The Pennsylvania State University
The Graduate School

DECIDABILITY AND OPTIMALITY IN PUSHDOWN CONTROL SYSTEMS: A NEW APPROACH TO DISCRETE EVENT CONTROL

A Thesis in
Industrial Engineering and Operations Research
by
Christopher Griffin

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2007

The thesis of Christopher Griffin was reviewed and approved* by the following:

Tom Cavalier
Professor of Industrial Engineering
Thesis Advisor, Chair of Committee

Terry Friesz
Harold and Inge Marcus Professor of Industrial Engineering

José Ventura
Professor of Industrial Engineering

George Kesidis
Professor of Computer Science and Engineering

Richard R. Brooks
Special Signatory
Associate Professor of Electrical and Computer Engineering
Clemson University

Stephen A. Racunas
Special Signatory
Research Scientist, Computational Learning Lab
Stanford University

Richard J. Koubek
Professor of Industrial Engineering
Department Head, Industrial and Manufacturing Engineering

*Signatures are on file in the Graduate School.

Abstract

Supervisory Control Theory (SCT) was developed by P. J. Ramadge and W. M. Wonham in the late 80's as a method of understanding control of a wide variety of discrete event dynamic systems. A discrete event dynamic system (DEDS) is characterized by a directed, labeled graph. The nodes of this system are called the states, the arcs of the system are called transitions. Each transition has an event label. In this sense, an event causes the DEDS to change from one state to another. There is a special state called the starting state and a set of special states called the terminal, marked or final states. A system modeled by a state machine is assumed to exist in one of the states and transition from state to state as a result of the generation or execution of events. The alphabet of events is partitioned into controllable and uncontrollable events. Controllable events may be *disabled* by a system supervisor. Given a system to control (called the plant) modeled by a DEDS, a supervisor will disable certain events from occurring in the system in order to achieve a certain desired behavior. The supervisor is often designed as a second DEDS whose transitions correspond to *enabled* events. When the two DEDS are run in parallel, the supervisor may exert enabling and disabling control over the plant.

SCT is well understood in the case when both the plant and the supervisor are modeled as finite state machines; i.e., the number of DEDS states in both the plant and supervisor are finite. Several authors have remarked that the theory of SCT need not be restricted to the finite state case and in many ways they are correct. However, with the notable exception of a few Petri Net models, almost all of the results on SCT have been restricted to finite state systems. Specifically, decision algorithms for controllability and optimality are all restricted to the finite state case because in the infinite state case they become poorly defined.

The purpose of this thesis is three-fold: First, we develop an extended SCT that enables the modeling of infinite state systems by adding pushdown stack memory to the finite state machine models of the DEDS supervisors. Second we show that there are strong computability bounds on the nature of our new extended models with respect to the primary controllability criterion for discrete event control. That is, certain classes of control specifications cannot be provably enforced. Third, we provide a new theory of parametric optimal control in DEDS systems that extends to cases when the controller is a pushdown machine. We explore the problem of synthesizing a controller that will act optimally for all time. We derive a novel branch and bound algorithm for synthesizing such an optimal controller.

Three applications of the theoretical framework developed in this thesis are provided. The first shows how pushdown machines are capable of modeling subsets of the 802.11 wireless network protocol and displays methods for analyzing such protocols for robustness to attacks. The second application demonstrates how to use discrete event control to model computer security policies and shows how the pushdown stack memory of a deterministic pushdown machine can be used to enumerate file access information. The third example shows how the pushdown stack model can generalize the NASA Livingstone modeling system for use in validating robotic control systems for missions in deep space.

Table of Contents

List of Figures ix

List of Tables xi

Glossary of Symbols and Terms xii

Acknowledgments xv

Chapter 1
Introduction and Thesis Road Map **1**
1.1 Discrete Event Control History and Project Summary 1
1.1.1 New Results on Infinite State Discrete Event Control Systems and Thesis Overview 5
1.2 Example Problems 6
1.2.1 Alice's' Supervision Problem 6
1.2.2 Machine Maintenance 6
1.3 Thesis Road Map 8

Chapter 2
An Introduction to Classical Discrete Event Control and Necessary Results from Formal Language Theory **11**
2.1 Introduction to Formal Languages 11
2.2 Finite State Machines 14
2.3 Pushdown Machines 18
2.4 Properties of **REG**, **DCFL** and **CFL** 23
2.5 Decidability and Turing Machines 24
2.6 Decidable Properties of the Languages **REG**, **DCFL** and **CFL** . . 25
2.7 Algorithms on FSM and DPDA 27
2.7.1 Prefix Closure of Regular and Deterministic Context Free Languages 27

2.7.2 Complementation of a language in **REG** 28
2.7.3 Complementation of a language in **DCFL** 28
2.7.4 Normal Form for PDA 29
2.7.5 Predicting Machines 30
2.7.6 Intersection of Languages 30
2.8 Introduction to Discrete Event Control 31
2.8.1 An Algorithm to Check Controllability 34
2.8.2 Identifying the Plant Model 37
2.8.3 Supremal Controllable Sublanguage 37
2.9 Decidability in Control Theory 38

Chapter 3
On the Decidability of the Control Predicate in Pushdown Systems 40
3.1 Introduction and Chapter Overview 40
3.2 Proof of Main Theorem 41
3.3 Testing Controllability 43
3.4 Example 46
3.5 Algorithmic Complexity 51
3.6 Undecidability Result 51
3.7 Conclusion 52

Chapter 4
Verification of Secure Network Protocols in Uncertain Environments 53
4.1 Introduction 53
4.2 Application of Discrete Event Systems to Network Protocols 55
4.2.1 Protocol Design 56
4.3 Resisting Network Attacks 58
4.3.1 Correcting Protocols 61
4.4 Future Directions 62
4.5 Conclusion 63

Chapter 5
An Algorithm to Compute the Supremal Controllable Sublanguage for a Class of Pushdown Systems 64
5.1 Introduction 64
5.2 An Algorithm to Compute A Controllable Sublanguage 65
5.2.1 Previous Results 65
5.2.2 Algorithm Derivation 66

5.2.2.1 Step 1 . . . 67
5.2.2.2 Step 2 . . . 68
5.2.2.3 Step 4 . . . 69
5.2.2.4 Algorithm Analysis . . . 71
5.3 Undecidability Results . . . 72
5.4 Parametric Discrete Event Control . . . 75
5.4.1 Algorithm . . . 77
5.5 Conclusion . . . 79

Chapter 6
State Space Models for Software Security Quantification 80
6.1 Introduction . . . 80
6.2 Modeling the Security State as a Finite State Machine . . . 81
6.3 Safety Properties of Sublanguages . . . 82
6.4 Undetected Transitions . . . 85
6.5 Extension to DPDA Example . . . 87
6.6 Conclusion . . . 89

Chapter 7
Optimal Parametric Discrete Event Control 90
7.1 Introduction . . . 90
7.2 Optimal Control . . . 92
7.2.1 Objective Function . . . 93
7.2.2 Computing the Objective Function . . . 94
7.2.3 The Function $M_K(X_1, \ldots, X_n)$ . . . 97
7.2.4 Constraints: $K \in \mathcal{C}(L)$ and $M_K(X_1, \ldots, X_n) \models \Pi$ . . . 97
7.2.5 Branch-and-Bound Algorithm for Optimization . . . 98
7.3 Computational Analysis of the Algorithms . . . 104
7.3.1 Computational Complexity Upper Bound . . . 104
7.3.2 Computational Experience . . . 105
7.3.3 Streamlining Operations for Executing Algorithm $\mathfrak{C}$ in the Large Scale . . . 106
7.4 Example . . . 107
7.5 Relation to the Work of Sengupta and LaFortune . . . 109
7.6 Conclusions . . . 110

Chapter 8
Exception Handling Control and an Application to Robotic Controller Design 111
8.1 Introduction . . . 111

8.2 Exception Handling Controllers 112
8.3 Properties of Exception Handling Controllers 114
8.4 Types of Stack Modification Operations 116
8.5 Application of the EHC Parametrization to Robot Control 118
8.6 Livingstone Models and Robotic Software Verification 118
8.6.1 Running the Algorithm 125
8.7 Induction Proof of Queuing Controllability 126
8.8 Conclusions and Future Directions 129

Chapter 9
Conclusion and Future Directions **130**
9.1 Summary 130
9.2 Future Directions 132

Bibliography **135**

List of Figures

1.1 The Life-Cycle of a Machine at Tina's Company 7
1.2 Graphical Representation of the Corporate Policy for Machine Repair 8

2.1 The Operation of a Finite State Machine on a Word 15
2.2 A Finite State Machine that accepts L. 15
2.3 A Finite State Machine that accepts Σ^* when $\Sigma = \{0, 1\}$. 15
2.4 The action of a PDA as it reads in a string of symbols. 19
2.5 An example PDA that accepts $\{0^n 1^n | n \geq 1\}$. 21
2.6 A simple three state plant model with a single starting state and a single final state. The starting state is illustrated by an arrow that connects to no state. 32
2.7 A simple control language that disables c whenever b is seen an even number of times. 35
2.8 An intersection of M and G. 35
2.9 The complement machine to M. 36
2.10 The intersection of M' and G. 36

3.1 Plant Model and Controller for Widget Production 48
3.2 An intermediate form of M that scans all input but accepts the same language as M. 49
3.3 A machine that accepts of $\left(\overline{K}\right)^c$. Note there are a number of useless transitions. 50
3.4 The weave of M^c and G. This automaton accepts $\mathcal{L}(G) \cap \mathcal{L}(M)^c$. Useless transitions have been grayed out. 50

4.1 Upper-level Automaton 56
4.2 Lower-level Automaton 57
4.3 SARAH Automaton 60
4.4 Spiral design method for wireless network protocols 62

6.1 A small state machine model of the security of a system using an unpatched version of *xterm* 83

6.2 A small state machine model of the security of a system using an unpatched version of *xterm* when an unknown attack disables the utility of patching *xterm*. 86
6.3 A small security policy for using an unpatched version of *xterm* when an unknown attack disables the utility of patching *xterm*. . . 87
6.4 System that can count the number of files modified by a system intrusion and correct each file. 88

7.1 A fully expanded tree; nodes are labeled in order they will be investigated. The immutable set is shown below each node. 99
7.2 The controller M_K that begins the branch-and-bound process. . . . 107
7.3 The branch-and-bound tree resulting from Algorithm $\mathfrak{C}$ when running on the problem given above. 108
7.4 Solution to problem given above. 108

8.1 Exception Handling Controller–General Structure 113
8.2 The flow of information in our scenario. 120
8.3 Robotic rover plant model. 120
8.4 Exception handlers recognize specific requests and store them on the stack below the currently executing command. 121
8.5 An quasi-EHC with looping exception handling transitions suppressed for space. 122
8.6 The branch-and-bound tree with various fathomed nodes. 126
8.7 The minimum cost quasi-exception handling control system for robotic control. 127
8.8 Exception handler that preserves the top two stack commands. . . . 128

List of Tables

3.1 Event definitions and descriptions 47

4.1 Controllable Events (Σ_c) 55

4.2 Uncontrollable Events (Σ_u) 55

7.1 Running times (seconds) for Java Application executing Algorithm $\mathfrak{C}$.105

7.2 Running times (seconds) for C++ Application executing Algorithm $\mathfrak{C}$ using only finite state machines for M_K. 106

Glossary of Symbols and Terms

Event An instantaneous occurrence that causes a state change in a dynamic system.

Discrete Event Dynamic System (DEDS) A dynamic system whose state evolution is driven by discrete events.

Finite State Machine (FSM) A dynamic system with a finite number of states. Transitions between states are caused by discrete events.

Stack A data structure that operates on a last-in-first-out principle. Symbols are pushed onto the top of the stack and may be popped off the top of the stack.

Pushdown Automata (PDA) A dynamic system whose true state is determined by a finite state and the configuration of a pushdown stack of symbols. Transitions between state/stack configurations are caused by discrete events.

Deterministic Pushdown Automata (DPDA) A PDA whose transition structure is completely deterministic; i.e., if the state/stack configuration of the system is known and an event occurs, then the next state/stack configuration is known precisely.

Predicate with variables $X_1, \ldots, X_n$ A statement about variables $X_1, \ldots, X_n$ that is either true or false.

Logical Sentence A statement about mathematical objects that is either true or false. Given a predicate with variables $X_1, \ldots, X_n$, if values are supplied to these variables, then a logical sentence is produced.

$M_K(X_1, \ldots, X_n) \models \Pi$ If Π is a set of logical sentences about the mathematical object $M_K(X_1, \ldots, X_n)$ that is defined by supplying values to the variables $X_1, \ldots, X_n$, then $M_K(X_1, \ldots, X_n) \models \Pi$ says that every sentence in Π is true for a given set of values for the variables $X_1, \ldots, X_n$.

Decidable A logical sentence is decidable if there is an algorithm (a computer program) that can be used to decide whether the sentence is true or false for any possible set of variable inputs.

$\emptyset$ The empty set.

$|\cdot|$ The size or length of anything.

$\subseteq$ The subset-equality relation.

$\cap$ The set intersection symbol.

Σ The set of discrete events that may occur in the system.

Σ_c The set of controllable system events.

Σ_u The set of uncontrollable system events.

String/Word An ordered sequence of events. (Also called an event trace or history.)

Concatenation The concatenation of two words (s and t) is the word that results when they are combined in order (i.e., st).

Prefix of a word w Any word s such that there exists t so that $w = st$.

ϵ The empty string–also the null event signifying no event has occurred.

Σ^* The set of all finite strings made up of elements of an alphabet Σ (including ϵ). Note: this also applies to Σ_u^*.

Language (in Σ^*) Any subset of Σ^*.

Language of a Dynamic System ($\mathcal{L}(\cdot)$) The set of possible event traces (words) that may be emitted by a DEDS as it evolves. (Here $\cdot$ denotes the name of the DEDS in question.)

Accepted Language of a Dynamic System ($\mathcal{L}_{\mathcal{M}}(\cdot)$) The set of possible event traces (words) that may be emitted by a DEDS as it evolves from a starting state to a legal ending state. (Here $\cdot$ denotes the name of the DEDS in question.)

Prefix closure (of a language) Let L be a language. Then the prefix closure of L (denoted $\overline{L}$) is the set of all prefixes of words in L.

Controllability of language K with respect to language L Language K is controllable with respect to language L (assuming $K \subseteq L$) if for every string s in the prefix closure of K and for every string $u \in \Sigma_u^*$, if su is an element of the prefix closure of L, the su is an element of the prefix closure of K. Note, controllability is a predicate on the languages K and L.

$\mathcal{C}(L)$ The set of controllable sublanguages of a language L. If $K \in \mathcal{C}(L)$, then $K \subseteq L$ and K is controllable with respect to L.

Supremal Controllable Sublanguage Given K and L languages with $K \subseteq L$, then the supremal controllable sublanguage of K with respect to L is the language $\sup_{\mathrm{C}}(K) \subseteq K$ such that (i) $\sup_{\mathrm{C}}(K)$ is controllable with respect to L and (ii) if there is another language $K' \subseteq K$ such that K' is also controllable with respect to L, then $K' \subseteq \sup_{\mathrm{C}}(K)$.

$\sup_{\mathrm{C}}(K)$ The supremal controllable sublanguage of a language K with respect to some known language L.

Acknowledgments

My adviser for this dissertation was Tom M. Cavalier of The Pennsylvania State University. I gratefully acknowledge his support during the development of this thesis. I would also like to acknowledge the support of Dr. Stephen Simpson of the Penn State Mathematics Department. Dr. Simpson acted as my adviser during my Masters work and introduced me to the concept of decidability. This concept is used throughout this thesis. I would also like to acknowledge my co-workers at the Penn State Applied Research Lab, who helped make some of this research possible. Finally, I acknowledge the anonymous reviewers who helped me revise and publish the contents of Chapters 3 - 6 of this thesis. I am forever indebted to these individuals.

Dedication

For my parents, who started me down the path of learning.

Chapter 1

Introduction and Thesis Road Map

1.1 Discrete Event Control History and Project Summary

In January of 1987 P. J. Ramadge and W. M. Wonham published the first paper on discrete event control [RW87c]. Their approach was to use a state machine to model a dynamical system–similar to a Markov chain without probabilities. As time passed, the dynamics of the system were expressed through streams of events, similar in spirit to the traces of a discrete event simulator. For the sake of this thesis, an event is an instantaneous occurrence that causes a system to change from one state to another [CL99]. By a transition, we mean a state transition caused by the occurrence of an event.

Control of the symbolic dynamic system was defined in terms of a supervisor that observed the event trace output and turned on or off a certain subset of the events. Events that could be disabled were called *controllable*, while the remaining events were called *uncontrollable*. In [RW87c] Ramadge and Wonham studied the following problem: Given the set of all possible event traces called L, and a subset of target (or objective or desired) event traces $K \subseteq L$, does there exist a supervisor that ensures the dynamical system will only output event traces in the target set K. Ramadge and Wonham identified a key necessary condition for a positive answer to this question and called it *controllability.* (Controllability is formally defined in Chapter 2.)

Two months after their first paper appeared, Ramadge and Wonham published a second paper on discrete event control [RW87b]. In this paper, they considered the problem of recovering when the desired set of event traces K is not controllable with respect to the set of all possible event traces L. In this case, Ramadge and Wonham provided an algorithm for determining a new set $\sup_{\mathrm{C}}(K) \subseteq K$ that was controllable and for which there was a supervisor to enforce the behavior of $\sup_{\mathrm{C}}(K)$. Thus, Ramadge and Wonham were able to provide a way to find an usable subset of the desired behaviors for which an enforcing supervisor could be found. Ramadge and Wonham [RW87a] then went on to explore the problem of designing K so that it is the intersection of several desired event traces. This is called the modular supervisory control problem.

The work was picked up by several authors, far too many to list exhaustively. Two of the most prolific of these authors were R. Kumar and V. K. Garg. In 1990 Kumar, Garg and others began investigating methods of computing $\sup_{\mathrm{C}}(K)$ more efficiently when K and L had special, more restrictive properties [BGK+90]. (We discuss this work in detail in Chapter 2.) This work culminated in a new algorithm for computing $\sup_{\mathrm{C}}(K)$ in 1991 [KGM91]. This algorithm was shown to be the most efficient algorithm for computing $\sup_{\mathrm{C}}(K)$. In 1995, Kumar and Garg summarized the state of the art in Discrete Event Control in their book Modeling and Control of Logical Discrete Event Systems [KG95b].

One of the key drawbacks in all this literature were the assumptions on the state machine model of the dynamical system (and thus the assumptions on the sets K and L). All the algorithms given in [RW87c, RW87b, RW87a, BGK+90, KGM91, KG95b] operated only on *finite* state machine models. All these authors (Ramadge, Wonham, Kumar etc.) noted that there was no reason that discrete event control should be limited to finite state models, while subsequently stating that they would restrict their attention to these models because the algorithms they published were not convergent with infinite state systems.

This was changed in 1993 when Sreenivas [Sre93] published the first paper on Discrete Event Control using Petri Nets. A Petri Net is a more complex model of a discrete event dynamic system that is capable of modeling an infinite number of states. Sreenivas showed that for certain Petri Net models (that produce event trace sets K and L) it was *impossible* to decide whether K was controllable with respect to L. This work was continued by Kumar and Halloway in 1996 [KH96] when they discussed modeling the discrete event dynamic system with a Petri net and modeling a system that outputs the target language K with a finite state machine. What is interesting about *both* of these works is they assume the restrictive properties *originally assumed* in [BGK$^+$90].

Investigation into optimality of discrete event dynamic systems began almost as soon as Ramadge and Wonham introduced their first papers. The subject of optimality in discrete event systems is still an active area of research and no single accepted view of optimality has yet come to the fore. Further, many of these approaches are highly distinct.

Optimal control of discrete event systems was originally investigated by Passino and Antsaklis [PA89]. This approach to optimal control attempts to drive the plant model along the shortest path to a final state from the initial state. As such, it uses a simple modification of Dijkstra's algorithm [CLRS01,BJS04]. Furthermore, it modifies the basic assumptions of the Supervisory Control Theory of Ramadge and Wonham by assuming a forced event model instead of an enabled event model. Kumar and Garg [KG95a] were the first to formally study optimal supervisory control using the enabled event paradigm. They consider finite state models of the plant and controller. They assume costs associated to transitions caused by events *and* event disabling. Further, pay-offs are associated with the various reachable states. An alternate formulation of optimal control in discrete event systems is in Brave and Heymann [BH93]. In their formulation, they attempt to minimize the cost of keeping a state machine within a finite set of states. Brave and Heymann do not include a control cost (a cost for disabling an event) and hence their approach is not as general as [KG95a].

The most general study of optimal control supervisory control is undertaken by Sengupta and Lafortune [SL98]. What is interesting about their approach is the underlying assumptions they make. They consider only *non-blocking* supervisors; i.e., they assume that any supervisory control law is controllable if and only if it *does not* disable an uncontrollable event. That is, they restrict the class of supervisors φ they consider to obtain fully realized theory. After defining a cost function, they attempt to minimize the *worst case cost* due to uncontrollable events. Hence, they obtain a minimax problem. Further, their analysis uses a dynamic programming (DP) approach to solving the underlying optimization problem. While extremely useful in solving small scale problems, DP suffers from the well known *curse of dimensionality* [Bel57]. Hence, this approach to optimal control of discrete event systems may only be suitable for small scale problems. This work is carried on by Marchand, Boivineau and LaFortune in [MBL00]. In this case, the authors pose several discrete event control problems (or goals) and then solve them using the methodology in [SL98]. They then attempt to stitch these problems together by means of a traveling salesman approach to create an optimal scheduler for executing multiple goals. While interesting, it seems that the traveling salesman approach is not only overly complicated but entirely unnecessary since multi-stage goal scheduling can be accomplished using a simplified mixed integer programming structure.

The latest investigation into optimal control of discrete event systems was undertaken by Ray et al. [WR02,RP02,FRL02]. In these papers, the authors proposed a *language measure*[1]. This measure is based on the transition structure of an automaton accepting the language (set of event traces). Unfortunately, this means that the measure defined is in fact a state machine measure and not unique to the language accepted by the state machine. This fact has been pointed out by other authors. In [WR02], a transition matrix is constructed that is similar to the transition matrix of a Markov chain, save for the fact that the values associated to the transitions leaving a state must add to less than or equal to 1 (as opposed to 1 in the case of Markov chains). Costs and rewards are assigned to states. The problem of determining an optimal controller essential is transformed into the problem of finding an optimal pure decision strategy in this unusual sub-Markovian structure. It is worth noting that all the optimal control discrete event control papers referenced operate only on finite state machine models of the discrete event dynamical systems.

[1] By measure we actually mean a mathematical measure, a mapping from a sigma-algebra to the real numbers.

1.1.1 New Results on Infinite State Discrete Event Control Systems and Thesis Overview

In 2006 Griffin [Gri06] began investigating a new model for infinite state discrete event dynamical systems. To do this, he added an infinite stack memory to the finite state machine model of the dynamics. This effectively provided the dynamic system with an infinite memory property. The impact this had on determining controllability was discussed in [Gri06] and [Gri07]. These results form the basis of the novel results contained in this thesis.

The second set of new material presented in this thesis concerns optimal discrete event control in infinite state discrete event systems defined using a pushdown stack. We take inspiration from the work of Sengupta and LaFortune [SL98] by extending the problem they study and presenting a solution method that works for our infinite state discrete event dynamic system models. One of the key differences between their paper and the work presented here is the optimizing algorithm. They use a dynamic programming solution to identify an optimal and controllable target set of event traces K, when given a set of event traces L produced by a discrete event dynamic system. Dynamic programming works in this case because the state space they consider is finite. We could not extend their dynamic programming approach because our state space is infinite in nature. Consequently, we define a bounded depth first search approach that is able to exhaustively search a finite tree of potential solutions. We are able to show that this is sufficient to identify an optimal supervisor, even though the resulting supervisor may be modeled by an infinite state system. In essence, we reduce an infinite state problem to an equivalent finite search problem.

Finally, three applications of the theoretical framework developed in this thesis are provided. The first shows how pushdown machines are capable of modeling subsets of the 802.11 wireless network protocol and displays methods for analyzing such protocols for robustness to attacks. The second application demonstrates how to use discrete event control to model computer security policies and shows how the pushdown stack memory of a deterministic pushdown machine can be used to enumerate file access information. The third example shows how the pushdown stack model can generalize the NASA Livingstone modeling system for use in validating robotic control systems for missions in deep space.

1.2 Example Problems

In this section, we illustrate some toy problems that capture the spirit of discrete event control problems.

1.2.1 Alice's' Supervision Problem

A supervisor Alice has hired a new worker Bob. Alice may task Bob with two tasks (for example filing and copying). In turn, Bob may succeed or fail at either of these tasks and will report this back to Alice. At Alice's company, there is a corporate cross-training policy, which states that workers cannot perform the same task more than twice in a row. Alice's approach to tasking is constrained by the corporate policy. The resulting behavior observed in the system (the sequence of tasking, successes and failures) is the result of Bob's behavior and Alice's policy of supervision. Suppose that Alice chooses a tasking policy for Bob. Techniques of discrete event control can answer the following questions:

1. Does Alice's policy conform to corporate policy?

2. Will Alice's policy cause a system dead-lock?

3. Is there a policy *like* Alice's policy that both conforms to corporate policy and does not cause deadlock?

4. When appropriate costs are assigned, we can ask: What is the least cost tasking policy for Alice that conforms to corporate policy?

