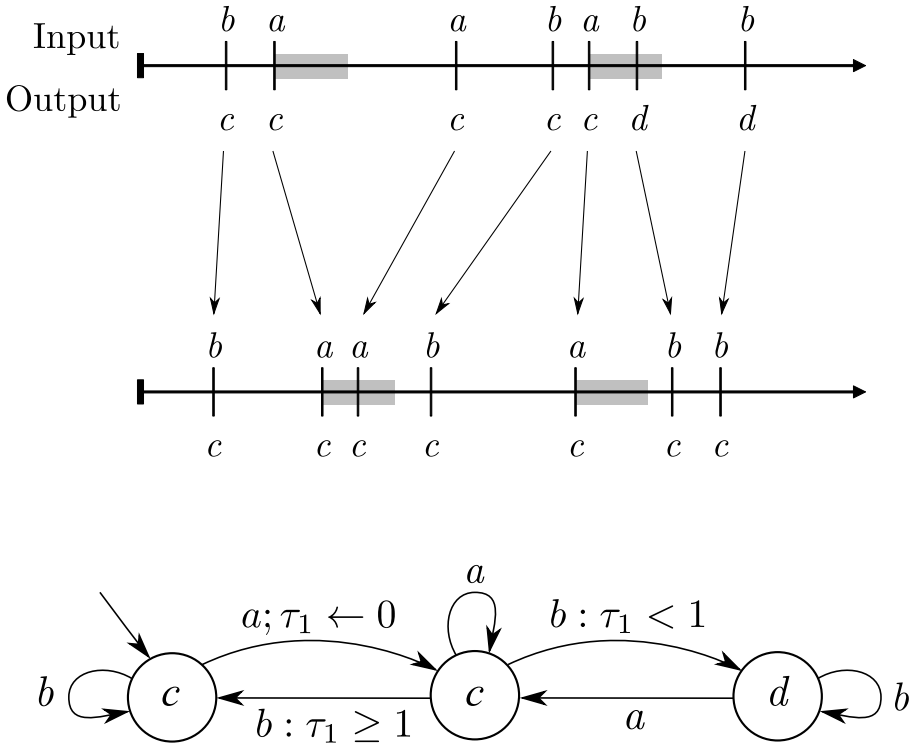

Figure 1.9: A timed automaton (the Alur-Dill model).

Timed Automaton

The classic timed automaton (TA) of Alur and Dill [29] has a fixed finite number of clocks. Unlike the ERA, the number of clocks is not related to the number of input symbols; rather, the number of clocks is arbitrary. The transitions have guard expressions involving the clocks, similar to the previous examples. Additionally, a clock reset, here denoted $\tau_c \leftarrow 0$, can be associated with any transition.

Consider the machine depicted in Figure 1.9. This machine happens to have only one clock, τ_1. This clock is reset whenever an input a occurs from the initial state. In the timelines, the gray bar indicates a one-second interval following a reset of the τ_1 clock.

This example happens to behave similarly to the DA of Figure 1.7. However, note that an input a from the middle state resets the clock in the DA but does not in this TA.

Discrete Event System

This formulation appears in [1]. It is noteworthy for being a fairly literal expression of a traditional discrete event simulation in an explicit state-space form. In this model, when an event is enabled, its clock (i.e., the time to event occurrence) starts ticking. When the event occurs, the time to the next event of this type is computed with a "roll of the dice." If the event is not enabled, its clock is reset upon any event occurring.

$$y^* = \min_{i \in \Gamma(x)} \{y_i\}$$

$$e' = \arg\min_{i \in \Gamma(x)} \{y_i\}$$

$$x' = f\left(x, e'\right)$$

$$t' = t + y^*$$

$$y_i{}' = \begin{cases} y_i - y^*, & i \neq e' \text{ and } i \in \Gamma\left(x\right) \\ v_{i,N_i+1}, & i = e' \text{ or } i \notin \Gamma\left(x\right) \end{cases}$$

$$N_i{}' = \begin{cases} N_i + 1, & i = e' \text{ or } i \notin \Gamma\left(x\right) \\ N_i, & \text{otherwise} \end{cases}$$

The current state is x. For each state there is a set $\Gamma(x)$ of enabled events. The y_i is the clock value, or remaining lifetime, of an event. The e' is the next event to occur. This event occurs next because it has the smallest clock value. The occurrence of the event advances the system state, via the state transition function f, to x', and advances the global time to t'.

The last two equations model events being "used up" as they are processed or expire. For the event type that just occurred (and also for any event types that were not enabled

in x), we obtain the next occurrence time v_{i,N_i+1} for an event of that type from a random number source. For all other event types, their clocks are advanced by $t' - t$.

Since the events can be inputs or outputs, this model does not require outputs to be synchronous with inputs. Also, since system state can influence which events may occur (via $\Gamma(x)$), it can influence which *input* events may occur.

1.3 Literature Review

With few exceptions, research on the identification of discrete event systems focuses on the untimed or logical DES model. Here we review prior work on identification of timed DES models such as timed automata, and its applications.

Supavatanakul et. al. [32] outline an algorithm for timed automaton identification. No correctness proof for the algorithm is apparent. The algorithm does not request any experiments on its own, but rather depends on completeness of the given data. In subsequent work [33] the authors demonstrate the applicability of their technique to an industrial fault detection and isolation problem. (This work can be compared with [34], which uses an untimed grammar as the underlying mathematical model in a similar identification and control problem.)

Choi and Kim [35] demonstrate the learning of a DES in the DEVS formalism with recurrent neural networks. The authors demonstrate the success of their technique with examples: several graphs depict identified system response to a random input overlaid on the true system response to the same input, showing a good match. The paper does not describe intended applications.

Verwer [30] has investigated the identification of delay automata (DA), timed systems restricted to a single clock. He is interested in the problem of learning from given examples (in other work he investigates learning in the limit), rather than a framework that allows the algorithm to propose experiments. He mentions applications only in very general terms.

The state of the art in identification of Alur-Dill timed automata (a model related to,

but distinct from, the one considered in this work) is certainly found in the series of papers by Grinchtein and collaborators [36–39]. A summary of the conclusions is as follows.

General timed automata are judged to be very difficult to learn, so the scope is limited to the subset of TAs known as event recording automata (ERA). Since ERAs can be determinized, it suffices to learn deterministic ERAs (DERAs). The DERAs are not canonical (there is no unique minimal DERA), but for every DERA there is a unique minimal so-called simple DERA (SDERA) with the same behavior; so, it suffices to learn SDERAs.

Then, Angluin's L^* algorithm [24] is adapted to learn SDERAs, where the identified automaton takes an input alphabet extended in a sophisticated way: the usual input symbols are augmented with guard expressions. So instead of processing a symbol paired with the clock values at the time the symbol appears, the identified automaton processes a symbol paired with a symbolic expression of inequalities on clocks, representing a whole set of possible clock valuations.

Several algorithms are given which make different tradeoffs in computational complexity. In particular, the algorithm TL_s^* is reworked into another algorithm that has worse worst-case performance than TL_s^*, but with expected better performance on average.

There is other identification work rooted in L^* that is not explicitly concerned with real time but is worth noting both for technique and applications. (It may be seen as a fulfillment of research going back to [40].)

Berg [41] identifies state machines in which the inputs are parameterized. Her application is the identification of formal specifications for communication protocols from hardware, since such documentation is often unavailable or outdated. Simulation tests demonstrate identification of randomly generated machines and of a partial model of a network presence protocol.

Bohlin [42] uses L^* to infer an automaton, then post-processes the result into a much more compact extended FSM model with state variables. They test the technique on A-MLC, a protocol deployed by several European telecom operators for communicating presence information for mobile devices (and not the same one as in [41] above). The technique

was successful, i.e., it passed the test posed by their equivalence oracle.

1.4 Overview

For a target model class for identification, none of the models described in section 1.2.2 is an ideal fit. The appropriate model should be oriented to simulation rather than verification, should make a clear distinction between input and output, and must allow spontaneous output events, i.e., output events not synchronous with some input event.

In Chapter 2 we introduce a model called timed event systems (TES) designed to meet these requirements. This model is developed as a special instance of an abstract system. One advantage of this formulation is that it allows us to study the relationship among different model types, in particular the relationship between timed event systems and finite state machines.

Chapter 3 is an exposition of the classic result of Angluin [24], generalized to apply to finite state machines. The result is an efficient algorithm for finite state machine identification. Unlike the traditional identification problem of inferring a system consistent with a given set of input-output examples, this algorithm formulates experiments designed to elicit exactly the behaviors from the unknown system needed for model formulation. This ability to actively experiment with the unknown system is the key to the algorithm's efficiency.

A simple approach to reconciling timed event systems and finite state machines is given in Chapter 4. Here we show that a timed event system can be made to have the external appearance of a state machine. This is done to allow an FSM identification algorithm to be applied without modification to the TES identification problem. Additionally, some specializations of the algorithm of Chapter 3 are given to better adapt the algorithm to the system identification problem.

The goal of Chapter 5 is to demonstrate that the identification technique, although it cannot be proven to work in general, is in fact effective on a non-trivial identification problem. We also empirically study the main theoretically indicated weakness of the approach.

While the main argument of this work makes absolutely minimal assumptions about

the structure of the unknown system, and therefore requires a very general model form and faces the identification problem at its most difficult, we do not ignore the possibility that some knowledge of the unknown system's form is available. Such a possibility is discussed in Appendix A.

1.5 Applications

There are some interesting applications for identification of timed machines. We expect it to be useful whenever constructing a model analytically is onerous or impractical, and to have applications similar to those for identification of linear systems (simulation, control system design) as well as in other areas.

Simulation; control system design. Suppose we have a manufacturing system that includes many subsystems in which timing is relevant. We would like to simulate the system in software to study the effects of modifications or new control algorithms.

Formal verification. We want to perform a verification study and some elements of the system have no available analytical model [39].

Automatic model 'flattening.' We have a sophisticated and detailed computer simulation of some process. We would like to embed the detailed simulation as a component in a wider-scope simulation, but the detailed simulation runs too slowly. So we automatically create a fast-running timed sequential machine model that reasonably approximates the detailed simulation, and embed this instead.

Chapter 2: Timed Event Systems

2.1 Introduction

It does not seem to be widely appreciated that timed discrete-event systems have a system theory analogous to that of linear systems. Here we introduce a Timed Event System (TES) formalism, and develop the elements of its system theory.

TES are closely related to the well-known DES model. Most knowledge of DES applies immediately to TES. We will argue here that a construct with so few restrictions has a mathematical structure that, though simple, is well worth exploring. Specifically, we show that TES and automata, like linear systems, are special cases of a general or abstract system.

There is a little risk that these results are perceived as an over-mathematization of straightforward concepts. But even with no claim to great depth or novelty, there are several reasons to develop these simple systems and their "abstract nonsense" [43] to this level of detail.

1. Since the models used here are not widely used in electrical engineering, it is worth taking some care in their description. As simple as these ideas are, I could not locate any reference that makes these points, so I have developed it here for my own purposes.

2. Software development is considerably aided by the precise expression of humdrum facts. Many ideas here are not mere abstractions but are closely followed in working software.

3. The ability to analyze systems formally is useful. For example, we will show the formal correctness of the usual DES simulation algorithm.

4. Most importantly, these models are foundational for the identification problem. For

example, the abstract formulation provides a language for expressing a relationship between a TES and an automaton.

There is interplay among these arguments as well, for example, if we need software to translate between TES and automaton based models of the same behavior, we should be precise about what those models are; and, writing a formal account of a simulation algorithm cannot help but influence how we write the software.

2.2 What is a block diagram?

Here we fill in details on the meaning of an interconnection of blocks in a way that fits very naturally with their usual interpretation.

2.2.1 Blocks, segments, and overlay

The *behavior* of a block in a block diagram is the rule it uses to transform an input to an output. These inputs and outputs are real-valued signals in the case of a linear system, or timelines in the case of a TES. When the system type is not important we generically call them *segments*. Segments are always functions of time and always have an associated duration.

A block maps an input segment to an output segment.

$$y = \beta(u)$$

It is worth emphasizing that the segments u and y cover an interval of time; they are not instantaneous values. If the input u has duration t, then the output y also has duration t, and is understood to take place over the same interval of time as the input.

Additionally, some diagrams (including linear systems and TES diagrams) may contain a junction where signals are combined before entering a block. In linear systems this is called an adder or a summing junction. In a TES, events from different timelines are

simply interleaved. We will generically refer to this operation as an overlay. Overlay is denoted with the common plus sign or summation notation.

2.2.2 A diagram as a system of equations

It follows that a diagram of N blocks indexed by $k = 1 \ldots N$ corresponds to a system of equations of the form

$$y_k = \beta_k (u_k)$$

$$u_k = \sum\nolimits_{j \in I_k} y_j$$

Each block has input u_k and output y_k. The I_k are the indices, valued $0 \ldots N$, of block outputs that feed into block k, and encode the block diagram interconnections. It can include a special exogenous input (denoted as u or y_0, as convenient) that is not the output of any block (or is the output of a block with no input — the important thing is that system behavior cannot affect it), and can be considered the exogenous input to the diagram overall.

Since the input to any block is a combination of the outputs that feed into it, we may also express the system as simply

$$y_k = \beta_k \left(\sum\nolimits_{j \in I_k} y_j \right)$$

In this form it is clear that the outputs y_k constitute a fixed point of the system.

2.2.3 Solving a diagram

We consider a block diagram to pose a problem in which the block behaviors and exogenous input are known, the outputs are unknowns, and solving a block diagram means solving for

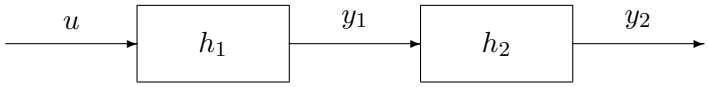

Figure 2.1: A cascade of two LTI blocks

the signals on all the outputs, such that all the block input-output relations hold. (Since the inputs are a combination of outputs, they can always be computed from the solution after the fact.)

Viewed as a system of equations, the unknown variables y_k to be solved for are not numbers, or functions taken pointwise, but entire functions of time (technically they are segments, but their length is already known to be equal to the length of the exogenous input), a situation similar to that of a differential equation problem.

Interconnecting blocks cannot be taken for granted. The topology of a diagram has a critical effect on how it may be solved, as discussed next.

2.2.4 Acyclic and cyclic interconnections

An interconnection with no cycles can always be solved directly in terms of the input signals and block behaviors. For example, consider a cascade of two LTI blocks, where the behavior of each is given as a convolution (Figure 2.1).

$$y_1 = h_1 * u$$

$$y_2 = h_2 * y_1$$

The special properties of linear systems allow us to combine the blocks h_1 and h_2 into one, and then process the input with this composite only once. Since we did not solve for y_1 this is technically not a complete solution of the diagram, but often in practice this is all we need. Indeed, if y_1 is considered only an intermediate result, it is advantageous to avoid allocating memory to store it.

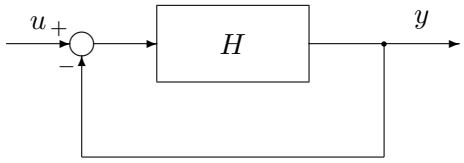

Figure 2.2: A simple cyclic interconnection

On the other hand, we could process the entire input through h_1 to compute y_1, then take this result and pass it through h_2 to compute y_2. This block-by-block style implementation does not depend on linearity or any other special block properties (even causality). It is apparent that any acyclic block diagram may be solved this way.

Cyclic interconnection in a diagram is also known as feedback, one of the pillars of systems and control theory. Diagrams with feedback are significantly qualitatively different from the acyclics.

Consider the simple cyclic interconnection depicted in Figure 2.2 ($H(t)$ denotes a Heaviside step function).

$$y(t) = \frac{1}{T} H(t) * (u(t) - y(t))$$

$$= \frac{1}{T} \int_0^t (u(\tau) - y(\tau)) H(t - \tau) \, d\tau$$

Note that the output $y(t)$ appears both inside and outside of the integral — this is the implicit equation that typically arises from feedback.

Again, since this system is a special linear type, it is immediately solvable with transform

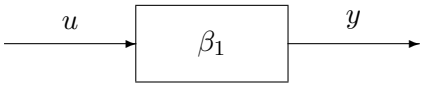

Figure 2.3: Diagram consisting of a lone block

techniques.

$$Y(s) = \frac{1}{T}\left(U(s) - Y(s)\right)\frac{1}{s}$$

$$Y(s) = \frac{1}{T}\frac{1}{s + \frac{1}{T}}U(s)$$

$$y(t) = \frac{1}{T}e^{-t/T} * u(t)$$

The system of equations of an acyclic diagram can be solved by composition of behavior functions, but feedback introduces implicitness into the equations. Unlike the acyclic case, there does not appear to be a general solution method for cyclic diagrams that is independent of any special block properties like linearity.

2.2.5 Issues in solving a diagram

Any lone block, interpreted as a trivial interconnection, has a unique solution (Figure 2.3). But it does not follow that an interconnection of such blocks has a unique solution.

Consider the classic example [44] of a closed loop made of an integrator, a square root nonlinearity, and a scalar gain of two. Each of the individual blocks has a unique input-output behavior. The differential equation for the interconnected system is $\dot{x} = 2\sqrt{x}$. But for the initial condition $x(0) = 0$ the equation/diagram does not have a unique solution, for it is satisfied by both $x(t) = 0$ and $x(t) = t^2$.

Consider this simplest possible example: the identity function is well-defined, but an interconnection in which an identity block's output is fed back to itself, is solved by any element of the function's domain.

We see that feedback interconnection of blocks changes the fundamental nature of the problem.

It is worth noting that block behaviors are enough to decide if some given proposed solution is in fact correct. The blocks can simply be visited one by one to determine whether the proposed input-output relationship holds at every block.

2.3 Abstract Systems Theory

Here we develop and summarize properties of segments, the state transition and output functions, etc.

2.3.1 Segments and their operations — overview

The inputs and outputs on a block diagram may be called signals or waveforms, sequences, or timelines, in the context of a specific system type. Here we define a segment as an object general enough to accommodate any of these specific structures. (The reason we do this is to have a theory general enough to include linear systems, timed event systems, and automata. In particular, we do not rely on the timebase being either discrete or continuous, in order to accommodate either case.)

A segment consists of two parts. First, a segment is a function that can be evaluated at different times. The domain of this function is called a timebase, whose elements are of an appropriate time type. Additionally, a segment has an associated length, of the timebase type.

Expressing properties like causality depends on a notation for before and after. This role is filled by splitting and splicing of segments. The prefix and suffix operators pref and suff return the parts of a segment before and after some given breakpoint, and a splice (denoted by $\oplus$) combines two segments sequentially into one. Naturally, when the part of a segment before a breakpoint is spliced with the part of a segment after the breakpoint,

we recover the original segment.

$$\text{pref}\,(u,t) \oplus \text{suff}\,(u,t) = u$$

2.3.2 Segments and their operations — the details

Here we provide some detailed definitions that are useful for simple proofs (which serve as much as anything to merely illustrate the definitions) as well as for expressing software implementation requirements. A general reference for the abstract algebra comments is [45].

This section is admittedly tedious, but working these proofs in detail was a useful check on the robustness and completeness of the definitions, and provides a record of the thought process behind the model.

Definition 2.1. A *timebase* is a set of quantities used to specify duration. It always includes a zero element. It has a less-than-or-equal-to relation that imposes a total ordering on the elements [46], and binary plus and unary minus that, with the zero, satisfy the definition of a group.

Proposition 2.2. In any timebase, there is no t such that $\tau \leqslant t < \tau$.

Proof. Assume there is some t. Since $\tau \leqslant t$ and the timebase is totally ordered, it follows that $t \not\leqslant \tau$. But we are also asserting that $t < \tau$, i.e., that $(t \leqslant \tau) \wedge (t \neq \tau)$, which contradicts $t \not\leqslant \tau$. Therefore, there is no such t. □

Definition 2.3. A *segment* is a tuple (s, t) where t is the length or duration of the segment and s is a partial function. The domain of s is some timebase, and t is an element of the same timebase. The duration t is always greater than or equal to zero. Evaluation of the segment at some time τ "falls through" to evaluation of its constituent function s whenever $0 \leqslant \tau < t$, otherwise the segment is undefined at τ.

It follows that two segments are equal precisely when their function parts are equal and

their durations are equal. Notationally, we will address the function part of a segment with function notation and the duration part with a len operator.

Definition 2.4. The splice of two segments is a segment. The splice operator is denoted with $\oplus$. The function and length of the splice are:

$$(u_1 \oplus u_2)(t) = \begin{cases} u_1(t) & 0 \leqslant t < \operatorname{len}(u_1) \\ u_2(t - \operatorname{len}(u_1)) & \operatorname{len}(u_1) \leqslant t < \operatorname{len}(u_1) + \operatorname{len}(u_2) \end{cases}$$

$$\operatorname{len}(u_1 \oplus u_2) = \operatorname{len}(u_1) + \operatorname{len}(u_2)$$

Proposition 2.5. The splice is associative, $(u_1 \oplus u_2) \oplus u_3 = u_1 \oplus (u_2 \oplus u_3)$.