1.2.2 Machine Maintenance

Tina's company[2] maintains a specialized machine that occasionally breaks down (event a). When this machine is broken, there are two possible courses of action. Either the machine is repaired in-house (event b) or it is sent out for repair (event

[2] Tina and Alice do not work at the same company.

c). A machine that is sent out for repair is eventually returned to the company (event r). The simple state diagram is shown in Figure 1.1 illustrating the life-cycle of the machine.

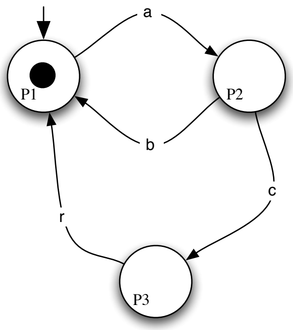


Figure 1.1. The Life-Cycle of a Machine at Tina's Company

We assume that the machine starts out working (State $P1$) and transitions to a broken state ($P2$) by transition a. From this state, the machine can either be repaired b, or it can be sent out for repair c. In the latter case, the machine is no longer located at the company (state $P3$). Once the machine is returned r, the machine is working again. We indicate that state $P1$ is the starting state by the arrow whose head touches state $P1$ and whose tail is free. The dark circle in state $P1$ indicates that we intend to consider only paths through this graph that terminate in state $P1$. This will be explained further in Chapter 2.

The company has a strict corporate policy that states that a machine can only be sent out for repair on alternate times of break-down. This policy was implemented as a cost saving measure. Tina may choose whether to send the machine out for repair or not, but she must act within company policy. A graphical representation of the corporate policy is shown in Figure 1.2.

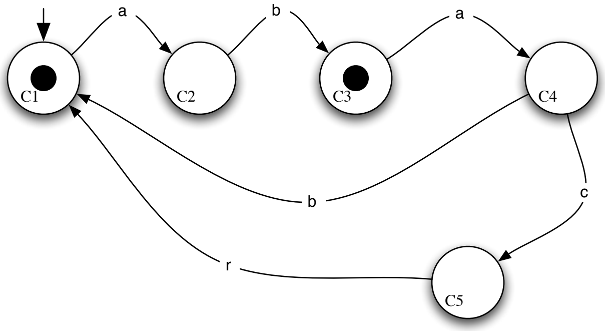


Figure 1.2. Graphical Representation of the Corporate Policy for Machine Repair

1.3 Thesis Road Map

This thesis is organized as follows[3]: In Chapter 2 we provide all necessary background information on Formal Language Theory and Discrete Event Control. By necessity, this chapter is quite long. In Chapter 3 we prove our first main result and show:

Theorem 1.3.1. *Controllability is decidable for a specification S generated by a deterministic pushdown automaton (DPDA) M and a plant language L generated by a finite state machine G. That is, if S is accepted by a DPDA and L is accepted by a finite state machine, then it is decidable whether $K = S \cap L$ is controllable with respect to L.*

We also provide an algorithm for deciding controllability in this case. (Algorithm $\mathfrak{A}$.)

Chapter 4 we apply the results of Chapter 3 to the problem of verifying complex wireless Internet protocols. In Chapter 5, we prove our next main result:

Theorem 1.3.2. *Let L be a prefix closed plant language and let K be a prefix closed language accepted by DPDA. Then the supremal controllable sublanguage of K ($\sup_{\mathrm{C}}(K)$) is also accepted by a DPDA.*

We further provide an algorithm (Algorithm $\mathfrak{B}$) for determining $\sup_{\mathrm{C}}(K)$ in this case. To do this, we use results proved in Chapter 3. We also define the

[3] All terms needed to understand these results are defined in Chapter 2. Additionally, key symbols are listed in the Glossary of Terms and Symbols that appears in the front matter. We provide this road map so that the reader can choose the order in which she wishes to read this thesis.

parametric control problem and show how to solve it using an algorithmic variation of Algorithm $\mathfrak{B}$.

In Chapter 6 we show how to use discrete event control models of computer security policies coupled with "Attack Graphs", models of computer system vulnerabilities, to derive a safe computer security policy using supremal controllable sublanguages.

In Chapter 7, we turn our attention to deriving an algorithm to solve the following non-traditional mathematical program:

$$\begin{aligned} \min \max_{w\in K} & C(w) \\ \text{s.t. } & \mathcal{L}_{\mathcal{M}}(M_K(X_1,\dots,X_n)) \subseteq L \\ & K = \mathcal{L}_{\mathcal{M}}(M_K(X_1,\dots,X_n)) \\ & M_K(X_1,\dots,X_n) \models \Pi \\ & K \in \mathcal{C}(L) \\ & (X_1,\dots,X_n) \in \mathbb{B} \end{aligned} \tag{1.3.1}$$

Here, w is a string in a discrete event language: L is an appropriately defined plant language; $X_1, \dots, X_n$ are binary variables; $M_K(X_1,\dots,X_n)$ is a function taking variables $X_1,\dots,X_n$ and returning a machine M_K; and K is the objective language. The statement $M_K(X_1,\dots,X_n) \models \Pi$ indicates that the machine $M_K(X_1,\dots,X_n)$ makes true a set of logical sentences Π that describe the structural properties of $M_K(X_1,\dots,X_n)$. The set $\mathcal{C}(L)$ denotes the set of all controllable sublanguages of L. To solve this problem, we derive a unique branch-and-bound algorithm and prove that it will derive an appropriate solution. The branch-and-bound algorithm makes use of Algorithms $\mathfrak{A}$ and $\mathfrak{B}$ derived in the preceding chapters. We discuss the computational complexity of this algorithm and discuss our experience in implementing it.

In Chapter 8, we apply our results from Chapter 7 to derive a specific form of parametrized discrete event controller called the *Exception Handling Controller*. We show that this control structure has properties making it ideal for use in our branch-and-bound framework. We then provide an example of its use as a natural extension to the NASA Livingstone software validation system and provide an example optimization and verification of a simple robotic control system. Finally, we provide conclusions and future directions in Chapter 9.

Chapter 2

An Introduction to Classical Discrete Event Control and Necessary Results from Formal Language Theory

2.1 Introduction to Formal Languages

Let Σ be a finite set called the *alphabet*. A word w is an ordered set of symbols that come from an alphabet Σ. Throughout, we may use the terms *string* and *word* synonymously. By Σ^* we denote the set of all finite words composed of elements of Σ. We denote the empty string by ϵ; since ϵ is a finite word, it is considered an element of Σ^*.

Example 2.1.1. Let $\Sigma = \{0, 1\}$, then Σ^* is the set of all finite binary strings. E.g., $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \dots\}$.

Any subset of Σ^* is called a *language*, though for brevity we will usually call the subsets of Σ^* languages. Naturally, Σ^* is itself a language.

Example 2.1.2. Let $\Sigma = \{0, 1\}$. Let $L = \{1, 11, 111, \dots\}$. Then L is a language in Σ^*.

When defining languages, we sometimes say that a set L is a *language over alphabet* Σ. This is equivalent to saying that Σ is an alphabet, and $L \subseteq \Sigma^*$. If

$K \subseteq L$ and L is a language over some alphabet Σ, then K is called a *sublanguage* of L and L is called a *superlanguage* of K. Hence, every language is a set of words. We may define the concatenation operation on two words w and v as you would expect. That is, w concatenated with v is the new word wv.

Example 2.1.3. Let $\Sigma = \{0, 1\}$ and let $w = 00$ and $v = 11$ be two words in Σ^*. Then $wv = 0011$.

Clearly, concatenation is associative but not (necessarily) commutative. It is further clear, that the empty word ϵ acts as an identity for this operation. Hence Σ^* is a monoid with the concatenation operator [Jac85].

When L and K are two languages, we may define their concatenation to be the pairwise concatenation of every string in L with every string in K. The operation is likewise associative, but not commutative.

Example 2.1.4. Let $L = \{00, 11\}$ and let $K = \{101, 11\}$, then

$$LK = \{00101, 0011, 11101, 1111\}$$

Let L and K be the two languages in question. Their concatenation is formally defined by the following predicate:

$$w \in LK \iff \exists u \in L \exists v \in K(uv = w) \tag{2.1.1}$$

Remark 2.1.5. Throughout this thesis, we use standard mathematical logic notation [Sim00]. By this we mean that existential ($\exists$) and universal ($\forall$) quantifiers are moved to the front of mathematical expressions. A parenthesis contains the main body of the statement. For example, the mathematical state $\exists x(P(x)$ should be read, "There exists and x such that statement $P(x)$ holds."

The $*$ notation (used on Σ^*) is a generalization of concatenation. Let $w \in \Sigma^*$ be a word. Then $w^2 = ww$; that is the concatenation of w with itself. We set $w^0 = \epsilon$ by definition. By w^* we mean the language containing all non-negative finite powers of w. That is, $w^* = \{\epsilon, w, ww, www, \dots\}$. We may extend this concept to languages immediately. Let $L \subseteq \Sigma^*$. By L^2 we mean LL. We define $L^0 = \{\epsilon\}$. Then we have $L^* = \{\epsilon\} \cup L \cup LL \cup \dots$. It is now easy to see why Σ^* is the set of all finite words composed of elements of Σ. This notation is called the *Kleene star* notation. Note because Σ^* is a *monoid* under concatenation we cannot define w^n for $n < 0$.

We say that a word u is a *prefix* of a word w if there is some other word v such that $uv = w$. Similarly, we say that v is a *suffix* of a word w if there is some other word u such that $uv = w$.

Example 2.1.6. Let $\Sigma = \{0, 1\}$ and let $w = 0011$ and $u = 0$ be two words in Σ^*. Then u is a prefix of w. Similarly, if $v = 11$, then v is a suffix of w.

It is trivial to note that a word w is a prefix of itself because ϵ, the empty string, allows us to write $w = w\epsilon$. Similarly, ϵ is a prefix of every word since for any word w, $w = \epsilon w$.

Fix Σ and let L be a language in Σ^*. The set of all prefixes of the words in L is called the *prefix-closure* of L and is denoted $\overline{L}$ (not to be confused with the set-theoretic complement of L in Σ^*, which we write as L^c). We have the following formal definition of $\overline{L}$:

$$u \in \overline{L} \iff \exists v \in \Sigma^*(uv \in L). \tag{2.1.2}$$

If L is a language and $L = \overline{L}$, then L is said to be *prefix-closed*.

We have the following formal definition of set-theoretic complement when $L \subseteq \Sigma^*$:

$$w \in L^c \iff w \in \Sigma^* \wedge w \notin L \tag{2.1.3}$$

Example 2.1.7. Let $\Sigma = \{0, 1\}$ and let $L = \{00, 11, 101\}$, then

$$\overline{L} = \{\epsilon, 0, 1, 00, 10, 11, 101\}.$$

Trivially we can see one property of $\overline{L}$ is that $L \subseteq \overline{L}$.

The concept of prefix closure is generalized by language quotients. Let L and K be two languages. Then the *right quotient of* L *by* K is defined as:

$$u \in L/K \iff \exists v \in K(uv \in L). \tag{2.1.4}$$

Put in simpler terms, a word u is in the quotient L/K if and only if there is some word v in the language K such that the concatenation uv is a word in L.

Example 2.1.8. Let L be a language over a fixed alphabet Σ. We can see from the formal definition of prefix-closure that $\overline{L} = L/\Sigma^*$.

There is likewise a natural concept of suffix-closure that is identical in every way to prefix closure, except we are concerned with the suffixes of a language instead of the prefixes. The *left quotient of K by L* generalizes this concept. Formally, the left quotient is defined as:

$$v \in L \backslash K \iff \exists u \in L(uv \in K) \tag{2.1.5}$$

Remark 2.1.9. The notation for left quotient is non-standard. Modern authors reverse the positions of L and K following conventions developed after 1979. We use the older form of the notation because Hunt and Rosenkrantz [IR74] use this notation and we make extensive use of one of their theorems, which use this notation.

Though these concepts seem dubiously useful, we will use of all of them in Chapters 3 and 5.

2.2 Finite State Machines

Imagine a word w written out on a ticker tape T and let M be some machine that when fed in T will indicate whether or not w is in some specified language L. We say that machine M recognizes a language L if when ever M is allowed to read a word, it indicates positively or negatively whether w is a member of L.

We often characterize the complexity of a language by the complexity of the machine required to recognize it. The simplest machine considered by computer scientists is the *finite state machine.* A *finite state machine*, abbreviated FSM and sometimes called a *finite state automaton*, is defined by a tuple $G = \langle Q, \Sigma, \delta, q_0, Q_f \rangle$ where Q is a finite set of states, Σ is the alphabet as it always is, δ is a relation in $Q \times \Sigma \times Q$ (i.e., $\delta \subseteq Q \times \Sigma \times Q$), $q_0 \in Q$ is called the starting state and $Q_f \subseteq Q$ are called the set of *final* or *marked* states. When we omit q_0 and Q_f, the system is sometimes called a *labeled transition system.*

We may envision a finite state machine operating in the following way: Again let T be a ticker tape with a word written on it. The finite state machine starts at the beginning of the tape in its initial state. It reads the first symbol on the tape and, using its transition relation, changes to a new state based on the input. The machine then moves to the next symbol and the process is repeated. If the machine is in a final state after it has consumed the entire tape, then the word is said to have been *accepted*. Otherwise, the word is not accepted. The process is illustrated in Figure 2.1.

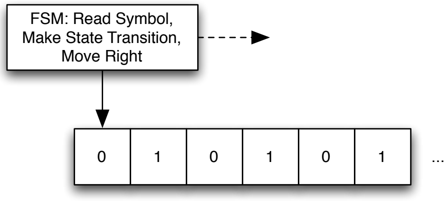


Figure 2.1. The Operation of a Finite State Machine on a Word

Example 2.2.1. Consider the language $L = (01)^* = \{\epsilon, 01, 0101, \dots\}$. This language is accepted by the finite state machine shown in Figure 2.2.

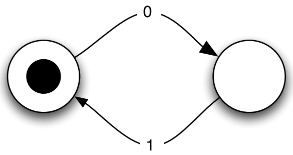


Figure 2.2. A Finite State Machine that accepts L.

It is important to note that language size is *not* an indicator of complexity. For example, the language Σ^* for any alphabet Σ is accepted by a finite state machine. To see this, consider the case when $\Sigma = \{0, 1\}$. The language Σ^* is accepted by the finite state machine shown in Figure 2.3.

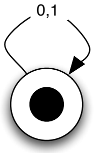


Figure 2.3. A Finite State Machine that accepts Σ^* when $\Sigma = \{0, 1\}$.

In discussing the operation of a finite state machine, we mentioned the problem of non-determinism. There are two types of non-determinism, ϵ-transitions and multiple transitions on a single symbol. Consider a finite state machine M and let q be a state and let σ be a symbol in Σ. If there are two (or more) states q' and q'' such that $(q, \sigma, q') \in \delta$ and $(q, \sigma, q'') \in \delta$, then clearly there is non-determinism when the machine is in state q and reads a single symbol σ from the input tape: the machine could transition either to q' or q''.

The second type of non-determinism arises from the use of ϵ-transitions. This transition type was not considered in our original definition of finite state machine. In our original definition, we only considered transitions defined on elements of Σ. Our definition can be extended to include it by considering the case when $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q$. By way of example, again let q, q' and q'' be states and let σ be a symbol in Σ. If $(q, \sigma, q') \in \delta$ and $(q, \epsilon, q'') \in \delta$, then we again are faced with the problem of non-determinism. Say that the finite state machine is in state q and is about to read symbol σ. There are two possibilities: the machine could read symbol σ and transition to q' *or*, since ϵ is the empty string and can be *inserted anywhere*, it could take an ϵ-transition to state q''.

In the face of non-determinism it seems all our notions of acceptance break down. However, we say that a word w is accepted by a non-deterministic finite state machine if there is *any* sequence of transitions that, when starting at the initial state, take the finite state machine to a final state while reading w. With this in mind, let G be a finite state machine. By $\mathcal{L}_{\mathcal{M}}(G)$ we mean the language accepted by G; i.e., the set of words w such that there exists a sequence of transitions that leads G to an accepting state when reading w.

There is another language associated with a finite state machine M, $\mathcal{L}(M)$. This is the set of words w such that there exists a sequence of transitions that leads M to *some* (not necessarily final) state in Q when reading w. When there is a labeled path (defined in the relation δ) leading from the starting state to every state a path from every state to *at least* one final state, then the finite state machine is called *trim*. Formally, a finite state machine $G = \langle Q, \Sigma, \delta, q_0, Q_f \rangle$ is trim if and only if the following holds:

1. For all q there exists $w = \sigma_1 \cdots \sigma_n$ such that

$$(q_0, \sigma_1, q_1) \in \delta \wedge \cdots \wedge (q_{n-1}, \sigma_n, q) \in \delta$$

for states $q_1, \ldots, q_{n-1} \in Q$ and n some finite non-negative integer, and

2. For all q there exists $w = \sigma_1 \cdots \sigma_m$ and $q_f \in Q$ such that

$$(q, \sigma_1, q_1) \in \delta \wedge \cdots \wedge (q_{m-1}, \sigma_m, q_f) \in \delta$$

for states $q_1, \ldots, q_{m-1} \in Q$ and m some finite non-negative integer.

In this case, $\overline{\mathcal{L}_{\mathcal{M}}(G)} = \mathcal{L}(G)$. That is, the prefix closure of $\mathcal{L}_{\mathcal{M}}(G)$ is $\mathcal{L}(G)$. We will call $\mathcal{L}_{\mathcal{M}}(G)$ the language accepted by final state and $\mathcal{L}(G)$ the language accepted by G or also the language *generated* by G. This verbiage will become clear when we discuss discrete event control.

A formal definition is now required to help us tell the difference between deterministic and non-deterministic finite state machines. A finite state machine is *deterministic* if

1. For all $q \in Q$ and for all $\sigma \in \Sigma$ there is *exactly one* q' in Q such that $(q, \sigma, q') \in \delta$.

2. For all $q \in Q$, $(q, \epsilon, q) \in \delta$ and there is no other q' such that $(q, \epsilon, q') \in \delta$.

Any finite state machine that is not deterministic is non-deterministic. Furthermore, when a finite state machine is deterministic, the transition relation is in fact equivalent to a partial function [Rog87] $\delta : Q \times \Sigma \to Q$. We shall treat it as such when appropriate.

As it turns out, the concept of non-determinism is not very important for finite state machines since the following theorem holds (Theorem 2.1 of [HU79]):

Theorem 2.2.2. *If G is a non-deterministic finite state machine, then there is a deterministic finite state machine G' such that $\mathcal{L}_{\mathcal{M}}(G) = \mathcal{L}_{\mathcal{M}}(G')$.*

As a result of this theorem, we may define the following class of languages: a language $L \subseteq \Sigma^*$ is called *regular* if there is a deterministic finite state machine $M = \langle Q, \Sigma, \delta, q_0, Q_f \rangle$ that accepts L. When we fix an alphabet Σ, the class of regular languages in Σ^* will be denoted by $\mathbf{REG}(\Sigma)$ or just $\mathbf{REG}$ when the alphabet Σ is known and fixed.

Throughout the rest of this thesis, we shall assume that all finite state machines are deterministic with transition *functions* δ. It is reasonable to ask, 'Why have we spent so much time on this distinction?' As it turns out, determinism plays an important role in more complex language classes; in fact it plays the most important role in the results we will display in Chapter 3. For this reason, a good grounding in determinism was required.

2.3 Pushdown Machines

Thus far we have examined the set of regular languages, those accepted by finite state machines. It is our goal now to enhance the power of the finite state machine by adding to it a pushdown stack–a device on which we may store data for later access. The resulting structure is called a pushdown machine (or pushdown automaton). Formally, a *pushdown automaton* (PDA) is a tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, Q_f \rangle$, where Q is a finite set of states as before, Σ is an alphabet, Γ is a *stack alphabet*, $q_0 \in Q$ is a starting state, $Z_0 \in \Gamma$ is the *starting stack symbol*, $Q_f \subseteq Q$ are the final (or marked) states and $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \Gamma \times Q \times \Gamma^*$ is a transition relation. It is sometimes convenient to think of δ as a mapping from $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$. In this case we will write $\delta(q, a, Z)$ to denote the finite subset of $Q \times \Gamma^*$ that results. Both notations will be used throughout this thesis and we will choose which notation to use based on convenience and clarity of expression.

As before, we may envision the operation of a PDA on an input tape T; the machine consists of a *finite state system*, and a *stack*. As symbols are read in from the tape, a (*possibly non-deterministic*[1]) transition will be made based on the top stack symbol, the input symbol and the current state of the machine; the transition changes the state of the finite state system, and replaces the top stack symbol with a (*possibly empty*) string of stack symbols. This operation is shown in Figure 2.4.

Let M be a PDA. By $\delta(q, a, Z) = \{(p_1, \gamma_1), \ldots, (p_n, \gamma_n)\}$ we mean that given an input symbol a when M is in state q with top stack symbol Z, then M may transition to a new state p_i and replace $Z \in \Gamma$ with the string $\gamma_i \in \Gamma^*$ for $1 \leq i \leq n$.

[1] Even though we will *not* consider non-deterministic finite state machines, we will be interested in non-deterministic pushdown machines.

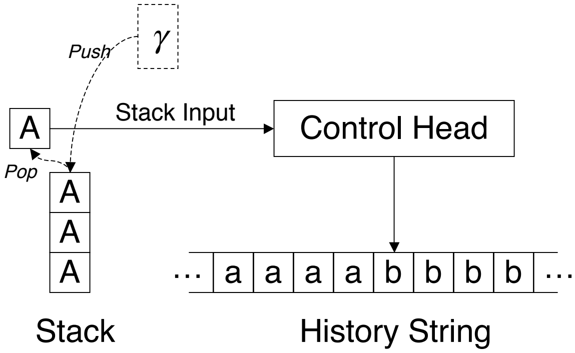


Figure 2.4. The action of a PDA as it reads in a string of symbols.

Under some conditions, the PDA can edit its top stack symbol independent of input events. This occurs when an ϵ-transition is fired. In this case, both the state and top stack symbol of the PDA may be altered, but no new symbols from the input string are accepted. By $\delta(q, \epsilon, Z) = \{(p_1, \gamma_1), \ldots, (p_n, \gamma_n)\}$ we mean that when M is in state q with top stack symbol Z, there may be an ϵ-transition which, independent of any input symbol, causes M to transition to a new state p_i and replace Z with γ_i for some $1 \leq i \leq n$.

Unlike finite state automata, the state of a PDA is captured by both the state of the machine and the symbols stored in the stack. To formally describe the state of a PDA, we must specify these values. Traditionally the remaining input string to be read is also included in this description [HU79].

An *instantaneous description* (ID) for a PDA M is a triple (q, w, γ), where $q \in Q$ is the state of M, $w \in \Sigma^*$ is the remainder of the string to be read and $\gamma \in \Gamma^*$ is the state of the stack.

Let M be a PDA. By $(q, aw, Z\gamma) \vdash (p, w, \beta\gamma)$, we mean that $\delta(q, a, Z)$ contains (p, β). Let $\vdash^*$ be the *transitive closure* of $\vdash$. Then by $(q, wv, \gamma) \vdash^* (p, v, \beta)$, we mean that there is a series of transitions in δ that causes M to read the prefix w and transition from state q to state p and transform the stack string from γ to β. The remaining string v is still available for reading.

A PDA may accept words in two different ways. We say that a PDA accepts a word $w \in \Sigma^*$ by final state, if reading w to completion causes a series of transitions leading from the starting state q_0 with Z_0 on the stack to some final state q. In contrast, we say that a PDA accepts a word w by empty stack if reading w to completion causes a series of transitions leading from the starting state q_0 with Z_0

on the stack to any state $q \in Q$ with an empty stack. Formally, the *final state language* and the *empty stack language* of M are defined as:

$$\begin{aligned} \mathcal{L}_{\mathcal{M}}(M) &= \{w \in \Sigma^* \mid \exists q \in Q_f \exists \gamma \in \Gamma^* \left((q_0, w, Z_0) \vdash^* (q, \epsilon, \gamma)\right)\} \\ \mathcal{L}_{\emptyset}(M) &= \{w \in \Sigma^* \mid \exists q \in Q \left((q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)\right)\}. \end{aligned} \tag{2.3.1}$$

We will make very little use of the empty stack acceptance criterion in this thesis. We have included it only for completeness. In fact, the following theorem, proved in [HU79], shows that the distinction between acceptance by final state and by empty stack is just a formality. (See Theorems 5.1 and 5.2 of [HU79].)

Theorem 2.3.1. *For any PDA M, there exists a second PDA M' such that $\mathcal{L}_{\mathcal{M}}(M) = \mathcal{L}_{\emptyset}(M')$. Likewise there exists a third PDA M'' such that $\mathcal{L}_{\emptyset}(M) = \mathcal{L}_{\mathcal{M}}(M'')$.*

Hence, any language L that is accepted by a PDA M by empty stack, is accepted by another PDA M' by final state and vice versa. A language accepted by a PDA (in any sense) is called *context free*. When we fix a Σ, the class of context free languages in Σ^* will be referred to as **CFL**(Σ) or just **CFL** if Σ is known and fixed.

For fixed Σ, it is immediately obvious that the regular languages are a proper subclass of the context free languages. (To see this, just simulate the moves of any finite state machine and continuously pop and push the symbol Z_0.) Put more formally, we have **REG** $\subset$ **CFL**.

Example 2.3.2. Consider a PDA $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, Q_f \rangle$ with the following properties: $Q = q_0, q_1, q_2$, $Q_f = \emptyset$, $\Sigma = \{0, 1\}$, $\Gamma = \{Z_0, A\}$, and

$$\begin{aligned} \delta = \{&(q_0, 0, Z_0, q_0, Z_0 A), (q_0, 0, A, q_0, AA), \\ &\qquad (q_0, 1, A, q_1, \epsilon), (q_1, 1, A, q_1, \epsilon), (q_1, \epsilon, Z_0, q_2, \epsilon)\} \end{aligned}$$

This PDA is shown in Figure 2.5.

The PDA we have defined accepts the language $L = \{0^n 1^n | n \geq 1\}$ by *empty stack*. To see this, consider what happens when the string 0011 is read in. During the first read: $(q_0, 0011, Z_0) \vdash (q_0, 011, AZ_0)$ because 0 is read, Z_0 is the top stack

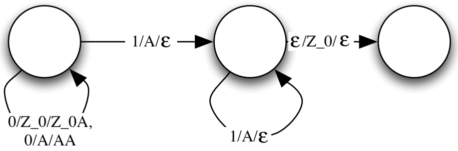


Figure 2.5. An example PDA that accepts $\{0^n 1^n | n \geq 1\}$.

symbol and a loop transition is made to q_0 and $Z_0 A$ is pushed onto the stack–this makes the top stack symbol now A because the symbols are pushed on in order of Z_0 and then A. During the next step, $(q_0, 011, AZ_0) \vdash (q_0, 11, AAZ_0)$. Then $(q_0, 11, AAZ_0) \vdash (q_1, 1, AZ_0)$ because the symbol A is read off the stack, but ϵ (nothing) is pushed on. Next $(q_1, 1, AZ_0) \vdash (q_1, \epsilon, Z_0)$ and finally, $(q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, \epsilon)$ by the ϵ-transition from state 1 to state 2. Note, the fact that $Q_f = \emptyset$ makes no difference in this case since we are allowing this PDA to *accept by empty stack*.

The stack keeps track of the number of 0's read in by pushing a symbol A onto the stack each time a zero is read. As 1's are read in, the stack unwinds the A's until only the Z_0 symbol remains. A non-deterministic transition empties the stack with no need to read additional symbols. Hence, the empty stack language of this machine is $\{0^n 1^n | n \geq 1\}$. We see it is $n \geq 1$ and not $n \geq 0$ since we start with a non-empty stack and we must read at least one 0 and one 1 to reach a state where we empty the stack of Z_0. It is fairly easy to see at this point, that this machine will accept no other strings other than the ones we have discussed.

The following "Pumping Lemma" will help us show that the language in the previous example is not regular.

Lemma 2.3.3 ("The Pumping Lemma"–Lemma 3.1 of [HU79])**.** *Let L be a regular language. Then there exists a constant n such that if z is any word in L and $|z| \geq n$ then we may write $z = uvw$ in such a way that $|uv| \leq n$, $|v| \geq 1$ and for all $k \geq 0$, $uv^k w$ is a word in L. Further, n is no greater than the number of states of the FSM with the fewest number of states that accepts L.*

Example 2.3.4. We can now show that $L = \{0^n 1^n | n \geq 1\}$ is not a regular language. For suppose that it is, then the Pumping Lemma holds [HU79]. But let n be arbitrarily large and suppose that we choose $s \geq n/2$, $s \in \mathbb{Z}$ then $z = 0^s 1^s$ has length at least n. It is clear for any decomposition of z into uvw we will not have $uv^kw \in L$ for $k > 0$. Hence, L cannot be regular.

A PDA M is *deterministic* (DPDA) if the following hold:

1. If $(q_1, a, Z, q_2, \gamma) \in \delta$ and $(q_1, a, Z, q_2', \gamma') \in \delta$, then $q_2 = q_2'$ and $\gamma = \gamma'$.

2. If $(q_1, \epsilon, Z, q_2, \gamma) \in \delta$, then for all $a \in \Sigma$, $\delta(q_1, a, Z) = \emptyset$.

Condition 1 of the definition ensures that for all $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$, $|\delta(q, a, Z)| = 1$. Thus there is no non-deterministic choice possible when making transitions. Condition 2 of the definition ensures that there is never a choice between accepting an input symbol or making an ϵ-transition without accepting an input symbol. We saw in Theorem 2.2.2 that non-determinism doesn't make a difference in the regular languages. In the context free languages, there is a difference. It is shown in Chapter 5.2 of [HU79] (Page 113) that there is a language accepted by a PDA that is not accepted by a DPDA. A language accepted by a DPDA (in any sense) is called *deterministic context free*. For fixed Σ, the class of deterministic context free languages in Σ^* will be denoted **DCFL**(Σ) or just **DCFL** when Σ is understood from context. It can now be seen that for fixed Σ: $\mathbf{REG} \subset \mathbf{DCFL} \subset \mathbf{CFL}$. Further, since the PDA given in Example 2.3.2 is deterministic, we can see that the language $\{0^n1^n : n > 0\}$ is in **DCFL**. In Example 2.3.4 we showed this language was not regular.

Remark 2.3.5. It will be important for us to talk about subautomata (or submachines). Let M be a finite state machine or DPDA. Then a submachine (or subautomaton) of M is a second finite state machine or DPDA M' such that:

1. The state set of M' is a subset of the state set of M.

2. The alphabet of M' is equal to the alphabet of M.

3. If appropriate, the stack alphabet of M' is equal to the stack alphabet of M.

4. The final states of M' are a subset of the final states of M.

5. If appropriate, the start stack symbol of M' is the same as the start stack symbol of M.

6. Finally, the transition relation of M' is appropriately defined on its state set and it is a subrelation of the transition relation of M.

2.4 Properties of REG, DCFL and CFL

The purpose of this section is to present several important closure properties of the regular languages, the deterministic context free languages and the context free languages that will be used later in this thesis. The proofs of all of these claims can be found in [HU79]. We will provide more details as we state individual results.

Lemma 2.4.1. *Fix an alphabet* Σ^**. The class* **REG** *is closed under (i) union, (ii) intersection, (iii) left and right quotient, (iv) concatenation, (v) Kleene star and (vi) complement in* Σ^**.*

Proof. See Theorems 3.1, 3.2, 3.3 and 3.6 of [HU79]. □

In particular, since $\mathbf{REG}(\Sigma)$ is closed under left and right quotient, it follows immediately that $\mathbf{REG}(\Sigma)$ is closed under prefix closure since for any $L \subseteq \Sigma^*$, $\overline{L} = L/\Sigma^*$; that is, if $L \in \mathbf{REG}$, then $\overline{L} \in \mathbf{REG}$.

Lemma 2.4.2. *Fix an alphabet* Σ^**. The class* **CFL** *is closed under (i) union, (ii) concatenation, (iii) intersection with regular language (iv) quotients with regular language, (v) Kleene star. The class* **CFL** *is not closed under (i) intersection, (ii) complement in* Σ^**.*

Proof. See Theorems 6.1, 6.4 and 6.5 and 11.3 (as it applies to **CFL**) of [HU79]. □

Again, note that if $L \in \mathbf{CFL}$, then $\overline{L} \in \mathbf{CFL}$ since $\Sigma^* \in \mathbf{REG}$ for any Σ. Contrast Lemma 2.4.2 with the properties of the class **DCFL**.

Lemma 2.4.3. *Fix an alphabet* Σ^**. The class* **DCFL** *is closed under (i) intersection with regular language, (ii) quotient with regular language and (iii) complement in* Σ^**. The class* **DCFL** *is not closed under (i) union, (ii) concatenation, (iii) intersection, or (iv) Kleene star.*

Proof. See Theorems 10.1, 10.2, 10.4, and 10.5 of [HU79]. Lemma 8.8 of [HU79] shows in great detail that **DCFL** is not closed under intersection. □

The properties we will use most are the closure of **DCFL** under complements and the closure of both **CFL** and **DCFL** under quotient with regular languages.

2.5 Decidability and Turing Machines

The definitions we have given for **REG**, **DCFL** and **CFL** are *machine based.* There are alternate *grammar* or *rule* based definitions that are outside the scope of this thesis. The interested reader is directed to [HU79]. The primary question early research in language theory attempted to address was the inclusion question: *Is there an algorithm to determine whether a word w is an element of language L?* It should be clear to the reader that for the classes of languages we have discussed, there is an algorithm: namely the word is fed into an appropriate machine, which either accepts or does not accept according to its acceptance criterion.

There are instances where the question of word inclusion becomes more subtle. In these cases, we must define what we mean by "algorithm." A *Turing Machine* (TM) is a tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Q_f \rangle$, where Q is a finite set of states, Σ is a finite input alphabet, $\Gamma \subseteq \Sigma$ is a finite tape alphabet, q_0 is a starting state, $Q_f \subseteq Q$ is a set of final states and $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q \times \Gamma \times \{L, R\}$ as before, we may also view δ as function from $Q \times \Sigma \cup \{\epsilon\}$ into a set of tuples in the set $Q \times \Gamma \times \{L, R\}$. There is a special symbol B in Γ called the *blank* symbol. Strictly speaking, this symbol is different from ϵ; to see why, if $a, b \in \Sigma$, then $a\epsilon b = ab$, whereas aBb literally represents "a b".

Again, let T be a ticker tape that extends infinitely far in both directions with a word $w \in \Sigma^*$ written upon it. A TM operates by reading in the ticker tape T starting at the left most symbol of w. Let $a \in \Sigma$ and suppose that a TM in state q has just read in a. Then the TM makes a state transition to some new state q', writes a symbol from Γ *over top of* a and either moves left (L) or right (R). If the TM reads off the right end of the word and no further move can be made and the state q' is in Q_f, then the TM is said to accept w. Since a TM can re-read the same position in w many times (by moving back and forth over it), it is very easy to see that it may take a long time to know whether a word w is accepted by an arbitrarily chosen TM. In fact, the TM may never stop moving once it starts reading w. When a TM eventually either accepts or rejects a word w, we say that the machine *halts* on w. A TM that *halts on all inputs* will eventually reach a decision about every word $w \in \Sigma^*$. Any language that is accepted by a TM that halts on all inputs is called *recursive.* Any language that is accepted by a TM (that may or may not halt on all inputs) is called a *recursively enumerable* language.

Without loss of generality, we may restrict out attention to deterministic TM's, since like finite state machines, the languages accepted by deterministic TM's are equivalent to the languages accepted by non-deterministic TM's. (See Theorem 7.3 of [HU79].) In a deterministic TM, the relation δ can be transformed immediately to a partial function from $Q \times \Sigma$ to $Q \times \Gamma \times \{L, R\}$.

As it turns out, every TM with $\Sigma = \{0, 1\}$ may itself be represented by a coded string over alphabet $\Sigma = \{0, 1\}$. That is to say any TM M may be assigned a unique value $w_M = \#(M)$ with $w_M \in \{0, 1\}^*$. This is called a code of the TM M and the function # is called a coding [Rog87,HU79]. A detailed description of this fact is given in Chapter 8 of [HU79]. This gives rise to a few interesting problems. For suppose that $H = \{\text{Codes for TM that halt on all inputs.}\}$. It would be nice to know whether $w \in H$, for then we would know that the TM coded by w could safely be used on any word without worrying about an infinite computation. Unfortunately, this is not possible. The following famous theorem was first proved by Alan Turing in 1936 [Tur36]. A proof is given in Chapter 8 of [HU79] (See Theorems 8.4 and 8.5).

Theorem 2.5.1. *There is no Turing Machine that halts on all inputs and decides whether $w \in H$ for $w \in \{0, 1\}^*$.*

This is called the *Halting Problem* and is the key result needed in decidability theory. In decidability theory, a *problem* is any set of strings $L \subseteq \Sigma^*$ for some Σ; the problem is tacitly understood to be the string inclusion problem: is there an algorithm–by which we mean a TM–that can decide whether or not a string $w \in L$. For any other mathematical problem that can be transformed (or *coded*) into an inclusion problem we may ask 'is this problem decidable?'

2.6 Decidable Properties of the Languages REG, DCFL and CFL

We already know a number of decidable properties of the languages we have discussed. In this section, we enumerate a number of decidable and undecidable properties of these languages. Unless otherwise stated, all of these properties are proved in [HU79].

Lemma 2.6.1. *Fix an alphabet Σ. Let $L \in \mathbf{REG}$. The following questions are decidable: (i) For any $w \in \Sigma^*$, $w \in L$?; (ii) If $H \in \mathbf{REG}$, $L = H$?; (iii) If $H \in \mathbf{REG}$, $H \subseteq L$?.*

Proof. (i) is obvious: input the word w into a finite state machine accepting L. (ii) and (iii) are proved in Theorem 3.7 and 3.8 of [HU79]. □

Lemma 2.6.2. *Fix an alphabet Σ. Let $L \in \mathbf{DCFL}$. The following questions are decidable: (i) For any $w \in \Sigma^*$, $w \in L$?; (ii) If $H \in \mathbf{DCFL}$, $L = H$?; (iii) If $R \in \mathbf{REG}$, $R \subseteq L$.*

Proof. (i) is obvious: input the word w into a DPDA accepting L. (ii) was just recently proved in 1997 in [Ś97]. This question had been open since formal language theory began. (iii) is discussed in Theorem 10.6 of [HU79]. □

What [Ś97] actually proves is that if M_1 and M_2 are two DPDA, then it is decidable whether $\mathcal{L}_{\mathcal{M}}(M_1) = \mathcal{L}_{\mathcal{M}}(M_2)$. Likewise, for all of these statements, we assume that we are given some accepting machine (FSM, DPDA, PDA, TM) for the specified languages.

Lemma 2.6.3. *Fix an alphabet Σ. Let $L \in \mathbf{DCFL}$. The following questions are undecidable: (i) If $H \in \mathbf{DCFL}$, $H \subseteq L$?; (ii) If $H \in \mathbf{DCFL}$, $H \cap L \in \mathbf{DCFL}$?; (iii) If $H \in \mathbf{DCFL}$, $H \cap L \in \mathbf{CFL}$?*

Proof. See Theorem 10.7 of [HU79]. □

Lemma 2.6.4. *Fix an alphabet Σ. Let $L \in \mathbf{CFL}$. The following questions are decidable: (i) For any $w \in \Sigma^*$, $w \in L$?; (ii) $L = \emptyset$?.*

Proof. (i) is obvious: Input the word w into a PDA accepting L. (ii) is proved in Theorem 6.6 of [HU79]. □

Lemma 2.6.5. *Fix an alphabet Σ. Let $L \in \mathbf{CFL}$. The following questions are undecidable: (i)$L = \Sigma^*$?; (ii) If $H \in \mathbf{CFL}$, $L = H$?; (iii) If $H \in \mathbf{CFL}$, $L \cap H \in \mathbf{CFL}$?*

Proof. See Theorems 8.11, 8.12 and 8.13 of [HU79]. □

There are many other decidable and undecidable properties of **CFL** and **DCFL**. In general for **REG** *almost* every property one can think of is decidable. The following useful lemma is proved in [IR74]. We will use it several times throughout the remainder of this thesis.

Lemma 2.6.6 (Theorem 2.3 of [IR74])**.** *Let P be any predicate*[2] *on the languages accepted by PDA over the alphabet $\Sigma = \{0, 1\}$, such that:*

1. *$P(\Sigma^*)$ holds and*
2. *Either $P_{\text{Left}} = \{L' \mid L' = \{x\} \backslash L,\ x \in \Sigma^*,\ P(L) \textit{ holds}\}$ or $P_{\text{Right}} = \{L' \mid L' = L/\{x\},\ x \in \Sigma^*,\ P(L) \textit{ holds}\}$ is a proper subset of the languages accepted by PDA over Σ, when L is accepted by a PDA over the alphabet Σ.*

Then for arbitrary PDA G, the predicate $P(\mathcal{L}_{\mathcal{M}}(G))$ is undecidable.

2.7 Algorithms on FSM and DPDA

In this section, we review several algorithms that will be used throughout the remainder of this thesis.

2.7.1 Prefix Closure of Regular and Deterministic Context Free Languages

Let $M = \langle Q, \Sigma, \delta, q_0, Q_f \rangle$ be a finite state machine. If M is trim, then the machine $\overline{M} = \langle Q, \Sigma, \delta, q_0, Q \rangle$ will accept the prefix closure of $\mathcal{L}_{\mathcal{M}}(M)$.

Similarly, if $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, Q \rangle$ is a PDA, then $\mathcal{L}_{\mathcal{M}}(M)$ is prefix closed. The generic algorithm for computing the prefix closure of M is discussed in detail in [Har80], when it is demonstrated that the family **DCFL** is closed under quotient with **REG**. Throughout this thesis, we will never need the more general algorithm and we will construct our DPDA so that all states are final. Hence, the languages they accept are prefix closed *a priori*. We will denote this operation as $\overline{M}$. So that $\mathcal{L}_{\mathcal{M}}(\overline{M}) = \overline{\mathcal{L}_{\mathcal{M}}(M)}$.

[2] In this case, a predicate is a statement about one or more inputs using an appropriately defined logic.

2.7.2 Complementation of a language in REG

Let $M = \langle Q, \Sigma, \delta, q_0, Q_f \rangle$ the following algorithm will construct a machine M' accepting the complement of $\mathcal{L}_{\mathcal{M}}(M)$.

Let $M' = \langle Q \cup \{d\}, \Sigma, \delta', q_0, (Q - Q_f) \cup \{d\} \rangle$

1. Define a state d not currently in Q.
2. For all $a \in \Sigma$, if there is no $p \in Q$ such that $(q, a, p) \in \delta$, then let $(q, a, d) \in \delta'$. Otherwise, if there is such a p, then let $(q, a, p) \in \delta'$.
3. For all $a \in \Sigma$, let $(d, a, d) \in \delta'$.
4. Finally, set the final states of M' to $(Q - Q_f) \cup \{d\}$.

Theorem 3.2 of [HU79] shows that this machine accepts $\mathcal{L}_{\mathcal{M}}(M)^c$. We will denote this operation on M by M^c.

2.7.3 Complementation of a language in DCFL

Let M be a DPDA and let $L = \mathcal{L}_{\mathcal{M}}(M)$. We describe an algorithm for computing the complement of L provided in Chapter 10 (pp. 235-239) of [HU79]. The first step is to transform M so that regardless of what word w is placed on its tape, it will always scan the entire word. Let $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ define $M' = \langle Q \cup \{q_0', d, f\}, \Sigma, \Gamma \cup \{X_0\}, \delta', q_0', X_0, F \cup \{f\} \rangle$ where:

1. $\delta(q_0', \epsilon, X_0) = (q_0, Z_0 X_0)$; i.e., we replace the original start state and stack symbol with new ones.
2. If for some q in Q, a in Σ and Z in Γ, $\delta(q, a, Z)$ and $\delta(q, \epsilon, Z)$ are both empty, then define $\delta'(q, a, Z) = (d, Z)$; and for all a in Σ and q in Q, define $\delta'(q, a, X_0) = (d, X_0)$; i.e., make state d act as it does when we find the complement of a regular language.
3. $\delta'(d, a, Z) = (d, Z)$ for all a in Σ and Z in $\Gamma \cup \{X_0\}$
4. If for q and Z and for all i there exists q_i and γ_i such that $(q, \epsilon, Z) \vdash_i (q_i, \epsilon, \gamma_i)$, then $\delta'(q, \epsilon, Z) = (d, Z)$; this detects conditions on which the DPDA may make an infinite number of ϵ-transitions without accepting.

5. $\delta'(f, \epsilon, Z) = (d, Z)$ for all $Z \in \Gamma \cup \{X_0\}$

6. For any q in Q, Z in Γ and a in $\Sigma \cup \{\epsilon\}$, if $\delta'(q, a, Z)$ has not been defined in Rules 1-5, then define $\delta'(q, a, Z) = \delta(q, a, Z)$.

Lemma 10.3 of [HU79] proves that this machine accepts the same language as $\mathcal{L}_{\mathcal{M}}(M)$, but that it will scan all inputs.

Now, assume that $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$ is a DPDA that has been transformed so that is will scan all inputs. Then the following algorithm will construct a DPDA M' that accepts $\mathcal{L}_{\mathcal{M}}(M)^c$.

Let $M' = \langle Q', \Sigma, \Gamma, \delta', q_0', Z_0', F'\rangle$ where

$$Q' = \{(q, k) | q \in Q, k = 1, 2, 3\}$$

Let $F' = \{(q, 3) | q \in Q\}$ and define:

$$q_0' = \begin{cases} (q_0, 1) & \text{if } q_0 \in F \\ (q_0, 2) & \text{if } q_0 \notin F \end{cases}$$

Now define δ' as follows:

1. If $\delta(q, \epsilon, Z) = (p, \gamma)$, then for $k = 1$ or $k = 2$ define $\delta'((q, k), \epsilon, Z) = ((p, k'), \gamma)$ where $k' = 1$ if $k = 1$ or $p \in F$; otherwise, $k = 2$.

2. If $\delta(q, a, Z) = (p, \gamma)$ for some $a \in \Sigma$, then $\delta'((q, 2), \epsilon, Z) = ((q, 3), Z)$ and $\delta'((q, 1), a, Z) = \delta'((q, 3), a, Z) = ((p, k), \gamma)$ where $k = 1$ for $p \in F$ and $k = 2$ for $p \notin F$.

Theorem 10.1 of [HU79] proves that the language $\mathcal{L}_{\mathcal{M}}(M') = \mathcal{L}_{\mathcal{M}}(M)^c$. In Chapter 5 we will discuss these two algorithms in detail. We will denote this operation on M by M^c.

2.7.4 Normal Form for PDA

A DPDA is said to be in *normal form* if the following is true: If $\delta(q, a, X) = (p, \gamma)$, then either:

1. $\gamma = \epsilon$ (pop),
2. $\gamma = X$ (no change), or
3. $\gamma = YX$ (push)

It is easy to see that any long stack push can be transformed into a series of pushes, where each stack symbol is added one at a time. Hence, we can transform any DPDA M into a new DPDA M in normal form. Lemmas 10.1 and 10.2 of [HU79] have a detailed proof of this fact; though even Hopcroft and Ullman [HU79] note that this is a fairly clear fact.

2.7.5 Predicting Machines

Let $M = (Q_M, \Sigma, \Gamma, \delta_M, q_0, Z_0, F_M)$ be a DPDA in normal form (see [HU79], Lemmas 10.1-10.2) and let $A = (Q_A, \Sigma, \delta_A, p_0, F_A)$ be a FSM. A *predicting machine* $\pi(M, A)$ is a DPDA with form

$$(Q_M, \Sigma, \Gamma \times \Delta, \delta, q_0, X_0, F_M)$$

where $\Delta = 2^{Q_M \times Q_A}$ (The power set of $Q_M \times Q_A$). If $(r, x, [Z, \mu]\gamma)$ is an ID of $\pi(M, A)$ (x a string), then μ consists of the pairs $(q, p) \in Q_M \times Q_A$ such that there is a $w \in \Sigma^*$ for which $\delta_A(p, w) \in F_A$ and $(q, w, Z\beta) \vdash^* (s, \epsilon, \alpha)$ for some $s \in F_M$, where β is the string of first components (the projection) of γ. The information associated with this top stack symbol $[Z, \mu]$ tells for states q and p whether there is some input string that causes both M and A to accept when started from their current states and run with that input. We will use Predicting Machines in Chapter 5, but only a passing familiarity with their technical details will be required and all other pertinent information is introduced in this chapter. Extensive details can be found in Chapter 10.3 of [HU79].

2.7.6 Intersection of Languages

Let $G_i = \langle Q^i, \Sigma, \delta^i, q_0^i, Q_f^i \rangle$ $(i = 1, 2)$ be two finite state machines. Define $M = \langle Q, \Sigma, \delta, q_0, Q_f \rangle$ where:

1. $Q = Q^1 \times Q^2$

2. $Q_f = Q_f^1 \times Q_f^2$

3. $q_0 = (q_0^1, q_0^2)$

4. $\delta((q^1, q^2), a) = (p^1, p^2)$ if and only if $\delta^1(q^1, a) = p^1$ and $\delta(q^2, a) = p^2$.

Then $\mathcal{L}_{\mathcal{M}}(M) = \mathcal{L}_{\mathcal{M}}(G^1) \cap \mathcal{L}_{\mathcal{M}}(G^2)$. It is clear that $\mathcal{L}_{\mathcal{M}}(M)$ is a regular language.

Now, suppose that $G^1 = \langle Q^1, \Sigma, \Gamma, \delta^1, q_0^i, Z_0^1, Q_f^i \rangle$ is a DPDA. Define $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, Q_f \rangle$ where:

1. $Q = Q^1 \times Q^2$

2. $Q_f = Q_f^1 \times Q_f^2$

3. $q_0 = (q_0^1, q_0^2)$

4. $\delta((q^1, q^2), a, Z) = ((p^1, p^2), \gamma)$ if and only if $\delta^1(q^1, a, Z) = (p^1, \gamma)$ and $\delta(q^2, a) = p^2$.

Then $\mathcal{L}_{\mathcal{M}}(M) = \mathcal{L}_{\mathcal{M}}(G^1) \cap \mathcal{L}_{\mathcal{M}}(G^2)$. This is proved in Theorem 6.5 of [HU79] It is clear that $\mathcal{L}_{\mathcal{M}}(M)$ is a deterministic context free language. We will denote these operations on G^1 and G^2 by $G^1 \cap G^2$.