Proof. Using the definition,

$$((u_1 \oplus u_2) \oplus u_3)(t) = \begin{cases} (u_1 \oplus u_2)(t) & 0 \leqslant t < \operatorname{len}(u_1 \oplus u_2) \\ u_3(t - \operatorname{len}(u_1 \oplus u_2)) & \operatorname{len}(u_1 \oplus u_2) \leqslant t < \operatorname{len}(u_1 \oplus u_2) + \operatorname{len}(u_3) \end{cases}$$

$$= \begin{cases} \left\{ \begin{array}{ll} u_1(t) & 0 \leqslant t < \operatorname{len}(u_1) \\ u_2(t - \operatorname{len}(u_1)) & \operatorname{len}(u_1) \leqslant t < \operatorname{len}(u_1) + \operatorname{len}(u_2) \end{array} \right\} & 0 \leqslant t < \operatorname{len}(u_1 \oplus u_2) \\ u_3(t - \operatorname{len}(u_1 \oplus u_2)) & \operatorname{len}(u_1 \oplus u_2) \leqslant t < \operatorname{len}(u_1 \oplus u_2) + \operatorname{len}(u_3) \end{cases}$$

Since both $0 \leqslant t < \operatorname{len}(u_1)$ and $\operatorname{len}(u_1) \leqslant t < \operatorname{len}(u_1) + \operatorname{len}(u_2)$ imply $0 \leqslant t < \operatorname{len}(u_1 \oplus u_2)$, we do not need to preserve the latter condition.

$$((u_1 \oplus u_2) \oplus u_3)(t) = \begin{cases} u_1(t) & 0 \leqslant t < \operatorname{len}(u_1) \\ u_2(t - \operatorname{len}(u_1)) & \operatorname{len}(u_1) \leqslant t < \operatorname{len}(u_1 \oplus u_2) \\ u_3(t - \operatorname{len}(u_1 \oplus u_2)) & \operatorname{len}(u_1 \oplus u_2) \leqslant t < \operatorname{len}(u_1 \oplus u_2 \oplus u_3) \end{cases}$$

Similarly,

$$(u_1 \oplus (u_2 \oplus u_3))(t) = \begin{cases} u_1(t) & 0 \leqslant t < \operatorname{len}(u_1) \\ (u_2 \oplus u_3)(t - \operatorname{len}(u_1)) & \operatorname{len}(u_1) \leqslant t < \operatorname{len}(u_1) + \operatorname{len}(u_2 \oplus u_3) \end{cases}$$

$$= \begin{cases} u_1(t) & 0 \leqslant t < \operatorname{len}(u_1) \\ u_2(t - \operatorname{len}(u_1)) & 0 \leqslant t - \operatorname{len}(u_1) < \operatorname{len}(u_2) \\ u_3(t - \operatorname{len}(u_1) - \operatorname{len}(u_2)) & \operatorname{len}(u_2) \leqslant t - \operatorname{len}(u_1) < \operatorname{len}(u_1) + \operatorname{len}(u_2 \oplus u_3) \end{cases}$$

Clearly the functions are equal.

We also see that

$$\begin{aligned}
\operatorname{len}\left((u_1 \oplus u_2) \oplus u_3\right) &= \operatorname{len}\left(u_1 \oplus u_2\right) + \operatorname{len}\left(u_3\right) \\
&= \operatorname{len}\left(u_1\right) + \operatorname{len}\left(u_2\right) + \operatorname{len}\left(u_3\right) \\
&= \operatorname{len}\left(u_1\right) + \operatorname{len}\left(u_2 \oplus u_3\right) \\
&= \operatorname{len}\left(u_1 \oplus \left(u_2 \oplus u_3\right)\right)
\end{aligned}$$

so the lengths are equal. Therefore, the splice is associative. □

Since the splice operation is associative, a class of segments under splices constitutes a semigroup.

Definition 2.6. A special segment denoted by Λ has a length of zero and a function equal to the empty set.

Proposition 2.7. $u \oplus \Lambda = \Lambda \oplus u = u$.

Proof. We show that the segments $u \oplus \Lambda$ and u are equal by showing that the lengths are equal and the function parts are equal. First,

$$\begin{aligned} \operatorname{len}(u \oplus \Lambda) &= \operatorname{len}(u) + \operatorname{len}(\Lambda) \\ &= \operatorname{len}(u) + 0 \\ &= \operatorname{len}(u) \end{aligned}$$

So the lengths are equal.

Since there is no t such that $\operatorname{len}(u) \leqslant t < \operatorname{len}(u)$, then $(u \oplus \Lambda)(\tau) = u(\tau)$, i.e., the function parts of the segments are equal.

The argument for the other operand ordering of $\Lambda \oplus u = u$ is practically the same. □

Since Λ is the identity of the splice operator, a segment semigroup including Λ is also a monoid.

Proposition 2.8. If segment length is zero, the segment equals Λ.

Proof. The function part of such a segment can be evaluated at t such that $0 \leqslant t < \operatorname{len}(u)$. But there are no such t, so the function has empty domain, so the function is the empty set. Such a segment satisfies the definition of Λ. □

Definition 2.9. The pref and suff are defined in terms of the segments they return.

$$(\mathrm{pref}\,(u,\tau))\,(t) = u\,(t)$$

$$\mathrm{len}\,(\mathrm{pref}\,(u,\tau)) = \begin{cases} \tau & 0 \leqslant \tau < \mathrm{len}\,(u) \\ 0 & \tau < 0 \\ \mathrm{len}\,(u) & \tau \geqslant \mathrm{len}\,(u) \end{cases}$$

The suff is similar.

$$(\mathrm{suff}\,(u,\tau))\,(t) = u\,(\max\,(0,\tau) + t)$$

$$\mathrm{len}\,(\mathrm{suff}\,(u,\tau)) = \begin{cases} \mathrm{len}\,(u) - \tau & 0 \leqslant \tau < \mathrm{len}\,(u) \\ 0 & \tau \geqslant \mathrm{len}\,(u) \\ \mathrm{len}\,(u) & \tau < 0 \end{cases}$$

Proposition 2.10. For any t, $\mathrm{pref}\,(u,t) \oplus \mathrm{suff}\,(u,t) = u$.

Proof. We proceed by cases of τ. First, for any τ,

$$(\mathrm{pref}\,(u,\tau) \oplus \mathrm{suff}\,(u,\tau))\,(t) =$$
$$\begin{cases} \mathrm{pref}\,(u,\tau)\,(t) & 0 \leq t < \mathrm{len}\,(\mathrm{pref}\,(u,\tau)) \\ \mathrm{suff}\,(u,\tau)\,(t - \mathrm{len}\,(\mathrm{pref})) & \mathrm{len}\,(\mathrm{pref}) \leq t < \mathrm{len}\,(\mathrm{pref}) + \mathrm{len}\,(\mathrm{suff}) \end{cases}$$

If $\tau < 0$,

$$(\mathrm{pref}(u,\tau) \oplus \mathrm{suff}(u,\tau))(t) = \begin{cases} \mathrm{pref}(u,\tau)(t) & 0 \leqslant t < 0 \\ \mathrm{suff}(u,\tau)(t-0) & 0 \leqslant t < 0 + \mathrm{len}(u) \end{cases}$$

$$= u(0+t), \quad 0 \leqslant t < \mathrm{len}(u)$$

$$\mathrm{len}(\mathrm{pref}(u,\tau) \oplus \mathrm{suff}(u,\tau)) = 0 + \mathrm{len}(u)$$

If $\tau \geqslant \mathrm{len}(u)$,

$$(\mathrm{pref}(u,\tau) \oplus \mathrm{suff}(u,\tau))(t) = \begin{cases} u(t) & 0 \leqslant t < \mathrm{len}(u) \\ \mathrm{suff}(u,\tau)(t - \mathrm{len}(u)) & \mathrm{len}(u) \leqslant t < \mathrm{len}(u) \end{cases}$$

$$\mathrm{len}(\mathrm{pref}(u,\tau) \oplus \mathrm{suff}(u,\tau)) = \mathrm{len}(u) + 0$$

Finally, if $0 \leqslant \tau < \operatorname{len}(u)$, then

$$(\operatorname{pref}(u,\tau) \oplus \operatorname{suff}(u,\tau))(t) = \begin{cases} \operatorname{pref}(u,\tau)(t) & 0 \leqslant t < \tau \\ \operatorname{suff}(u,\tau)(t-\tau) & \tau \leqslant t < \tau + \operatorname{len}(u) - \tau \end{cases}$$

$$= \begin{cases} u(t) & 0 \leqslant t < \tau \\ u(\tau + t - \tau) & \tau \leqslant t < \operatorname{len}(u) \end{cases}$$

$$= u(t), \quad 0 \leqslant t < \operatorname{len}(u)$$

$$\operatorname{len}(\operatorname{pref}(u,\tau) \oplus \operatorname{suff}(u,\tau)) = \tau + \operatorname{len}(u) - \tau = \operatorname{len}(u)$$

Therefore, for any t, $\operatorname{pref}(u,t) \oplus \operatorname{suff}(u,t) = u$. □

Since the above holds for any t, it can be used to decompose a segment into a splicing together of subsegments.

Proposition 2.11. $\operatorname{suff}(u_1 \oplus u_2, \operatorname{len}(u_1)) = u_2$

Proof. Since we always have $0 \leqslant \operatorname{len}(u_1)$,

$$\begin{aligned}
(\operatorname{suff}(u_1 \oplus u_2, \operatorname{len}(u_1)))(t) &= (u_1 \oplus u_2)(\operatorname{len}(u_1) + t) \\
&= \begin{cases} u_1(\operatorname{len}(u_1) + t) & 0 \leqslant \operatorname{len}(u_1) + t < \operatorname{len}(u_1) \\ u_2(\operatorname{len}(u_1) + t - \operatorname{len}(u_1)) & \operatorname{len}(u_1) \leqslant \operatorname{len}(u_1) + t < \operatorname{len}(u_1 \oplus u_2) \end{cases} \\
&= u_2(\operatorname{len}(u_1) + t - \operatorname{len}(u_1)), \quad 0 \leqslant t < \operatorname{len}(u_2)
\end{aligned}$$

The first case in the conditional expression never applies, because suff is only defined for $0 \leqslant t$.

As for length, if $\operatorname{len}(u_1) < \operatorname{len}(u_1 \oplus u_2)$, then

$$\begin{aligned}
\operatorname{len}(\operatorname{suff}(u_1 \oplus u_2, \operatorname{len}(u_1))) &= \operatorname{len}(u_1 \oplus u_2) - \operatorname{len}(u_1) \\
&= \operatorname{len}(u_2)
\end{aligned}$$

If $\text{len}(u_1) = \text{len}(u_1 \oplus u_2)$, then

$$\text{len}(\text{suff}(u_1 \oplus u_2, \text{len}(u_1))) = 0$$

which equals $\text{len}(u_2)$ because the condition implies, via cancellation in the timebase group, that $\text{len}(u_2)$ is zero:

$$\text{len}(u_1 \oplus u_2) = \text{len}(u_1)$$

$$\text{len}(u_1) + \text{len}(u_2) = \text{len}(u_1) + 0$$

$$\text{len}(u_2) = 0$$

Therefore, $\text{suff}(u_1 \oplus u_2, \text{len}(u_1)) = u_2$. □

2.3.3 Overlay operation

We introduce an overlay operator, denoted by the common plus sign. It corresponds to a signal combiner junction on a block diagram. It can take different specific interpretations over different types. Splice and overlay are related through:

$$\operatorname{pref}(u+v,t) = \operatorname{pref}(u,t) + \operatorname{pref}(v,t)$$

$$\operatorname{suff}(u+v,t) = \operatorname{suff}(u,t) + \operatorname{suff}(v,t)$$

In what follows, use of a summation symbol refers to the overlay operator.

2.3.4 Overlay of splices

Lemma 2.12. A splice of overlays equals an overlay of splices, i.e.,

$$(u_0 + v_0) \oplus (u_1 + v_1) \oplus \cdots \oplus (u_n + v_n) = (u_0 \oplus u_1 \oplus \cdots \oplus u_n) + (v_0 \oplus v_1 \oplus \cdots \oplus v_n)$$

Proof. We proceed by induction. The base case is:

$$\begin{aligned}
(u_0 + v_0) \oplus (u_1 + v_1) &\triangleq (\operatorname{pref}(u,t) + \operatorname{pref}(v,t)) \oplus (\operatorname{suff}(u,t) + \operatorname{suff}(v,t)) \\
&= \operatorname{pref}(u+v,t) \oplus \operatorname{suff}(u+v,t) \\
&= u + v \\
&= (\operatorname{pref}(u,t) \oplus \operatorname{suff}(u,t)) + (\operatorname{pref}(v,t) \oplus \operatorname{suff}(v,t)) \\
&\triangleq (u_0 \oplus u_1) + (v_0 \oplus v_1)
\end{aligned}$$

For the inductive step, we must show that

$$(u_0 + v_0) \oplus \cdots \oplus (u_n + v_n) = (u_0 \oplus \cdots \oplus u_n) + (v_0 \oplus \cdots \oplus v_n)$$

implies

$$(u_0 + v_0) \oplus \cdots \oplus (u_{n+1} + v_{n+1}) = (u_0 \oplus \cdots \oplus u_{n+1}) + (v_0 \oplus \cdots \oplus v_{n+1})$$

And the inductive step does hold:

$$\begin{aligned}
&[(u_0 + v_0) \oplus \cdots \oplus (u_n + v_n)] \oplus (u_{n+1} + v_{n+1}) \\
&= [(u_0 \oplus \cdots \oplus u_n) + (v_0 \oplus \cdots \oplus v_n)] \oplus (u_{n+1} + v_{n+1}) \\
&= [U_n + V_n] \oplus (u_{n+1} + v_{n+1}) \\
&= (U_n \oplus u_{n+1}) + (V_n \oplus v_{n+1}) \\
&= (u_0 \oplus \cdots \oplus u_n \oplus u_{n+1}) + (v_0 \oplus \cdots \oplus v_n \oplus v_{n+1})
\end{aligned}$$

□

2.3.5 Segment spaces

Segments ultimately are elements of sets called segment spaces that serve as domains and ranges of behavior block functions. Here are some examples of segment spaces.

Real-time waveforms

In this case the timebase is the real numbers, and the generators are some continuous functions on a finite interval. The splices of these give piecewise continuous functions.

(N.B. The $H(t)$ below denotes a Heaviside step function.)

$$u_1(t) = H(t - 1.0),\ 0.0 \leqslant t < 2.0$$

$$u_2(t) = H(t - 1.0) - H(t - 2.0),\ 0.0 \leqslant t < 3.0$$

$$u_1 \oplus u_2 = \begin{cases} H(t - 1.0) & 0 \leqslant t < 2.0 \\ H(t - 3.0) - H(t - 4.0) & 2.0 \leqslant t < 5.0 \end{cases}$$

Discrete sequences

Discrete sequences take a timebase of the integers and are perhaps the simplest segment type.

$$u_1(t) = \{(0, a), (1, b)\},\ \mathrm{len}(u_1) = 2$$

$$u_2(t) = \{(0, c), (1, d), (2, e)\},\ \mathrm{len}(u_2) = 3$$

$$u_1 \oplus u_2 = \{(0, a), (1, b), (2, c), (3, d), (4, e)\},\ \mathrm{len}(u_1 \oplus u_2) = 5$$

Note that for discrete time, the timebase serves only to order the sequence. Therefore the segments may be much more conveniently denoted as tuples, for example, leaving the indices implicit (or notationally suppressed). (We will not 'really' get rid of the indices,

since the time shift in the concatenation formula depends on them.)

$$u_1 = (a, b)\,,\ \mathrm{len}\,(u_1) = 2$$

$$u_2 = (c, d, e)\,,\ \mathrm{len}\,(u_2) = 3$$

$$u_1 \oplus u_2 = (a, b, c, d, e)\,,\ \mathrm{len}\,(u_1 \oplus u_2) = 5$$

Event timelines

Here the timebase is more abstract. It consists of a set of tuples (k, τ) each encoding an integer index $k \in \mathbb{Z}$ annotated with a real-valued sim time $t \in \mathbb{R}$. The zero is $(0, 0.0)$.

Intuitively, the τ indicates what we typically think of as the simulation time. The integer k provides an ordering for events that occur at the "same" τ. Therefore there are no simultaneous events properly speaking, although events can be meaningfully described as happening at the same time (the same τ). (N.B. In this example we again notationally suppress the integer part of the timebase and write tuples instead.)

$$u_1\,(t) = ((1.0, a)\,, (3.0, b))\,, \mathrm{len}\,(u_1) \triangleq (2, 3.0)$$
$$u_2\,(t) = ((3.0, a)\,, (5.0, b))\,, \mathrm{len}\,(u_2) \triangleq (2, 3.0)$$
$$u_1 \oplus u_2 = ((1.0, a)\,, (3.0, b)\,, (3.0, a)\,, (5.0, b))\,, \mathrm{len}\,(u_1 \oplus u_2) = (4, 6.0)$$

2.3.6 System theory in a nutshell

Earlier we introduced the behavior function $y = \beta\,(u)$. We will now introduce some associated functions that constitute an alternative account of the behavior.

Definition 2.13. Introduce a memory function μ, state transition function f, and output

function g which together must satisfy

$$f\left(\mu\left(u_{01}\right), u_{12}\right) = \mu\left(u_{01} \oplus u_{12}\right)$$

$$g\left(\mu\left(u_{01}\right), u_{12}\right) = \text{suff}\left(\beta\left(u_{01} \oplus u_{12}\right), \text{len}\left(u_{01}\right)\right)$$

This is called a realization of the behavior.

Proposition 2.14. Given a behavior function, then corresponding memory, output, and state transition functions exist.

Proof. We will proceed by construction. Let μ be the identity function (i.e., $\mu\left(u\right) = u$), and define the output and state transition functions as

$$f\left(x, u_{12}\right) = x \oplus u_{12}$$

$$g\left(x, u_{12}\right) = \text{suff}\left(\beta\left(x \oplus u_{12}\right), \text{len}\left(x\right)\right)$$

Since μ is the identity, the functions obviously satisfy the requirement. □

Definition 2.15. The *state space* is the image of the memory function. (Sometimes this is called the *reachable* state space, to distinguish it from some easily expressed superset that is called the state space for convenience.)

The output function condition shows that the value of the memory function satisfies the classical definition of state, namely, that it, along with the input, is all the information needed (by the output function) to compute system output indefinitely into the future.

Intuitively, the memory function returns the state arrived at by some input sequence, starting from a fixed initial state. This state summarizes all information in past input that is relevant to future output.

The memory, state transition, and output functions exist but are not unique. Indeed, the construction used in the proof, called the free realization, is an "inefficient" solution. In practice we prefer a memory function that forgets everything about past input except what is essential to future system behavior.

Definition 2.16. There is a distinguished state in the state space $x_0 = \mu(\Lambda)$ called the *initial state*.

Observe that a given state transition equation, with knowledge of the initial state, determines the memory function.

$$f(x_0, u) = f(\mu(\Lambda), u) = \mu(\Lambda \oplus u) = \mu(u)$$

Therefore for many applications, the memory function never need be mentioned.

Remark 2.17. An alternative formulation of state transition function requirement that does not explicitly refer to the memory function. Since $\mu(u) = f(x_0, u)$, then the requirement

$$f(\mu(u_{01}), u_{12}) = \mu(u_{01} \oplus u_{12})$$

holds whenever

$$f(f(x_0, u_{01}), u_{12}) = f(x_0, u_{01} \oplus u_{12})$$

holds.

Proposition 2.18. $g(x_0, u) = \beta(u)$

Proof.

$$\begin{aligned} g(x_0, u) &= g(\mu(\Lambda), u) \\ &= \text{suff}(\beta(\Lambda \oplus u), \text{len}(\Lambda)) \\ &= \text{suff}(\beta(u), \text{len}(\Lambda)) \\ &= \text{suff}(\Lambda \oplus \beta(u), \text{len}(\Lambda)) \\ &= \beta(u) \end{aligned}$$

Therefore, when an output equation and initial state are known, we do not necessarily ever need to refer to the behavior function. □

Remark 2.19. An alternative formulation of the output function requirement that does not explicitly refer to the memory or behavior functions is

$$g(\mu(u_{01}), u_{12}) = \text{suff}(\beta(u_{01} \oplus u_{12}), \text{len}(u_{01}))$$

$$g(f(x_0, u_{01}), u_{12}) = \text{suff}(g(x_0, u_{01} \oplus u_{12}), \text{len}(u_{01}))$$

Interestingly, we have not yet made any explicit mention of causality.

Definition 2.20. *Behavioral causality* is the property that

$$\beta(u_{01} \oplus u_{12}) = \beta(u_{01}) \oplus \text{suff}(\beta(u_{01} \oplus u_{12}), \text{len}(u_{01}))$$

For our purposes, we expect a block behavior function to have the property that later inputs will not change earlier outputs. The condition in the definition above means that if

t marks the time where u_{01} ends and u_{12} begins, the output before t cannot be affected by the input after t, though the output after t may depend on the input both before and after t.

Remark 2.21. It might appear that causality is implied by the transition and output function properties. But consider this behavior function.

$$\beta((a)) = (1)$$

$$\beta((a,a)) = (2,2)$$

$$\beta((a,a,a)) = (3,3,3)$$

$$\beta((a,a,a,a)) = (4,4,4,4)$$

$$\vdots$$

It is not causal, but it can still be perfectly well represented by state and output equations that satisfy the properties above.

$$\mu(s) = \text{len}(s)$$

$$f(\mu(s_1), s_2) = \mu(s_1) + \mu(s_2)$$

$$g(\mu(s_1), s_2) = \underbrace{(\mu(s_1) + \mu(s_2), \ldots, \mu(s_1) + \mu(s_2))}_{\text{len}(s_2)}$$

Thus, an underlying transition and output function realization does not, by itself, imply behavioral causality.

Next we show that a causality property in the behavior function implies a similar property in the realization.

Proposition 2.22. The transition-output system realization of a causal behavior, is causal.

Proof. First,

$$\begin{aligned}\beta\left(u_{01} \oplus \left(u_{12} \oplus u_{23}\right)\right) &= \beta\left(u_{01}\right) \oplus \operatorname{suff}\left(\beta\left(u_{01} \oplus u_{12} \oplus u_{23}\right), \operatorname{len}\left(u_{01}\right)\right) \\ &= \beta\left(u_{01}\right) \oplus g\left(\mu\left(u_{01}\right), u_{12} \oplus u_{23}\right)\end{aligned}$$

Also,

$$\begin{aligned}\beta\left(\left(u_{01} \oplus u_{12}\right) \oplus u_{23}\right) &= \beta\left(u_{01} \oplus u_{12}\right) \oplus \operatorname{suff}\left(\beta\left(u_{01} \oplus u_{12} \oplus u_{23}\right), \operatorname{len}\left(u_{01} \oplus u_{12}\right)\right) \\ &= \beta\left(u_{01} \oplus u_{12}\right) \oplus g\left(\mu\left(u_{01} \oplus u_{12}\right), u_{23}\right) \\ &= \beta\left(u_{01}\right) \oplus \operatorname{suff}\left(\beta\left(u_{01} \oplus u_{12}\right), \operatorname{len}\left(u_{01}\right)\right) \oplus g\left(\mu\left(u_{01} \oplus u_{12}\right), u_{23}\right) \\ &= \beta\left(u_{01}\right) \oplus g\left(\mu\left(u_{01}\right), u_{12}\right) \oplus g\left(\mu\left(u_{01} \oplus u_{12}\right), u_{23}\right)\end{aligned}$$

Taking the $\operatorname{suff}\left(\cdot, \operatorname{len}\left(u_{01}\right)\right)$ of each of these, we get

$$g\left(\mu\left(u_{01}\right), u_{12} \oplus u_{23}\right) = g\left(\mu\left(u_{01}\right), u_{12}\right) \oplus g\left(\mu\left(u_{01} \oplus u_{12}\right), u_{23}\right)$$

$$g\left(x_1, u_{12} \oplus u_{23}\right) = g\left(x_1, u_{12}\right) \oplus g\left(f\left(x_1, u_{12}\right), u_{23}\right)$$

The final equation is the output causality relation. □

System causality can be extended by induction to show that the output function has a distributive property over splices.

Lemma 2.23. A causal output equation distributes over splices.

Proof. The property itself is the base case. The inductive step:

$$g\left(x_0, u_{01} \oplus u_{12} \oplus \cdots \oplus u_{ab}\right) = g\left(x_0, u_{01}\right) \oplus g\left(x_1, u_{12}\right) \oplus \cdots \oplus g\left(x_a, u_{ab}\right)$$

This implies

$$\begin{aligned} g\left(x_0, u_{01} \oplus u_{12} \oplus \cdots \oplus \left(u_{ab} \oplus u_{bc}\right)\right) &= g\left(x_0, u_{01}\right) \oplus g\left(x_1, u_{12}\right) \oplus \cdots \oplus g\left(x_a, \left(u_{ab} \oplus u_{bc}\right)\right) \\ &= g\left(x_0, u_{01}\right) \oplus g\left(x_1, u_{12}\right) \oplus \cdots \oplus g\left(x_a, u_{ab}\right) \oplus g\left(x_b, u_{bc}\right) \end{aligned}$$

Where $x_1 = f\left(x_0, u_{01}\right)$, $x_2 = f\left(x_1, u_{12}\right)$, etc. □

Proposition 2.24. System causality implies behavior causality.

Proof. Since system causality holds, it holds from the initial state x_0.

$$g\left(x_0, u_1 \oplus u_2\right) = g\left(x_0, u_1\right) \oplus g\left(f\left(x_0, u_1\right), u_2\right)$$

The state transition from the initial state is an alias for the memory function.

$$g\left(x_0, u_1 \oplus u_2\right) = g\left(x_0, u_1\right) \oplus g\left(\mu\left(u_1\right), u_2\right)$$

Relating output and behavior, and the output function property,

$$\beta\left(x_0, u_1 \oplus u_2\right) = \beta\left(x_0, u_1\right) \oplus \operatorname{suff}\left(\beta\left(u_1 \oplus u_2\right), u_1\right)$$ □

Proposition 2.25. The state transition function requirement and output causality relation

together imply that the output function requirement holds.

Proof.

$$\begin{aligned}\text{suff}\left(g\left(x_0, u_{01} \oplus u_{12}\right), \text{len}\left(u_{01}\right)\right) &= \text{suff}\left(g\left(x_0, u_{01}\right) \oplus g\left(f\left(x_0, u_{01}\right), u_{12}\right), \text{len}\left(u_{01}\right)\right) \\ &= g\left(f\left(x_0, u_{01}\right), u_{12}\right)\end{aligned}$$

□

2.3.7 Linear systems example

Consider a linear system of the form:

$$\dot{x}\left(t\right) = Ax\left(t\right) + Bu\left(t\right)$$

$$y\left(t\right) = Cx\left(t\right) + Du\left(t\right)$$

The behavior is $y = \beta\left(u\right)$ where the output segment y has a duration same as u and a function given by the solution of the above system of ordinary differential equations, namely,

$$y\left(t\right) = C\left(e^{(t-t_0)A}x_0 + \int_{t_0}^{t} e^{(t-\tau)A}Bu\left(\tau - t_0\right)d\tau\right) + Du\left(t\right)$$

Viewed as a convolution integral, the upper integration limit of t ensures that the argument $t-\tau$ to the convolution kernel is always nonnegative, making this a causal behavior function.

A good candidate for the state transition function in the sense meant here is [47]:

$$f\left(x_0, u\right) = e^{(t-t_0)A}x_0 + \int_{t_0}^{t} e^{(t-\tau)A}Bu\left(\tau - t_0\right)d\tau$$

This function satisfies the state transition requirement: we know that

$$\begin{aligned} &f\left(f\left(x_0, u_{01}\right), u_{12}\right) = f\left(e^{(t_1-t_0)A}x_0 + \int_{t_0}^{t_1} e^{(t_1-\tau)A}Bu_{01}\left(\tau - t_0\right)d\tau, u_{12}\right) \\ &= e^{(t_2-t_1)A}\left(e^{(t_1-t_0)A}x_0 + \int_{t_0}^{t_1} e^{(t_1-\tau)A}Bu_{01}\left(\tau - t_0\right)d\tau\right) + \int_{t_1}^{t_2} e^{(t_2-\tau)A}Bu_{12}\left(\tau - t_1\right)d\tau \\ &= e^{(t_2-t_0)A}x_0 + \int_{t_0}^{t_1} e^{(t_2-\tau)A}Bu_{01}\left(\tau - t_0\right)d\tau + \int_{t_1}^{t_2} e^{(t_2-\tau)A}Bu_{12}\left(\tau - t_1\right)d\tau \end{aligned}$$

But also (and using the formal definition of the splice),

$$\begin{aligned} &f\left(x_0, u_{01} \oplus u_{12}\right) = e^{(t_2-t_0)A}x_0 + \int_{t_0}^{t_2} e^{(t_2-\tau)A}B\left(u_{01} \oplus u_{12}\right)\left(\tau - t_0\right)d\tau \\ &= e^{(t_2-t_0)A}x_0 + \int_{t_0}^{t_1} e^{(t_2-\tau)A}Bu_{01}\left(\tau - t_0\right)d\tau + \int_{t_1}^{t_2} e^{(t_2-\tau)A}Bu_{12}\left(\tau - t_1\right)d\tau \end{aligned}$$

Therefore,

$$f\left(x_0, u_{01} \oplus u_{12}\right) = f\left(f\left(x_0, u_{01}\right), u_{12}\right)$$

Since the behavior is causal and the state transition requirement holds, it follows that the output function requirement also holds. Therefore we have established the state transition and output functions for the behavior of a system of linear time-invariant differential equations.

2.4 Timed Event Systems Theory

A timed event system, like any system, accepts an input signal and produces an output signal. The signals are of a novel type: they do not consist of quantities varying in time, but of events occurring at associated times. (This TES-specific signal type is called a timeline.) The events do not have any special restrictions on them. In practice they are implemented as arbitrary software objects (in the sense of object-oriented programming).

The computation of outputs from inputs is also quite free, but one important constraint relied on here is causality, namely, an input event at some time cannot produce an output event at some earlier time.

Causality follows naturally from a TES block when its behavior is described not explicitly with a behavior function, but rather in terms of a more convenient function called the event response function (ERF).

There is a difference of emphasis or viewpoint between the behavior and ERF descriptions of a block. Behavior emphasizes a relationship that holds among the input and output segments "all at once." The ERF viewpoint emphasizes the pointwise dynamic evolution of the output as a response to the input.

2.4.1 Event response function and external dynamics

Here we define the ERF and show that it defines a behavior.

In the ERF viewpoint, the state of a block is partitioned into two parts, called the external state q and internal state x, for a total state (x, q). The rationale for this is to support the partitioning of block dynamics into external dynamics and ERF (or internal dynamics), as described below.

Intuitively, the external state contains pending output events scheduled by the block. The internal state contains any relevant summary of input history, and is private to the block.

The ERF computes the changes to total block state caused by some input event e.

Therefore, an input event can affect both external state (e.g., it can schedule an output event) and internal state.

$$x', q' = h\,(x, q, e)$$

Clearly the ERF by itself is not enough to define a behavior, because it does not account for the passage of time. The effect of the simple passage of time, i.e. a stretch of time in which no input events occur, is modeled with external dynamics. External dynamics is not dependent on internal state and cannot affect internal state. Over the lapse of time τ the block output segment y_τ is computed. Expressing external dynamics as a function H,

$$y_\tau, q' = H\,(q, \tau)$$

The above expression is necessary but not sufficient to constrain H; by itself it ascribes H too much freedom. But if q is modeled formally as a timeline type (which is quite appropriate), then external dynamics can be precisely defined as

$$y_\tau = \text{pref}\,(q, \tau)$$

$$q' = \text{suff}\,(q, \tau)$$

The interpretation of this is straightforward. In a lapse of time τ, any pending output events scheduled to occur in the next τ time units appear on the output. The remaining pending events are those that are scheduled to occur after τ. The waiting time for all of these events is reduced by τ (this is the effect of the suff).

Note that since in this model the passage of time does not affect internal state, a block cannot simulate something as simple as an analog clock. If a block needs to know the time of an event arrival, this information must be included with the event.

2.4.2 TES block behavior

We take for granted that an input segment is always equal to some splicing together of pure events and pure time lapses. (An argument that such a decomposition is possible is as follows: as a segment, the timeline is a tuple of a function and a duration; the domain of the function is totally ordered; therefore the events can be totally ordered (i.e., can be sorted; the number of events is finite) and the time delays between them can be computed.)

Given such a decomposition of the input segment, we can compute the corresponding output segment.

Let the decomposition of the input segment be:

$$u_1 \oplus u_2 \oplus \cdots \oplus u_n$$

where each u_i is either a time lapse segment or a segment containing one event (NB a segment containing one event has length $(1, 0.0)$, which is not a length of zero in the timeline type). Then

$$f\left((x,q),u_i\right) = \begin{cases} h\left(x,q,e\right) & \text{if } u_i = \text{Event}\left(e\right) \\ \left(x, \text{suff}\left(q,\tau\right)\right) & \text{if } u_i = \text{Lapse}\left(\tau\right) \end{cases}$$

$$g\left((x,q),u_i\right) = \begin{cases} \Lambda & \text{if } u_i = \text{Event}\left(e\right) \\ \text{pref}\left(q,\tau\right) & \text{if } u_i = \text{Lapse}\left(\tau\right) \end{cases}$$

With $u = u_1 \oplus u_2 \oplus \cdots \oplus u_n$ and using the distributive property of the output function over splices, the behavior follows immediately as $\beta\left(u\right) = g\left(\left(x_0, q_0\right), u\right)$.

The question remains of how to orchestrate internal and external dynamics in an interconnection of blocks, so that the resulting outputs for all blocks satisfy the interconnection's block diagram equation.

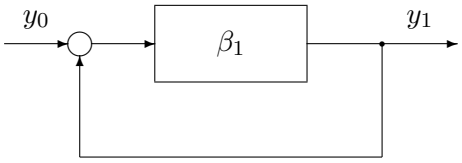

Figure 2.4: Block diagram of the simulation example

2.4.3 TES simulation example

Here we simulate a simple system (Figure 2.4) by hand. The block β_1 is a unit delay.

The block diagram equation for this system is

$$y_1 = \beta_1 (y_0 + y_1)$$

Let the exogenous input y_0 be a timeline of length 2.5 with a single event a at 0.0.

$$y_0 = (\{(0.0, a)\}, 2.5)$$

Event a at 0.0 passes through the summing junction to the delay block. The delayed version appears at 1.0 on y_1. This feeds back through the summer to the block, producing a delayed version on y_1 at 2.0. This also feeds back and causes the delay block to schedule another output at 3.0, but since the simulation runs for only 2.5 seconds, this last output event never appears (instead it remains pending in q_1). The total output of the simulation is

$$y_1 = (\{(1.0, a), (2.0, a)\}, 2.5)$$

Note that y_1 could not be computed by a simpleminded application of β_1, because the input $y_0 + y_1$ to the block over the whole input interval was not known before the simulation was run. (It is interesting to note that a solution using β_1 alone does exist, though it is more expensive: fixed point iteration [48] of β_1, starting from $y_1 = (\{\}, 2.5)$, arrives at the solution in a finite number of steps. A related technique for differential equations is called Picard iteration, which converges in the limit [49].)

However, a given solution can be verified by a straightforward application of β_1. We give an example of how the block behaviors can verify the solution. Suppose we were given the above solution without explanation and asked if it does indeed satisfy the block diagram equation. Since this diagram has only one block, there is only one function to check. The processing performed by a delay block can be written down by inspection. Does the solution y_1 computed by the simulation indeed satisfy the block diagram equation $y_1 = \beta_1 (y_0 + y_1)$?

$$y_0 + y_1 = (\{(0.0, a), (1.0, a), (2.0, a)\}, 2.5)$$

$$\beta_1 (y_0 + y_1) = (\{(1.0, a), (2.0, a)\}, 2.5)$$

$$= y_1$$

Therefore, the block diagram equation is satisfied by the simulation output.

These arguments are formalized and made general in the next section.

2.5 Simulation Solves a TES Diagram

Here we will show that the well-known DES/TES simulation algorithm solves the block diagram.

Here the specific notations for internal (ERF) and external dynamics are wrapped up by the behavior function. However, the algorithm presented is TES-specific, since the total external state must be examined to determine the time of the next pending event.

2.5.1 The pending event

As far as the algorithm here is concerned, the critical feature of a TES is that the time of the next event can always be determined. We simply assume this capability here, but note in passing that while it is often simple to implement in software, it is not a triviality;

references [50–52] are entry points to the interesting literature on this subject.

Knowing the next event (including its time of occurrence) means that all output signals are known from the current time up to the event time, namely, they are all uneventful except for one, on which the pending event occurs at the end of the interval.

2.5.2 Simulation algorithm

Suppose the total diagram is in state (X, Q), where X is the collection of the block internal states and Q is the collection of block external states (i.e., pending output events).

From Q we can determine the next event e_i, its time of occurrence t_i, and the block k sending it.

Denote the output of block k from the previous to the current event as $y_{k,i}$. Typically this will be a segment of length $t_i - t_{i-1}$ containing exactly one event e_i at time t_i. The outputs $y_{j,i}$ for all other blocks $j \neq k$ are uneventful on this interval.

This determines all outputs for this interval. The outputs determine the inputs for this interval. How do we know these inputs will not include some new interfering event? An interfering event could not be caused by e_i because the blocks are causal. And, an interfering event could not have been scheduled by some other block because this would contradict the determination that e_i was the next pending event. Therefore, the outputs satisfy the block diagram equations on the current interval.

Processing the event e_i removes it from the sender's queue and updates the receive block's state to (x_i, q_i). Iterate until there is no pending event, upon which the algorithm terminates, returning the outputs y_n for all blocks, spliced together across the intervals.

2.5.3 Correctness of algorithm output

A straightforward application of the above lemmas (namely, output equation distributes over splices; a splice of overlays equals an overlay of splices) shows that this simulation output solves the diagram, i.e., that satisfying the implicit relation per block and per interval as the algorithm does, is sufficient to satisfy it for all blocks in the diagram over the entire

simulation timeline.

$$
\begin{aligned}
y_n &= g_n\left(x_0, \sum\nolimits_{m\in I_n} y_{m,1}\right) \oplus g_n\left(x_1, \sum\nolimits_{m\in I_n} y_{m,2}\right) \oplus \cdots \oplus g_n\left(x_{N-1}, \sum\nolimits_{m\in I_n} y_{m,N}\right) \\
&= g_n\left(x_0, \sum\nolimits_{m\in I_n} y_{m,1} \oplus \sum\nolimits_{m\in I_n} y_{m,2} \oplus \cdots \oplus \sum\nolimits_{m\in I_n} y_{m,N}\right) \\
&= g_n\left(x_0, \sum\nolimits_{m\in I_n} y_{m,1} \oplus y_{m,2} \oplus \cdots \oplus y_{m,N}\right) \\
&= g_n\left(x_0, \sum\nolimits_{m\in I_n} y_m\right)
\end{aligned}
$$

Chapter 3: FSM Identification

This chapter is not yet about timed event systems, but rather about finite state machines. We review the problem of FSM identification, and then consider in detail how the problem is solved by an adaptation of a classic result on learning regular languages. The FSM identification algorithm presented here includes the original language learning algorithm as a special case. Although the differences in the algorithm and its proofs between the original and generalized versions are relatively minor, the broadening of applicability is substantial, a theme to be continued in the following chapter.

3.1 From language learning to FSM identification

A major goal of computational learning theory (CoLT) is learning from examples the rule that characterizes a subset of a known set.

For example, a central problem is the inference of a Boolean function from input-output examples. In this case the set consists of all possible input variable valuations, and the subset consists of those valuations that evaluate to "true." Reconstructing the Boolean function is straightforward in principle if the output to every possible input is considered, except that this involves exponentially many terms. A key concern is efficiency (computational complexity) and its related questions such as, how "learnable" are different classes of Boolean functions? Is an algorithm guaranteed to converge to the correct answer, and if so how quickly? Questions such as this touch on deep topics in computer science [53] as well as being of some practical interest [54].

A regular language is a special form of subset of the closure of an alphabet. Because of its simplicity, familiarity, and importance in computer science, learning a regular language is an interesting CoLT problem. In [23], Gold showed that inferring a minimal automaton

from a given set of example strings, labeled with an indicator of membership in the regular language, is intractable. In 1987 Angluin developed an algorithm that efficiently identifies a regular language [24]. This positive result was made possible by making different assumptions than Gold — in contrast to a given a priori set of examples, Angluin allows the algorithm to query for the language membership of arbitrary strings of its own devising.

CoLT has not gotten much attention from the system identification community, perhaps because, as noted in [55], traditional CoLT "is aimed at trying to learn a 'static' set or function."

Now when considering an infinite set such as a regular language, it is apparent that an algorithm does not represent this language literally as an infinite set (for which finite time would not suffice, let alone finite memory), but rather uses some finite representation.

An automaton is an important finite representation of a regular language. And while the language represented by an automaton is a static set, the automaton itself is a dynamical system! Therefore, a solution to a static language learning problem has a dynamical systems interpretation.

In fact, tracing back through the references (especially from [24] to [56] to [57]), the ideas leading up to Angluin's algorithm have their roots in automata and abstract systems theory.

This observation does not diminish Angluin's contribution in the slightest. From the systems identification viewpoint, her paper originates two crucial advances: the algorithm is efficient, and (in its PAC variant) does not require access to system internals (i.e., to current state, the state space, or even state space size).

What is the difference between Angluin's algorithm and the one presented here? The output of Angluin's algorithm is a regular language represented as an automaton. An automaton can be considered a special case of a finite state machine, namely, an FSM with a binary output alphabet. The algorithm here differs from Angluin's in that the output alphabet is not assumed to be binary. In this sense, it is a strict generalization of Angluin's result. Similarly, the proof given here simply shows that Angluin's proof more or less goes

through without assuming a binary output.

In the algorithm and proofs this is a modest change. But the resulting broadening of applicability is appreciable, as is the shift in viewpoint: we go from identifying a regular language to identifying a machine, and from querying membership in a language to performing an experiment on the unknown system.

Another modest novelty is the explanation of algorithm behavior in terms of observation tree diagrams. While diagramming system evolution as a tree is hardly novel [58,59], using a tree to visualize the meaning of approximate Nerode equivalence does not seem to appear in the tutorial literature on Angluin's algorithm, and is important to intuitive understanding and therefore to application and adaptation of the algorithm.

3.2 The finite state machine

Definition 3.1. A finite state machine of Moore type is a tuple $(Q, q_0, \delta, \gamma, A, Z)$ of:

- A finite set Q of states, and an initial state $q_0 \in Q$
- A finite set A of input symbols
- A finite set Z of output symbols
- A transition function $\delta : Q \times A \to Q$ that, given a current state and current input, returns the next state
- An output function $\gamma : Q \to Z$ that associates an output value with each state

The set A is called the input alphabet, Z the output alphabet, and Q the state space. Sometimes the input and output alphabets are implicit in the definitions of δ and γ, respectively, so the FSM can also be given as (Q, q_0, δ, γ).

A finite state machine is commonly diagrammed as in Figure 3.11. In that figure there are three states. The initial state q_0 is depicted with a heavy stroke (initial state is also commonly designated with an unlabeled incoming arrow or other device [60]). If the uppermost state in the diagram is q_1, then for an input symbol of 0, from q_0 we follow the

outward arrow labeled 0 to q_1. This corresponds to a transition function that evaluates $\delta(q_0, 0) = q_1$.

Also, the output of q_0 is 1 and the output of q_1 is 0; the states in the diagram are labeled with their outputs. In the formal model these facts correspond to an output function that evaluates $\gamma(q_0) = 1$, $\gamma(q_1) = 0$.

We call the above model *internal* since it explicitly uses a state $q \in Q$ that in general cannot be uniquely determined from the output. A strictly behavioral or *external* model can be constructed from the internal model as follows.

First, it is convenient to extend the definition of the transition function from input symbols to input sequences, i.e., to extend its domain from a state and an input symbol to a state and an input sequence.

Definition 3.2.

$$\delta(q, \Lambda) \triangleq q$$

$$\delta(q, a \cdot s) \triangleq \delta(\delta(q, a), s)$$

This then allows us to immediately write an external description, given by $\beta'(u) = \gamma(\delta(q_0, u))$. (We will often write the latter expression as $\gamma\delta(u)$ for brevity.) This function returns the final symbol output by the FSM in processing input sequence u. This external model is a good model of an FSM in a black box. We can feed it input sequences and see the outputs they generate, but the internal workings of state transitions are completely unknown.

Again referring to the diagram in Figure 3.11, we can read out that for the input sequence 1100, we end up at q_0 whose output is 1. In the formal model this corresponds to $\gamma\delta(1100) = 1$.

3.3 The Identification Problem

The FSM identification problem can be informally stated as: given a behavioral FSM model β', infer a state space, transition function, and output functions satisfying it. By being given β' we mean that we can determine the machine's response to any input sequences. We assume the behavior β' is not arbitrary but is in fact an external model of an FSM.

State spaces such as a Euclidean space come equipped with concepts of linear dependence, a metric, etc. In contrast, the discreteness of an FSM state space leaves it relatively devoid of structure. The state space is a mere set, and the kinds of objects in the set are not important. All that matters is that the set is finite.

The task is to somehow build up a representation of state from the only information available, namely, input-output traces. Most fundamentally, this representation will need to answer identity questions such as deciding whether two states are equal to each other or not. Once this is possible, we can work out the transition and output functions.

Note that our goal is not quite an exact reconstruction of the FSM underlying β' — this is impossible in principle if, for example, the underlying FSM has redundant states. Rather, we aim only for behavioral equivalence.

3.4 Introduction to the algorithm

Here we provide definitions and preliminary discussion of constructs used in the algorithm.

3.4.1 Preliminaries

First we briefly review some definitions and notation.

As used here, a *sequence* is simply a synonym for a tuple. Given a set A, the set A^* (called the Kleene closure of A) consists of sequences of symbols in A. It includes Λ, the sequence of length zero.