2.8 Introduction to Discrete Event Control

In this section we finally begin our discussion of discrete event control. Discrete Event Control (or the Supervisory Control Theory (SCT) of Discrete Event Dynamic Systems (DEDS)) was founded by Ramadge and Wonham in 1987 [RW87c, RW87b] in their two seminal papers. Let P be a system that is to be controlled. This is called the *plant model* (or just the plant). We suppose that the dynamics of the plant may be modeled by a state machine $G = \langle Q, \Sigma, \delta, q_0, Q_f \rangle$. When Q is finite, this is just a finite state machine as we have defined it in the previous sections. As time moves forward, the plant *generates* symbols in Σ and changes state according to the rules set forth in δ. In discrete event control, we call the symbols *events*. We suppose at time 0, the plant is in state q_0. The set of states Q_f are states in which the plant may stop functioning "legally." By convention, we assume that the plant models are *trim* (See Section 2.2).

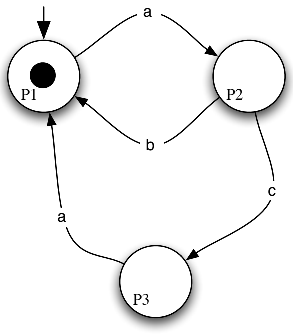


Figure 2.6. A simple three state plant model with a single starting state and a single final state. The starting state is illustrated by an arrow that connects to no state.

Example 2.8.1. Consider a simple plant model shown in Figure 2.6.

As time moves forward, the symbols produced by the plant form a *history* $w(t) \in \Sigma^*$. Here, $w(t)$ is a word for each time step t and t may be either used to count the steps executed by the plant or may be an actual time counter in $\mathbb{R}_+$. By $\mathcal{L}(G)$ we mean the set of possible history strings $w(t)$ we may see *at any time*. It is now easy to see that $w(t) \in \mathcal{L}(G)$ for all t. This is why we referred to $\mathcal{L}(G)$ as the language *generated* by a finite state machine G. To reduce the complexity of our mathematical expressions, we will often drop the time component of $w(t)$ and just refer to histories of G as $w \in \mathcal{L}(G)$.

When modeling plants, we partition Σ into Σ_c and Σ_u. The events in Σ_c are *controllable*. An event is controllable *if it can be prevented from occurring by choice.* The events in Σ_u are uncontrollable. These events *cannot be prevented by choice.*

Example 2.8.2. Consider the plant model shown in Figure 2.6. Suppose that $\Sigma_c = \{a, c\}$ and $\Sigma_u = \{b\}$. Then if we desired, we could permanently turn off event c but b may occur no mater what we choose.

A supervisor for G is a mapping $\varphi : \mathcal{L}(G) \to \mathcal{P}(\Sigma)$, where $\mathcal{P}(\Sigma)$ is the power set of Σ. If $\varphi(w) = E \subseteq \Sigma$, then whenever the supervisor observes sequence w being generated by the plant model G, it will *enable* events in E and *disable* events in $\Sigma \setminus E$. In this way, the supervisor may exert some control over the plant model G. It is easy to see that by disabling certain events in Σ_c, the language produced by the plant must be contained in $\mathcal{L}(G)$. We refer to the language generated by G when under the control of φ as $\mathcal{L}(\varphi/G)$. This is the set of words w such that there is a sequence of transitions in G that lead from the initial state to some state *and* so that at each step, the supervisor φ enables the transition used. The definition of $\mathcal{L}_{\mathcal{M}}(\varphi/G)$ is completely analogous. Note, this notation is used originally in [RW87c] and is not intended to indicate a quotient of any kind. The notation φ/G is supposed to represent the supervisor φ acting in control *over* the plant G. Even the authors of [RW87c] noted that this could be confusing. We will always use the Greek letter φ to denote a supervisor, thus minimizing the confusion.

We may now state the fundamental question of classical discrete event control: Let $K \subseteq \mathcal{L}_{\mathcal{M}}(G)$. Does there exist a supervisor φ such that $\mathcal{L}_{\mathcal{M}}(\varphi/G) = K$? The fundamental theorem (Theorem 6.1) of [RW87c] answers this question:

Theorem 2.8.3. *Let $K \subseteq \mathcal{L}_{\mathcal{M}}(G)$. Then there exists a supervisor φ such that $\mathcal{L}_{\mathcal{M}}(\varphi/G) = K$ if and only if*

1. *K is controllable with respect to $\mathcal{L}_{\mathcal{M}}(G)$, that is: $\overline{K}\Sigma_u \cap \mathcal{L}(G) \subseteq \overline{K}$ and*

2. *K is $\mathcal{L}_{\mathcal{M}}$ closed that is: $\overline{K} \cap \mathcal{L}_{\mathcal{M}}(G) = K$.*

The key criterion is Condition 1, which is called the *controllability* criterion, or the definition of controllability for discrete event systems.

The language K is the *target* or *objective* language; it describes desirable patterns of execution for the plant. The goal of the supervisor is to force the plant to execute a desired pattern of behavior. To do this, the supervisor observes the output w at any time and formulates a disabling plan $\varphi(w)$. Ramadge and Wonhams's theorem states that a supervisor exists that can force G to behave in a desired way only if controllability holds.

Kumar et al. [KGM91] showed that controllability is equivalent to $\overline{K}\Sigma_u^* \cap \mathcal{L}(G) = \overline{K}$. Griffin [Gri06] noted that this expression is itself equivalent to the logical sentence:

$$\forall w \in \overline{K} \forall u \in \Sigma_u (wu \in \mathcal{L}(G) \implies wu \in \overline{K}). \tag{2.8.1}$$

We will use expressions like Expression 2.8.1 as the definition of controllability throughout this thesis. If $L \subseteq \Sigma^*$, we define $\mathcal{C}(L)$ to be the set of controllable sublanguages of L. Hence, if K is controllable with respect to L, then we can write $K \in \mathcal{C}(L)$.

The principle objective of discrete event control is to determine when, given a language K and a plant G, there is a supervisor (a strategy for disabling events) so that $\mathcal{L}_{\mathcal{M}}(\varphi/G) = K$. Hence, the objective of discrete event control is to understand controllability. Ramadge and Wonham showed that controllability is decidable for the case when K is an element of $\mathbf{REG}(\Sigma)$ (that is both K and $\mathcal{L}_{\mathcal{M}}(G)$ are regular languages). They gave an algorithm to check this property in [RW87c].

At first, controllability may seem like a mysterious concept. However, it is really a planning concept. Think of the controllability criterion as the following: There is a plan K for the behavior of a plant. You want to build a controller to make the plant G behave according to the specified plan. One can only do this if the plan K is well thought out enough so that whenever you have been executing the plan for any amount of time and something terrible occurs (an uncontrollable event), then you have already planned for that eventuality. Put simply, discrete event control is all about planned exception handling.

2.8.1 An Algorithm to Check Controllability

One way to understand the action of the supervisor φ is as a second state machine M that *runs synchronously* with G. Machine G may generate a symbol from Σ only if machine M may read it. The resulting language generated by the synchronously running machines is $\mathcal{L}(M) \cap \mathcal{L}(G)$.

Example 2.8.4. We will work with the plant model shown in Figure 2.6. Suppose that K is the language produced by the finite state machine M shown in Figure 2.7. It is easy to see that $\mathcal{L}_{\mathcal{M}}(M) \subseteq \mathcal{L}_{\mathcal{M}}(G)$. We can observe the effect of running M in parallel with G by observing the machine that accepts the intersection of the two languages: Figure 2.8 shows the state changes that occur when the two machines are run simultaneously. Starting in states $(P1, C1)$, G generates an a and M reads an a. The machines transition to states $(P2, C2)$. Now, since c is not present in state $C2$ in M, we see that c is disabled. Hence, G may generate a b and M will read a b and the machines will transition to states $(P1, C3)$. The plant G now generates an a again and the machines move to state $(P2, C4)$. Since

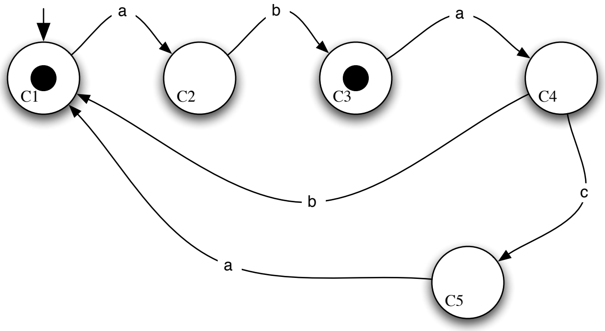


Figure 2.7. A simple control language that disables c whenever b is seen an even number of times.

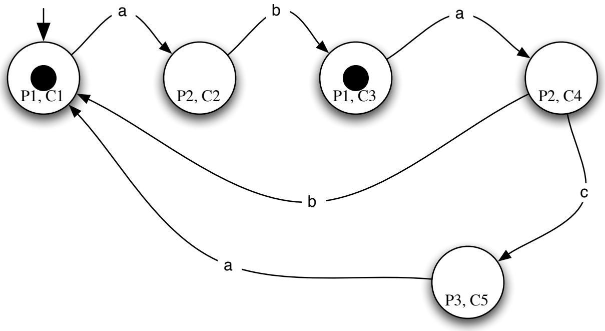


Figure 2.8. An intersection of M and G.

c is available now, we say that c is enabled and a transition to state $(P3, C5)$ may be taken.

We will illustrate an algorithm for checking for controllability by an example. This is not the algorithm given in [RW87c]. It is a simpler form of the algorithm. A formalized version of this algorithm will be used in Chapter 3.

Example 2.8.5. Consider the Plant and Controller in Figures 2.6 and 2.7. To check controllability, we first append a *dump* state q_d to the machine M and form the *complement* of M. The complement of M is a machine M' such that $\mathcal{L}_{\mathcal{M}}(M') = \Sigma^* \setminus \mathcal{L}_{\mathcal{M}}(M)$ *and* $\mathcal{L}(M') = \Sigma^*$ *and* M' has only the extra dump state and extra transitions leading to the dump state.

When constructing the complement of M, if a symbol transition σ is not defined at a state q, then we define $\delta(q, \sigma) = q_d$. We define $\delta(q_d, \sigma) = q_d$ for all $\sigma \in \Sigma$. Lastly, we set the only final state of M' to be q_d. We show the complement of M

in Figure 2.9.

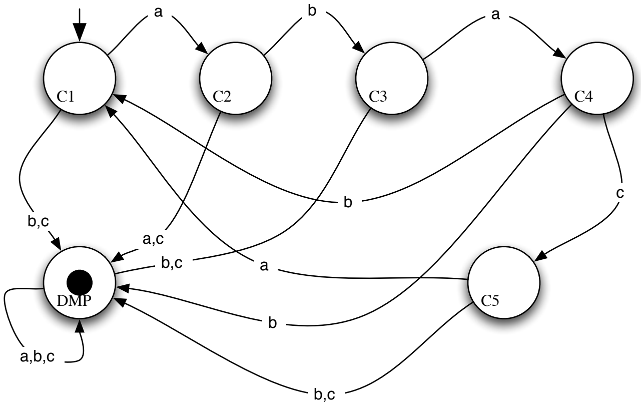


Figure 2.9. The complement machine to M.

The second step is to form the *synchronous composition* of M' and G. This will be done just as we did in the previous example. The result is shown in Figure 2.10 The states corresponding to the dump state in M' are shown circled in the figure

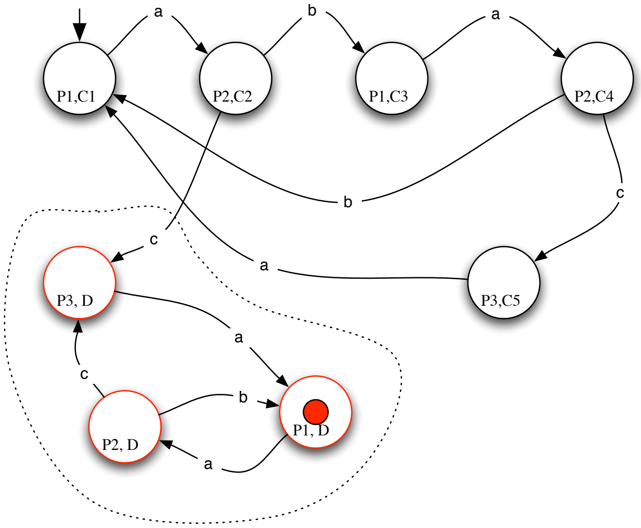


Figure 2.10. The intersection of M' and G.

and are labeled $(P1, D), (P2, D), (P3, D)$. We will call these “dump” states and the remaining states “non-dump states”. We now have the following fact: if there is an uncontrollable transition leaving the non-dump states and entering the dump states, then the language K is uncontrollable with respect to the plant model G.

We will prove a very strong form of this statement in Chapter 3.

Recall our discussion about controllability. We suggested that the reader think of K as a plan and the controllability criterion as saying something about the completeness of the plan in the face of exceptions (uncontrollable events). Transitions leading to the dump state in M' are precisely the transitions that are not in the plan. So, if an uncontrollable event leads into the dump states, then there is no plan for that uncontrollable event. In our example, no uncontrollable events lead to the dump states, so we've planned for all exceptions. We can disable ("turn off") c thus preventing ourselves from entering the dump states. This is precisely what is meant by controllability.

2.8.2 Identifying the Plant Model

All discrete event control problems begin by determining a plant model. Most discrete event control problems assume that the plant model is provided, or can be determined using some system description. An alternative approach is to use a hidden Markov model [Rab89] discovery algorithm along with an observed sequence of events w. A better hidden Markov model algorithm for this problem is the CSSR Algorithm of Crutchfield and Shalizi [SS04, SSC02b, SSC02a]. Their algorithm takes a series of symbols and a single numeric value d, the maximal history length of interest, and produces a plant model with associated transition probabilities. The only draw back to this approach is no unique start state or final states can be identified. However, the controller designer may be able to determine this information from additional data.

2.8.3 Supremal Controllable Sublanguage

For the remainder of this section, let $L = \mathcal{L}_{\mathcal{M}}(G)$ and suppose that G is trim so that $\mathcal{L}(G) = \overline{L}$. This will simplify our notation significantly. One nice property of controllability is that it is preserved under union.

Lemma 2.8.6. *Let $K_1, K_2 \subseteq L$ be controllable sublanguages. Then $K_1 \cup K_2$ is also a controllable sublanguage of L.*

Proof. See Proposition 7.1 of [RW87c]. □

It may be the case that given $K \subseteq L$, there is no supervisor to enforce K. The *supremal controllable sublanguage* of K (with respect to L or G) is the largest sublanguage $\sup_{\mathrm{C}}(K) \subseteq K$ such that $\sup_{\mathrm{C}}(K)$ is controllable with respect to G (or L). It follows from Lemma 2.8.6 and from a simple Zorn's Lemma argument that $\sup_{\mathrm{C}}(K)$ exists for any given K and L. In the case when K is controllable, then $\sup_{\mathrm{C}}(K) = K$. In [RW87b] Ramadge and Wonham showed that if $K \in \mathbf{REG}$ and $L \in \mathbf{REG}$, then $\sup_{\mathrm{C}}(K) \in \mathbf{REG}$. They gave an algorithm to determine $\sup_{\mathrm{C}}(K)$ in this case. Kumar et al. [KGM91] gave a simpler algorithm that runs in quadratic time when $K = \overline{K}$ and $L = \overline{L}$, that is both the plant language and objective language are prefix closed. We will return to the subject of supremal controllable sublanguages in detail in Chapters 5 and 6.

2.9 Decidability in Control Theory

In [RW87c], Ramadge and Wonham note that the state set Q need not be restricted to a finite set. This statement was repeated in most of the earliest papers on discrete event control, yet no one really addressed the problem of discrete event control with infinite state sets. Why? Consider the definition of controllability in Expression 2.8.1. Let us replace $\mathcal{L}_{\mathcal{M}}(G)$ with L and consider the full definition in the absence of our convenient prefix closure notation. Expression 2.8.1 is equivalent to

$$\forall w \forall u (u \in \Sigma_u \wedge \exists s (ws \in K) \wedge \exists t (wut \in L) \implies \exists v (wuv \in K)). \tag{2.9.1}$$

This statement says, "For all strings w and for all symbols u, if u is in Σ_u and there is some s such that ws is in K and there is some t such that wut is in L, then there is some v such that wuv is in K also." By migrating the existential quantifiers to the front of the logical expression, we can write this equivalently as:

$$\forall w \forall u \exists s \exists t \exists v (u \in \Sigma_u \wedge (ws \in K) \wedge (wut \in L) \implies (wuv \in K)). \tag{2.9.2}$$

Without loss of generality, let either L or K be the language accepted by a Turing Machine that halts on all inputs; i.e., a recursive language. In the parlance of Mathematical Logic, Expression 2.9.2 is called a Π^0_2 sentence [Rog87]. A Δ^0_1

sentence is any combination (with AND's, OR's and NOT's) of expressions of the form $w \in L$ or $w \in K$, when K and L are recursive languages. For this reason, Δ_1^0 expressions are also called recursive. They *can be decided* by a Turing machine. A sentence that is of the form $\exists x_1, \ldots, \exists x_n R(x)$, where $R(x)$ is a Δ_1^0 expression is called Σ_1^0. The truth of these sentences is *not decidable* for all values of x. A Π_2^0 sentence is obtained from a Σ_1^0 sentence by prepending any number of universal ($\forall$) quantifiers to it. Figuratively speaking, if a Σ_1^0 sentence is undecidable, then a Π_2^0 sentence is *really* undecidable. Hence we have found an interesting result:

Theorem 2.9.1. *For arbitrarily chosen recursive languages K and L with $K \subseteq L$, it is undecidable whether or not K is controllable with respect to L.*

This is why there were no results on infinite state machines early in the history of discrete event control. Once you leave finite state machines behind, you must exercise extreme caution or the fundamental criterion of controllability becomes undecidable. This fact was shown in practice first by Sreenivas [Sre93]. Sreenivas showed that when K and L are languages accepted by so-called *labeled Petri nets*, then it is undecidable whether K is controllable with respect to L. He also showed that when K and L are *labeled free Petri nets*[3], then it is decidable whether K is controllable with respect to L. This was the first instance of an infinite state machine being used for discrete event control. Until 2006, this was the only instance of the use of a machine capable of expressing an infinite number of states in discrete event control.

[3] The precise definition of Petri Net, Labeled Petri Net and Labeled Free Petri Net is *not* important. The concepts are not used elsewhere in this thesis.

Chapter 3

On the Decidability of the Control Predicate in Pushdown Systems

3.1 Introduction and Chapter Overview

In this Chapter we show the decidability of the controllability predicate when objective language K is accepted by a DPDA and the plant language L is accepted by a FSM. The majority of the results presented here are published in [Gri06].

Before proceeding, we first introduce some terminology that is essential to understanding this chapter. We have previously stated that the object of a supervisor φ is to enforce some language $K \subseteq L$, where $L = \mathcal{L}_{\mathcal{M}}(G)$ is the marked language accepted by a state machine plant model G. Often times, the target (or objective) language K is derived by intersecting L with some given *specification language* $S \subseteq \Sigma^*$. We think of S as describing the desired behavior of the system irrespective of the physical constraints (L) in the plant. The objective language $K = S \cap L$ is the physically obtainable acceptable behavior. As we know from Chapter 2, a supervisor φ exists only if $K = S \cap L$ is *controllable* with respect to L (or equivalently G). We say that a specification S is controllable with respect to L if $K = S \cap L$ is controllable. In the event that $S \subseteq L$, then clearly $S = K$ and the result is still the same.

Our main goal in this chapter is to prove the following theorem:

Theorem 3.1.1 (Griffin 2006)**.** *Controllability is decidable for a specification S*

generated by a DPDA M and a plant language L generated by a finite state machine G. That is, if S is accepted by a DPDA and L is accepted by a finite state machine, then it is decidable whether $K = S \cap L$ is controllable with respect to L.

To prove this theorem, we will first derive an equivalent and more useful expression for the controllability predicate (Expression 2.8.1). We then use the results of Sénizergues [S9́7] to show that our new, equivalent, definition of controllability is decidable when S (or K) is accepted by a DPDA and L, the plant language, is accepted by a finite state machine. Having established this result, we go onto provide an algorithm that will determine whether a given S (or K) is controllable with respect to a given L. We end this chapter with a protracted example of the use of this algorithm.

As an ancillary result, we also show that it is undecidable for an arbitrary specification S generated by a non-deterministic pushdown automaton and plant language L generated by a finite state machine whether $K = S \cap L$ is controllable with respect to L.

3.2 Proof of Main Theorem

The proof of the first lemma relies on several laws of logic [Sim00].

Lemma 3.2.1 (Griffin 2006, [Gri06])**.** *Let S be a specification and let L be a plant language. If $K = S \cap L$, then the following are equivalent:*

1. $\overline{K}\Sigma_u^* \cap \overline{L} = \overline{K}$.

2. $K = S \cap L$ *is controllable with respect to* L.

3. $\left(\left(\overline{K}\right)^c \cap \overline{L}\right)/\Sigma_u^* = \left(\overline{K}\right)^c \cap \overline{L}$.

Proof. We shall prove 1 and 2 to be pairwise equivalent and likewise 2 and 3 to be pairwise equivalent.

(1 $\iff$ 2) This is Theorem 3.2 of [KGM91].

(2 $\iff$ 3) We first prove 2 $\implies$ 3: The fact that ϵ is an element of Σ_u^* shows that $\left(\overline{K}\right)^c \cap \overline{L} \subseteq \left(\left(\overline{K}\right)^c \cap \overline{L}\right)/\Sigma_u^*$, since $\left(\left(\overline{K}\right)^c \cap \overline{L}\right)\epsilon = \left(\overline{K}\right)^c \cap \overline{L}$. Suppose that σ is a member of $\left(\left(\overline{K}\right)^c \cap \overline{L}\right)/\Sigma_u^*$. Then it follows that $\sigma \in \overline{L}$, since there is a $u \in \Sigma_u^*$

such that $\sigma u \in \overline{L}$ and $\overline{L}$ is prefix closed. It now suffices to show that $\sigma \in \left(\overline{K}\right)^c$. Suppose not; there is a $\sigma \in \overline{K}$ (and hence $\overline{L}$) and a $u \in \Sigma_u^*$ such that $\sigma u \in \overline{L}$ but $\sigma u \notin \overline{K}$. This contradicts controllability. Hence, $\sigma \in \left(\overline{K}\right)^c$.

To prove 3 $\implies$ 2, assume that $\left(\left(\overline{K}\right)^c \cap \overline{L}\right)/\Sigma_u^* = \left(\overline{K}\right)^c \cap \overline{L}$. In set theoretic notation that is: $\left\{s : \exists u \in \Sigma_u^* \left(su \in \overline{L} \wedge su \notin \overline{K}\right)\right\} = \left\{s : s \in \overline{L} \wedge s \notin \overline{K}\right\}$. By equality, we have the following universal sentence:

$$\forall s \left(\exists u \in \Sigma_u^* \left(su \in \overline{L} \wedge su \notin \overline{K}\right) \implies \left(s \notin \overline{K} \wedge s \in \overline{L}\right)\right).$$

By contrapositive and DeMorgan rules, we may transform the implication inside the universal quantifier to obtain the new (equivalent) sentence:

$$\forall s \left(\left(s \in \overline{K} \vee s \notin \overline{L}\right) \implies \neg\exists u \in \Sigma_u^* \left(su \in \overline{L} \wedge su \notin \overline{K}\right)\right)$$

We now apply three logical rules: (i) $(A \vee B) \implies C \equiv (A \implies C) \wedge (B \implies C)$, (ii) $\forall x(A(x) \wedge B(x)) \equiv \forall x A(x) \wedge \forall x B(x)$, and (iii) $\neg\exists x(A(x)) \equiv \forall x(\neg A(x))$ to obtain:

$$\begin{gathered}\forall s \left(s \in \overline{K} \implies \forall u \in \Sigma_u^* \left(su \notin \overline{L} \vee su \in \overline{K}\right)\right) \wedge \\ \forall s \left(s \notin \overline{L} \implies \forall u \in \Sigma_u^* \left(su \notin \overline{L} \vee su \in \overline{K}\right)\right)\end{gathered}$$

We know that $(\neg A \vee B) \equiv (A \implies B)$ and we use this to deduce the equivalent sentence:

$$\begin{gathered}\forall s \left(s \in \overline{K} \implies \forall u \in \Sigma_u^* \left(su \in \overline{L} \implies su \in \overline{K}\right)\right) \wedge \\ \forall s \left(s \notin \overline{L} \implies \forall u \in \Sigma_u^* \left(su \in \overline{L} \implies su \in \overline{K}\right)\right).\end{gathered}$$

The first conjunct, namely $\forall s \left(s \in \overline{K} \implies \forall u \in \Sigma_u^* \left(su \in \overline{L} \implies su \in \overline{K}\right)\right)$ is the definition we've given for controllability. The second conjunct is tautological. To see this, note that $s \notin \overline{L}$ implies that $su \notin \overline{L}$ by prefix closure. Hence $su \in \overline{L} \implies su \in \overline{K}$ is always true (because $su \in \overline{L}$ is false). This, in turn, implies that $s \notin \overline{L} \implies \forall u \in \Sigma_u^* \left(su \in \overline{L} \implies su \in \overline{K}\right)$ is true (because $su \in \overline{L} \implies su \in \overline{K}$ is true).