We frequently will express concatenation of symbols and sequences with a dot operator. For symbols a_i, a concatenation of symbols denoted $a_1 \cdot a_2 \cdot a_3$ means the sequence (a_1, a_2, a_3)

consisting of those symbols. Let $s = a_1 \cdot a_2 \cdot a_3$ be a sequence; then the concatenation of this sequence and a symbol, denoted $s \cdot a$, means the sequence $a_1 \cdot a_2 \cdot a_3 \cdot a$. Concatenation of sequences s_1 and s_2, denoted $s_1 \cdot s_2$, means the sequence consisting of the symbols in s_1 followed by the symbols in s_2. (Note that sequences built up with the dot operator always consist of symbols, never of other sequences.) Finally, the concatenation of sets S_1 and S_2, denoted $S_1 \cdot S_2$, is a set consisting of all $s_1 \cdot s_2$ for all $s_1 \in S_1$ and $s_2 \in S_2$.

For any sequence s, we have that $s \cdot \Lambda = \Lambda \cdot s = s$.

A set of sequences is *prefix-closed* whenever all prefixes of every sequence in the set are also in the set.

Similarly, a set of sequences is *suffix-closed* whenever all suffixes of every sequence in the set are also in the set.

The *prefix closure* of a set S is a set containing all the sequences in S, and all the prefixes of all the sequences in S. The *prefix reduction* of a set S is a subset of S in which no element is a prefix of another element.

Set difference is denoted with a minus sign.

3.4.2 Data structures

The algorithm maintains three structures: the sets S and E, and the function T. These are not fixed, but change as the algorithm executes.

The set S consists of sequences in the input alphabet. A sequence in S can be thought of as representing the state reached by that sequence starting from the initial state. (Note that different sequences in S may reach the same state.)

The set E also consists of sequences in the input alphabet. The sequences in E represent distinguishing experiments for states. They give the subtree on which state behavior is compared.

The algorithm stores the results of its experiments in a function T. The domain of T is the set of experiments performed so far, and T evaluates to the result of the experiment, namely, the final output symbol. Thus T captures a subset of the SUT's behavior. As more

	0	0				
1	0	1				
	1	0	0	0	0	
1	0	0	0	0	0	
	1	0	0	1		
1	0	0	0	1		
	1	0	1	0	0	
1	0	0	0	1	0	
	1	0	1	1		
1	0	0	0	0		
	1	1	0	1	0	0
1	0	1	0	0	0	0
	1	1	0	1	1	
1	0	1	0	0	0	

Figure 3.1: Input-output traces.

experiments are performed, the domain of T grows.

3.4.3 Discussion: tree diagram

Here we discuss representation and diagramming of input-output information stored in S, E, T.

In Figure 3.1, we see a diagram of raw input-output traces. The input symbol is recorded above the line and the corresponding output symbol is recorded below the line. These traces always begin from the system's initial state.

This same data is redrawn in Figure 3.2. Since the system is deterministic, the same input sequences always lead to the same output sequences. Also, since the system is causal, later inputs cannot affect earlier outputs. Therefore many input-output traces have an early portion in common. This figure is derived from Figure 3.1 by merging common input-output portions from different traces onto a single line. For example, the last two traces in Figure 3.1 both begin with input sequences 1,1,0,1 and necessarily have equal output sequences on this interval. Therefore the same information appears in Figure 3.2 collapsed onto a single

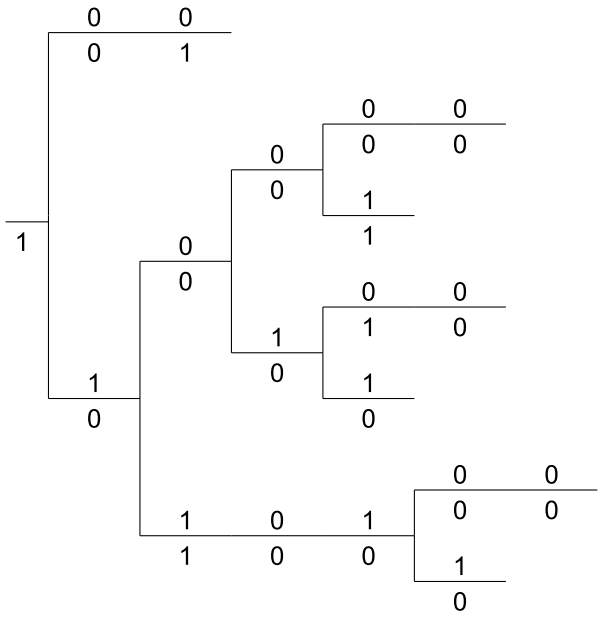

Figure 3.2: Since the system is deterministic and causal, the traces can be redrawn as a tree.

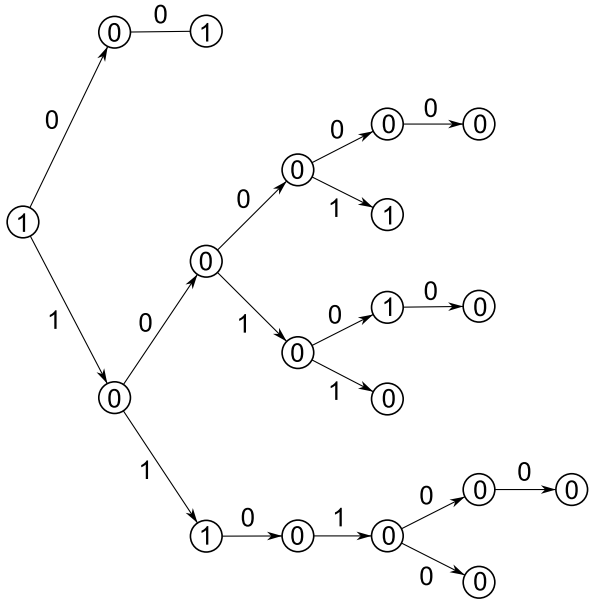

Figure 3.3: The input-output data resembles an FSM.

line. Note that the first input symbol is given a line shared by all traces that begin with that symbol.

The same data is redrawn yet again in Figure 3.3. This is done simply to make clear the similarity of an observation tree with an FSM. The similarity is that we can interpret the diagram as having nodes connected by transitions with associated inputs and outputs. The difference is that an FSM has transitions and outputs defined for every state, but the observation tree does not. In particular, while numerous nodes near the root or interior of the tree do have all transitions defined, the leaf nodes have no transitions defined. Since a finite tree always has leaf nodes, and leaf nodes have no transitions, a finite tree is not an FSM. But as will be shown, under appropriate conditions it can be transformed into an FSM.

3.4.4 Discussion: characterizing state by behavior

When a system is described to us in a way not explicitly involving state, is state still a meaningful concept? To infer a state-based model from input-output sequences we must address this question.

Applying an input sequence to a system from its initial state, drives it to some state. Certainly we can associate in some sense the resulting state with the input sequence that led to it. But it would be inadequate to define state in terms of the sequence leading to it, because then it would not be possible to have two different input sequences that lead to the same state.

If state is hidden from us, how would we ever know that two different input sequences led to the same state in a system? If two input sequences lead to the same state, then system behavior will be the same from this point on, regardless of which input sequence was used. This is an informal expression of the concept of Nerode equivalence of states.

Subtrees with common behavior from the starting node are indicated in Figure 3.4. It is entirely possible for two nodes in the observation tree to represent the same state, i.e., they diagram the same state being reached by different input sequences. In fact, for an

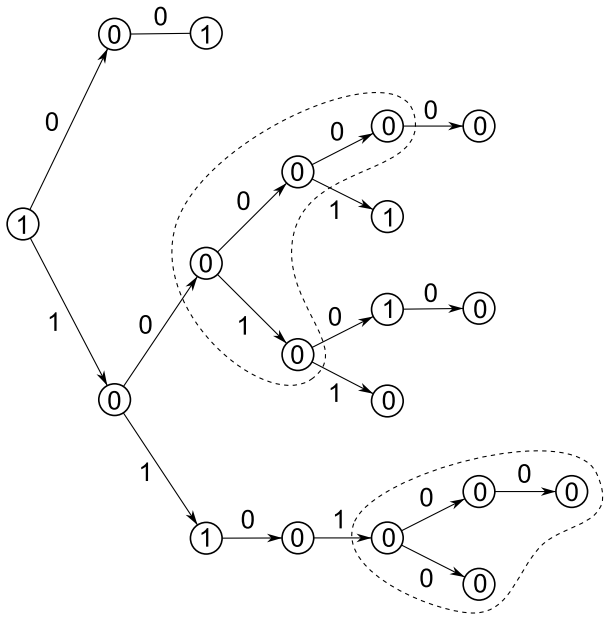

Figure 3.4: The dashed regions indicate subtrees with common behavior. These subtrees consist of input sequences Λ, 0, 00, 1.

SUT with a finite number of states, this eventually must happen. The critical idea here for determining when two nodes represent the same state is the characterization of state by behavior.

If two nodes produce different outputs for the same input sequences, they certainly do not represent the same state. If they produce the same outputs for several input sequences they might be the same state, or they might be different with their difference not yet discovered by an input sequence that distinguishes them. All we know is that there is no evidence that demonstrates that they represent different states.

Consider the nodes in Figure 3.4 reached by input sequences 10 and 1101. The dashed regions show that system behavior matches from these points on subsequent inputs Λ, 0, 00, 1. True Nerode equivalence would have them match for all possible subsequent inputs. The more limited concept of equivalence is the type used in this work; its advantage is that it can be checked by finite experimentation.

The observation tree viewpoint and the concept of equivalent states provides great insight into the FSM identification problem, and additionally highlights several challenges.

- Under what conditions can an observation tree be transformed into an FSM?
- How do we build up an observation tree such that these conditions are met?
- How do we know the resulting FSM is consistent with the tree it was derived from?

All of these questions will be addressed in what follows.

3.4.5 The sequence characterization function

We define a sequence characterization function ρ in terms of the observation table (S, E, T). It maps an input sequence to a structure[1] characterizing system behavior following that sequence, namely, its behavior in response to E. It is defined on $S \cup S \cdot A$.

Definition 3.3.

$$\rho(u) = \{(e_1, T(u \cdot e_1)), (e_2, T(u \cdot e_2)), \ldots, (e_n, T(u \cdot e_n))\}$$

For all elements $e_k \in E$.

It is important to emphasize that since ρ draws from S, E, T, it is *not* defined for all input sequences A^*, but only on $S \cup S \cdot A$.

Equivalence of states is defined in terms of ρ.

Definition 3.4 (Equivalence of states)**.** Let s_1 and s_2 be sequences in S or $S \cdot A$. The states reached by s_1 and s_2 are called *equivalent* whenever $\rho(s_1) = \rho(s_2)$.

Note that $\rho(s_1) = \rho(s_2)$ implies $T(s_1 \cdot e) = T(s_2 \cdot e)$ for all $e \in E$.

[1] Although this interpretation will not be emphasized, it is worth noting that as a set of ordered pairs, $\rho(u)$ satisfies the set-theoretic definition of a function [61], which means that using expressions like $\rho(u)(s)$ would be quite reasonable.

It was earlier noted that S, E, T change as the algorithm runs. Therefore, the definitions of ρ and of equivalence also change as the algorithm runs[2]. If $E = \{\Lambda, 0, 00, 1\}$, then the regions in Figure 3.4 are a schematic of $\rho(10)$ and $\rho(1101)$; since these subtrees are equal, the (states reached by the) sequences 10 and 1101 are equivalent.

3.4.6 Closure and Consistency

Closure and consistency are properties of the table S, E, T that may or may not hold at any given time.

Closure means that every element of $S \cdot A$ is equivalent to some element in S, i.e., there is always an $s \in S$ such that for all $s_1 \in S$ and $a \in A$, $\rho(s_1 \cdot a) = \rho(s)$. Informally, closure means that every next-state is equivalent to some state.

Consistency means that for $s_1, s_2 \in S$, $\rho(s_1) = \rho(s_2)$ implies $\rho(s_1 \cdot a) = \rho(s_2 \cdot a)$ for all $a \in A$. Informally, consistency means that if two states are equivalent, then all of their corresponding next-states are also equivalent.

Closure and consistency are significant because when S, E, T has these properties, it can be transformed into an FSM.

3.4.7 Discussion: diagramming the concepts

Here we will preview the above concepts in action, in terms of the illustrations that accompany the detailed example in section 3.7.

The tree diagrams represent the sets S, E, T at a certain point in algorithm execution. The sequence of edge labels from the initial node (rendered with emphasis) to any node represents an input sequence. The value in a node holds the SUT output in response to the sequence leading to that node. An input sequence, or sequence of arrow labels, is an input to T, and the value in a node is the corresponding output. The nodes to the left of the heavy dotted line correspond to the elements in the set S, i.e., the sequences in S lead to these nodes.

[2]This definition of equivalence stands in contrast to others that commonly appear in the literature, such as equal behavior on all input sequences [57, 62] or on all input sequences up to some length k [16, 62].

The set of nodes in $S \cdot A$, informally called the next-states, are the nodes one step to the right of the nodes in S. They include the "layer" of nodes $(S \cdot A) - S$ that lie immediately to the right of the dashed line.

All nodes taken together correspond to the elements of $(S \cup S \cdot A) \cdot E$, i.e., the complete domain of T. A behavior on the diagram is the subtree reachable by the sequences in E from some starting node in $S \cup S \cdot A$.

For example, in Figure 3.10, $S = \{\Lambda, 0, 1, 11\}$, $S \cdot A = \{0, 1, 00,\ 01,\ 10, 11,\ 110,\ 111\}$, and $T(\Lambda) = 1$, $T(0) = 0$, $T(1) = 0$, $T(00) = 1$, etc.

For a sequence s, the characterization $\rho(s)$ gives the behavior of the system for sequences in E starting from the node s. At the stage of the algorithm's execution in this Figure, $E = \{\Lambda, 0\}$. Behaviors on this E will include only the starting node's output, and the output of the node reached after an input of 0. Therefore, characterizations for the elements of S are:

$$\rho(\Lambda) = \{(\Lambda, 1), (0, 0)\}$$

$$\rho(0) = \{(\Lambda, 0), (0, 1)\}$$

$$\rho(1) = \{(\Lambda, 0), (0, 0)\}$$

$$\rho(11) = \{(\Lambda, 1), (0, 0)\}$$

Note that among these four elements of S, there are only three distinct behaviors. In particular, since $\rho(\Lambda) = \rho(11)$, the sequences Λ and 11 are equivalent.

For another example of ρ we refer to Figure 3.13. By this point in the algorithm's execution, $E = \{\Lambda, 0, 1\}$. Therefore the characterizations at this time will include more behavior than before, namely, output due to the input sequence 1. So for the same arguments as

above, we now have:

$$\rho(\Lambda) = \{(\Lambda, 1), (0, 0), (1, 0)\}$$

$$\rho(0) = \{(\Lambda, 0), (0, 1), (1, 0)\}$$

$$\rho(1) = \{(\Lambda, 0), (0, 0), (1, 1)\}$$

$$\rho(11) = \{(\Lambda, 1), (0, 0), (1, 0)\}$$

There are still only three distinct behaviors on these four nodes (since Λ and 11 are still equivalent), but at a different sequence a fourth behavior has appeared, namely, $\rho(01) = \{(\Lambda, 0), (0, 0), (1, 0)\}$.

We finish this section by highlighting examples of closure and consistency.

Consider Figure 3.6. At this point in the algorithm's execution, $E = \{\Lambda\}$, therefore behavior is compared only on the immediate node output. Both nodes in $S \cdot A$ have outputs of 0. There is only one node in S, and its output is 1. Since there is a sequence in $S \cdot A$ (actually there are two: 0 and 1) that is not equivalent to any sequence in S (because their outputs differ), this S, E, T is not closed.

Consider Figure 3.9. At this point we still have $E = \{\Lambda\}$. The two shaded nodes are equivalent, $\rho(0) = \rho(1)$. For consistency to hold, this requires that we also have $\rho(00) = \rho(10)$ and $\rho(01) = \rho(11)$. But in fact neither of these hold, because the outputs for both pairs are different. Therefore this S, E, T is not consistent.

Finally, consider Figure 3.10. At this point in the algorithm's execution, $E = \{\Lambda, 0\}$, therefore behavior is compared at the immediate node output as well as on the output after an input of 0. In this case all the behaviors in $S \cdot A$ can also be found in S, so S, E, T is closed. Also, for the equivalent sequences 0 and 1, now that E has been augmented, we do have that $\rho(00) = \rho(10)$ and $\rho(01) = \rho(11)$. So this S, E, T is also consistent.

3.4.8 Equivalence of machines

How does the algorithm know if the conjecture matches the SUT? This can be considered a problem of empirically determining if two functions, the SUT model $\beta'(u)$ and the conjecture $\gamma\delta(u)$, are equal for all inputs u. Clearly this cannot be done by enumeration, since the number of tests required would be infinite.

Angluin addressed this problem in two steps. First, it is simply assumed that a query regarding the equivalence of two models can in fact be answered by an "oracle." If the two systems are in fact equivalent the oracle would report it; otherwise it will return an input sequence u that evaluates to a different output in the two models, $\gamma\delta(u) \neq \beta'(u)$, and thus serves as a counterexample to the hypothesis that the conjecture matches the SUT.

In some circumstances such an oracle is in fact available, such as a case where we have access to the underlying SUT, and can therefore check for homomorphism of SUT and conjecture.

However, in applications such as ours no such oracle is available. To handle such cases, we take the step of providing a surrogate for the exact equivalence query. This is a statistical test of equivalence that forswears access to internal structure and relies on experimentation only. In effect, the ideal oracle's testing (in principle) for output equality over every possible input is approximated by empirical testing for output equality over a substantial but finite number of random inputs. This is a widely-used criterion known as Probable Approximate Correctness, or PAC [63].

Informally, the statistical principle at work is, if $\beta'(u) = \gamma\delta(u)$ for a large number of random u's, there is a good chance that $\beta'(u)$ will continue to equal $\gamma\delta(u)$ most of the time (for u's that continue to be drawn from that same random source).

If one of the random inputs leads to a different output, this sequence is returned as the counterexample. Otherwise, once the SUT and conjecture match in behavior a certain number of times, the conjecture is deemed probably approximately correct.

For our identification application, there is no oracle of exact equivalence, and we use the statistical approach exclusively.

Initialization: S and E are sets containing Λ, and T is a function defined on Λ and the elements of A.
Repeat forever: *(main loop)*
  Repeat until (S, E, T) is closed and consistent:
    If (S, E, T) is not consistent, **then** augment E with $a \cdot e$, and extend T.
    If (S, E, T) is not closed, **then** augment S with $s_1 \cdot a$, and extend T.
  Construct FSM M from (S, E, T).
  Compare M to β with the equivalenceTest.
  If M is not equivalent to β, **then** augment S with prefixClosure(counterexample), and extend T;
  Else, return M and terminate.

Figure 3.5: The identification algorithm

3.5 Algorithm

The identification algorithm is outlined in Figure 3.5. It is a working implementation of this algorithm that is referred to in section 5.1.4.

3.5.1 Inputs

The algorithm is given an alphabet A of input symbols and a behavior function β' that takes an input sequence to an output symbol. The function β' is assumed to be an external model of a bona fide FSM. It models the black box or SUT which we can experiment on but whose internals we have no access to. The algorithm is free to assemble input sequences u in A^*, and observe the behavior function's response to them. (Note that in this section, β' always refers to the SUT, and the symbols δ and γ always refer to the current conjecture.)

3.5.2 The query function `extendT`

The algorithm will routinely add new elements to S or E. Since the domain of T is $(S \cup S \cdot A) \cdot E$, experiments must be run to define T for these new sequences. They are

determined by calling β', i.e., by performing an experiment on the SUT. Therefore every call to `extendT` means conducting experiments on the SUT. This operation is centralized in a function called `extendT` in the pseudocode.

This function takes a list of input sequences, applies each input sequence u to β', and then records the output in T, i.e., $T(u) \triangleq \beta'(u)$.

3.5.3 Initialization

Assign the sets S and E their initial values of Λ. Call `extendT` on Λ and the elements of A.

After initialization we enter the main loop. The main loop consists of two major parts: the closure and consistency loop, and the conjecture test.

3.5.4 Closure and consistency loop

The closure and consistency loop iterates until S, E, T is closed and consistent.

First it performs the consistency check. Recall that consistency requires $\rho(s_1) = \rho(s_2)$ to imply $\rho(s_1 \cdot a) = \rho(s_2 \cdot a)$ for all $s_1, s_2 \in S$ and $a \in A$. If the table is not consistent, this means that there are some s_1, s_2, a, e for which $\rho(s_1) = \rho(s_2)$ but $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$. Thus, the consistency check function returns True if S, E, T is consistent, otherwise it returns the s_1, s_2, a, e that violate consistency. If S, E, T is not consistent, then $a \cdot e$ is added to E. This broadens the domain of T, and new experiments need to be run by `extendT` to fill in its values.

Next we perform the closure check. Recall that closure requires every s_a in $S \cdot A$ to be equivalent to some s in S. If S, E, T is not closed, there is some $\rho(s_a)$ not equal to any $\rho(s)$. Thus, the closure check function returns True if S, E, T is closed, or the sequence s_a that violates closure. If S,E,T is not closed, s_a is added to S. This broadens the domain of T, and new experiments need to be run by `extendT` to fill in its values.

If S, E, T was found to be both closed and consistent, we exit the closure and consistency loop and move on to the conjecture test; otherwise we repeat the loop.

3.5.5 FSM Construction and Conjecture Test

Once the table is closed and consistent, we can construct an FSM M consistent with all observations so far. This is our guess or conjecture of the actual system.

Definition 3.5. The FSM M is the tuple (Q, q_0, δ, γ) where:

$$Q = \rho(S)$$

$$q_0 = \rho(\Lambda)$$

$$\delta(\rho(s), a) = \rho(s \cdot a)$$

$$\gamma(\rho(s)) = T(s)$$

It is worth noting that this definition can be readily implemented as an algorithm in a high-level programming language; that equivalent states are "merged" simply by virtue of Q being a set, which cannot contain duplicates; and that the states in this realization are *not* mere integers or labels, but behaviors that can be meaningfully compared for equality.

Next we perform a statistical test comparing the outputs of the SUT behavior $\beta'(u)$ and the conjecture M's behavior $\gamma\delta(u)$ on random input sequences u to determine whether the conjecture is an adequate match to the SUT. We use a formula for the required number r of matches, and then enter a loop that iterates r times. In each iteration, we generate a random input sequence u and pass it to both the SUT and M. If any of these u lead to different outputs between the systems, it is a counterexample disproving the equivalence of the SUT and M — we exit the statistical testing loop, add u's prefix closure to S, and `extendT` to this new domain. Then we repeat the main loop by returning to the closure and consistency step.

On the other hand, if the statistical testing loop terminates after the outputs of the SUT and M matched r times, this means that the SUT and M are PAC equivalent. We

return the machine M as the solution, and terminate.

3.6 Algorithm correctness

Here we demonstrate the correctness of the realization step at the heart of the algorithm, and summarize its algorithmic properties.

3.6.1 Invariants

The set S initially contains only Λ, and is augmented with either $s \cdot a$ (i.e., a sequence already in S appended with a single symbol from A) or the prefix closure of a counterexample; therefore, S is a prefix-closed subset of A^*.

Similarly, E initially contains only Λ, and is augmented with $a \cdot e$ (i.e., a single symbol from A suffixed with a sequence already in E), resulting in it being a suffix-closed subset of A^*.

3.6.2 Conjecture FSM is well-defined

Here we show that the construction in definition 3.5 is well-defined.

Let us obviate a potential problem with $\delta(\rho(u), a) = \rho(u \cdot a)$ and show that it is well-defined.

Proof. Suppose we have two distinct sequences $u_1 \neq u_2$ in S that refer to equivalent states, $\rho(u_1) = \rho(u_2)$. To avoid an inconsistent definition of δ, we must have that $\rho(u_1 \cdot a) = \rho(u_2 \cdot a)$; but this property is exactly what is called for by the consistency requirement.

Also, the range of δ must be a subset of Q, in particular, $\rho(u \cdot a)$ must lie in Q. But this is exactly what is called for by closure. □

We need similar assurances that the output function $\gamma(\rho(u)) = T(u)$ is well-defined. If sequences u_1 and u_2 in S are distinct, but $\rho(u_1) = \rho(u_2)$, how do we know that $T(u_1) = T(u_2)$?

Proof. Since $\rho(u_1) = \rho(u_2)$, it follows that $T(u_1 \cdot e) = T(u_2 \cdot e)$ for any $e \in E$, and since $\Lambda \in E$, then $T(u_1) = T(u_2)$. □

In definition 3.2 we extended the domain of the transition function to sequences. It follows that, as long as $u \in S$, so that $u \cdot a$ is in ρ's domain of $S \cdot A$,

$$\delta(\rho(u), a \cdot s) = \delta(\delta(\rho(u), a), s)$$

$$= \delta(\rho(u \cdot a), s)$$

Proposition 3.6. By induction, the above extends to $\delta(\rho(u_1), u_2 \cdot u_3) = \delta(\rho(u_1 \cdot u_2), u_3)$, as long as $u_1 \cdot u_2 \in S \cup S \cdot A$.

3.6.3 Conjecture FSM matches observations

Proposition 3.7. Starting from the initial state, the transition function maps the accessor sequence to a state, $s_a \in S \cup S \cdot A$, to its corresponding state.

Proof. NB this only makes sense for $s_a \in S \cup S \cdot A$, the domain of ρ.

$$\delta(\rho(\Lambda), s_a) = \delta(\rho(\Lambda \cdot s_a), \Lambda)$$

$$= \rho(\Lambda \cdot s_a)$$

$$= \rho(s_a)$$

where we use result 3.6, definition 3.2, and a concatenation property of Λ. □

Proposition 3.8. For $s_a \in S \cup S \cdot A$, then $\delta(\rho(\Lambda), s_a \cdot u)$ is equal to $\delta(\rho(\Lambda), s_1 \cdot u)$ for some $s_1 \in S$.

Proof.

$$\begin{aligned}\delta(\rho(\Lambda), s_a \cdot u) &= \delta(\rho(s_a), u) \\ &= \delta(\rho(s_1), u) \\ &= \delta(\rho(\Lambda), s_1 \cdot u)\end{aligned}$$

where we use result 3.6, the closure property, and result 3.6 again. □

Proposition 3.9. The machine M is consistent with all observations recorded so far; or, $\gamma(\delta(\rho(\Lambda), s_{ae})) = T(s_{ae})$ for any $s_{ae} \in (S \cup S \cdot A) \cdot E$.

This result is nontrivial because T is defined on $(S \cup S \cdot A) \cdot E$, but the values returned by γ only come from T defined on S. We must take care in the proof to be sure we are not applying these equations where the functions involved are undefined.

Proof. We want to show that $\gamma(\delta(\rho(\Lambda), s_a \cdot e)) = T(s_a \cdot e)$, for any $s_a \in S \cup S \cdot A$, $e \in E$. The proof is by induction.

For the base case we suppose that $e = \Lambda$. We will show that the result holds whether s_a is in S or $S \cdot A$.

- If s_a is in S, then $\gamma(\delta(\rho(\Lambda), s_a)) = \gamma(\rho(s_a)) = T(s_a)$, by result 3.7 and definition 3.5.
- If s_a is in $S \cdot A$, there is by closure some $s_1 \in S$ such that $\rho(s_1) = \rho(s_a)$. So $\gamma(\delta(\rho(\Lambda), s_a)) = \gamma(\rho(s_a)) = \gamma(\rho(s_1)) = T(s_1)$. But $\rho(s_1) = \rho(s_a)$ implies $T(s_1) = T(s_a)$. Therefore, $\gamma(\delta(\rho(\Lambda), s_a)) = T(s_a)$.

Next we address the inductive step. Let e_{k+1} be an element of E of length $k+1$. Since E is suffix-closed, we have the factorization $e_{k+1} = a \cdot e_k$, and e_k is in E. We show that for $s_a \in S \cup S \cdot A$, if $\gamma(\delta(\rho(\Lambda), s_a \cdot e_k)) = T(s_a \cdot e_k)$, then $\gamma(\delta(\rho(\Lambda), s_a \cdot e_{k+1})) = T(s_a \cdot e_{k+1})$.

So what is $\gamma(\delta(\rho(\Lambda), s_a \cdot e_{k+1}))$?

$$\begin{aligned}
\gamma(\delta(\rho(\Lambda), s_a \cdot e_{k+1})) &= \gamma(\delta(\rho(\Lambda), s_1 \cdot e_{k+1})) \\
&= \gamma(\delta(\rho(\Lambda), s_1 \cdot a \cdot e_k)) \\
&= T(s_1 \cdot a \cdot e_k) \\
&= T(s_1 \cdot e_{k+1}) \\
&= T(s_a \cdot e_{k+1})
\end{aligned}$$

Where the steps above follow by applying result 3.8, the definition of e_{k+1}, the inductive hypothesis, the definition of e_{k+1} again, and definition 3.4 (because $\rho(s_a) = \rho(s_1)$ in 3.8).

□

3.6.4 Minimality and termination

Here we give brief summary "proofs" of results that are important in themselves but not a key concern for this work. The results are stated here in weakened form, and many details have been omitted. For a full development of these nontrivial proofs consult [24, 64, 65].

Proposition 3.10. Suppose M' is some FSM consistent with T. This M' does not have fewer states than the realization M.

Proof. The states in Q are not equivalent to each other, because they are all distinguishable by some sequence in E. Let Q' be the state space of M'. Every state in Q is equivalent to some state in Q', because M' must contain states producing the behaviors recorded in T. Therefore there are at least $|Q|$ states in Q' that are not equivalent to each other (because they also are distinguishable by some sequence in E). Therefore there are at least as many states in Q' as in Q. □

Proposition 3.11. The algorithm terminates.

Proof. Any FSM consistent with T must have at least $|\rho(S)|$ states. The underlying unknown system is obviously consistent with T. Suppose its minimal form has n states.

If T is not closed or not consistent, we must increase the size of $|\rho(S)|$ by at least one. Since $|\rho(S)| = 1$ after initialization, it can be increased at most $n-1$ times, after which T must be closed and consistent.

Similarly, if a counterexample is found for a conjecture M, this increases $|\rho(S)|$ by at least one. So an incorrect conjecture can be made at most $n-1$ times.

Since the algorithm can make no more than n closure or consistency adjustments before making a conjecture, and can make no more than n conjectures, the algorithm terminates. □

3.7 Example

Here is an example of the algorithm at work. For simplicity, the input and output alphabets for the SUT are both $\{0, 1\}$. For a peek ahead at what the underlying system is, see the correct final answer produced by the algorithm in Figure 3.14.

A guide to interpreting the diagrams was given in section 3.4.7.

1. The algorithm begins with an initialization. $\beta'(\Lambda)$, $\beta'(0)$, $\beta'(1)$. Then we enter the main loop of the algorithm.

   Our observation tree T at this point is diagrammed in Figure 3.6. This tree is trivially consistent because there is only one node in S. But it is not closed, since there is no node in S whose behavior matches those seen in $S \cdot A$ (at this point E still contains only Λ, the behavior in S is $\{(\Lambda, 1)\}$ but the behaviors in $S \cdot A$ are both $\{(\Lambda, 0)\}$). Therefore we move (the node reached by) sequence 0 into S. (In the figure this is done by redrawing the line.) Since we are now missing next-states (i.e., since S has been augmented, T is no longer completely defined on all of its domain $(S \cup S \cdot A) \cdot E$), we must query for them by running more experiments.

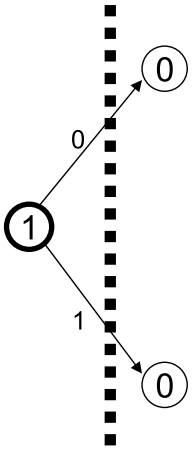

Figure 3.6: Observation tree upon entering main loop.

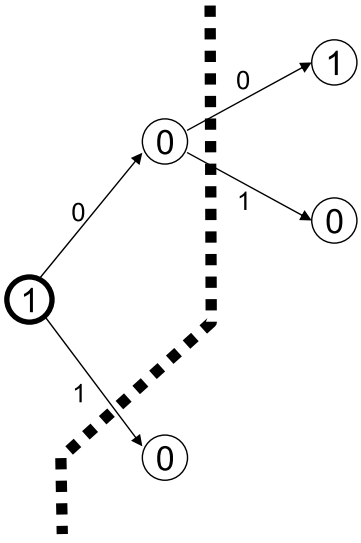

Figure 3.7: A closed and consistent tree with two unique behaviors; $E = \{\Lambda\}$.

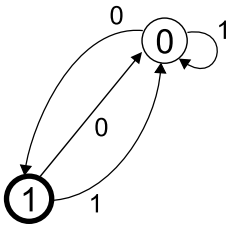

Figure 3.8: The first conjecture, M_1.

2. After querying the SUT with input sequences 00, 01, we arrive at the observation table in Figure 3.7. It is consistent, again trivially, because all behaviors in S are unique; and closed, since all behaviors in $S \cdot A$ appear in S. With closure and consistency established, we realize our first conjecture M_1 from T. (In this case this can be visualized by rerouting every arrow to a node in the $(S \cdot A) - S$ layer to its behavioral equivalent in S.)

3. This conjectured automaton, 3.8, fails the equivalence test. The counterexample is the input sequence 11. In our conjectured system this gives output 0 but the real SUT gives output 1 in response to the same input sequence. So we move the counterexample 11 (and all its prefixes) into S, and run experiments (namely 10, 11, 110,111) to extend the definition of T for its augmented domain.

   The resulting observation table, Figure 3.9, is closed but not consistent, because we have two behaviorally equivalent states in S whose next-states are behaviorally inconsistent, i.e., the two shaded nodes in S have output 0 but one changes to a state with output 1 in response to input 0, the other gives output 0 in response to input 0. (They also differ in response to 1, but we are interested in the fewest inputs that can distinguish among them.)

   The current entries in E do not adequately distinguish these states. But clearly, the input sequence 0 does, so add this input sequence to E, so that now $E = \{\Lambda, 0\}$. Judgments about behavior are no longer restricted to output only (i.e., response to input Λ), but must also take the later response to input 0 into account. Since the

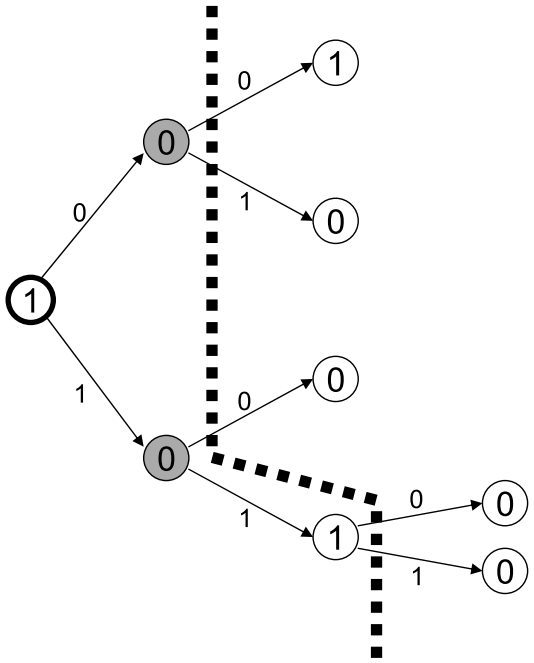

Figure 3.9: The shaded nodes are equivalent but have inconsistent next-states.

domain of T has grown, run membership queries to extend the table.

4. The result (Figure 3.10) is closed — there are three distinct behaviors in $S \cdot A$, and all can be found in S. It is also consistent — there is one pair of sequences in S (namely Λ and 11) with the same behavior, and their corresponding successors also have the same behavior.

   We transform the closed and consistent tree into our second conjecture M_2 (Figure 3.11) and subject it to an equivalence test. This test fails on the counterexample input sequence 011. Our conjecture M_2 predicts an output of 1, but the actual system gives an output of 0. We add this correction to the observation tree, and run more queries to define T on its extended domain.

5. The result is Figure 3.12. This tree is closed but not consistent – the behaviorally equivalent shaded nodes transition to behaviorally inequivalent successors upon input 1. So we add 1 to E and run queries to define T on its extended domain.

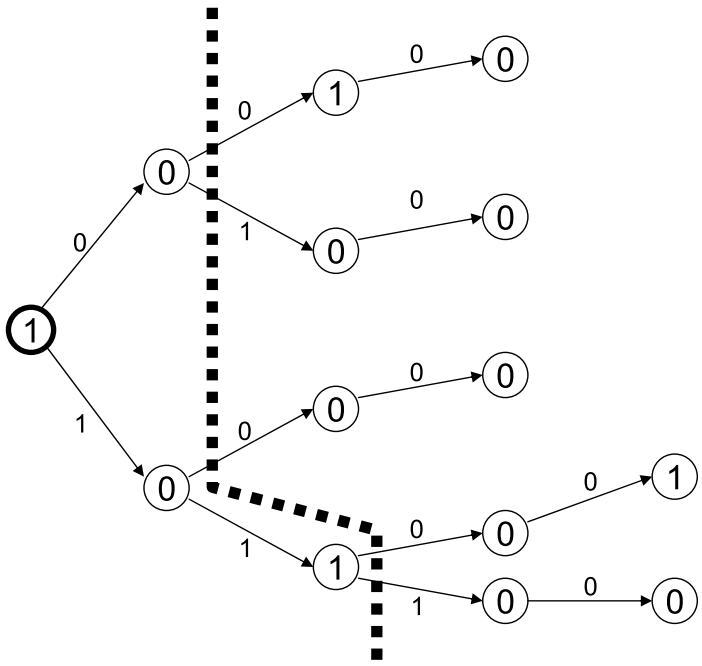

Figure 3.10: A closed and consistent tree with three unique behaviors; $E = \{\Lambda, 0\}$.

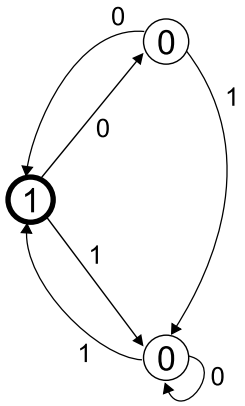

Figure 3.11: The second conjecture, M_2.

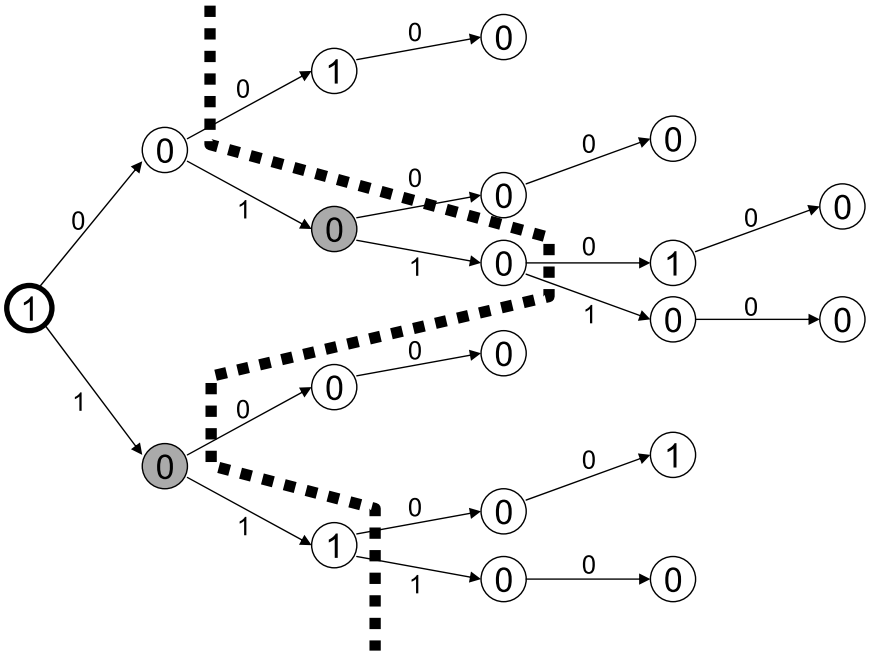

Figure 3.12: Once again, a tree that is closed but not consistent.

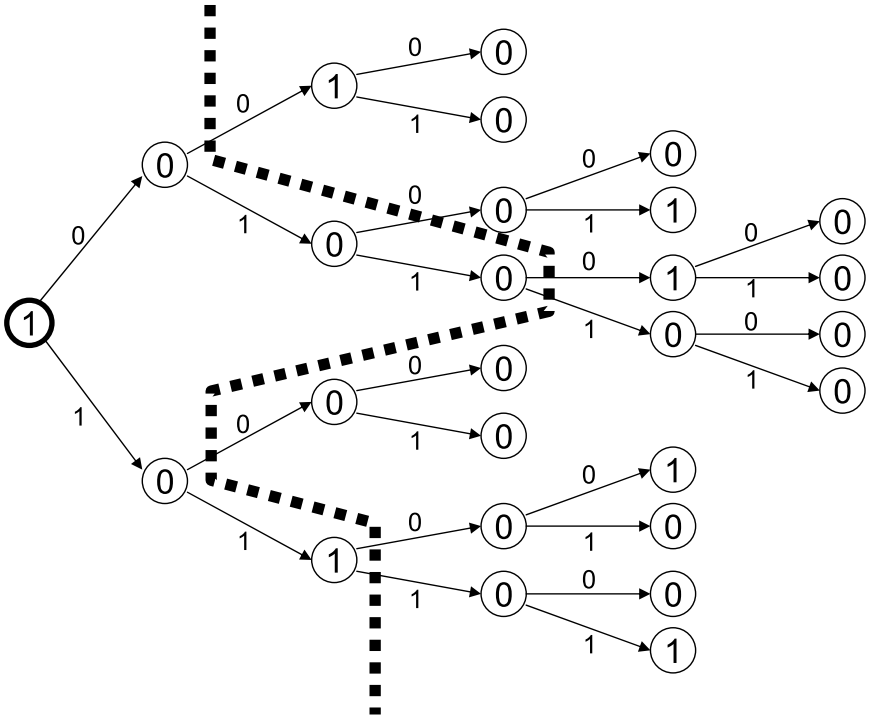

Figure 3.13: A closed and consistent tree with four unique behaviors; $E = \{\Lambda, 0, 1\}$.

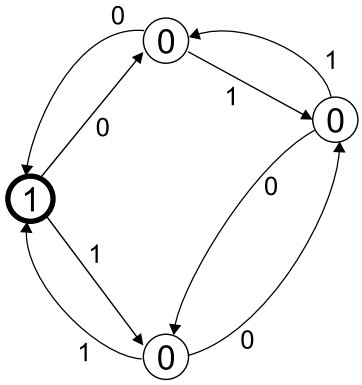

Figure 3.14: The third conjecture M_3 passes the equivalence test; it is a correct identification.

6. The result in Figure 3.13 is closed and consistent. Of the five nodes in S, there are four unique behaviors. Build a conjecture M_3 from the tree. This machine passes the equivalence test (Figure 3.14). It is a correct identification.

Chapter 4: TES Identification

Chapter 3 was a strict generalization of Angluin's algorithm to finite state machines. In this chapter we address the theoretical and practical concerns of specializing those results to timed event systems.

First we will describe how the identification algorithm of Chapter 3 applies to timed event systems. Then we will provide changes to the algorithm that better adapt it to this new application.

4.1 Finitization

In this section we address the practical problem of reconciling the timed event system model of Chapter 2 with the finite state machine identification algorithm of Chapter 3.

4.1.1 Introduction

Let us revisit the family of traditional timed automaton models. Roughly speaking, they consist of classical automata augmented with real-time clocks and transition guards. They typically have two ways to advance the total system state: a transition due to the passage of time, which simply advances the clock portion of the state, or a transition due to the occurrence of an event, which causes a transition in the automaton portion of the state. Note that an output event can only occur synchronous with an input event. The only effect of the clocks is to influence or modulate, via guard expressions, system behavior when an event occurs.

Now, an important FSM property is that each input symbol has an associated output symbol. If we take the input alphabet to include both the set A of input events and the set $\mathbb{R}^{\geqslant 0}$ of non-negative delays, this same correspondence can be set up to a certain extent in

the family of traditional timed automaton models. Aligning input and output sequences, an input delay corresponds to an output delay. Also, every output event is synchronous with some input event.

However, it is not clear in this family of models how to express a "spontaneous" output event, i.e., one that is not synchronous with an input event. Without this capability, systems as simple as a unit delay block cannot be expressed.

For example, with output spontaneity, a single input symbol may actually generate an output delay, event, and another delay. For the modeling needs of this work, output spontaneity is essential. This forces us to shift our viewpoint away from the approach taken in the traditional models.

Significantly, the example suggests that instead of input and output alphabets consisting of delays and events, the alphabets rather should consist of relatively brief timeline snippets. Once we make this change, an output timeline corresponding to an input timeline can be determined quite naturally, spontaneous events and all. The idea of using timelines as input and output symbols is the key to applying FSM identification to TES.

4.1.2 Notation review

For convenience, here we summarize notation important in this chapter.