Hence we have shown that if $\left(\left(\overline{K}\right)^c \cap \overline{L}\right)/\Sigma_u^* = \left(\overline{K}\right)^c \cap \overline{L}$ if and only K is controllable in L. □

Remark 3.2.2. At this point, we are ready to prove the main theorem of this chapter. The proof relies on the results mentioned in Lemma 2.6.2. Specifically, we use the power of the result of Sénizergues, which states that it is decidable whether the language generated by a given DPDA M_1 is the same as the language generated by a second DPDA M_2; i.e., whether $\mathcal{L}_{\mathcal{M}}(M_1) = \mathcal{L}_{\mathcal{M}}(M_2)$ [S9́7]. Sénizergues' result gives us an extremely quick and elegant way to prove the main theorem of this chapter. However, as we will see in the next section, using Sénizergues' result is not a particularly efficient way to check controllability.

Proof of Main Theorem. Let $K = S \cap L$. Then $\overline{K}$ is generated by a DPDA, hence $\left(\overline{K}\right)^c$ is generated by a DPDA by Theorem 10.3 of [HU79]. Likewise, $\left(\overline{K}\right)^c \cap \overline{L}$ and $\left(\left(\overline{K}\right)^c \cap \overline{L}\right)/\Sigma_u^*$ are generated by DPDA by Theorem 10.5 of [HU79]. Apply Lemma 2.6.2 with Lemma 3.2.1 to obtain the result. □

Corollary 3.2.3. *Let L be a plant language, let S be a specification and let $K = S \cap L$. Then the following are equivalent:*

1. *The question: "Is $K = S \cap L$ controllable with respect to L?" is decidable.*
2. *$\left(\overline{K}\right)^c \cap \overline{L} = \left(\left(\overline{K}\right)^c \cap \overline{L}\right)/\Sigma_u^*$ is decidable.*
3. *$\overline{K}\Sigma_u^* \cap \overline{L} = \overline{K}$ is decidable.*

3.3 Testing Controllability

One problem in using the result of Sénizergues [S9́7] is that his algorithm for determining the equivalence of the two DPDA does not have a running time bound. This problem has been investigated by Stirling [Sti02], who has proved that the time bound is at worst a primitive recursive function. However, this could still imply that the time required for the procedure is super-exponential.

Our goal is to show that we can compute whether $\left(\overline{K}\right)^c \cap \overline{L} = \left(\left(\overline{K}\right)^c \cap \overline{L}\right)/\Sigma_u^*$ by checking whether there is a path labeled by a string $v \in \Sigma_u^*$ leading backward from a marked state of a machine accepting $\left(\overline{K}\right)^c \cap \overline{L}$ to a non-marked state. This is analogous to current controllability checking procedures [CL99] as outlined in Chapter 1.

If M is a DPDA, then a Σ_u reverse path in M between states q_m and q_1 is a sequence of states $q_1, \ldots, q_m$ and a string $\upsilon = u_1 \ldots u_n \in \Sigma_u^*$ such that for $\zeta_1 \in \Gamma^*$, and $\sigma \in \Sigma^*$ (i) $(q_0, \sigma\upsilon, Z_0) \vdash^* (q_1, \upsilon, \zeta_1)$ and (ii) $(q_1, \upsilon, \zeta_1) \vdash^* (q_m, \epsilon, \zeta_f)$ for some $\zeta_f \in \Gamma^*$. Note we *are* considering the possibility of ϵ-transitions in this path.

This approach will not work for arbitrary DPDA's since it may be the case that we can never reach a given state q from the starting state. Furthermore, we may be unable to determine whether or not the transition used to arrive at q will even be fired. If this is the case, then the fact that there is a string υ in Σ_u^* that leads from this state to a final state is unimportant. This problem can be solved by determining whether each transition used in the construction of υ could actually be fired and whether or not the state q can be reached.

Lemma 3.3.1. *There is an algorithm to remove all unreachable states and useless transitions from a DPDA M.*

Proof. To determine whether a state q in M is unreachable, construct a new DPDA M' whose only marked state is q. It is easy to see that $\mathcal{L}_{\mathcal{M}}(M') = \emptyset$ if and only if q cannot be reached, i.e., q is unreachable. By Theorem 6.6 of [HU79] it is decidable whether $\mathcal{L}(M') = \emptyset$. To remove unreachable states from M, iterate through the states of M, testing each for reachability. Remove unreachable states.

Consider a transition $(q, x, X, q', \gamma) \in \delta$. That is, when the DPDA M is in state q with top stack symbol X and symbol x is read, then M transitions to state q' and pushes string γ onto its stack. To determine whether this transition is useless, construct a new PDA M' with only one marked state, q_d, not in the state set of M. Replace the transition (q, x, X, q', γ) by the transition (q, x, X, q_d, γ). Then add transitions at q_d to empty the stack. By determinism, it is clear that $\mathcal{L}_{\mathcal{M}}(M') = \emptyset$ if and only if the transition (q, x, X, q', γ) could never fire in M. Applying Theorem 6.6 of [HU79] again, we see that this question is decidable. To remove useless transitions from M, iterate through the transitions of M, testing each for uselessness. Remove useless transitions. This completes the proof. □

It is worth noting that it is *not* sufficient to check for unreachable states in order to remove useless transitions, since the configuration of the DPDA is determined not only by its state but also by its stack. Hence, a transition may never fire because the appropriate stack symbol never appears on top of the stack at the correct time and not just because a certain state is never reached.

Theorem 3.3.2. *Let L be a language generated by a DPDA and let*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, Q_f)$$

be a DPDA accepting L by final state with no useless transitions or unreachable states. Then $L/\Sigma_u^* = L$ if and only if no Σ_u reverse path leaves Q_f [1].

Proof. Suppose that $L = L/\Sigma_u^*$ and Q_f is not closed under Σ_u^* reverse paths. Then there is a word $s = \sigma \upsilon$ such that s is accepted by M in some final state q_f and there is a reverse path from q_f along υ^R leading to a state $q \notin Q_f$. By the determinism of M, if σ were to be accepted, then it would be accepted in q or there would be a series of ϵ-transitions leading to another accepting state q'. Hence σ is not an element of L thus contradicting our assumption.

Conversely, suppose that $L \neq L/\Sigma_u^*$ and Q_f is closed under Σ_u^* reverse paths. Then there is a string σ in L/Σ_u^* but σ is not in L. Let υ be a string in Σ_u^* such that $\sigma\upsilon \in L$. There is a final state q_f such that M accepts $\sigma\upsilon$ in q_f. Since σ is a prefix of $\sigma\upsilon$, there is at least one state q not in Q_f such that $(q_0, \sigma, Z_0) \vdash^* (q, \epsilon, \gamma)$ and $(q, \upsilon, \gamma) \vdash (q_f, \epsilon, \gamma')$. Then this defines a Σ_u^* reverse path that leaves the marked states, since we assumed that $\sigma \notin L$. This completes the proof. □

It is easy to see that we can check whether a DPDA has its final states closed under Σ_u reverse paths by traversing the graph backward from the marked states following ϵ-transitions and uncontrollable transitions, once we have removed unreachable states and useless transitions. Alternatively, we may use a predicting machine as we discuss in Chapter 5.

Algorithm $\mathfrak{A}$ checks for the controllability of a DPDA controller and a finite state machine plant model. Assume we are given a DPDA M that accepts S a specification and a plant model G that accepts a plant language L. The first step is to compute a machine M_K that accepts that language K we wish to analyze for controllability. We do this by forming a machine $M_K = M \cap G$ that accepts

[1] We call this being closed under Σ_u reverse paths.

$K = S \cap L$. The next step is to form a machine N that accepts $\overline{K}^c \cap L$. We do this because we must check whether $(\overline{K}^c \cap L)/\Sigma_u^* = \overline{K}^c \cap L$. To do this check, we look at machine N and determine whether or not N has any Σ_u reverse paths. We discussed how to do this above. If we find a Σ_u reverse path, then K is not controllable with respect to L. Otherwise, K is controllable with respect to L.

Algorithm Description 3.3.1 – Algorithm $\mathfrak{A}$

1. Given M accepting S the specification and G accepting L, the plant language.
2. Compute $M_K = M \cap G$.
3. Compute machine $\overline{M_K}$ accepting $\overline{K}$.
4. Compute machine $\overline{M_K}^c$ accepting $\overline{K}^c$.
5. Compute machine $\overline{G}$ accepting $\overline{L}$.
6. Compute a machine $N = \overline{G} \cap \overline{M_K}^c$.
7. Remove useless states and transitions from N.
8. If N has any Σ_u reverse paths, then K is not controllable with respect to L. If N does not have any Σ_u reverse paths, then K is controllable with respect to L.

3.4 Example

We consider a contrived automated fabrication scenario. A machine has been constructed that can take requests from users who require widgets. The users and machine are separated, so user requests may come into the machine while it is building a widget. Manufacturing of a *widget* requires five basic steps:

1. Cut (the basic shape),
2. Bend (make a bend in the shape),
3. Rotate (rotate the product counter-clockwise 180°),
4. Bend (make a second bend in the shape),
5. Punch (make a round hole in the shape), and

Suppose that manufacturing studies have indicated that bending rarely fails, so after steps 1, 3, and 5, the product is inspected by an electronic eye for quality assurance. If the electronic eye detects a fault, the widget is discarded and the process begins again. At the completion of each job, an operator may choose to start a new production round or halt production. We define the following event alphabet for this process:

(a) Σ_c

Event	Description
c	Cut
b	Bend
r	Rotate
p	Punch

(b) Σ_u

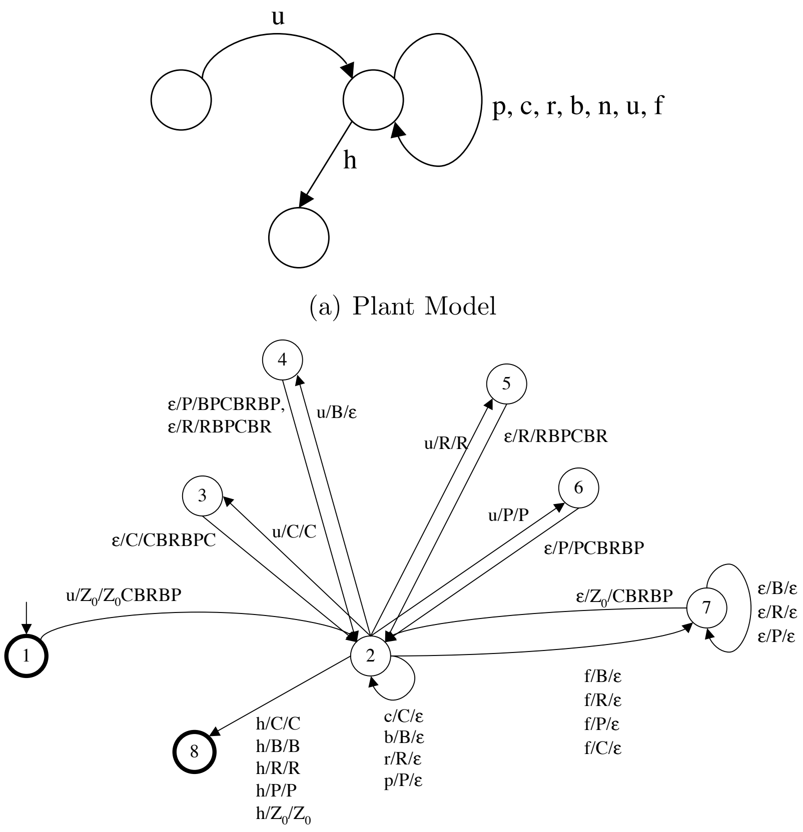
Event	Description
f	Fault in quality assurance test
u	New production request
h	Halt production

Table 3.1. Event definitions and descriptions

Let $\Gamma = \{C, B, R, P, Z_0\}$, where lower-case events correspond to their upper-case counterparts. The controller for this process is shown in Figure 3.1(b). Marked states are denoted by a thick state border. Denote this machine by M.

The proposed control system has nine states but cannot be modeled by a finite state machine. To see this we observe that the control system must be able to store instructions for each u event that occurs. Apply the Pumping Lemma for Regular Languages (Chapter 2). Choose $n \geq 0$ and consider the string $w = u^k(cbrbp)^k \in \mathfrak{L}_{\mathfrak{M}}(M)$, for $k > 0$ so that w is the shortest string possible with length greater than n. (In particular, $k = \lceil n/6 \rceil$, so $|w| - n \leq 5$.) Choose $x, y, z \in \Sigma^*$ so that $xyz = w$ and $|xy| \leq n$. Then it is easy to see that no matter how x, y and z are chosen, if $i \geq 2$ for which $xy^iz \notin \mathfrak{L}_{\mathfrak{M}}(M)$. Thus, $\mathfrak{L}_{\mathfrak{M}}(M)$ is not regular and hence cannot be generated by a finite state machine because it fails to meet the criteria of the Pumping Lemma.

If we assume that the widget making machine is capable of executing any combination of bending, rotating etc. before receiving a halt signal, then it is clear that the proposed control system is controllable. Such a plant model is shown in Figure 3.1(a). Denote this plant model by G. We shall prove the controllability by Algorithm $\mathfrak{A}$. We have constructed M so that $\mathfrak{L}_{\mathfrak{M}}(M) \subset \mathfrak{L}(G) = \mathfrak{L}_{\mathfrak{M}}(G)$. This fact can be checked by inspection. If it were not the case that $\mathfrak{L}_{\mathfrak{M}}(M) \subseteq \mathfrak{L}_{\mathfrak{M}}(G)$, then we could compute a DPDA that accepts $K = \mathfrak{L}_{\mathfrak{M}}(M) \cap \mathfrak{L}_{\mathfrak{M}}(G)$ using the procedure defined in Chapter 2. In the present case we see that M accepts $L \cap S$



(a) Plant Model

(b) Pushdown Automata Controller

Figure 3.1. Plant Model and Controller for Widget Production

as required by step one of Algorithm $\mathfrak{A}$, where $L = \mathcal{L}_{\mathcal{M}}(G)$ and $S = \mathcal{L}_{\mathcal{M}}(M)$. To consider $\overline{K}$ we mark every state of M. Figure 3.2 shows an intermediate form of M required to compute a machine accepting $\left(\overline{K}\right)^c$.

This automaton accepts $\mathcal{L}(M)$ but scans an entire input string; *i.e.*, we add transitions that may or may not be used to ensure that regardless of the string input on the tape, the control head of the DPDA will scan the whole string. This intermediate form is defined in Chapter 2. Note useless transitions have been added to insure that the state machine does not halt on any input. For example, consider the transition $u/X_0/X_0$ defined at state 2. This transition will never be fired because there is no transition to pop the stack symbol Z_0 that is positioned above X_0 at the start state. Denote this automaton by N.

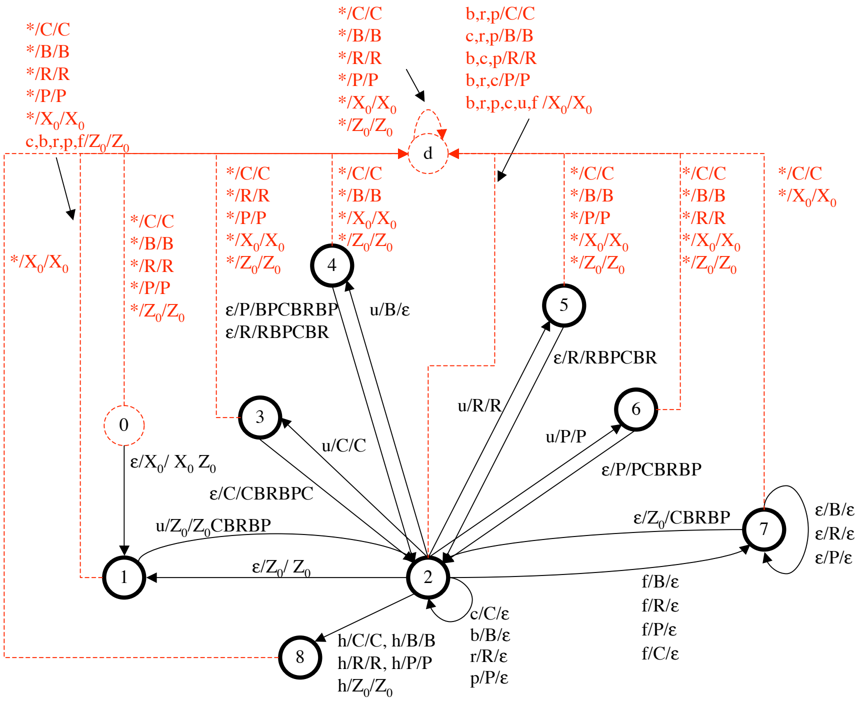


Figure 3.2. An intermediate form of M that scans all input but accepts the same language as M.

We next apply the procedure defined in Chapter 2 to find the complement of $\mathcal{L}_{\mathcal{M}}(N)$. Denote this machine by N^c. It is shown in Figure 3.3.

The weave of G and N^c is shown in Figure 3.4. Unreachable transitions have been grayed out. Thus we see that the step reducing M to N^c will leave useless transitions in the complementary DPDA. This shows that the reduction step of Algorithm $\mathfrak{A}$ is necessary.

It is now clear that every uncontrollable transition that enters a marked state is a useless transition. The only transition entering the marked states is labeled by controllable events and connects state $(2,1)$ with $(d,2)$, an ϵ transition then connects state $(d,2)$ to the marked state $(d,3)$. Thence we conclude that the system is in fact controllable. If we had failed to define an uncontrollable event or a deadlock occurred that caused this system to be uncontrollable, then one of the transitions that were added in the intermediate form would not be useless and hence an uncontrollable transition would lead from a non-marked state to a marked state (possibly using ϵ transitions). It follows at once that our widget

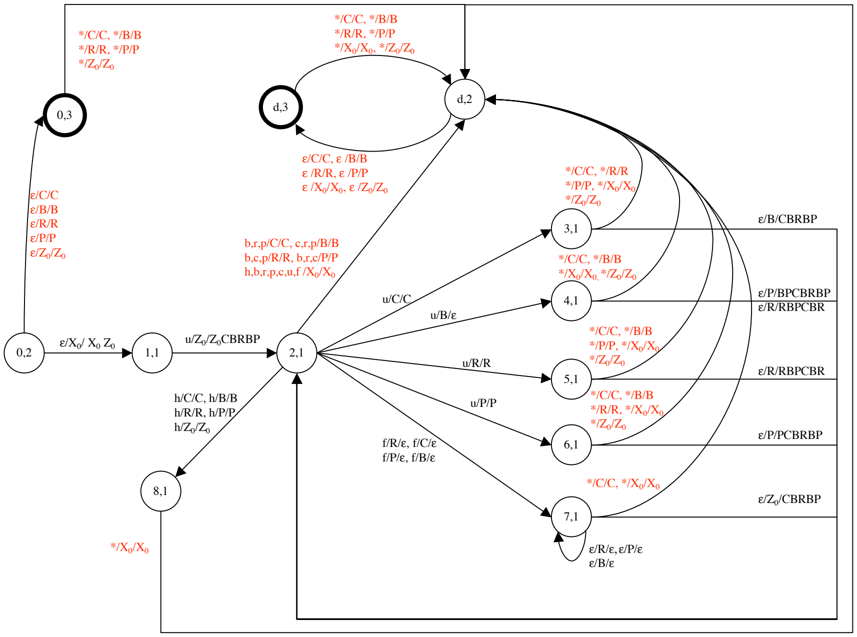


Figure 3.3. A machine that accepts of $\left(\overline{K}\right)^c$. Note there are a number of useless transitions.

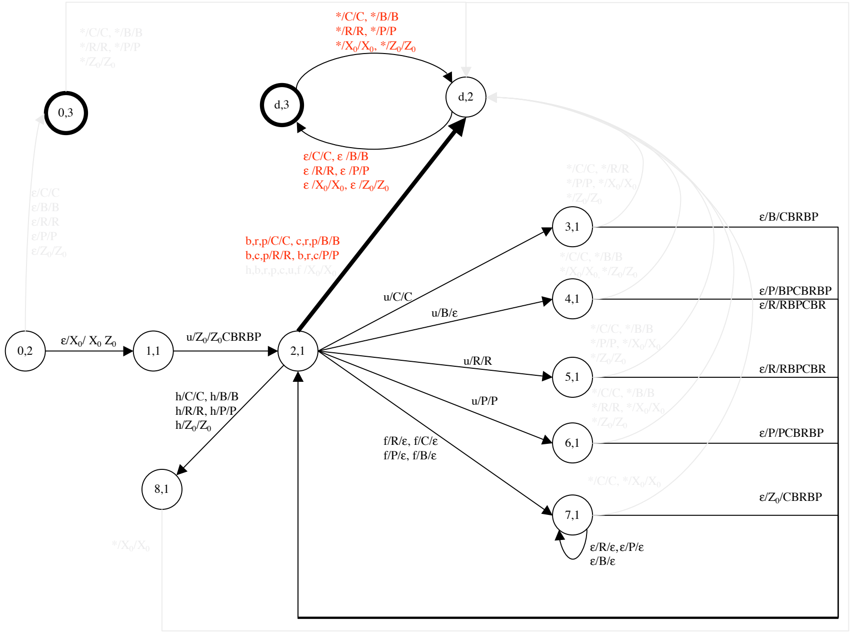


Figure 3.4. The weave of M^c and G. This automaton accepts $\mathfrak{L}(G) \cap \mathfrak{L}(M)^c$. Useless transitions have been grayed out.

making control system is controllable.

3.5 Algorithmic Complexity

In this section, we prove that Algorithm $\mathfrak{A}$ is P-complete [2].

Theorem 3.5.1. *Algorithm $\mathfrak{A}$ is P-complete with log-space reductions.*

Proof. Step 6 of Algorithm $\mathfrak{A}$ requires us to remove useless states and transitions from machine N. In Lemma 3.3.1 we showed that to check this required verifying that a certain DPDA accepted the empty language. Theorem 13.12 of [HU79] shows that the emptiness problem for Context Free Grammars is complete in P with log-space reductions. Hence, it follows that the emptiness problem for arbitrary DPDA is at worst P-complete. Now, since we may choose arbitrary DPDA S, and let $\mathcal{L}_{\mathcal{M}}(G) = \Sigma^*$, it follows at once that Algorithm $\mathfrak{A}$ is P-complete. This concludes the proof. □

3.6 Undecidability Result

Lemma 3.6.1. *Let $\Sigma = \{0,1\}$; either let $\Sigma_u = \{0\}$ or $\Sigma_u = \{1\}$ and let $P(K)$ be the predicate on Σ saying $\overline{K}\Sigma_u^* = \overline{K}$, for some language K accepted by PDA over Σ. Then P is undecidable.*

Proof. We apply Lemma 2.6.6. First, if $K = \Sigma^*$, then clearly $\overline{K}\Sigma_u^* = \overline{K}$, hence P holds for Σ^*. Now, let $x \in \Sigma^*$ be a word. If $\{x\}\backslash K \neq \emptyset$, then $\{x\}\backslash K$ is infinite; suppose that $y \in \{x\}\backslash K$. Then $xy \in K$. Choose any prefix z of y. Then $xz \in \overline{K}$. Let $u \in \Sigma_u^*$. Then $xzu \in \overline{K}$ and hence there is some $w \in \Sigma^*$ such that $xzuw \in K$. Therefore, $zuw \in \{x\}\backslash K$. Thus we have shown that given any word $y \in \{x\}\backslash K$, we can find a longer word $y'(= zuw)$ also in $\{x\}\backslash K$. Hence it follows that $\{x\}\backslash K$ is infinite. The finite, non-empty, context free languages are not in the set

$$P_{\text{Left}} = \{K' \mid K' = \{x\}\backslash K, x \in \Sigma^*, P \text{ holds for } K\}.$$

[2] P is the space of all problems computable in polynomial time on a deterministic Turing Machine. Compare this to the class NP, which is the set of all problems computable in polynomial time on a non-deterministic Turing machine. Discussion of these problems is *far* outside the scope of this thesis and would add excessive volume. The reader is referred to Chapters 12-13 of [HU79] or [CLRS01] for details.

Hence P_{Left} is a proper subset of the languages accepted by PDA over Σ. Applying the conclusion of Theorem 2.3 of [IR74] (see Lemma 2.6.6) we have that P is undecidable for a language K given by an arbitrary PDA G. □

Theorem 3.6.2 (Griffin 2006)**.** *It is undecidable for an arbitrary specification S generated by a non-deterministic pushdown automaton and plant language L generated by a finite state machine whether $K = S \cap L$ is controllable with respect to L.*

Proof. Let $L = \Sigma^*$, this is regular. Controllability of the language K reduces to the predicate considered in the theorem. □

3.7 Conclusion

We have shown that it is not decidable whether arbitrary specifications generated by PDA are controllable against regular plant models. We have demonstrated the positive result that controllability can be tested for specifications generated by DPDA and regular plant models.

Deterministic pushdown automata can be used to form specifications that are more complicated than regular specifications. In particular, the pushdown stack can be used as a to-do list for future action. In this case, uncontrollable events can be seen to alter the composition of the list causing the controller to re-plan future actions. Furthermore, nested response to uncontrollable events can be encoded using DPDA. For example, suppose that a string w must be enabled each time an uncontrollable event u is observed. Then the resulting behavior $\{u^n w^n | n \geq 1\} \subseteq K$ cannot be specified by a finite state automaton and hence a DPDA must be used.