The external model β' (introduced in Section 3.2) takes an input sequence to an output symbol, i.e., $\beta' : A^* \to Z$. (The prime is intended to suggest a single output symbol.) In contrast, the external model β (introduced in Section 2.2.1) takes an input timeline to an output timeline, $\beta : U \to Y$.

For sequences such as the input sequence passed to β', a dot operator (used in chapter 3) gives concatenation of sequences. Since sequences are always made up of symbols, we "overloaded" the dot operator for convenience so that for sequences s_j and symbols a_k, all of $s_1 \cdot s_2$, $a_1 \cdot a_2$, and $s_1 \cdot a_1$ yield sequences of symbols.

Timelines also have their own notion of concatenation. We will keep the circle-plus symbol introduced in Chapter 2 to denote concatenation of timelines. While sequences are

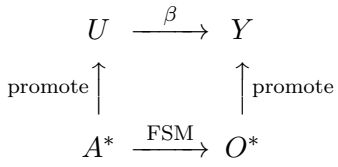

Figure 4.1: The commutative diagram.

always built from symbols, timelines are simply built from other timelines[1]. Therefore there is no need to define additional behaviors for the circle-plus operator.

Both the circle-plus and dot operations are splices in the sense of Definition 2.4, but refer to different types. Instead of using the circle-plus as the notation in both cases and disambiguating on context, we continue to use the dot notation when the operands are sequences and the circle-plus when they are timelines.

Finally we introduce an operation that "promotes" sequence concatenation to timeline concatenation.

Definition 4.1.

$$\text{promote}\,(a_1 \cdot \ldots \cdot a_n) = a_1 \oplus \ldots \oplus a_n$$

4.1.3 Identification

Our identification approach is to find an FSM whose behavior matches the timed-event SUT. We will express this relationship in the form of a commutative diagram [66], Figure 4.1. Such diagrams can be very helpful for expressing concepts like "system A simulates system B" succinctly and precisely.

In general, commutative diagrams make a statement about different paths that connect the same endpoints. As used here, each arrow on the diagram represents a function from its domain to its codomain. If there are different paths connecting the same endpoints, the diagram implies that applying the sequence of functions along either path to some element

[1]If in the input alphabet we changed each symbol to a sequence of length one containing that symbol, this apparent contrast between sequences and timelines would disappear. This approach is actually advantageous in the software implementation. It would eliminate the apparent relative complexity of the dot concatenator, at the cost of extra explanation it would have required early in the narrative.

from the starting set, yields a common element in the ending set. (Note that this is a much stronger statement than merely expressing the relationship of domains and codomains of a collection of functions.)

In this particular diagram there are two paths from A^* to Y, namely, up–then–across, and across–then–up. In either case we begin with some element $a_1 \cdot a_2 \cdots a_n \in A^*$, which is a sequence of timelines. First we consider the up–then–across path. The promote operation takes the sequence $a_1 \cdot a_2 \cdots a_n$ and, using the splice proper to timelines, concatenates them together into a single timeline $u \in U$, namely, $u = a_1 \oplus a_2 \oplus \cdots \oplus a_n$. Then we pass this input timeline through β to compute the corresponding output timeline y.

Taking the other path, the input sequence of timelines $a_1 \cdot a_2 \cdots a_n$ is fed to the finite state machine, which produces a corresponding sequence of output timelines $z_1 \cdot z_2 \cdots z_n \in Z^*$. This output sequence is promoted to a timeline using the splice proper to timelines, $z_1 \oplus z_2 \oplus \cdots \oplus z_n$.

We can understand the "rows" of this diagram as expressing a correspondence. Every element of A^* has a corresponding element in U. We can find this corresponding element with the promote operation. In the bottom row, the meaning of the a's as timelines is left uninterpreted; they are treated as arbitrary symbols. To carry this sequence to the top row, the promote operation must recognize the a's as timelines. Similarly, every element of Z^* has a corresponding element in Y, which can also be computed with promotion. A sequence of z's is treated as a sequence of arbitrary symbols, but promotion acknowledges them as timelines in order to map them to an element of Y. So in the bottom row, the FSM transforms an input sequence to an output sequence without caring that the symbols represent timelines; in contrast, the top row is only meaningful because it involves timelines. If β and the FSM map corresponding inputs to corresponding outputs, then β and the FSM themselves correspond with each other.

Finally, our expression of the identification problem is, find an FSM such that this diagram commutes, i.e., such that when corresponding inputs are fed to the FSM and the target system, their outputs also correspond.

4.1.4 Approach

We have described the FSM we are looking for. Here we show how it may be found with FSM identification. The key is to make a TES appear to take a sequence of symbols as input and produce a sequence of symbols as output. We show how a behavior β and input alphabet A may effectively construct a finitized behavior ϕ that has the "interface" of a state machine.

Consider a causal behavior function that takes an input timeline to an output timeline, $\beta : U \to Y$, and choose a finite subset $A \subset U$ of the set of input timelines. We will denote an element of A as a and of A^* as $a_1 \cdot a_2 \cdots a_n$. Note that a is in U, $a_1 \cdot a_2 \cdots a_n$ is not in U, but $a_1 \oplus a_2 \oplus \ldots \oplus a_n$ is in U.

Definition 4.2.

$$\phi'(a_1 \cdot a_2 \cdots a_n) \triangleq \text{suff}(\beta(a_1 \oplus \cdots \oplus a_n), \text{len}(a_1 \oplus \cdots \oplus a_{n-1}))$$

Let us interpret this definition. We begin with a behavior β and input timeline snippets $a_1 \cdot a_2 \cdots a_n$. We can splice these input timeline snippets together and compute system response to them. The suff keeps only the portion of the output timeline corresponding to the final snippet a_n in the input timeline. Therefore ϕ' gives the output timeline corresponding to the final input snippet.

Definition 4.3.

$$\phi(a_1 \cdot \ldots \cdot a_n) = \phi'(a_1) \cdot \phi'(a_1 \cdot a_2) \cdot \ldots \cdot \phi'(a_1 \cdot \ldots \cdot a_n)$$

And now let us interpret this definition. We know that $\phi'(a_1)$ gives the output timeline response to a_1, $\phi'(a_1 \cdot a_2)$ gives the response to input a_2 (after having processed a_1), and $\phi'(a_1 \cdot \ldots \cdot a_n)$ gives the response to a_n (after having processed $a_1 \cdot \ldots \cdot a_{n-1}$). It follows that ϕ takes a sequence of input timeline snippets to a corresponding sequence of output timeline snippets.

The motivation behind these definitions is that they relate a timeline-based behavior β to a sequence-based behavior ϕ.

Proposition 4.4. For causal β, if the FSM in figure 4.1 has the above behavior ϕ, then the diagram commutes.

Proof. To show that the diagram commutes, we will show that the two paths through the diagram give equal results, i.e.,

$$\text{promote}\left(\phi\left(a_1 \cdot a_2 \cdot \ldots \cdot a_n\right)\right) = \beta\left(\text{promote}\left(a_1 \cdot a_2 \cdot \ldots \cdot a_n\right)\right)$$

First, using the definitions above and 2.13 (and where g refers to a realization of β),

$$\begin{gathered}\phi\left(a_1 \cdot a_2 \cdot \ldots \cdot a_n\right) = \phi'\left(a_1\right) \cdot \phi'\left(a_1 \cdot a_2\right) \cdot \ldots \cdot \phi'\left(a_1 \cdot a_2 \cdot \ldots \cdot a_n\right) \\ = \beta\left(a_1\right) \cdot \text{suff}\left(\beta\left(a_1 \oplus a_2\right), \text{len}\left(a_1\right)\right) \cdot \ldots \cdot \text{suff}\left(\beta\left(a_1 \oplus \cdots \oplus a_n\right), \text{len}\left(a_1 \oplus \cdots \oplus a_{n-1}\right)\right) \\ = g\left(\mu\left(\Lambda\right), a_1\right) \cdot g\left(\mu\left(a_1\right), a_2\right) \cdot \ldots \cdot g\left(\mu\left(a_1 \oplus \cdots \oplus a_{n-1}\right), a_n\right)\end{gathered}$$

Promoting this sequence we obtain

$$\begin{aligned}\text{promote}\left(\phi\left(a_1 \cdot a_2 \cdot \ldots \cdot a_n\right)\right) &= \text{promote}\left(g\left(\mu\left(\Lambda\right), a_1\right) \cdot g\left(\mu\left(a_1\right), a_2\right) \cdot \ldots \cdot g\left(\mu\left(a_1 \oplus \cdots \oplus a_{n-1}\right), a_n\right)\right) \\ &= g\left(\mu\left(\Lambda\right), a_1\right) \oplus g\left(\mu\left(a_1\right), a_2\right) \oplus \ldots \oplus g\left(\mu\left(a_1 \oplus \cdots \oplus a_{n-1}\right), a_n\right)\end{aligned}$$

Then, using 2.23 (which holds when β is causal) and 2.18,

$$\begin{aligned}g\left(\mu\left(\Lambda\right), a_1\right) \oplus g\left(\mu\left(a_1\right), a_2\right) \oplus \ldots \oplus g\left(\mu\left(a_1 \oplus \cdots \oplus a_{n-1}\right), a_n\right) &= g\left(\mu\left(\Lambda\right), a_1 \oplus \cdots \oplus a_n\right) \\ &= \beta\left(a_1 \oplus \cdots \oplus a_n\right) \qquad \square\end{aligned}$$

So from β and A we have defined a new behavior ϕ. This new behavior processes sequences instead of timelines, i.e., it has the interface of a state machine. (Note that we have not shown this behavior to necessarily correspond to a *finite* state machine.)

And now the identification approach is simply this.

1. Express the sequence-based behavior ϕ in terms of the timeline behavior β.
2. Choose an input alphabet $A \subset U$.
3. Use this A and ϕ as the inputs to the FSM identification algorithm variant described in Section 4.2.

We have shown that we can construct a sequential behavior ϕ that has the behavior of a timed event system β. But we have not shown that ϕ can be realized with a *finite* number of states. In fact it generally cannot. Since the reachable subspace $\mu(A^*)$ of ϕ depends on A, the selection of A is of substantial importance. These considerations are explored experimentally in chapter 5.

4.1.5 A pragmatic concern

This theoretical discussion actually raises a minor issue discovered in practice. Our approach relies on the algebraic property that a splice of a split of timelines equals the original timeline. However, if the split time does not have a finite binary representation (such as 1/3, which is a repeating decimal in binary as it is in base-10 and therefore not exactly representable in conventional floating point), this property does not exactly hold.

As far as this work is concerned this problem is easily addressed. Since the user has the ability to choose the input alphabet, we always choose the duration of these input timelines to be friendly to binary fraction representations.

4.2 Algorithm enhancements

In the previous chapter we described an adaptation of Angluin's algorithm that identifies finite state machines instead of regular languages.

The algorithm of chapter 3 is directly applicable to the TES identification problem as described above, except for one technical flaw concerning probable approximate correctness, which is addressed and corrected below. Additionally, some other heuristic changes based on the shift from languages to machines are considered, with a view mostly to reducing the total number of experiments required.

4.2.1 Relevant output

As an automaton or FSM operates, it produces a sequence of output symbols. In the special case of an automaton, these symbols indicate the language membership of the input sequence so far. For determining language membership, only the final output symbol matters.

For example, consider the FSM in Figure 3.14 and interpret it as an automaton whose output indicates language membership. Consider an input sequence $u = 001101$. The corresponding output sequence (suppressing the output of the initial state) is $y = 010100$. As indicators, this output tells us that the sequences 00 and 0011 are in the language, but all other prefixes of u are not, including the complete sequence. If we are interested in language membership of the input sequence as a whole, we only care about the final output symbol. Normally the language memberships of the prefixes of the input sequence are irrelevant. This is why the original algorithm modeled the system behavior as β', a function taking an input sequence to a single output symbol.

Consider the output of the systems in Figures 3.14 and 3.11 to the input sequence $u = 0111$. Their output sequences are 0000 and 0010, respectively. They are not equal, but they do agree in the last symbol, namely, they both indicate that this input sequence is not in the language.

If the input sequence 0111 was generated in a statistical test of equivalence of 3.14 and 3.11, it would be appropriate to count the comparison as a match as far as languages are

concerned, but if our goal is a probably approximately correct system, this would mean we were ignoring an error.

Instead of an elaborate investigation into what agreement in the final symbols tells us about agreement of the entire output sequences, we do something much simpler. We change from using a function β' that returns only the final output, to β that returns the entire output sequence. (In the context of the identification problem this makes perfect physical sense.) We compare entire output sequences in the statistical test, so that when the algorithm terminates, its PAC assurance applies to the output sequences, not only to the final output symbols.

4.2.2 Corollary changes

The substantial difference made by considering entire sequences is that the statistical acceptance test is made stronger — probable approximate correctness holds for the whole output sequence, not just the final output symbol. But other advantages emerge from this shift from a languages to a systems viewpoint.

One easy benefit is that instead of running a batch of new experiments, it suffices to run their prefix reduction. For example, if the algorithm calls for the results to input sequences 0, 00, and 001, it suffices to actually run only 001. The output symbols for all three input sequences can be filled into the observation table using the results of the single output sequence.

Another advantage of this change is that since we compare the whole output sequence, counterexamples can be trimmed down to the first place they are wrong. A shorter counterexample requires fewer experiments for its processing.

4.2.3 The silent suffix heuristic

What is here called the silent suffix heuristic assumes that making experiments slightly longer in order to reduce the total number of experiments is a good tradeoff. The experiments are lengthened by appending an extension s_{suff} to the input sequence. The idea is

that the extended experiment also includes other longer experiments likely to be requested in the future.

How to choose s_{suff} is quite dependent on the system at hand, but some observations that often hold in practical systems suggest a starting point.

- Many systems revert to a well-known reference mode or initial condition in case of a long uneventful spell.
- In some cases, an uneventful input is "least expensive" to run in some sense.
- Most of the time, no input event is occurring.

While the design of s_{suff} is ultimately up to the analyst, these observations suggest setting it to a period of time in which no input events occur when no better options present themselves.

As for the algorithm, at certain points it has a choice of input symbols, and in this implementation it chooses them in order from A. However s_{suff} is chosen, it should be coordinated with the ordering[2] of A. When the algorithm needs to extend an experiment, it begins with the first element in A. Therefore, the first symbol in s_{suff} and the first element in A should be the same symbol.

For example, when experiment s is called for, we actually run $s \cdot s_{\text{suff}}$. We return only the portion of the output corresponding to s but store the entire result. Then later if any prefix of $s \cdot s_{\text{suff}}$ is called for, it need not be run, but the result simply recalled from a database.

4.3 The modified algorithm

The resulting algorithm is really just a variant of the one described in the previous chapter. Instead of repeating the bulk of that presentation, only the changes will be described.

A working implementation of this algorithm is demonstrated in the next chapter.

[2] Although A is a set for all mathematical purposes, in software practice it is implemented with an ordering of elements, a property which turns out to be useful here.

4.3.1 Input

In Figure 3.5, the input to the algorithm was an alphabet A of input symbols and a final-output behavior function β'. In this variant, the input is an alphabet A of input symbols and a whole-output behavior function β. Note that now the behavior maps to an output sequence instead of a single output symbol.

4.3.2 The query function `extendT`

The general role of `extendT` is the same as before — it takes a list of input sequences, runs experiments on the SUT to find their associated outputs, and enters the results into the observation tree — but its details and implementation are different.

First we take the given list of input sequences and compute their prefix reduction.

Then, for each input sequence u in the reduction that was not already in T:

1. Run the experiment $\beta(u \cdot s_{\text{suff}})$, where s_{suff} is the silent suffix,

2. Loop over the prefixes u_i of $u \cdot s_{\text{suff}}$ and their associated output symbol z_i in the output sequence, and enter them into T, i.e., $\beta(u_i) \triangleq z_i$.

Note that the signature of T is still $T : A^* \to Z$ as it was before.

Also, the operation of the silent suffix is very simple. The results of the extended part of the input sequence are simply recorded in T. Since this lies beyond T's official domain of $(S \cup S \cdot A) \cdot E$, the main algorithm is not even aware of this data's presence. If and when some part of the algorithm augments S or E and calls for filling out T's domain on an input sequence that has already been run, `extendT` detects the fact that the answer is already in place in T and skips ahead to the next input sequence.

4.3.3 Conjecture test

As before, we build the conjecture M and then test it in a loop with random inputs. The changes are that the entire output sequences are now compared, instead of just the final

symbol; and a counterexample input sequence is truncated to the first disagreement in the output sequences.

Chapter 5: Experimental Results

The contribution of this dissertation is a system identification technique. Theoretical considerations form an important part of the argument, but ultimately the value of a tool lies in its usefulness, which is not a mathematical property and therefore cannot be proven to hold. Rather, it is a judgment made on the basis of practice and experience.

The examples chosen here stand in place of extensive industrial application. They are artificial cases meant to illustrate intended usage of the technique, and additionally, to highlight its strengths and weaknesses.

The search-track example is an SUT of appreciable complexity. The number of experiments required for identification may be considered rather high for many applications. The example takes advantage of serendipitously chosen input timing.

In contrast, the queue example is an extremely simple SUT; however, its input alphabet is poorly chosen, putting the identification algorithm at a disadvantage.

It is worth noting the difference between the intended application of this technique and the way the technique is used here. In practice, the algorithm is expected to operate on a piece of hardware such as an embedded system. The system will be instrumented so that a computer can reset the system, send it input events, and record its output events. However, in the examples here, the SUT is a piece of software interfaced to the identification algorithm, running on the same computer. Therefore all of the engineering issues of instrumenting real hardware are ignored. These issues are expected to include timing jitter and the mapping between events and physical signals. The particular problems posed by such issues are expected to vary considerably from system to system.

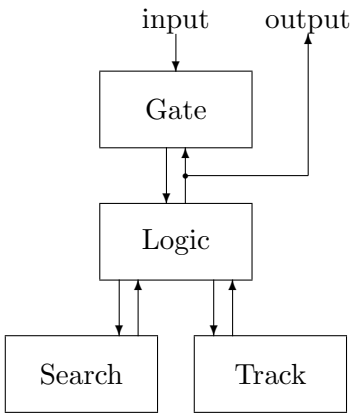

Figure 5.1: Block diagram of the example SUT

5.1 Search-Track Example

Here we illustrate the method by applying it to an SUT of a notional search-track radar system.

5.1.1 SUT description

An environment is divided into a 4×4 grid of cells. A "radar" can place a gate over any selected cell. This gate allows the radar to sense the presence or absence of a target in that cell only. A logic block arbitrates between search and track behaviors (Figure 5.1). In terms of our formalism, all blocks in the figure between the input and output, taken together, correspond to the SUT behavior function β.

The input is a sequence of target maps for the whole 4×4 environment. The gate block filters the data in these maps, passing through information about the presence or absence of a target only for the currently selected cell. This is intended to roughly model the information available in a sequence of radar returns.

Initially the logic block gives the search block control over the currently selected cell. The search block moves the cell over the entire environment in a raster pattern, spending one second in each cell. When the search block finds a cell containing a target, it notifies the logic block, which then hands control of the cell to the tracker. The tracker block attempts to keep the selected cell over the target. If the radar returns indicate a target present, the

tracker takes no action. If the target is lost, the tracker quickly checks the four adjacent cells (or their available subset, if the target was last tracked in an edge cell) in an attempt to re-establish track. (In this experiment it spends 0.25 seconds in each adjacent cell.) If the tracker loses track it notifies the logic block, which transfers control of the cell back to the search block.

5.1.2 Identification arrangements

In the experimental setup, the periodic sequence of target maps is provided by the identification algorithm at a rate of four per second as input to the gate block, and the sequence of cell selection events constitutes the output. The identification task is to compute a model of the complete system. (The model only aims to be behaviorally equivalent to the SUT in Figure 5.1 — we have no interest in, or hope of, recovering the block structure, hidden events, etc.)

Before attempting identification of this system, the analyst needs to design an input alphabet A and a random process in that alphabet, i.e., assign a distribution to A^*.

For this system, a single input event is a target map that encodes per-cell occupancy of the entire 4×4 environment. Allowing every possible occupancy combination of the 4×4 grid would lead to an alphabet containing $2^{16} = 65536$ distinct elementary input timelines. We prefer to begin with a smaller alphabet of 17 timelines: 16 in which any one cell may be occupied in the interval, and a 17th in which no cell is occupied during the interval.

The input random process design is important, since the quality of the identification will be with respect to this process. The design involves a tradeoff: if the process ranges over fewer possibilities, the identification problem is easier but applies to a narrower range of situations; a process that includes a very wide range of inputs is broadly applicable but could make identification quite expensive.

In this example, we decide to use a simple independent identically distributed (IID) input process. Each sample is 200 symbols long, and each symbol is uniformly and independently chosen from A. (This is a choice of convenience over realism, since actual targets would

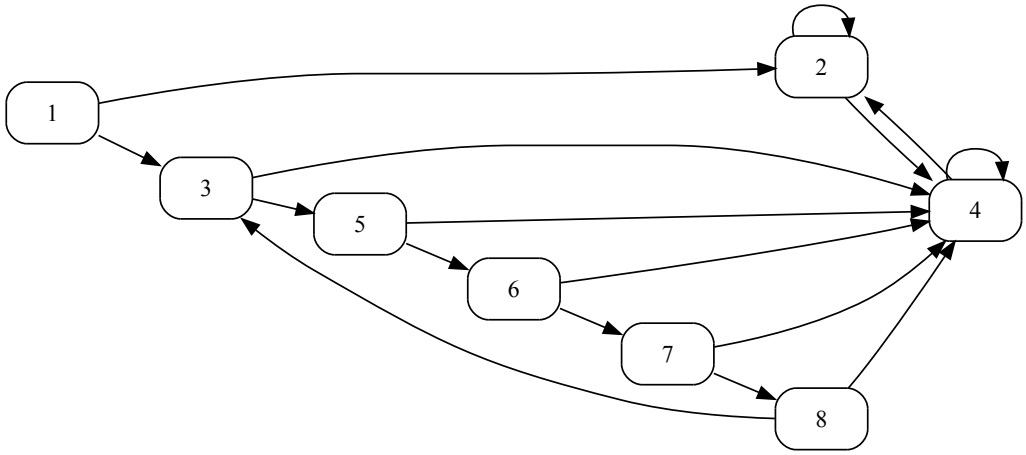

Figure 5.2: The first conjectured FSM

move through adjacent cells rather than jumping around or disappearing at random.)

5.1.3 Simulation results

The identification algorithm enters the closure-consistency loop and crafts a sequence of experiments. After 129 experiments its observations are sufficient to conjecture a model of the SUT. This conjecture is illustrated in Figure 5.2 and has 8 states.

After constructing a conjecture, the identification algorithm transitions to equivalence test mode, where it sends input sequences generated by the random process to both the SUT and its conjecture and compares the results. In this run, the conjecture predicted a different outcome than the actual system on the very first random experiment. This counterexample is incorporated in the observations, and the algorithm returns to the closure-consistency loop.

It takes 1091 more experiments (for a total of 1221 experiments so far) before the algorithm finds its database of observations sufficient to make its second conjecture, an FSM of 73 states illustrated in Figure 5.3. When facing the battery of random tests, this conjecture fails on the first randomly-generated input sequence, just as its predecessor. The

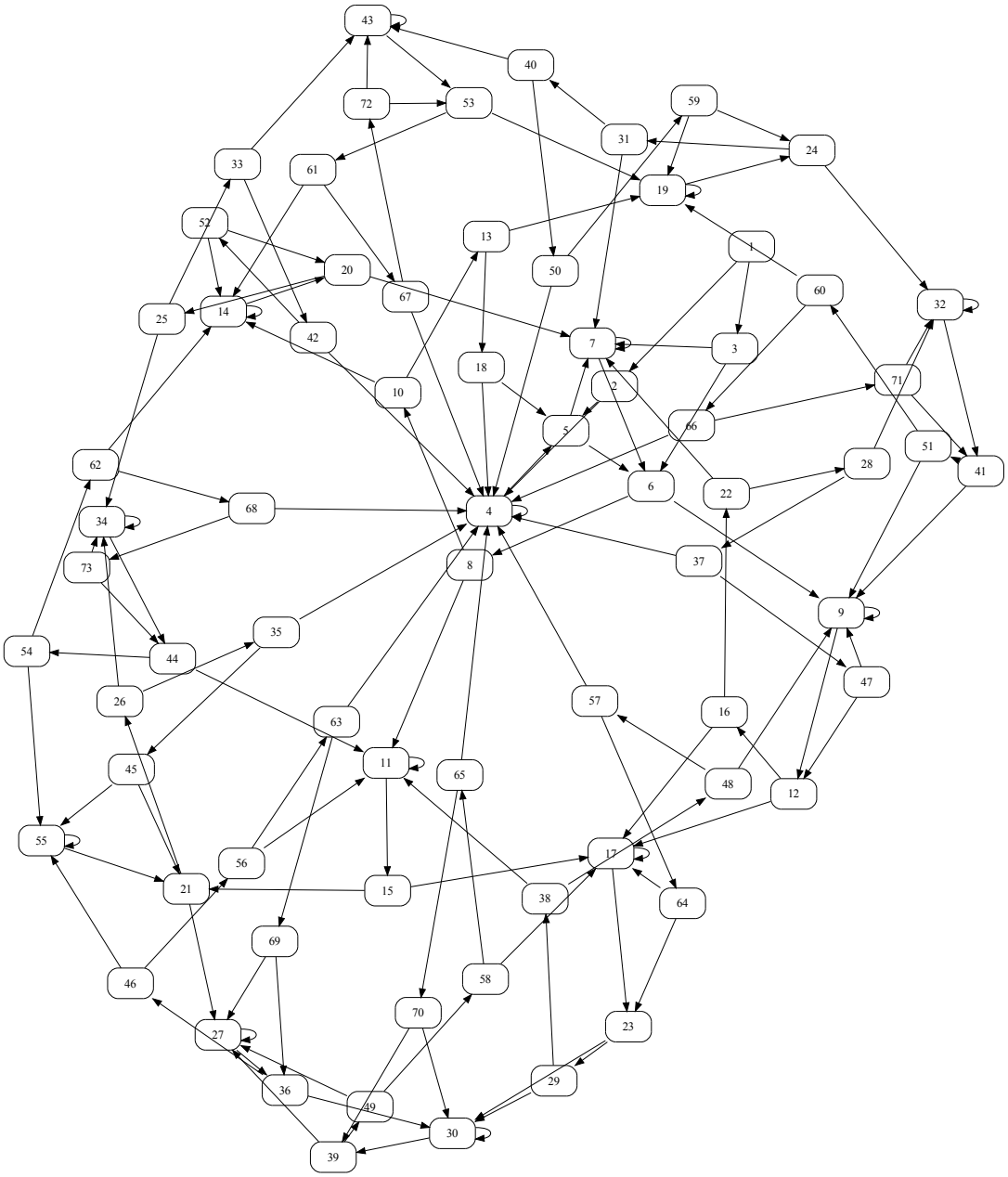

Figure 5.3: The second conjectured FSM

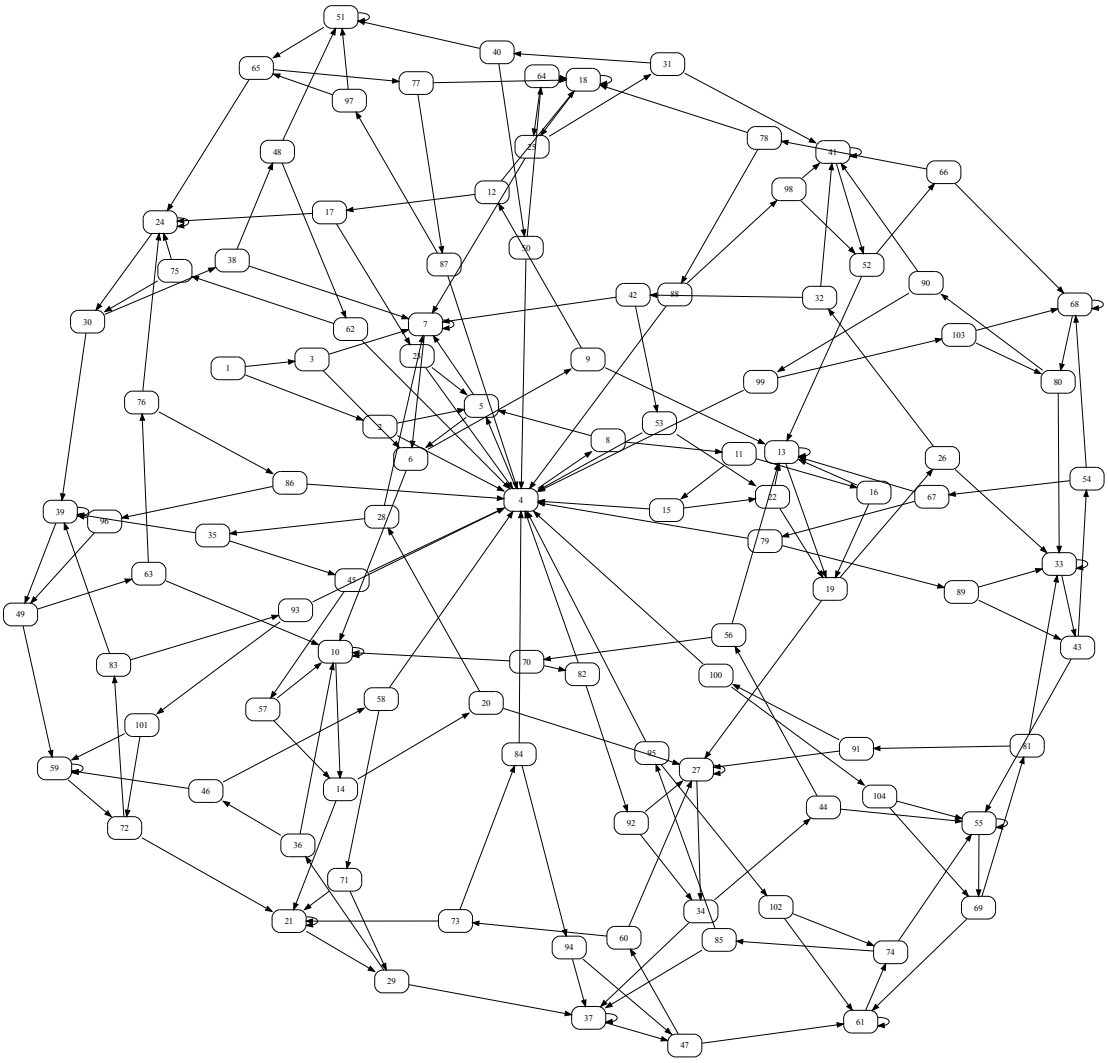

Figure 5.4: The third conjectured FSM

Figure 5.5: The fourth and final conjectured FSM

counterexample is incorporated into the observations and the algorithm goes back to the closure-consistency drawing board.

After 535 more experiments, the algorithm makes its third conjecture, this time consisting of 104 states and illustrated in Figure 5.4. Once again, the conjectured system fails to match SUT performance on the first random trial. The counterexample is noted and, 2489 experiments later, for a total of 4247 experiments, the identification algorithm makes its fourth conjecture, an FSM of 253 states illustrated in Figure 5.5. This one is able to pass the required 116 random trials without any errors. It is accepted as probably approximately correct.

These results were obtained using ideal input alphabet timing, e.g., the 0.25 second input event period evenly divides the one second period of the search behavior. The effects of non-ideal timing are investigated in Section 5.2, but first we will use the search-track example to measure the effectiveness of the heuristics introduced in Chapter 4.

5.1.4 Heuristic effectiveness

Here we present the raw data showing the difference in learning effort between the algorithm of Chapter 3, and the algorithm enhanced with the heuristics of Chapter 4. Each algorithm was run for fifteen trials on the search-track problem, where each trial used a different random seed for the input process. The results are depicted in Figure 5.6.

Every trial of the algorithm without heuristics needed at least 20000 experiments (an average of about 23448 experiments). Every trial of the algorithm with heuristics needed fewer than 5000 experiments (an average of about 4351 experiments) for the same identification quality. In this case, use of the heuristics allowed a model of the same quality to be inferred with approximately one-fourth the learning effort.

5.2 Queue Example

It goes without saying that identification techniques applied blindly cannot be expected to succeed; and as the complexity of the system type increases, so must the user's care. In the

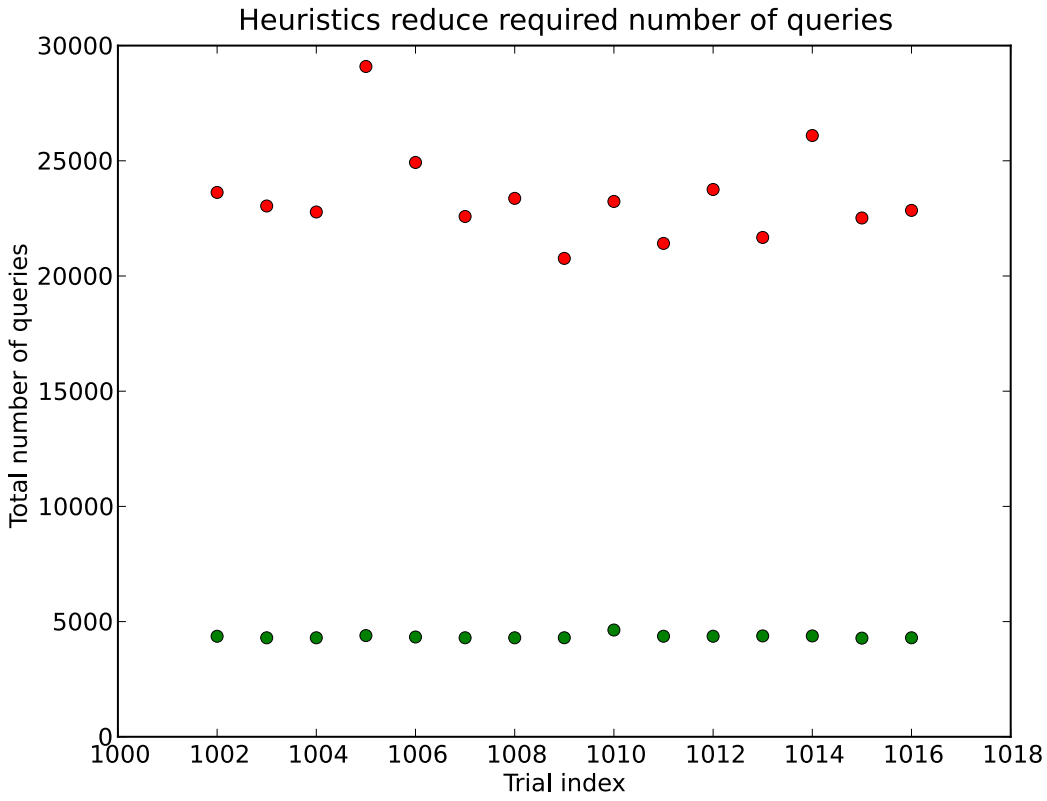

Figure 5.6: Impact of heuristics on learning effort

previous section we showed simulation results for a reasonably complex system identified with the benefit of ideal input timing. Here we explore the opposite extreme of a very simple system analyzed with unfortunate timing.

In these experiments the SUT is a very simple queue block. When an input arrives at an empty queue, it departs T_q seconds later. If the queue is busy, a new input arrival leaves T_q seconds after the last already-pending departure.

For an input alphabet we use

$$A = \{(\{\}, \tau_q), (\{(\tau_q/2, a)\}, \tau_q)\}$$

i.e., the symbols are a pure time lapse of duration τ_q, and an interval of duration τ_q containing event a at its midpoint. Every sample of the random process is a sequence of 30 alphabet symbols. Five symbols, uniformly distributed in the sequence, contain the event a, the other 25 are the time lapse.

We vary τ_q to study its effect on identification performance. The τ_q values lie in a reasonable range but otherwise were chosen quite arbitrarily. We chose to quantize τ_q to 2^{-6}. This is mostly to avoid the annoyance of roundoff error when splicing. However, the queue processing time is a practically incommensurable $T_q = 0.314159$.

Since one of the inputs to the identification algorithm is a random process, its final output is also random. The scatter plot in Figure 5.7 illustrates the learning effort (i.e., number of experiments) needed to arrive at a PAC conjecture, for several designs of the input alphabet over several trials (ten trials per τ_q).

The clearest feature in this figure is the "sour spot" at $\tau_q = 0.15625$ where the learning effort skyrockets. Also noteworthy is the flat region for the larger τ_q values. In these cases, the elementary input timelines are so long that the queue has finished processing them before the elementary timeline ends.

These results illustrate the hazard of applying the technique without careful preparation. Some choices of input timing can conceal an SUT's underlying simplicity.

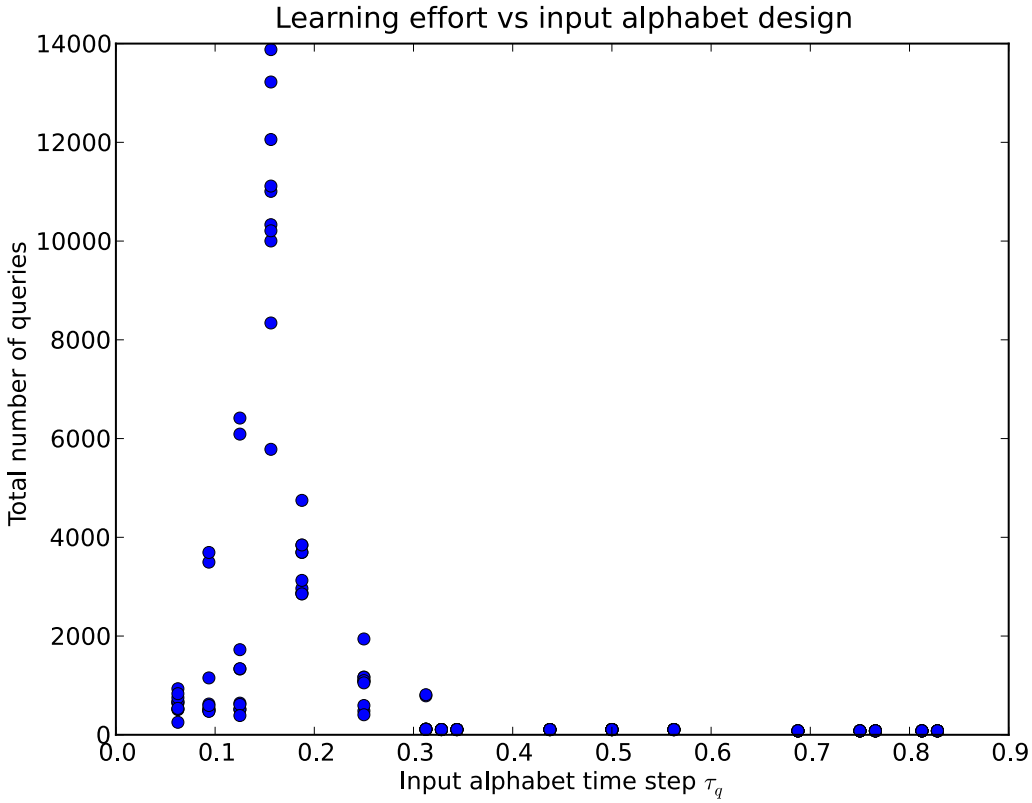

Figure 5.7: Effort needed to identify a simple queue model

Chapter 6: Conclusions

To conclude, we review the contributions made in this work, and present some of the new questions raised by this research.

6.1 Contributions

1. Development of a timed event system model in an abstract systems context. The model allows for arbitrary event timing, spontaneous output events that are not required to be synchronous with an input event, subsystem composition (one that completely avoids explicit construction of synchronized products of automata), an effectively arbitrary number of clocks (through they are not explicitly addressed as such), a simple and familiar mechanism for scoping of events (namely, the block diagram), and the ability to retract scheduled events that have not yet occurred. An important feature is the simplicity with which all of these features can be exercised. The model is developed in enough detail to allow formulation of an identification problem in its terms, for implementation in working software, and for proof of simple theorems such as the correctness of the discrete event system simulation algorithm[1].

2. Development and presentation of an algorithm for identification of finite state machines. This algorithm is a strict generalization of Angluin's seminal contribution, which addressed the problem of identification of regular languages. A detailed proof is given of the critical property that every FSM constructed by the algorithm is consistent with all experimental data. It is shown historically that one influence on Angluin's algorithm was systems theory, and the presentation of the algorithm is from

[1]While intended as a means to an end, the inherent value of this part of the problem in its own right was a pleasant surprise in the course of this research. Thus the inclusion of systems theory material that goes beyond what is strictly necessary for the identification task.

the systems theory viewpoint.

3. Development of a finitization construction that permits the interpretation of a timed event system as an abstract system, or state machine. This is done to bridge the gap between timed event systems and finite state machines, so that a finite state machine identification algorithm can be applied to a timed event system. The theoretical concept also finds a concrete implementation in software.

4. Development and implementation of heuristic adaptations of the identification algorithm. These exploit the additional information available from an FSM not available from a regular language oracle, to improve identification efficiency. The heuristics are shown to give a fourfold improvement in learning efficiency over the unenhanced algorithm.

5. While recognizing that the technique has theoretical limitations, show by a detailed example a case in which the goal of automatic identification of a timed event system is achieved.

6.2 Future research

The work described in this paper represents a first step in adapting a technique from computational learning theory to a system identification problem. Early experience suggests numerous lines of investigation and development, including:

1. Extending the algorithm with noise tolerance, along the lines of [67].

2. Allowing the input process to depend on SUT behavior (this will sometimes be necessary to give the input process the desired probability distribution).

3. Finding heuristics for reducing the number of experiments needed. Could occasional user intervention help guide identification? Can exploratory data analysis, data mining, or optimization of initial results suggest better input alphabets?

4. Can applying knowledge of SUT subsystems reduce identification complexity? For example, in the search-track identification problem, an analyst might hypothesize that a subsystem in the SUT is generating commands to move the gate of interest either left, right, up, or down, and that the gate position is then being updated in a predictable way. Externally implementing this hypothesized predictable part could reduce the problem to that of identifying the generation of left, right, up, and down events, and would substantially reduce the size of the state space of the unknown system.

5. What is the natural analogue of the region automaton [28, 29] for an ERF?

6. Can the discrete event system simulation correctness proof be interpreted as a system-theoretic justification for the technique of deadlock resolution via null messages used in distributed discrete event simulation [50, 52]?

7. Can the concept of a code, i.e. a set of symbol sequences with a unique factorization in that set [68], and related concepts aid in input alphabet design?

8. How do we translate the identification product from its FSM form into other representations?

6.3 Epilogue

Currently, an analyst faced with the need to elicit the behavior of a complex event-based system does not have a wealth of techniques available. An identification technique is proposed here not as a complete solution but as a starting point. Like any sophisticated tool it should be used with other complementary methods when possible, and can only really be effective when applied with insight and engineering judgment. In spite of its limitations, it is hoped that the approach presented here can be a useful aid in addressing this difficult problem.

Appendix A: Analog Guards Example

The problem and examples used in the main body of this work consider identification problems in their most difficult form. In this appendix we consider scaling down to a less demanding model class, with a proportionate decrease in caveats. The result here is one arrived at relatively early in the course of this research. One benefit of considering a simpler problem is that we do not use the additional layer of abstraction described in Chapter 4.

A.1 The analog guard machine

A very simple model with some timed automaton-like features is an "analog guard" machine (AGM). An AGM is an FSM with optional conditions or guards included on the transitions. These guards take the form of a simple comparison on a real variable, e.g., $x \leqslant 1.6$. (How intricate the guards are allowed to get will certainly affect identification complexity.) For system behavior to be well defined, guards must be mutually exclusive and exhaustive, of course.

The AGM was not found in the literature but rather was developed as a foil to the ERA. The motivation for it is as follows. Recall that in an ERA, each input symbol has an associated clock, which gets reset to zero whenever its corresponding symbol is received. Since the analog clock values can be completely determined from the input sequence, it appears that ERA identification reduces to AGM identification.

Consider the AGM in Figure A.1. The states, from left to right, are referred to as $S0$, $S1$, $S2$. A typical run with the machine may be as follows.

1. Start in the initial state $S0$. Its output is 0 before any input is applied.

2. An input b arrives, causing a self transition to $S0$.

3. An input a causes transition to $S1$.

4. Input b occurs. There are two b transitions out of $S1$, and they depend on the value of x. (The value of x means the value of x associated with this input, e.g., the value of x "when" the input b occurs; x can take different values every time.) Say the value of x when b occurs is 2.5. This value determines that the transition we take is back to $S0$.

5. Another a takes us to $S1$.

6. It is followed by another b. This time the value of x is 3.14. The transition is to $S2$, and the output changes to 1 for the first time in the run.

A.2 Identification technique

We seek to apply Angluin's algorithm directly to this AGM. With a sensible choice of input alphabet this is possible. The idea is to extend the input alphabet to include a sampling of values for x. Here I choose values for x representative of the range of values it will take in the input distribution of interest. This is reasonable from the PAC criterion point of view.

$$A = \{(a, 0.375), (a, 1.0), (a, 2.25), (b, 0.375), (b, 1.0), (b, 2.25)\}$$

This input alphabet and a software implementation of the AGM in the Figure are passed into the identification algorithm. The result is depicted in Figure A.2.