It is worth noting that we have not said anything about the case when both the specification and plant are given by DPDA. We have reasons to suspect that in this case the controllability predicate is undecidable. However, we are not aware of any proof of this statement.

Chapter 4

Verification of Secure Network Protocols in Uncertain Environments

4.1 Introduction

In this chapter, we provide a method for determining whether or not a given protocol specified by automata models can cope with all known uncontrollable hazards, and a method of estimating the confidence level for controllability of a protocol in the case where this fails. Many of the results presented here were published in [DGP$^+$06] Our method for protocol checking and verification takes formalisms from the literature of discrete event control. By protocol checking and verification, we mean verifying that a protocol is logically correct, that it does not cause deadlocks, and that it has been defined to respond to uncontrollable events that may occur in a system implementing it. As protocols and open standards have become increasingly important, a significant amount of work has been done on verifying and designing protocols [HHL$^+$93]. In the past, others have suggested that discrete event control was insufficient for verifying protocols because of certain inadequacies in the power of the models used [BCV90]. We have corrected some of these problems by extending the elementary theory of discrete event control to include specifications written as deterministic pushdown automata; these machines are capable of accepting a wider class of languages than finite state machines and hence can model more complicated systems. In addition to this, we have defined a maximum probability method for analyzing a protocol's response to *a priori* unspecified uncontrollable events. Our approach uses a specialized hidden Markov model that serves as a generator of uncontrollable events. Using this approach we can precisely define what we mean by *robustness in uncertain environments*.

The analysis we suggest provides a simple solution to many of the problems noted in [BCV90]. It was observed that systems as complicated as networks are often impossible to model in a discrete event control context. By extending the class of models that can be analyzed using discrete event control, we have extended the class of protocols that can be successfully verified using this approach. Furthermore, by defining an optimal robustness measurement for un-modeled uncontrollable events, we have reduced the amount of complexity that must be modeled in order to verify a given protocol. Once these problems are removed, it becomes clear that hierarchical discrete event control techniques are a natural approach for analyzing and verifying protocols for logical correctness. Moreover, hierarchically designed protocols allow the complexity to be spread out over several layers, thus allowing us to design more complicated protocols built up from simpler, layered protocols. Each of these layers and their interactions can by verified using the approach we define below.

We apply our approach to the verification of a simple two-level protocol specifying the behavior of two wireless nodes as they attempt to establish a secure connection. Recent work has been done in the formal modeling and analysis of network protocols [BCV90, SAFK^{+}03]. As we become more dependent on wireless networks for communication, the design and implementation of secure network protocols will become more important. It is simply impossible to design *ad hoc* protocols that will be responsible for secure communications throughout the world. Off-line analysis and verification of protocols before they are standardized will save time and money in the long run. The use of pushdown-automata to model aspects of protocols that cannot (or should not) be modeled by finite state automata represents a novel approach to protocol modeling and verification.

The remainder of this chapter is organized as follows: Section 4.2 presents a simple two-level hierarchical protocol that will be used to demonstrate the techniques we describe. Section 4.3 discusses the hidden Markov model approach to

Out of Range with Return	Temporarily Out of Range
Association Request	Response
Management Frame Exchange	Beacon Sent/Received
Timed Out/No Retries	Receive Acknowledgment
Connection Authorized/Associated	Send Packet/Start Timer
Timed Out	Timed Out/Retries Left
Receive Packet	Send Acknowledgment

Table 4.1. Controllable Events (Σ_c)

Security Exception	Permanent Out of Range
Out of Range	T Seconds Elapse
D Seconds Elapse	

Table 4.2. Uncontrollable Events (Σ_u)

verifying robustness of protocols in uncertain environments. Section 4.4 presents future work and conclusions respectively.

4.2 Application of Discrete Event Systems to Network Protocols

In this section we use techniques from discrete event control to analyze a two-level example protocol. Consider the events shown in Tables 4.1 and 4.2:

The protocol we describe defines a procedure to be followed when two wireless devices attempt to make a secure connection for transmitting packets. The protocol specifies the following network behaviors:

1. Allow a number of retries before aborting transmission attempt if packet transmission fails

2. Allow nodes to communicate only when connection has been both associated and authorized

3. Abort connection request if receiving node is permanently out of range

4. Receiving node should send an acknowledgment upon receipt of packet; transmitting node should wait for acknowledgment or timeout before aborting a connection

5. Beacon should be both sent and received before beginning association or authorization procedure for connection

In order to apply techniques from discrete event control, we must define a suitable plant model against which to test controllability. We make no domain knowledge assumptions; hence let Σ^* be the plant language in the example. We wish to restrict the behavior of the plant language to allow only the strings in Σ^* that meet the specifications.

4.2.1 Protocol Design

In this section we give an example protocol to be followed when two nodes attempt to establish communication and exchange data. The lower level of this hierarchy models information exchange and retry logic. (Retry logic procedures similar to these have been implemented in many wireless network protocols [Lin03, Cor03].) The upper level represents the negotiations needed to establish a secure connection between two nodes. Figure 4.1 shows the negotiations that occur when two nodes attempt to make a secure connection. The most obvious restriction made at this level is that all accepted strings must begin with the event *Beacon Sent & Received.* If the state *Authorized & Associated* is reached, then the lower-level specification determines further behavior (see Figure 4.2). (The control scheme of this model is consistent with current deployments of the 802.11 wireless protocol.)

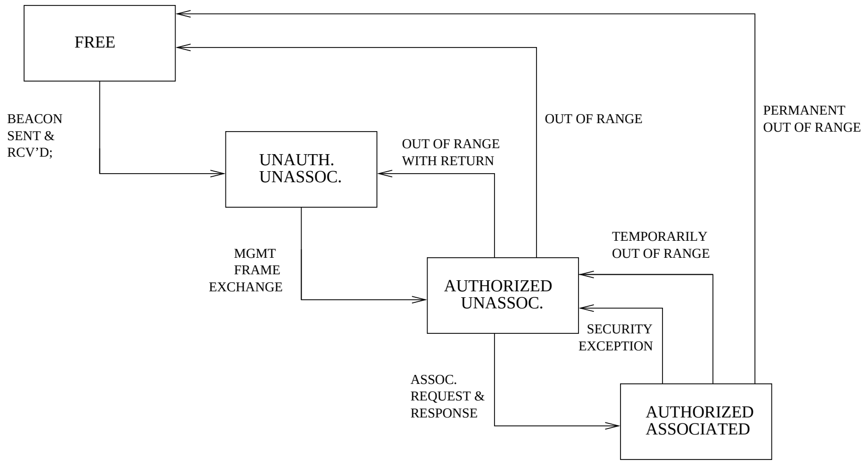


Figure 4.1. Upper-level Automaton

When one node determines that it needs to communicate with another node, it listens for available access point (AP) node beacons, chooses one, initiates an authorization and association sequence, and then begins the data exchange. Each of these compound events may involve lower-level events. (For instance, authorization entails the exchange of eight messages built from a "shared secret" string known to both initiator and recipient: query, challenge, encrypt-with-secret, authorization, and their symmetric counterparts.) As sample complications, we signal a security exception should an appropriate trigger occur (i.e., a duplicate MAC address is received), and abort the connection if a permanently out of range status is determined (perhaps due to a mid-message interrupt).

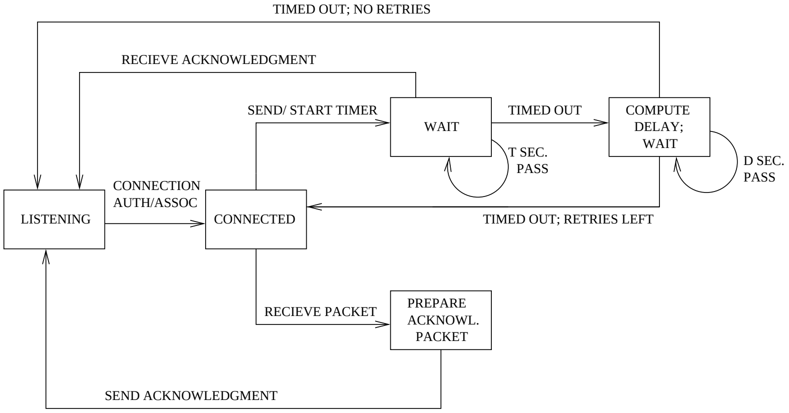


Figure 4.2. Lower-level Automaton

The upper level in the hierarchy is modeled by an FSA. Hence, the language it generates is regular. The node is modeled using a deterministic PDA, thus its language is deterministic context free. Taking the intersection of these two languages gives the deterministic context free specification language of the protocol model. The plant language is given by Σ^*, a regular language. By Theorem 3.1.1 the controllability of the protocol can be decided.

In the upper level specification, the marked states are *Free* and *Authorized Associated*. We consider *Security Exception* an uncontrollable event since it is executed in response to something uncontrollable happening (i.e., a duplicate MAC address received). Should this event occur, the *Association Request and Response* sequence is again initiated which leads back to a marked state. The other uncontrollable events, *Out of Range* and *Out of Range - Permanent* both lead the network back into a marked state.

The data exchange process is modeled by the lower level. A PDA is used here since a stack is used to keep track of the timeout and delay factors, as shown in the generic node model of Figure 4.2.

At this level, the uncontrollable events are the elapsing of the timeout (while the transmitting node waits for an acknowledgment packet) and delay (the node waits before retransmitting) periods. The loop transitions at State "Wait" and "Compute Delay: Wait" both pop symbols off the stack to simulate the passage of time. The "Start Timer" event leading to those states will push a certain number of *time unit stack symbols* onto the stack. Since a stack is used to keep track of the timeout period, the stack emptying is a controllable event (i.e., it can be disabled), and similarly for the delay period. The marked states at this level are *Listening* and *Connected*. The lapse of the timeout period and subsequently the lapse of the delay period are controllable events which lead to one of the marked states, thus the protocol is controllable with respect to the base plant model.

This system is simple enough to analyze by hand; it is clear simply from the definitions that the system is controllable. Hence the protocol has been designed to respond to all uncontrollable events that may occur at any state.

Controllability verification is the simplest protocol verification possible. It guarantees that the protocol will not deadlock in a non-accepting state and that it is sufficiently well designed to accept uncontrollable events that may occur. For larger systems, we are implementing a software tool [DGP03] that will assist in designing and automatically verify controllability for large protocols described via deterministic pushdown automata.

4.3 Resisting Network Attacks

In this section, we describe a method for verifying that a protocol maximizes the probability that control is maintained even if unobservable, uncontrollable events occur. To illustrate our approach, we describe a method for securing the wireless network protocol we've described against intrusion detection. Our approach is a form of specification based intrusion detection [SCS98, Den87]. This approach was pioneered in [Den87] and since then has become one of the major approaches to intrusion detection.

In our model, we do not account for the fact that an adversary node may eavesdrop on the first and third transmissions of the authorization exchange, thereby gaining the ability to forge valid packets without access to the WEP "shared secret." An adversary might also send packets with "stolen" MAC headers, in an attempt to fool the system into believing it was a legitimate node.

Suppose we discovered that an adversary is attempting to gain unauthorized access, and that the previously unobservable events "MAC spoof" and "packet sniff" are observed.

Since these events were not part of the original model, they are unobservable to the original controller and their dynamics are not modeled by the existing automata. However, if we can make a probabilistic "guess" automata describing the attacker's strategy, we may still say something about the (probabilistic) controllability of the new system.

Definition 4.3.1 (SARAH)**.** A Synchronous Automata Representing Attackers' Heuristics (SARAH) is a quadruple, $M = (\Sigma_S, Q_S, p_{ij}, e_i)$, where:

1. Σ_S is an alphabet,
2. Q_S is a finite set of states capable of emitting symbols from the alphabet,
3. p_{ij} is a set of state transition probabilities, and
4. e_i is a set of (state-wise) event-generation probabilities

A SARAH is a hidden Markov model whose transition and emission probabilities are linked. Moreover, a SARAH is an (unmarked) FSA whose state transitions are probabilistic and represent changes in attacker strategy that are unobservable events for the network, and whose self-loops are deterministic and contain all of the target network's controllable events. Within a SARAH's states, uncontrollable events which cause state transitions in the network model are probabilistically generated.

The synchronous automata of Figure 4.3 defines a possible time evolution for an attack strategy acting on the wireless network described above. We do not know the state evolution of this SARAH *a priori*, and thus cannot say with certainty

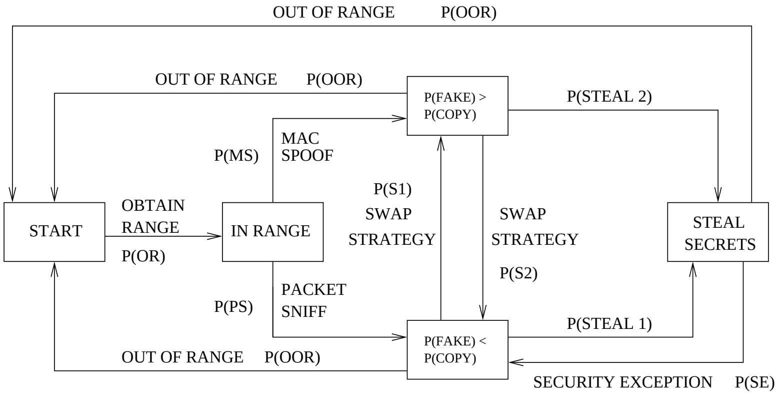


Figure 4.3. SARAH Automaton

which attacks may occur at which times. However, we can identify the most likely state evolution, as follows:

Let $\sigma = x_1 \ldots x_l$ be the string of newly observed SARAH events. Let η be a sequence denoting the state-wise time evolution of the adversary model (i.e. the l terms of η are all taken from SARAH's state set, Q.)

We wish to find a sequence of SARAH states ($\eta_i \in Q$) that leads to the maximal probability of generating the i^{th} prefix of σ.

Let $v_k(i)$ denote the maximal probability over all paths ending at k:

$$v_k(i) := \max_{\eta|\eta_i=k} P\left(x_1, \ldots, x_i|\eta\right) \tag{4.3.1}$$

It is possible to identify the most likely strategy by computing the *a posteriori* probabilities for each of the l sequences; however, this computation is not scalable for large automata. Alternately, we may construct a tree to represent the possible paths through the SARAH. The Markov property of SARAH requires that optimal paths through this tree must have optimal prefix-paths, and so it is straightforward to apply dynamic programming [Bel57] techniques to find the most likely path. The key is to consider (at each time point) only the best path leading to each state. We do this as follows:

$$v_k := \begin{cases} 1 & \text{if } k = q_0 \\ 0 & \text{else} \end{cases} \tag{4.3.2}$$

$$v_q(i+1) := e_q\left(x_{i+1}\max_{k\in Q}[v_k(i)p_{kq}]\right) \tag{4.3.3}$$

where,

$$\begin{aligned} p_{lm} &= P(\eta_i = m|\eta_{i-1} = l) \text{ and} & (4.3.4)\\ e_l(x) &= P(x_i = x|\eta_i = l)). & (4.3.5)\end{aligned}$$

We obtain the probability of realizing the observed uncontrollable sequence σ from the state sequence which generates the best fit to the observed sequence from:

$$P(\sigma|\eta_{\text{bestfit}}) = max_{k\in Q}[v_k(l)p_{k,Q_f}] \tag{4.3.6}$$

This process uses only $O(l|Q|)$ storage locations to compute $v_k(i)$ in $O(l|Q|^2)$ steps, making it computationally attractive.

If the protocol is controllable under parallel composition with a state machine constructed from the best fit sequence, the network is robust to corruption from the most (historically) likely attack style. By running the network model in parallel with SARAH and performing the controllability analysis on the combined system, we can determine the probabilistic controllability of the network given our attack model. Let $\mathfrak{C}(S||N)$ be the indicator function that returns 1 if the combination is controllable, and 0 if not.

Definition 4.3.2. Let $\mathcal{N}$ be a set of protocols. A protocol N in $\mathcal{N}$ is optimally robust against a SARAH S if it maximizes the sum:

$$\sum_s \mathfrak{C}(S||N)P(\sigma|\eta_s)P(\eta_s)$$

4.3.1 Correcting Protocols

If a protocol has been verified against common network attacks and has been found to be vulnerable to a particular attack, then it is necessary to correct the problems in the protocol. There is a large literature on correction of discrete event control specifications, [KGM91, RW87b] contain a good introduction. Most of the approaches used are automated and correct an uncontrollable specification by removing control paths within the specification that lead to system deadlock for a given plant model. We have argued in [DGP03] that this is **not sufficient** for correcting all types of discrete event control specifications because it can lead to the removal of key control paths in the specification. In [DGP03] we provide an algorithm that determines control pathologies for finite state machines. Using this algorithm, a human can debug specifications systematically to ensure that the resulting specification is controllable. These pathologies are general across deterministic context free languages and apply in the more general case we've outlined in this thesis. By analyzing the Σ_u reverse paths determined by Algorithm $\mathfrak{A}$, we can identify causes of uncontrollability and correct them in the protocol by hand.

Using definition 4.3.2 and the methods outline in [DGP03] or Chapter 5 we can systematically construct a protocol that is optimally robust using a spiral design method (see Figure 4.4).

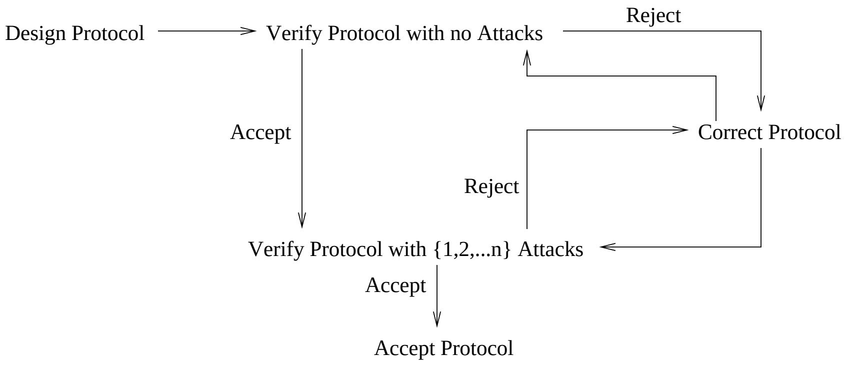

Design Protocol
Verify Protocol with no Attacks
Reject
Accept
Correct Protocol
Reject
Verify Protocol with {1,2,...n} Attacks
Accept
Accept Protocol

Figure 4.4. Spiral design method for wireless network protocols

This procedure is illustrated in [DGP03]. It can be simplified using automated design and verification tools that can help the user check and correct incorrect specifications.

4.4 Future Directions

In this chapter, we applied discrete event control methodology to a generic protocol with a single environmental perturbation. An obvious next step is to obtain a real-world network environment characterization and simulate a detailed protocol in depth.

We have identified a likely avenue for additional theoretical work: Formulate a parametric optimization for protocols modeled as discrete event systems. This will be useful, for instance, in optimizing timeout periods for retry logic. Given statistical data for the distribution of uncontrollable events, we would like to be able to optimize the protocol-defining automata and other design parameters for maximal reliability. It would be highly desirable to extend this method to provide a rigorous algorithm for determining a measure of security of a protocol.

4.5 Conclusion

We have demonstrated how discrete event control theory can be used in the design and verification of wireless network protocols, even in the case where certain uncontrollable events cannot be identified until the protocol is deployed. For example, this is the case with unanticipated strategies for attacking the network.

We have also provided a method for determining whether or not a given protocol specified by automata models can cope with all known uncontrollable hazards, and a method of estimating the confidence level for controllability of a protocol in the case where this fails. This approach can be used for designing new protocols for wireless networks that are robust to attack.

Chapter 5

An Algorithm to Compute the Supremal Controllable Sublanguage for a Class of Pushdown Systems

5.1 Introduction

In this chapter, we study positive decidability results on supremal controllable sublanguages of pushdown machines. The majority of the results presented in this chapter are available in [Gri07], which has been accepted for publication and is available as a pre-print. Section 5.4 is not included in [Gri07] and is used in Chapters 6 - 8. The main goal of this chapter is to prove the following theorem:

Theorem 5.1.1 (Griffin 2007 [Gri07])**.** *Let L be a prefix-closed plant language and let K be a prefix closed sublanguage of L that is accepted by a DPDA. Then* $\sup_{\mathrm{C}}(K)$ *is accepted by a DPDA.*

To prove this theorem, we develop an algorithm that extends the supremal controllable sublanguage finding algorithm of [KGM91]. We show how to use the results proved in Chapter 3 to derive this algorithm.

As an ancillary result, we also show when K is accepted by an arbitrary context free sublanguage of a regular plant language L, then it is undecidable whether

$\sup_C(K) = \emptyset$. This demonstrates conclusively that the non-deterministic context free languages are extremely difficult to use in discrete event control.

5.2 An Algorithm to Compute A Controllable Sublanguage

In this section we derive an algorithm to compute $\sup_C(K) \subseteq K$, when K is uncontrollable with respect to L and both K and L are prefix closed and K is accepted by a DPDA and L is accepted by a finite state machine. To accomplish this, we use a few lemmas that were proved in [Gri06]. We then apply these lemmas to the construction of our algorithm.

5.2.1 Previous Results

The first lemma relates a certain quotient of the plant and the complement of the control language to the controllability criterion.

Lemma 5.2.1. *Let S be a specification and let L be a plant language. If $K = S \cap L$, then the following are equivalent: (i) $K = S \cap L$ is controllable with respect to L. (ii) $\left(\left(\overline{K}\right)^c \cap \overline{L}\right)/\Sigma_u^* = \left(\overline{K}\right)^c \cap \overline{L}$. (See Lemma 3.2.1.)*

The second lemma asserts that there is an algorithm to remove any unreachable state from a DPDA. It also asserts that we can remove any useless transitions from a DPDA. A useless transition is one that, by nature of the machine's structure, will never fire during DPDA reading.

Lemma 5.2.2. *There is an algorithm to remove all unreachable states and useless transitions from a DPDA M. (See Lemma 3.3.1).*

The third lemma details how to use the two preceding lemmas to check for controllability in deterministic pushdown systems. Recall if M is a DPDA, then a Σ_u reverse path in M between states q_m and q_1 is a sequence of states $q_1, \dots, q_m$ and a string $\upsilon = u_1 \dots u_n \in \Sigma_u^*$ such that for $\zeta_1 \in \Gamma^*$, and $\sigma \in \Sigma^*$ (i) $(q_0, \sigma\upsilon, Z_0) \vdash^* (q_1, \upsilon, \zeta_1)$ and (ii) $(q_1, \upsilon, \zeta_1) \vdash^* (q_m, \epsilon, \zeta_f)$ for some $\zeta_f \in \Gamma^*$. Note we *are* considering the possibility of ϵ-transitions in this path.

Lemma 5.2.3. *Let L be a language generated by a DPDA and let*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, Q_f)$$

be a DPDA accepting L by final state with no useless transitions or unreachable states. Then $L/\Sigma_u^* = L$ if and only if no Σ_u reverse path leaves Q_f. (See Theorem 3.3.2.)

5.2.2 Algorithm Derivation

Before getting into the details of the algorithm, we first summarize the approach. Let $K \subseteq L$ be the desired (uncontrollable) sublanguage. Our algorithm is motivated by the algorithm in [KGM91]. Let both K and L be prefix closed. Let M_K be a DPDA that accepts K by final state and let a finite state machine A accept L. Since K and L are prefix closed, suppose every state of M_K is final and every state of A is final. Finally, assume that there are no useless transitions or unreachable states in M. If there are, we know from Lemma 3.3.1 that we can remove them.

Algorithm $\mathfrak{B}$ works as follows: we first form a machine that we can check for Σ_u reverse paths, just as in Algorithm $\mathfrak{A}$. We denote this machine by N'. An interesting fact about N' is that it has the structure of M_K embedded inside it. We use this fact extensively. We identify all the states that have Σ_u reverse paths leading from them. The idea now is to remove any controllable transition that leads to these states. Suppose we can isolate these states (i.e., cut them out of the DPDA), then the uncontrollability will disappear (because we no longer have any Σ_u reverse paths that are reachable. If we cannot isolate these states, then the supremal controllable sublanguage is empty. We use the fact that the transition structure of M_K is embedded inside N' to remove these transitions from M_K as we remove them from N'. The result is a new structure M_K'' that generates $\sup_{\mathrm{C}}(K)$. This algorithm is a variation on Kumar's algorithm for obtaining a finite state machine that generates the supremal controllable sublanguage when $K, L \in \mathbf{REG}$ and K and L are both prefix closed [KGM91]. In a sense, this algorithm extends this classical result to the case when $K \in \mathbf{DCFL}$.

We will prove the following about Algorithm $\mathfrak{B}$.

Theorem 5.2.4. *$\mathfrak{L}(M_K'') = \sup_{\mathrm{C}}(K)$.*

Algorithm Description 5.2.1 – Algorithm $\mathfrak{B}$

1. Form a new machine $\overline{M}_K$ that scans all inputs but still accepts K.
2. Use $\overline{M}_K$ to construct a machine N accepting $K^c \cap L$. The structure of this machine retains information about K and L in its states.
3. Trim N of useless states and transitions obtaining N'.
4. Form a predicting machine $\pi(N', U)$ where U is the one state finite state machine accepting Σ_u^*. During an execution of $\pi(N', U)$, the top stack symbol tells for each state q of N' whether there is some input string in Σ_u^* that causes N' to accept if started in state q. That is, whether or not there is a Σ_u reverse path terminating at this state.
5. Find all controllable transitions in $\pi(N', U)$ that are between original states of M_K and that create a condition where the top stack symbol indicates a Σ_u reverse path connects the next state to a final state. Remove these transitions from M_K to form M'_K.
6. If $\mathcal{L}_{\mathcal{M}}(M'_K)$ is still uncontrollable with respect to L, then $\sup_{\mathrm{C}}(K) = \emptyset$. Otherwise:
7. Trim M'_K of useless states and transitions obtaining M''_K.

Before proving Theorem 5.2.4, we first show that we can keep track of the original states and transitions of M throughout the steps of Algorithm $\mathfrak{B}$. All of the construction steps are taken directly from Hopcroft and Ullman, Chapters 6 and 10. The novelty herein lies in the use of these steps to produce a supremal controllable sublanguage.

5.2.2.1 Step 1

We show that we can keep track of the transitions of M_K when creating $\overline{M}_K$. Let $M_K = \langle Q, \Sigma, \Gamma, \delta, q_0, Q \rangle$ The procedure for forming $\overline{M}_K$ is described in [HU79] and Chapter 2. Let

$$\overline{M}_K = \langle Q \cup \{q'_0, d, f\}, \Sigma, \Gamma \cup \{X_0\}, \delta', q'_0, X_0, Q \cup \{f\} \rangle, \tag{5.2.1}$$

where:

1. $\delta'(q'_0, \epsilon, X_0) = (q_0, Z_0 X_0)$, where X_0 marks the bottom of the stack.

2. If for some $q \in Q$, $a \in \Sigma$ and $Z \in \Gamma$, $\delta(q, a, Z)$ and $\delta(q, \epsilon, Z)$ are both empty, then $\delta'(q, a, Z) = (d, Z)$. Further, for all $q \in Q$ and $a \in \Sigma$, $\delta(q, a, X_0) =$

(d, X_0).

3. $\delta'(d, a, Z) = (d, Z)$ for all $a \in \Sigma$ and $Z \in \Gamma \cup \{X_0\}$.

4. Let $q \in Q$ and $Z \in \Gamma$. If for all i there exists q_i and γ_i such that $(q, \epsilon, Z) \vdash_i (q_i, \epsilon, \gamma_i)$, then $\delta'(q, \epsilon, Z) = (d, Z)$ if no q_i is final and $\delta(q, \epsilon, Z) = (f, Z)$ otherwise. (This rule allows us to detect structures that allow an infinite number of ϵ-transitions. [HU79], pp. 237-238 shows this can be made effective.)

5. $\delta(f, \epsilon, Z) = (d, Z)$ for all $Z \in \Gamma \cup \{X_0\}$.

6. For any other $q \in Q$, $a \in \Sigma \cup \{\epsilon\}$ and $Z \in \Gamma$, if $\delta'(q, a, Z)$ is not defined by rule 2 or 4, then $\delta'(q, a, Z) = \delta(q, a, Z)$.

It is easy to see that the only rules using the definition of δ in M_K are 4 and 6. Clearly, we can keep track of the transitions from M_K in Rule 6. The ϵ-transitions that are changed from M_K to $\overline{M}_K$ are not particularly important to us in the use of Algorithm $\mathfrak{B}$, however we can book-keep any transition $\delta(q, \epsilon, Z) = (d, Z)$ or $\delta'(q, \epsilon, Z) = (f, Z)$ that arose from the use of Rule 4. Thus we have shown that we can keep track of transitions in $\overline{M}_K$ that are also in M. Note now, every state of M' is final except d and q_0'.

5.2.2.2 Step 2

We first show that we can keep track of the transitions of M_K when computing a machine $\overline{M}_K^c$ accepting K^c. By the very construction of $\overline{M}_K^c$ from Hopcroft and Ullman, we know that $\overline{M}_K^c$ must simulate $\overline{M}_K$. Let

$$\overline{M}_K^c = \langle Q'', \Sigma, \Gamma \cup \{X_0\}, \delta'', q_0'', X_0, F \rangle, \tag{5.2.2}$$

where $Q'' = \{(q, k) | q \in Q \cup \{q_0', f, d\}, k = 1, 2, 3\}$. Let $F = \{(q, 3) | q \in Q \cup \{q_0', f, d\}\}$ and let $q_0'' = (q_0', 2)$, because as we remarked, q_0' is not final in $\overline{M}_K$. For notational ease, let $Q' = Q \cup \{q_0', f, d\}$. We define δ'' as follows

1. If $\delta'(q, \epsilon, Z) = (p, \gamma)$, then for $k = 1$ or $k = 2$,

$$\delta''((q, k), \epsilon, Z) = ((p, k'), \gamma)$$

where $k' = 1$ if $k = 1$ or $p \in Q \cup \{f\}$ and $k' = 2$ if $k = 2$ and $p \notin Q \cup \{f\}$.

2. If $\delta'(q, a, Z) = (p, \gamma)$ for $a \in \Sigma$, then $\delta''((q, 2), \epsilon, Z) = ((q, 3), Z)$ and

$$\delta''((q, 1), a, Z) = \delta''((q, 3), a, Z) = ((p, k), \gamma)$$

where $k = 1$ if $p \in Q \cup \{f\}$ (the original final states of $\overline{M}_K$) and $k = 2$ if $p \notin Q \cup \{f\}$.

Observe that $k = 2$ only when a transition to state d is made because d is the only non-final state aside from q_0' and there are no transitions to q_0'. Hence, all transitions generated in Rule 6 of Step 1 will become transitions connecting states of the form $(q, 1)$ to other states of the form $(p, 1)$. Original transitions will either connect $(q, 1)$ with $(f, 1)$, in which case they are just like normal transitions in M_K. Added transitions will connect states of the form $(q, 1)$ to $(d, 2)$. Either way, we can differentiate these transitions as we construct $\overline{M}_K^c$.

We now show that we can differentiate transitions in the original machine and a machine accepting $L \cap K^c$. Since our notation is becoming cumbersome, we will show this fact for an arbitrary deterministic pushdown machine M and a deterministic finite state machine A. Let $M = \langle Q_M, \Sigma, \Gamma, \delta_M, q_0, Z_0, F_M\rangle$ and let $A = \langle Q_A, \Sigma, \delta_A, p_0, F_A\rangle$. Define:

$$M' = \langle Q_A \times Q_M, \Sigma, \Gamma, \delta, (p_0, q_0), Z_0, F_A \times F_M\rangle$$

Let $\delta((p, q), a, X)$ contain $((p', q'), \gamma)$ if and only if $\delta_A(p, a) = p'$ and $\delta_M(q, a, X)$ contains (q', γ). If $a = \epsilon$, set $\delta(p, a) = p$. Clearly, if a transition (q, a, Z, q', γ) were in M, and a transition $((p, q), a, Z, (p', q'), \gamma) \in \delta$, then clearly it is copied to M'. By determinism of A and M, we know that M' will also be deterministic. Hence, we have shown we can keep track of the transitions of our original machine M accepting K after Step 2 of Algorithm $\mathfrak{B}$.

5.2.2.3 Step 4

Trivially, since Step 3 of Algorithm $\mathfrak{B}$ only removes states and transitions, we will still be able to differentiate the transitions originally in M_K after Step 3. In fact, if we can prove that the transition structure of M_K is still present after Step 4, then we are done. We know the algorithm will be executable. It will remain to prove that it provides us with a useful sublanguage; i.e., to prove Theorem 5.2.4.

Before proceeding, it is necessary for the machine N' to be in normal form for a deterministic pushdown machine. Hopcroft and Ullman [HU79] define normal form as a condition where the only stack operations are to (i) erase the top stack symbol or (ii) to push a single new stack symbol onto the stack. It is easy to construct such a machine from an existing machine and further, we can assume that our original M is a machine of this form. The construction procedure in Steps 1-3 will not change this fact.

Let $N' = \langle Q_M, \Sigma, \Gamma, \delta_M, q_0, Z_0, F_M \rangle$. We know the states of N' are of the form $(p, (q, k))$, where $q \in Q \cup \{q_0', f, d\}$ $k = 1, 2, 3$ and p is a state of the deterministic finite state machine accepting L. We know that the accepting states look like $(p, (d, 3))$, since in the machine $\overline{M}_K$, d was the only non-final state with transitions going in. We can similarly describe Γ, Z_0 and δ_M. Note, Σ has been constant throughout.

Let $U = \langle \{p\}, \Sigma_u, \delta_U, p, \{p\} \rangle$ be the finite state machine accepting Σ_u^* with one state. The predicting machine $\pi(N', U) = (Q_M, \Sigma, \Gamma \times \Delta, \delta, X_0, F_M)$ where $\Delta = Q_M \times \{p\}$. If $(r, x, [Z, \mu]\gamma)$ is an ID of $\pi(M_1', U)$, then μ contains the set of states $(q, p) \in F_M$ such that there is a string $u \in \Sigma_u^*$ such that $(r, u, Z\gamma_1) \vdash^* (q, \epsilon, \alpha)$, where γ_1 is the string of first components of γ and α is some string in Γ^*.

Following the convention of Hopcroft and Ullman, let $N'(q, Z)$ be a deterministic pushdown machine with the structure of N' but started in state q with stack symbol Z. Let $\mathcal{L}_{\mathcal{M}}(N'(q, Z))$ be as usual. Let $N_r(N'(q, Z))$ be the set of strings that cause $N'(q, Z)$ to erase its stack and enter state r. We now define δ for $\pi(N', U)$ as follows:

1. If $\delta_M(r, a, Z) = (s, \epsilon)$, then $\delta(r, a, (Z, \mu)) = (s, \epsilon)$.

2. If $\delta_M(r, a, Z) = (s, Z)$, then $\delta(r, a, (Z, \mu)) = (s, (Z, \mu))$.

3. If $\delta_M(r, a, Z) = (s, YZ)$, then $\delta(r, a, (Z, \mu)) = (s, (Y, \nu)(Z, \mu))$, where ν is the set of pairs (q, p) such that either $\mathcal{L}_{\mathcal{M}}(M_1'(q, Y)) \cap \Sigma_u^*$ is non-empty or there is some t in Q_M such that $N_t(N'(q, Y)) \cap \Sigma_u^*$ is non-empty and (t, p) is in μ.

Let $X_0 = (Z_0, \mu_0)$ where $\mu_0 = \{(q, p) |\, \mathcal{L}_{\mathcal{M}}(M(q, Z_0)) \cap \Sigma_u^* \neq \emptyset\}$.

The construction of $\pi(N', U)$ shows that its structure is identical to the structure of N', except that the stack symbols have been modified to carry information about U. Hence we can identify the original transitions of M_K that were identified in Step 1.

5.2.2.4 Algorithm Analysis

Suppose that there is a transition in $\pi(N', U)$ that originated in M_K of the form $\tau = (q, c, (Z, \mu), q', (Y, \nu))$, with q not a final state, q' not a final state and ν, whose first symbol contains an element of the form (q_f, p), where $q_f = (s, (d, 3))$ and s is a final state of A, the machine accepting L; by assumption, every state of A is final, since L is prefix closed, d is the dead state introduced in Step 1, p is the only state of U and $c \in \Sigma_c$. Then there is a string in Σ_u^* leading from state q' to a final state of N'. That is to say, a Σ_u reverse path leading out of the final states. We know that we have trimmed N' of useless transitions and unreachable states. Hence, we know that this transition τ can fire in N'. We can therefore conclude that if we disable this one transition, we will eliminate a single instance of uncontrollability. To see this, consider repeating Steps 1 through 4 again on the modified M_K. The transition we just removed, would be added to $\overline{M}_K$, only now it would go to state d in Step 1 instead of state q'. It follows immediately that we have "closed off" a Σ_u reverse path from entering the final states. After repeating this process for all transitions of this type, one of two conditions will be true:

1. There is a transition $(q, u, (Z, \mu), q', (Y, \nu))$ with $u \in \Sigma_u$ satisfying all the same properties as τ. In this case, since we have eliminated all controllable transitions in M_K leading to uncontrollability, we can conclude that some uncontrollable transition in M_K leads to uncontrollability. Thus $\sup_{\mathrm{C}}(K) = \emptyset$.

2. There are no more transitions satisfying the properties of τ with either a controllable event or uncontrollable event.

In either case, we know that every state of M_K'' is final and hence $\mathcal{L}_{\mathcal{M}}(M_K'')$ is prefix closed. We further can see that $\mathcal{L}_{\mathcal{M}}(M_K'') \subseteq K$. Finally, in Case 2, we have shown that $\mathcal{L}_{\mathcal{M}}(M_K'')$ is controllable with respect to L, since by construction a machine accepting $L \cap \mathfrak{L}_{\mathcal{M}}(M''_K)$ will have its final states closed under Σ_u reverse paths. Using this argument and Lemma 5.2.3, we have proved the following.

Theorem 5.2.5. *$\mathfrak{L}_{\mathcal{M}}(M''_K) \subseteq K$ and $\mathfrak{L}_{\mathcal{M}}(M''_K)$ is controllable with respect to L.*

It now suffices to prove that $\mathfrak{L}(M''_K)$ is supremal.

Proof of Theorem 5.2.4. Suppose $\mathfrak{L}(M''_K)$ is controllable (Case 2). To show that $\mathfrak{L}(M''_K)$ is supremal, proceed by contradiction. Suppose $s \in \mathfrak{L}(M''_K)$ and $s \in K$ and for some $c \in \Sigma_c$, let $sc \in K$ and $sc\Sigma_u^* \cap L \subseteq K$. Since we only removed controllable transitions, we know such a situation must arise if $\sup_{\mathrm{C}}(K) \neq \mathfrak{L}(M''_K)$. By determinism, the string s leads M''_K and M_K to the same (corresponding) state and these states are unique. We must have deleted a transition of the form (q, c, A, q', γ) from M_K to get M''_K. We did this because there is some string $u \in \Sigma_u^*$ such that from a corresponding state in $\pi(N', U)$ there is a path following the elements of u causing $scu \in K^c \cap L$. By determinism of $\pi(N', U)$, we know that there could not have been another path leading to a non-final state in $\pi(N', U)$. It follows at once that $\sup_{\mathrm{C}}(K) = \mathfrak{L}(M''_K)$. □

Theorem 5.2.6. *When K and L are prefix closed and K is accepted by a deterministic pushdown machine and L is a regular language, then* $\sup_{\mathrm{C}}(K)$ *is also accepted by a deterministic pushdown machine.*

Proof. This follows immediately from the results above. □

Theorem 5.2.7. *Algorithm $\mathfrak{B}$ is P-complete.*

Proof. Algorithm $\mathfrak{B}$ contains the steps of Algorithm $\mathfrak{A}$ as a sub-algorithm. Hence it is at least P-complete. The construction of a predicting machine given in Chapter 10 of [HU79] shows that this problem is also contained in P. Hence, this algorithm is P-complete. □

5.3 Undecidability Results

We now turn our attention to proving the following proposition.

Theorem 5.3.1. *Let G be a finite state machine with language $\mathcal{L}(G)$ and marked language $\mathcal{L}_{\mathcal{M}}(G)$. If $K \subseteq \mathcal{L}_{\mathcal{M}}(G)$, $K \neq \emptyset$ is accepted by an arbitrary PDA M_K, then it is undecidable whether* $\sup_{\mathrm{C}}(K) = \emptyset$ *when $|\Sigma_u| \geq 2$.*

To prove this proposition we will use four lemmas that, when taken together, establish the result. The first lemma was established by Hunt and Rosenkrantz [IR74].

Lemma 5.3.2. *Let P be any predicate on the languages accepted by PDA over the alphabet $\Sigma = \{0,1\}$, such that: (i) $P(\Sigma^*)$ holds and (ii) Either $P_{\mathrm{Left}} = \{L' \mid L' = \{x\} \backslash L,\ x \in \Sigma^*,\ P(L) \text{ holds}\}$ or $P_{\mathrm{Right}} = \{L' \mid L' = L/\{x\},\ x \in \Sigma^*,\ P(L) \text{ holds}\}$ is a proper subset of the languages accepted by PDA over Σ, when L is accepted by a PDA over the alphabet Σ. Then for arbitrary PDA G, the predicate $P(\mathcal{L}_{\mathcal{M}}(G))$ is undecidable.*

We use this result to prove the second lemma.

Lemma 5.3.3. *Let $\Sigma = \{0,1\}$ and let $P(K)$ be the predicate on Σ saying $\Sigma^* \subseteq \overline{K}$, for some language K accepted by PDA over Σ. Then P is undecidable.*

Proof. If $K = \Sigma^*$, then $\Sigma^* \subseteq \overline{K} = \Sigma^*$ and $P(\Sigma^*)$ is true. Now let K be a language for which P holds. We will show that if $\{x\}\backslash K$ is non-empty, then $\{x\}\backslash K$ is infinite. Suppose that $y \in \{x\}\backslash K$. Then $xy \in K$ and x is a prefix of K. Since $x \in \overline{K}$ and $\Sigma^* \subseteq K$, it follows that for any arbitrary $z \in \Sigma^*$, $xzy \in \overline{K}$. Hence, there exists some $w \in \Sigma^*$ such that $xzyw \in K$. Thus we have shown that if $|z| \geq 1$, there is a longer string $zyw \in \{x\}\backslash K$ if $y \in \{x\}\backslash K$. It follows at once that $\{x\}\backslash K$ is infinite in size.

From this argument, P_{Left} cannot contain any finite context free languages and hence cannot contain all context free languages. Applying the conclusion of Lemma 5.3.2, we see that the predicate P is undecidable for arbitrary PDA G. □

Corollary 5.3.4. *If $|A| \geq 2$ is an alphabet and G is an arbitrary PDA over alphabet Σ with $A \subseteq \Sigma$, then it is undecidable whether $A^* \subseteq \overline{\mathcal{L}_{\mathcal{M}}(G)}$.*

Proof. We proceed by double induction on the size of A and the size of Σ. Clearly, we have established base case $|A| = 2$, $|\Sigma| = 2$. Suppose that the statement holds for $|\Sigma| \leq n$ and $|A| = 2$, then we will show that it holds for $|\Sigma| = n + 1$.

By contradiction, suppose not, there is an Algorithm $\mathfrak{Z}$ to determine whether $A^* \subseteq \overline{\mathcal{L}_{\mathcal{M}}(G)}$ when G is an arbitrary PDA over alphabet Σ and $|\Sigma| = n + 1$. Take a language L generated by some PDA M over alphabet $\Sigma' \subseteq \Sigma$.

If $A \not\subseteq \Sigma'$, then $A^* \not\subseteq L$. If $A \subseteq \Sigma'$, then construct a new PDA M' using all the symbols of Σ such that M' first executes all the moves of M and then push some arbitrary finite string of symbols from $\Sigma \setminus \Sigma'$ on to the end of all elements of L. It is easy to see, $A^* \subseteq \overline{L}$ if and only if $A^* \subseteq \overline{\mathcal{L}_{\mathcal{M}}(M')}$. Hence use Algorithm $\mathfrak{Z}$ to test whether $A^* \subseteq \overline{\mathcal{L}_{\mathcal{M}}(M')}$. This contradicts our induction hypothesis. Hence, Algorithm $\mathfrak{Z}$ does not exist.

To pass to the general case, suppose that the statement is true for $|\Sigma| \leq n$ and $|A| \leq m \leq n$. It is clear that the contradiction argument we used above applies just as well in this case. And hence, the corollary follows immediately by the double induction. □

Remark 5.3.5. There is an algorithm to decide whether $A^* \subseteq \overline{\mathcal{L}_{\mathcal{M}}(G)}$ in the case when $|A| = 1$. To test this case, it suffices to analyze the parse tree of the corresponding context free grammar that generates $\mathcal{L}_{\mathcal{M}}(G)$ to determine whether there is a branch containing A^*. This can be done with a recursive algorithm similar to the Coverability Algorithm used in Petri Net analysis (see [CL99], Chapter 4, Section 4.2). It is easy to see that it is trivially decidable whether $A^* \subseteq \overline{\mathcal{L}_{\mathcal{M}}(G)}$ when $|A| = 0$.

Our third lemma establishes a lower bound on the supremal controllable sublanguage of a prefix closed language. The result should be clear without proof, though the proof is trivial.

Lemma 5.3.6. *Let $K \neq \emptyset$ be accepted by an arbitrary pushdown automaton G. Let L be a plant language. Consider* $\sup_{\mathrm{C}}(\overline{K})$ *with respect to $\overline{L}$. Then,* $\sup_{\mathrm{C}}(\overline{K}) \neq \emptyset$ *iff* $\overline{L} \cap \Sigma_u^* \subseteq \sup_{\mathrm{C}}(\overline{K})$.

Proof. Suppose that $\sup_{\mathrm{C}}(\overline{K}) \neq \emptyset$. We know from Cassandras and LaFortune that since $\overline{K}$ is prefix closed so to will $\sup_{\mathrm{C}}(\overline{K})$ be prefix closed. From this it follows that $\epsilon \in \sup_{\mathrm{C}}(\overline{K})$. Since $\epsilon \in \sup_{\mathrm{C}}(\overline{K})$, it follows from the definition of controllability that $\epsilon\Sigma_u^* \cap \overline{L} \subseteq \sup_{\mathrm{C}}(\overline{K})$ or $\overline{L} \cap \Sigma_u^* \subseteq \sup_{\mathrm{C}}(\overline{K})$. Conversely, suppose that $\overline{L} \cap \Sigma_u^* \subseteq \sup_{\mathrm{C}}(\overline{K})$. Then $\sup_{\mathrm{C}}(\overline{K})$ is non-empty because $\epsilon \in \overline{L} \cap \Sigma_u^*$. □

Corollary 5.3.7. *Let $K \neq \emptyset$ be accepted by an arbitrary pushdown automaton G. Let L be a plant language.* $\sup_{\mathrm{C}}(\overline{K}) \neq \emptyset$ *iff* $\overline{L} \cap \Sigma_u^* \subseteq \overline{K}$.

Proof. It is trivial that if $\sup_{\mathrm{C}}(\overline{K}) \neq \emptyset$, then $\overline{L} \cap \Sigma_u^* \subseteq \overline{K}$ since $\sup_{\mathrm{C}}(\overline{K}) \subseteq \overline{K}$. To prove the converse, suppose that $\overline{L} \cap \Sigma_u^* \subseteq \overline{K}$. Then $\overline{L} \cap \Sigma_u^*$ is a controllable subset of $\overline{K}$. To see this, let $w \in \overline{L} \cap \Sigma_u^*$. Choose $u \in \Sigma_u^*$. Then $w \in \Sigma_u^*$ and hence $wu \in \Sigma_u^*$. If $wu \in \overline{L}$, then $wu \in \overline{L} \cap \Sigma_u^*$. Hence, $[\overline{L} \cap \Sigma_u^*]\Sigma_u^* \cap \overline{L} \subseteq \overline{L} \cap \Sigma_u^*$. Hence, $\sup_{\mathrm{C}}(\overline{K}) \neq \emptyset$. □

In our final lemma, we show that it is undecidable whether $\sup_{\mathrm{C}}(\overline{K}) = \emptyset$ with respect to $\overline{L}$ when K is accepted by an arbitrary pushdown automaton G and L is an arbitrary regular plant language.

Lemma 5.3.8. *Let K be accepted by an arbitrary pushdown automaton G. Let L be a regular plant language. Finally, suppose that $|\Sigma_u| \geq 2$. Then it is undecidable whether* $\sup_{\mathrm{C}}(\overline{K}) = \emptyset$ *with respect to L.*

Proof. Suppose that is was decidable whether $\sup_{\mathrm{C}}(\overline{K}) = \emptyset$ and consider the case when the plant language $L = \Sigma^*$. By Corollary 5.3.7 we know that $\sup_{\mathrm{C}}(\overline{K}) \neq \emptyset$ if and only if $\overline{L} \cap \Sigma_u^* \subseteq \overline{K}$. For the case we are considering, $\sup_{\mathrm{C}}(\overline{K}) \neq \emptyset$ if and only if $\Sigma_u^* \subseteq \overline{K}$. Applying Lemma 5.3.3, we see for arbitrary K, it is undecidable whether $\Sigma_u^* \subseteq \overline{K}$ for $|\Sigma_u| \geq 2$. □

Finally, we can prove Theorem 5.3.1.

Proof of Theorem 5.3.1. By way of contradiction, suppose that it is decidable whether $\sup_{\mathrm{C}}(K) = \emptyset$ with respect to arbitrary regular language L. We know from [HU79] that $\overline{K}$ is also accepted by a PDA and that $\overline{L}$ is also regular. Hence, it follows that it is decidable whether $\sup_{\mathrm{C}}(\overline{K}) = \emptyset$ with respect to $\overline{L}$. But this contradicts Lemma 5.3.8. Thus, it is impossible to decide whether $\sup_{\mathrm{C}}(K) = \emptyset$. □

5.4 Parametric Discrete Event Control

In this section we propose a problem like that of the problem of parametric control for discrete event control systems. By parametric control, we mean that we must choose some structural parameters that will produce a controller with a chosen form; for example, given a PID control structure, a solution to the parametric control problem is a set of coefficients that provides a specific PID form [Cas02].

In this case, we will give the structural form: a pushdown automaton with a desired transitional structure, and then attempt to identify a specific instance that satisfies desired properties.