The only "problem" in the results figure is the middle state, which has two outgoing

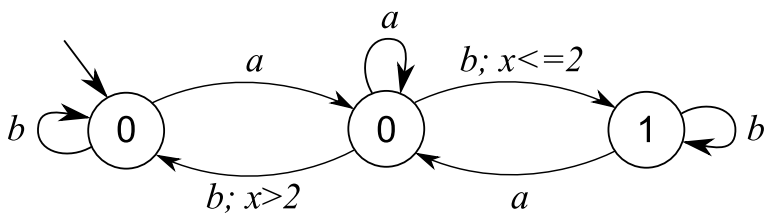

Figure A.1: An analog guard machine

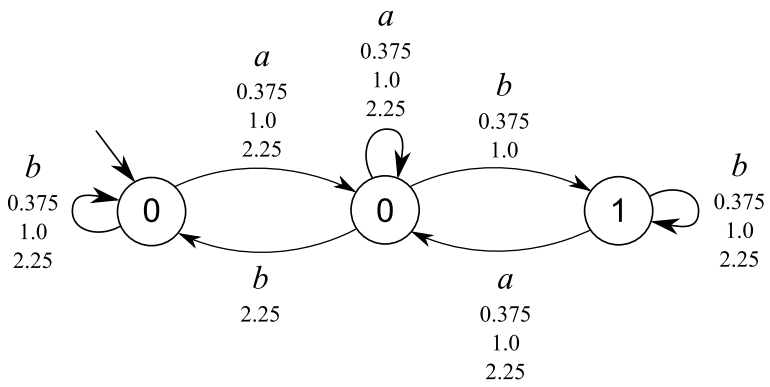

Figure A.2: Result of identification

transitions on b. In such a situation, the transition is disambiguated by the analog component. It remains to infer a guard expression consistent with these particular data values.

For small problems such as this, a guard can be inferred by inspection. However, it is of interest to note that inferring such a guard from examples is a problem ideally suited to numerous data mining or statistical learning algorithms, such as classification trees [69]. The sophistication of the inference algorithm should be matched to the richness of the guard language.

This identification approach stands as a rough-and-ready complement to the elegant theoretical results of [39]. It must be admitted that the approach here naively applied is exponential in the number of analog inputs.

Appendix B: FSM Minimization

The rationale underlying Angluin's algorithm can become considerably clearer when it is seen as an evolution of techniques used in classical finite state machine minimization algorithms. To this end, the classical FSM minimization algorithms are reviewed here.

It is interesting to note that while the algorithms here rely on a "state splitting" principle, many practical grammatical inference algorithms work from a complementary "state merging" principle, e.g. [70].

Some important definitions are as follows (see [62]). States are equivalent if and only if they produce the same output sequence in response to any possible input. If two states are not equivalent, then they are distinguishable, and an input sequence that leads them to different outputs is a distinguishing sequence for the pair. Equivalence of states S_i and S_j is denoted $S_i \equiv S_j$, and $S_i \not\equiv S_j$ denotes that S_i and S_j are not equivalent.

It is important to note that the definition of equivalence implies that if two states are equivalent, then their successor states after any input symbol are also equivalent.

B.1 Propositional approach

Here we develop the implication table using a propositional logic interpretation. The big picture is, we show how facts about distinguishability of pairs of states imply distinguishability of other pairs.

Consider the machine M_1 in Figure B.1.

Step P1. Some states are distinguishable because they have different outputs. This can be read directly from the FSM diagram. For example, since the states A and B in Figure B.1 have different outputs, i.e., the outputs given input 0 and 1 are 0 and 1 for A but 0 and 0 for B, the states A and B are distinguishable. Therefore we note that $A \not\equiv B$, and

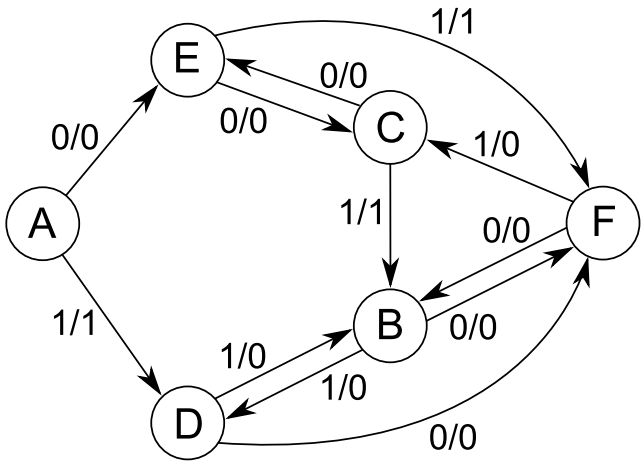

Figure B.1: Example FSM M_1.

so forth for the other pairs of states. These become our starting facts. (It should be clear that we are writing these facts in an arrangement corresponding to the familiar implication table — columns correspond to the states A to E, rows correspond to the states B to F, and a fact regarding a pair of states is entered in the table at the pair's unique coordinate.)

$$
\begin{array}{llll}
B \not\equiv A & & & \\
 & C \not\equiv B & & \\
D \not\equiv A & & D \not\equiv C & \\
 & E \not\equiv B & & E \not\equiv D \\
F \not\equiv A & & F \not\equiv C & \qquad F \not\equiv E
\end{array}
$$

Next we find a way to apply the principle mentioned above, namely, if two states S_i and S_j are equivalent, then their successors for all inputs are also equivalent. This principle lets us go to the FSM diagram and derive rules from it.

As an example, consider the states A and E. The successors to A and E given the input 0 are E and C, respectively, and the successors to A and E upon input 1 are D and F, respectively. Therefore by the principle, if A is equivalent to E, it follows then that E is equivalent to C and D is equivalent to F, or in symbols, $A \equiv E \rightarrow E \equiv C \wedge D \equiv F$.

However, our starting facts above pertain to distinguishability, not equivalence. We can apply the contrapositive, $(p \to q) \leftrightarrow (\neg q \to \neg p)$, to transform rules such as the one above into a more useful form, namely, $E \not\equiv C \vee D \not\equiv F \to A \not\equiv E$. This rule allows us to deduce new facts about distinguishability from previously known facts about distinguishability.

There are some minor issues to be aware of in this procedure. Consider the pair of states A and B from the FSM diagram. This pair gives us the rule $A \equiv B \to E \equiv F \wedge D \equiv D$. But we already know that $D \equiv D$, so this can be left unsaid. More interestingly, we know this rule will contrapose into a form $\ldots \to A \not\equiv B$. But we already know that $A \not\equiv B$ because it is one of our starting facts. There is nothing that can be concluded from this rule that we do not already know, so the whole rule may be skipped.

Taking these kinds of details into account, we generate all the rules from the FSM diagram and note them appropriately in the table. The skipped rules such as the $A \not\equiv B$ example above correspond to spaces already filled with state distinguishability facts in our table of propositions.

$B \not\equiv A$				
$D \not\equiv B$ $\to C \not\equiv A$	$C \not\equiv B$			
$D \not\equiv A$		$D \not\equiv C$		
$E \not\equiv C \vee F \not\equiv D$ $\to E \not\equiv A$	$E \not\equiv B$	$F \not\equiv B$ $\to E \not\equiv C$	$E \not\equiv D$	
$F \not\equiv A$	$D \not\equiv C$ $\to F \not\equiv B$	$F \not\equiv C$	$F \not\equiv B \vee C \not\equiv B$ $\to F \not\equiv D$	$F \not\equiv E$

The pair A, C gives $C \equiv A \to D \equiv B$, which contraposes into the rule $D \not\equiv B \to C \not\equiv A$.

For the pair D, B we get a tautology which tells us nothing we do not already know, so we let it go unsaid (and leave its cell in the table blank).

For the pair E, C we get $E \not\equiv C \vee F \not\equiv B \to E \not\equiv C$, for which we only need to write

the interesting part $F \not\equiv B \rightarrow E \not\equiv C$.

With such a regular form, a simple algorithm can use the starting facts and the rules to deduce all facts. In step P2 we apply the rules to the facts from step P1 to find that $F \not\equiv B$ and $F \not\equiv D$. These are the new facts in P2.

In step P3 we apply the rules to the new facts from P2 to determine that $E \not\equiv C$ and $E \not\equiv A$.

In step P4 we see that there is nothing new to learn from the new facts from P3, so we terminate.

B.1.1 Distinguishing sequences

The results of the algorithm can be used to find distinguishing sequences for states, including the length of the distinguishing sequence.

The facts of step P1 give the pairs that are 1-distinguishable, because they can be distinguished by looking at the output on some single input.

The facts of step P2 give the pairs that are 2-distinguishable. For example, we use $C \not\equiv B$ (from P1) and $C \not\equiv B \rightarrow F \not\equiv D$ to conclude $F \not\equiv D$. This proof that $F \not\equiv D$ suggests how to find the distinguishing sequence for F, D: first apply an input of 1 to drive F, D to C, B. We know C and B are distinguishable by different outputs (a P1 fact) for an input of 1. So the distinguishing input sequence for F, D is 1,1.

B.2 State partitioning

An algorithm substantially the same as the above but diagrammed rather differently, known variously as the Huffman-Mealy method [71] and the Moore reduction procedure [62], is worth reviewing. This approach emphasizes successive refinement of a partition on the states.

We begin with an initial partition of the states based on their outputs. In response to inputs 0,1, the states A, C, and E have outputs 0,1, and the states B, D, and F have outputs 0,0. For the first row of the diagram we write the states grouped this way. Also,

each state is annotated below with its next-states for all inputs.

A	C	E	B	D	F
E, D	E, B	C, F	F, D	F, B	B, C

Then we begin to iteratively refine this partitioning until it is proper. In a proper partition, for each input, the corresponding next-states are all in the same partition. (This requirement is trivially satisfied if a partition contains only one state.) For the first partition in the diagram above, E, E, C are all in the same partition (the first), and D, B, F are all in the same partition as well (they are all in the second). So far, so good.

For the second partition, F, F, B are all in the same partition, but D, B, C are not. So we refine the second partition to fix this, putting B and D (which on an input of 1 go to D and B, respectively) on one side, and F (which goes to C on a 1 input) on the other.

A	C	E	B	D	F
E, D	E, B	C, F	F, D	F, B	B, C

And we iterate. Putting F in its own partition "breaks" the first partition on input 1 — though E, E, C are still in the same partition, D and B are no longer in the same partition as F, so we must split again.

A	C	E	B	D	F
E, D	E, B	C, F	F, D	F, B	B, C

Since every partition now satisfies the requirement, we terminate.

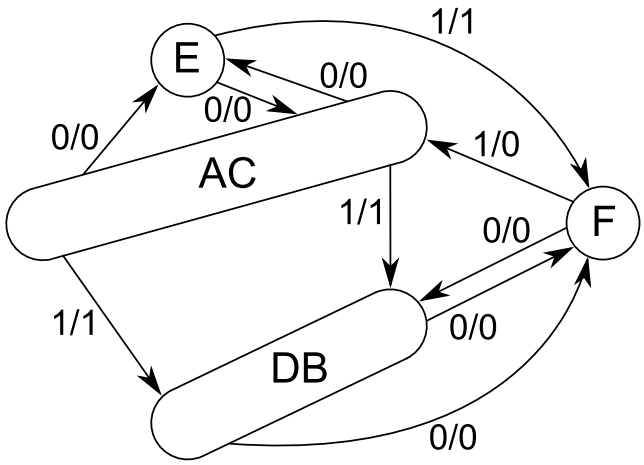

Figure B.2: The minimal equivalent to M_1.

B.3 Result

By two different means we have arrived at an equivalence relation on the states of the machine M_1.

Both the final set of facts from the propositional approach, and the final partition in the refinement approach, lead us to conclude that the states A and C are equivalent, and D and B are equivalent. The resulting minimized machine is depicted in Figure B.2. In the Figure, states A and C are merged into one new state labeled AC, and the merge of D and B is labeled DB.

Bibliography

Bibliography

[1] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*, 2nd ed. New York: Springer, 2008.

[2] W. D. Kelton, R. P. Sadowski, and N. B. Swets, *Simulation with Arena*. Boston: McGraw-Hill Higher Education, 2010.

[3] B. P. Douglass, *Real time UML*. Boston: Addison-Wesley, 2004.

[4] G. N. Roberts and R. Sutton, *Advances in unmanned marine vehicles*. London: Institution of Electrical Engineers, 2006.

[5] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[6] M. S. Branicky, "A unified framework for hybrid control: model and optimal control theory," *IEEE Transactions on Automatic Control*, vol. 43, no. 1, pp. 31–45, 1998.

[7] L. Ljung, *System identification : theory for the user*. Upper Saddle River, NJ: Prentice Hall PTR, 1999.

[8] O. Nelles, *Nonlinear system identification : from classical approaches to neural networks and fuzzy models*. Berlin; New York: Springer, 2001.

[9] E. F. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies*, C. E. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton University Press, 1956.

[10] W. Pincus, "Nuclear weapon's refurbishing woes draw congressional attention to treaty," *The Washington Post*, 2009, Tuesday, August 4.

[11] J. V. Last, "The fog of war: forgetting what we once knew," *The Weekly Standard*, vol. 14, no. 33, 2009, May 18.

[12] T. Baugh, *MSP430 state machine programming : with the ES2274*. SoftBaugh, 2008.

[13] J. Andersson, J. Huselius, C. Norstrom, and A. Wall, "Extracting simulation models from complex embedded real-time systems," in *Software Engineering Advances, International Conference on*, October 2006, p. 7.

[14] H. Hungar, T. Margaria, and B. Steffen, "Model generation for legacy systems," in *RISSEF 2002*, ser. LNCS, M. Wirsing et al., Ed., vol. 2941. Springer-Verlag, 2004, pp. 167–183.

[15] T. H. Feng, L. Wang, W. Zheng, S. Kanajan, and S. A. Seshia, "Automatic model generation for black box real-time systems," in *Design, Automation and Test in Europe (DATE)*, 2007.

[16] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers*, vol. C-21, no. 6, pp. 592–597, 1972.

[17] K. S. Fu, *Syntactic pattern recognition and applications*. Englewood Cliffs, N.J.: Prentice-Hall, 1982.

[18] H. Bunke and A. Sanfeliu, *Syntactic and structural pattern recognition : theory and applications*. Singapore ; New Jersey: World Scientific, 1990.

[19] G. W. Hart, "Minimum information estimation of structure," Ph.D. dissertation, MIT Dept. of Electrical Engineering and Computer Science, 1987.

[20] E. M. Gold, "Language identification in the limit," *Inf. and Control*, vol. 10, pp. 447–474, 1967.

[21] F. Denis, "Learning regular languages from simple positive examples," *Machine Learning*, vol. 44, pp. 37–66, 2001.

[22] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications*. New York: Springer, 1997.

[23] E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, pp. 302–320, 1978.

[24] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, 1987.

[25] H. Jack, "Automating manufacturing systems with PLCs," April 2010, version 7.0.

[26] H. Jack and A. Sterian, "Control with embedded computers and programmable logic controllers," in *Mechatronic System Control, Logic, and Data Acquisition*, R. H. Bishop, Ed. CRC Press, 2008, ch. 25.

[27] *101 Basics Series, Learning Module 24: Programmable Logic Controllers (PLCs)*, Eaton Corp., February 1999, publication No. TR.09E.02.T.E.

[28] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie, *Systems and software verification : model-checking techniques and tools*. Berlin Heidelberg: Springer-Verlag, 2001.

[29] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[30] S. E. Verwer, M. M. de Weerdt, and C. Witteveen, "Identifying an automaton model for timed data," in *Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands (Benelearn)*, Y. Saeys, E. Tsiporkova, B. D. Baets, and Y. van de Peer, Eds., 2006, pp. 57–64.

[31] R. Alur, L. Fix, and T. A. Henzinger, "Event-clock automata: a determinizable class of timed automata," *Theoretical Computer Science*, vol. 211, pp. 1–13, 1999.

[32] P. Supavatanakul, C. Falkenberg, and J. Lunze, "Identification of timed discrete-event models for diagnosis," Ruhr-University Bochum, Tech. Rep. 2003.4, 2004.

[33] P. Supavatanakul, J. Lunze, V. Puig, and J. Quevedo, "Diagnosis of timed automata: Theory and application to the DAMADICS actuator benchmark problem," *Control Engineering Practice*, vol. 14, pp. 609–619, 2006.

[34] J. Martins, J. Dente, A. Pires, and R. V. Mendes, "Language identification of controlled systems: modeling, control, and anomaly detection," *IEEE Trans. SMC-C*, vol. 31, no. 2, 2001.

[35] S. J. Choi and T. G. Kim, "Identification of discrete event systems using the compound recurrent neural network: extracting DEVS from trained network," *Simulation*, vol. 78, no. 2, pp. 90–104, 2002.

[36] O. Grinchtein, B. Jonsson, and M. Leucker, "Learning of event-recording automata," in *Proceedings of the Joint International Conferences on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. LNCS, vol. 3253. Springer, 2004, pp. 379–396.

[37] ——, "Inference of timed transition systems," in *Proceedings of International Workshop on Verification of Infinite State Systems*, ser. Electronic Notes in Theoretical Computer Science, vol. 138(3), 2005, pp. 87–99.

[38] O. Grinchtein, B. Jonsson, and P. Pettersson, "Inference of event-recording automata using timed decision trees," in *Proceedings of International Conference on Concurrency Theory*, ser. LNCS, vol. 4137. Springer, 2006, pp. 435–449.

[39] O. Grinchtein, "Learning of timed systems," Ph.D. dissertation, Uppsala University, Uppsala, Sweden, 2008.

[40] D. Lee and K. Sabnani, "Reverse-engineering of communication protocols," in *Proceedings of the 1st International Conference on Network Protocols (ICNP)*, 1993, pp. 208–216.

[41] T. Berg, B. Jonsson, and H. Raffelt, "Regular inference for state machines with parameters," in *FASE*, ser. LNCS, vol. 3922. Springer, 2006, pp. 107–121.

[42] T. Bohlin and B. Jonsson, "Regular inference for communication protocol entities," Uppsala University, Tech. Rep. TR-2008-024, 2008.

[43] E. D. Sontag, *Mathematical control theory : deterministic finite dimensional systems*, 2nd ed., ser. Texts in Applied Mathematics. New York: Springer, 1998, no. 6.

[44] L. A. Zadeh and C. A. Desoer, *Linear System Theory*. New York: McGraw-Hill, 1963.

[45] W. J. Gilbert and W. K. Nicholson, *Modern algebra with applications*, 2nd ed. Hoboken, New Jersey: John Wiley & Sons, 2004.

[46] J. D. Lipson, *Elements of algebra and algebraic computing.* Reading, Massachusetts: Addison-Wesley, 1981.

[47] W. L. Brogan, *Modern Control Theory*, 3rd ed. Upper Saddle River, New Jersey: Prentice Hall, 1991.

[48] T. Sauer, *Numerical Analysis.* Boston: Pearson Addison Wesley, 2006.

[49] G. F. Simmons, *Differential equations : with applications and historical notes.* New York: McGraw-Hill Book Company, 1972.

[50] J. Misra, "Distributed discrete-event simulation," *Computing Surveys*, vol. 18, no. 1, pp. 39–65, 1986.

[51] V. K. Garg, *Elements of distributed computing.* New York: Wiley-Interscience, 2002.

[52] R. M. Fujimoto, *Parallel and distributed simulation systems.* New York: Wiley-Interscience, 2000.

[53] H. Vollmer, *Introduction to circuit complexity : a uniform approach.* Berlin: Springer, 1999.

[54] T. M. Mitchell, *Machine learning.* Boston: WCB/McGraw-Hill, 1997.

[55] M. Vidyasagar, "An introduction to some statistical aspects of PAC learning theory," *Systems & Control Letters*, vol. 34, pp. 115–124, 1998.

[56] E. M. Gold, "System identification via state characterization," *Automatica*, vol. 8, pp. 621–636, 1972.

[57] M. Arbib and H. Zeiger, "On the relevance of abstract algebra to control theory," *Automatica*, vol. 5, pp. 589–606, 1969.

[58] J. N. Webb, *Game theory : decisions, interaction and evolution.* New York: Springer, 2007.

[59] S. Haykin, *Communication Systems*, 3rd ed. New York: John Wiley & Sons, 1994.

[60] M. Fowler, *UML Distilled*, 2nd ed. Boston: Addison-Wesley, 2000.

[61] P. R. Halmos, *Naive Set Theory*, ser. Undergraduate texts in mathematics. New York: Springer-Verlag, 1974.

[62] Z. Kohavi, *Switching and finite automata theory*, 2nd ed. Boston: McGraw-Hill Book Company, 1978.

[63] P. D. Laird, *Learning from good and bad data.* Boston: Kluwer Academic Publishers, 1988.

[64] M. J. Kearns and U. V. Vazirani, *An introduction to computational learning theory.* Cambridge, Mass.: MIT Press, 1994.

[65] B. K. Natarajan, *Machine learning : a theoretical approach.* San Mateo, CA: M. Kaufmann Publishers, 1991.

[66] S. Awodey, *Category Theory*, ser. Oxford logic guides. Oxford; New York: Oxford : Clarendon Press, 2006, no. 49.

[67] Y. Sakakibara, "On learning from queries and counterexamples in the presence of noise," *Information Processing Letters*, vol. 37, pp. 279–284, 1991.

[68] J. Berstel, D. Perrin, and C. Reutenauer, *Codes and Automata*, ser. Encyclopedia of Mathematics and its Applications. Cambridge: Cambridge Universtiy Press, 2010, no. 129.

[69] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning.* New York: Springer-Verlag, 2001.

[70] A. V. Raman and J. D. Patrick, "The sk-strings method for inferring PFSA," in *Proceedings of the 14th International Conference on Machine Learning, ICML97*, 1997.

[71] F. J. Hill and G. R. Peterson, *Introduction to switching theory and logical design*, 3rd ed. New York: Wiley, 1981.

Curriculum Vitae

Donald E. Jarvis is an electrical engineer with the Naval Research Laboratory, Washington, D.C. He received the B.S.Engr. (computer science/engineering) and Ph.B. (bachelor of philosophy) in 1994 and M.S.Engr. (electrical engineering) in 1999 from the Catholic University of America, Washington, D.C. His research interests involve the application of systems theory methods to simulation, identification, and control problems arising in Naval electronic warfare.