We provide Algorithm $\mathfrak{D}$ to solve the parametric discrete event control problem. The algorithm begins by assuming that a plant model G is provided. Based on the results we have proved so far about decidability in discrete event control (see Chapter 3), we assume that G is a finite state machine. Suppose that $L = \mathfrak{L}_{\mathcal{M}}(G)$. We then assume that a specification machine M is provided. Naturally, M may be either a DPDA or a FSM. Machine M accepts some language S (i.e., $S = \mathfrak{L}_{\mathcal{M}}(M)$) that describes the discrete event strings that are desirable to the controller designer. Machine M may be constructed without regard to the true plant model G–it simply describes the behaviors that are preferred by the designer. We compute a machine $M_K = M \cap G$ that accepts $K = S \cap L$. This is the language that results when the desired behavior (S) is combined with the possible behavior (L). The language K is to be the controllable sublanguage of L. If K is not controllable with respect to L, then we use Algorithm $\mathfrak{B}$ on the DPDA M_K with plant model G. If Algorithm $\mathfrak{B}$ returns $\emptyset$, then there is no substructure of M_K that yields a controllable sublanguage for L. On the other hand, if Algorithm $\mathfrak{B}$ does not return $\emptyset$, then the new DPDA produced, call it M_K', will generate a language $K' = \mathfrak{L}_{\mathcal{M}}(M_K')$. This language will be controllable with respect to L. Further, M_K' has a structure similar to M_K except that some of the transitions that were present in M_K have been removed (by Algorithm $\mathfrak{B}$). The result is a solution to the parametric control problem. We've found a substructure of M_K (namely M_K') such that K' is controllable with respect to L.

The problem in Step 1 is a design problem associated with defining what is acceptable to the controller designer. Step 2 is readily solved by algorithms for intersecting a finite state machine and a pushdown machine. Step 3 can be checked using the algorithms from Chapter 3 or [Gri06]. Finally, Steps 5 and 6 are end steps. The key to executing this algorithm is to determine a method for computing Step 4.

The remainder of this section is given over to showing that Algorithm $\mathfrak{B}$

Algorithm Description 5.4.1 – Algorithm $\mathfrak{D}$

1. Produce a pushdown machine M that accepts a specification S. This specification may be generated without reference to a plant model, it simply details desired behavior of any plant with a given alphabet Σ.
2. Given a plant model G compute machine $M_K = M \cap G$ accepting $K = \mathfrak{L}_{\mathfrak{M}}(L) \cap \mathfrak{L}_{\mathfrak{M}}(M)$. Machine M_K is the *parametric controller*.
3. If K is controllable, GOTO 5. Otherwise, K is not controllable, GOTO 4.
4. Apply Algorithm $\mathfrak{B}$ to M_K and G to remove transitions from M_K that cause uncontrollability. Denote this new machine M'_K. If $\mathfrak{L}(M'_K) = \emptyset$ GOTO 6. Otherwise GOTO 5.
5. The machine M_K is the solution to the parametric control problem.
6. There is no controller meeting the specification S and developed from the structure prescribed by M. The problem is ill-posed.

for computing supremal controllable sublanguages of deterministic pushdown machines can be used to identify a new controller M'_K that is a substructure of M_K and that accepts a subset of K that is controllable with respect to L; i.e., to show that Step 4 is algorithmically possible.

5.4.1 Algorithm

In Algorithm $\mathfrak{B}$ we created a transition deletion procedure for when K was prefix closed and L was prefix closed. This transition deletion procedure is inspired by the work in [KGM91]; in fact it is the same basic algorithm with modifications made for the pushdown structure. In Theorem 5.2.5, we proved that the language $K' = \mathfrak{L}_{\mathfrak{M}}(M'_K)$ is controllable with respect to L. This proof *did not require* that L be prefix closed. On the other hand the proof for supremacy **did** require the prefix closure. Hence we have the following fact:

Theorem 5.4.1. *Let S be a specification produced by M and G be a plant model accepting language L and suppose $K = S \cap L$ is not controllable. Suppose Algorithm $\mathfrak{B}$ is run with $M_K = G \cap M$ and G and produces a non-empty result. Then a new specification $S' = \mathfrak{L}_{\mathfrak{M}}(M'_K)$ produced by Algorithm $\mathfrak{B}$ is a controllable sublanguage of S and L.*

Proof. Controllability follows from Theorem 5.2.5. The fact that S' is a subspecification is an immediate consequence of the construction of the algorithm. □

Remark 5.4.2. Naturally, if $\mathcal{L}(M_K)$ and $\mathcal{L}_{\mathcal{M}}(G)$ are prefix closed, this algorithm also has the pleasing property that it is the supremal controllable sublanguage. If L is *not* prefix closed, then K' may not be the supremal controllable sublanguage, but it will be a language derived from the original parametric controller M_K requiring the least number of modifications to force controllability. (This fact follows implicitly from Kumar's optimality proof [KGM91].

Remark 5.4.3. Suppose that we apply Algorithm $\mathfrak{D}$ to machines M_K and G. Recall, we are operating on a machine that accepts the language $\overline{\mathcal{L}_{\mathcal{M}}(M_K)}^c \cap \overline{\mathcal{L}_{\mathcal{M}}(G)}$. Suppose that we remove a set of transitions $\mathcal{T}$ to obtain M_K'. It follows that if any of the transitions in $\mathcal{T}$ were not removed, then the resulting M_K' would produce a language K' controllable with respect to $\mathcal{L}_{\mathcal{M}}(G)$.

Suppose that we remove another controllable transition τ' from M_K'. Suppose further that after removing element τ' from M_K' we replace some transition $\tau \in \mathcal{T}$. We now have a machine M_K'' with some transition $\tau \in \mathcal{T}$ enabled and another transition $\tau' \notin \mathcal{T}$ disabled. Then there are two possibilities:

1. The language $\mathcal{L}_{\mathcal{M}}(M_K'')$ is still controllable with respect to $\mathcal{L}_{\mathcal{M}}(G)$. If this is the case, then we showed that τ must be effectively disabled by the removal of τ'. To see this, recall that transition τ is only added to $\mathcal{T}$ *because* it leads to a Σ_u-reverse path (defined in Chapter 3.3); i.e., τ now becomes a useless transition in the machine M_K''. If it were not useless, then τ could still fire and a Σ_u-reverse path could be reached and $\mathcal{L}_{\mathcal{M}}(M_K'')$ would be uncontrollable with respect to $\mathcal{L}_{\mathcal{M}}(G)$. Hence, disabling τ' has the effect of also disabling τ.

2. The language $\mathcal{L}_{\mathcal{M}}(M_K'')$ is not controllable with respect to $\mathcal{L}_{\mathcal{M}}(G)$. In this case at one Σ_u-reverse path still exists that creates the condition of uncontrollability.

Hence we have proved that the set $\mathcal{T}$ is *the unique* set of transitions that must be removed to obtain machine M_K' from M_K by effectively disabling the fewest number of transitions. That is, there is *no alternate set* of transitions that can still be disabled to achieve controllability.

Remark 5.4.4. In a very real way, Kumar et al.'s algorithm for supremal controllable sublanguage is brilliant. It makes the minimal number of changes to a supervisor necessary to return controllability. We are harnessing its structure preserving properties here to address the problem of Step 4 of Algorithm $\mathfrak{D}$.

5.5 Conclusion

In this chapter we developed an algorithm for computing the supremal controllable sublanguage of a language K when K is accepted by a DPDA and the plant language L is accepted by a FSM and both are prefix closed. We further showed that this supremal controllable sublanguage was *also* accepted by DPDA's. As a complement to this, we showed that when K is accepted by arbitrary non-deterministic PDA, it is undecidable whether $\sup_{\mathrm{C}}(K) = \emptyset$ with respect to an arbitrary regular plant language L. When taken with the results of Chapter 3 or [Gri06] this shows that languages accepted by arbitrary PDA are not practical as controller target languages. On the contrary, these results show promise for using languages accepted DPDA and in particular prefix closed languages accepted by DPDA's as control specifications.

We have left several open questions worth investigation. First, we have not developed an algorithm for computing $\sup_{\mathrm{C}}(K)$ when K is accepted by a DPDA and not prefix closed. We have also not discussed any closure properties; i.e., in this case, it may not be the case that $\sup_{\mathrm{C}}(K)$ is accepted by a DPDA. We believe that these questions are worth further investigation.

Finally, we have introduce the concept of Parametric Control in discrete event systems. This concept will be used throughout the remainder of this thesis.

Chapter 6

State Space Models for Software Security Quantification

6.1 Introduction

In this chapter we investigate state space models of the security of computer systems. The majority of the results presented in this chapter were published in [GMT05]. We use a discrete event dynamic system model of security dynamics. We show how to derive events and transitions from existing security taxonomies. We then apply the theory of discrete event control to define safety properties of the computer system in terms of the basic concepts of controllability used in discrete event control for two special sublanguages $K_{\mathbf{S}}$ and $K_{\mathbf{V}}$. These languages correspond to maximally robust controllable sublanguages. We also use this application to foreshadow our theory of optimal parametric control presented in Chapter 7.

A discrete event dynamic system (DEDS) model of the security of a system is a finite state machine G whose states correspond to varying levels of security and whose transitions are labeled with specific attacks and recovery methods, generically referred to as events. The number of states and transitions in the model is limited only by the sophistication of the information available about the system.

We assume that the restorative actions of the system are *controllable*; i.e., that they can be disabled at the discretion of a system administrator. We assume that the attacks in the system are uncontrollable; i.e., they are executed outside the control of a system administrator. Let Σ be the set of all events and suppose that Σ_c are the controllable events and Σ_u are the uncontrollable events. Our model assumes that all events can be partitioned into either Σ_c or Σ_u. We shall further partition our states so that they are either *safe* or *vulnerable* or *compromised*. We define a state as safe if it represents a condition in which no known vulnerabilities can be exploited. A state is defined to be vulnerable if there is a known attack that can compromise the system. Finally, a system is compromised if an attacker has successfully executed an attack or series of attacks on a system and no restorative measures have been taken.

Our analysis will proceed in three stages:

1. Model the security state of the system as a finite state machine.
2. Identify a suitable language K corresponding to a desirable security policy and determine the safety properties of K.
3. If cost information is available, formulate the problem as an optimization problem. We will take up the issue of optimal control in Chapter 7.

6.2 Modeling the Security State as a Finite State Machine

Other authors have discussed modeling system security state as a finite state machine. Phillips and Swiler [PS98] introduced the notion of modeling system state in a graph structure. This was taken up by Sheyner *et al.* [SW03, JSW02a, JSW02b] who reintroduced the concept as an attack graph and applied elementary computational tree logic to the analysis of safety properties. Sheyner [SW03] provides a detailed description of methods for modeling security states of networked systems, however neither Swiler nor Sheyner *et al.* suggest using the existing security taxonomies.

Landwehr [LBMC93] presents a taxonomy of security flaws. While this overview is out of date, it is a reasonable place to begin cataloging potential elements for the set of uncontrollable events Σ_u. Howard and Longstaff [HL98] have extended the original taxonomy concept to include a common language for discussing security incidents. Likewise, Axelsson [Axe00] presents a taxonomy of intrusion detection systems (IDS) that can help users modeling computer system security when IDS are in use. Finally, [LKJ98] presents a unique taxonomy of security pathologies and their remedies. This ontology is unique in its presentation of restorative actions and, for that reason, is particularly useful in defining the elements in the set Σ_c (the controllable events).

Example 6.2.1. Consider the *xterm* example from [LKJ98]. The *xterm* program in X Windows running under the effective root user (`su`) has a flawed logging facility in some Unix variants. This flaw allows an attacker to create new files or modify existing files by appending information to it. The vulnerability stems from the use of root privileges when making key system calls instead of the original user's privilege level. The discovery of this vulnerability may constitute an uncontrollable event. Its exploitation is certainly an uncontrollable event.

Three immediate remedies are available for this vulnerability:

1. Remove super-user privileges from *xterm*

2. Disable the logging facility of *xterm*

3. Patch *xterm* to remove this vulnerability

Each of these actions is a controllable event. The state machine model shown in Figure 6.1 adequately describes a fragment of the security of the system under consideration (i.e., a Unix system running X Windows with an unpatched version of *xterm*). In this case, States 0 and 5 are safe states, State 1 is a vulnerable state and States 2, 3 and 4 are compromised states. As we said, this represents only a fragment of the true security state of the entire system. Additionally, state machine models for entire networks can be developed as in [SW03].

6.3 Safety Properties of Sublanguages

In order to use the tools of discrete event control on our system, we must define a sublanguage of interest for a supervisor to enforce. Let $G_{\mathbf{S}}$ be a second finite

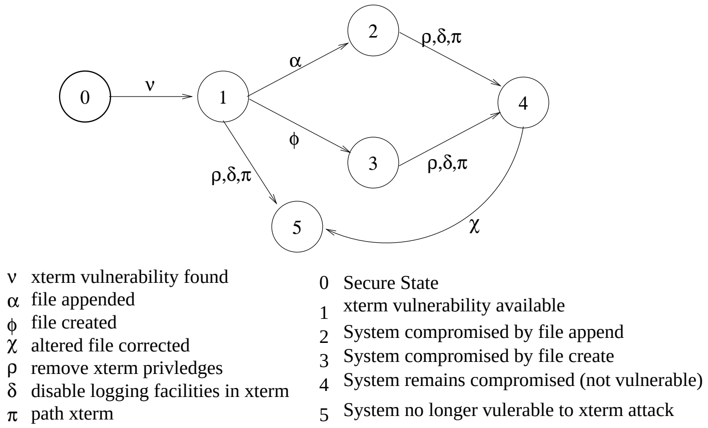

0
1
2
3
4
5
ν
α
φ
ρ,δ,π
ρ,δ,π
ρ,δ,π
χ
ν xterm vulnerability found
α file appended
φ file created
χ altered file corrected
ρ remove xterm privledges
δ disable logging facilities in xterm
π path xterm
0 Secure State
1 xterm vulnerability available
2 System compromised by file append
3 System compromised by file create
4 System remains compromised (not vulnerable)
5 System no longer vulerable to xterm attack

Figure 6.1. A small state machine model of the security of a system using an unpatched version of *xterm*

state machine with the same structure as G in which we mark the safe states. Let $K_{\mathbf{S}} = \mathcal{L}_{\mathcal{M}}(G_{\mathbf{S}})$.

Proposition 6.3.1. *The language $K_{\mathbf{S}}$ is controllable with respect to $\mathcal{L}(G)$ if and only if for any series of system compromises there is a series of restorative actions κ that return the system to a safe condition.*

Proof. The proof follows immediately from the formulation of controllability. □

We shall call a system for which $K_{\mathbf{S}}$ is controllable *repairable* since any vulnerabilities or attacks (i.e., uncontrollable events) can be repaired and the system can be driven back into a safe state. As a consequence of [RW87c] we have the following result:

Corollary 6.3.2. *A system can be verified for repairability in $O(|G|^4)$ time.*

Proof. Ramadge and Wonham [RW87c] prove that controllability can be verified in quartic time. □

In an ideal world, computer systems are always repairable. It may be the case however that $K_{\mathbf{S}}$ is *not* controllable. In this case, it is reasonable to wonder whether $K_{\mathbf{S}}$ is not controllable because of some vulnerability inherent in the system or because some potential security policy is itself creating an unsafe condition.

The maximal controllable sublanguage of $K_{\mathbf{S}}$ is denoted $\sup_{\mathrm{C}}(K_{\mathbf{S}})$ and is defined as the largest controllable sublanguage of $K_{\mathbf{S}}$ (with respect to $\mathcal{L}(G)$).

Proposition 6.3.3. *Assuming that safe states are correctly identified, if* $\sup_{\mathrm{C}}(K_{\mathbf{S}}) = \emptyset$*, then* $K_{\mathbf{S}}$ *is not controllable because of vulnerabilities inherent in the system.*

Proof. The maximal controllable sublanguage $\sup_{\mathrm{C}}(K_{\mathbf{S}})$ is obtained by disabling controllable events leading to an uncontrollable condition. If no such events exist, then $\sup_{\mathrm{C}}(K_{\mathbf{S}}) = \emptyset$ and consequently inherent vulnerabilities in the system (i.e., uncontrollable events) cause the resulting security flaw. □

In the case when $\sup_{\mathrm{C}}(K_{\mathbf{S}}) \neq \emptyset$, then we call the security policy $K_{\mathbf{S}}$ securable and refer to $\sup_{\mathrm{C}}(K_{\mathbf{S}})$ as the *secured* policy. Kumar *et al.* [KGM91] have provided a quadratic time algorithm for constructing a finite state machine accepting $\sup_{\mathrm{C}}(K_{\mathbf{S}})$.

Corollary 6.3.4. *We can determine whether a system is securable in* $O(|G|^3)$*.*

Proof. Compute a machine to accept $\sup_{\mathrm{C}}(K_{\mathbf{S}})$ ($O(|G|^2)$). Grow a Dijkstra Tree from the start state ($O(|G|^3)$). If the tree contains no path from the start state to a final state, then $\sup_{\mathrm{C}}(K_{\mathbf{S}}) = \emptyset$. □

Example 6.3.5. In Example 6.2.1, if $\mathbf{S} = \{0, 5\}$, then it is easy to see that $K_{\mathbf{S}}$ is controllable. Conversely, if $\mathbf{S} = \{0\}$, then it is easy to see that $K_{\mathbf{S}}$ is not controllable and furthermore, $\sup_{\mathrm{C}}(K_{\mathbf{S}}) = \emptyset$. In this case however, it was a poor choice of safe states that led to the uncontrollability and not (necessarily) the uncontrollable events.

If $K_{\mathbf{S}}$ is uncontrollable and $\sup_{\mathrm{C}}(K_{\mathbf{S}}) = \emptyset$, it is still possible to say something about the security of the underlying system. Let $\mathbf{V}$ be the set of vulnerable states and consider a finite state machine $G_{\mathbf{V}}$ with the same structure as G but with states in both $\mathbf{S}$ and $\mathbf{V}$ marked. Let $K_{\mathbf{V}} = \mathcal{L}_{\mathcal{M}}(G_{\mathbf{V}})$.

Proposition 6.3.6. *The language* $K_{\mathbf{V}}$ *is controllable with respect to* $\mathcal{L}(G)$ *if and only if for any series of system compromises there is a series of restorative actions* κ *that return the system to an uncompromised condition.*

We call a system in which $K_{\mathbf{V}}$ is controllable *salvageable*. In this case, we see that the salvageable property is much weaker than the repairable property, for even a system that is salvageable is still in a vulnerable state. Furthermore, it is easy to see that the analysis we performed for $K_{\mathbf{S}}$ holds for $K_{\mathbf{V}}$ and hence that salvageability can be checked in $O(|G|^4)$ time. Likewise, the maximal controllable sublanguage $\sup_{\mathrm{C}}(K_{\mathbf{V}})$ can also be computed and checked for emptiness in $O(|G|^3)$ time. If $\sup_{\mathrm{C}}(K_{\mathbf{V}}) \neq \emptyset$, then we refer to it as the secured salvageable security policy. Clearly such a policy is sub-optimal in terms of overall system security, but better than the case when $\sup_{\mathrm{C}}(K_{\mathbf{V}}) = \emptyset$. In this case, the system is inherently insecure.

It is worth noting that the controllability analysis is similar to the attack graph analysis proposed by Sheyner *et al.* [SHJ$^+$02, SW03, JSW02a, JSW02b]. Like the formal analysis proposed in [JSW02b] our controllability analysis corresponds to the verification of a formula in computational tree logic. We view our analysis as being more sophisticated however because of our use of supremal controllable sublanguages for analyzing restorative security policies. Such activities are not considered in the cited works.

6.4 Undetected Transitions

As noted earlier, perhaps one of the most serious security issues is an undetected vulnerability, that is an attack that can be performed on a system without the users' knowledge. In such an instance, the state of the system may be compromised but the user believes that the system exists in an safe state.

Traditional discrete event control models use a mask to represent *unobservable* events (see [KG95b], Chapter 4). We find this practice unnecessarily complicated and useless in our case since it may require the modeler to give names to attacks or vulnerabilities that have not yet been conceived. Consequently, we take an equivalent and alternate approach, ϵ-transition non-determinism. An ϵ-transition in a finite state machine is a *silent* transition that produces no event. They are ideal for representing unobservable phenomena.

Example 6.4.1. Consider the modified security model of Figure 6.2. In this case, we have added an unobservable transition (ϵ) from State 1 to State 6. We assume that this compromise state disables the utility of patching *xterm*.

In this case, we can *still* analyze the controllability of $K_{\mathbf{S}}$ (respectively $K_{\mathbf{V}}$) to determine whether our system is repairable. However, our analysis will not yield as much useful information. Non-determinism means that it is impossible

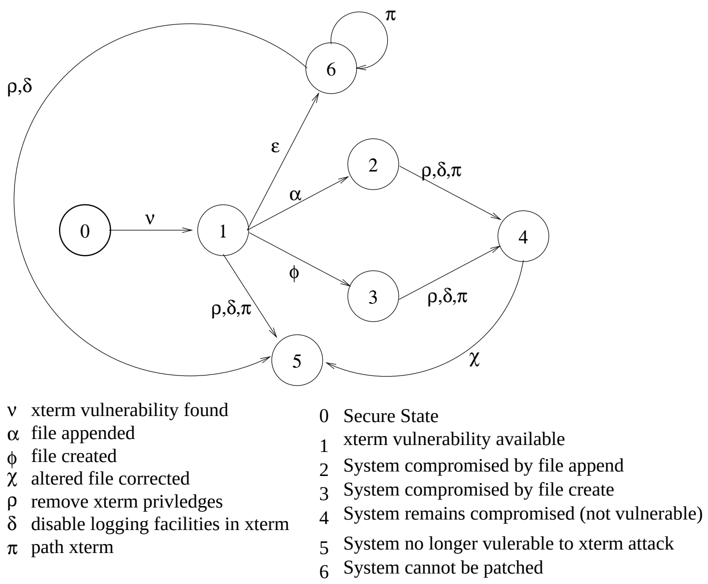

π
6
ρ,δ
ε
2
ρ,δ,π
α
0
ν
1
4
φ
3
ρ,δ,π
ρ,δ,π
5
χ
ν xterm vulnerability found
α file appended
φ file created
χ altered file corrected
ρ remove xterm privledges
δ disable logging facilities in xterm
π path xterm
0 Secure State
1 xterm vulnerability available
2 System compromised by file append
3 System compromised by file create
4 System remains compromised (not vulnerable)
5 System no longer vulerable to xterm attack
6 System cannot be patched

Figure 6.2. A small state machine model of the security of a system using an unpatched version of *xterm* when an unknown attack disables the utility of patching *xterm*.

to determine when we are in a specific state and executing a specific restorative action. In Example 6.4.1, we will still have the string $\nu\pi$ in $K_{\mathbf{S}}$, but we will also have the string $\nu\pi\delta$ in $K_{\mathbf{S}}$ as a result of the non-determinism associated with the single ϵ transition.

The growth in $K_{\mathbf{S}}$ can make analysis of controllability in determining a security policy nearly useless since non-determinism may cause the policy to specify useless restorative actions. A solution to this problem is to deterministically select a control policy $K \subseteq K_{\mathbf{S}}$ and verify it for controllability.

The choice of K is very much a design problem and no theory is available for designing K from scratch. If K is uncontrollable, the policy designer may choose to use the maximal controllable sublanguage to correct K. This process is not without its caveats, since K may be uncontrollable as a result of human error and not inherent system insecurity. Damiani, Griffin and Phoha [DGP03] have developed methods for correcting it if K is uncontrollable that do *not* involve the use of maximal controllable sublanguages. The techniques they present can be adapted to DPDA controllers using the algorithms set forth in Chapter 3. The algorithm presented in [DGP03] will identify the sources (and corrective suggestions) for the security policy K in quadratic time.

A simple approach to designing K is to separate the control policy into deterministic and non-deterministic components. In this case, restorative actions are taken only when it is certain they will lead to an improvement in system security. If the resulting language K is uncontrollable, then a trade-off must be made between executing a restorative action that is useless and doing nothing (which leads to the uncontrollability of K). In this case, the policy designer must use a heuristic approach to settling these design decisions.

Example 6.4.2. We can divide the security policy for the problem in Example 6.4.1 into the case when a known attack has occurred and the case when we do not know whether an attack has occurred. At State 1 we enable only ρ and δ since we are certain those will secure the system. At States 2 and 3 we enable only π, for that is the better solution to our problem and we can be sure that patching *xterm* will be effective. It is clear that K is controllable in this small example.

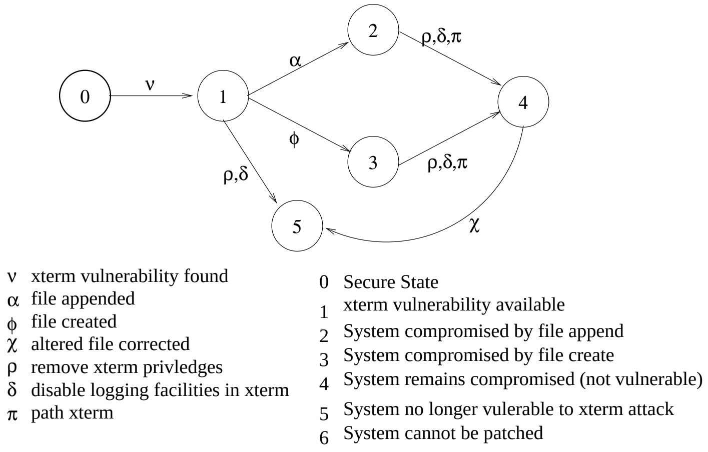

0
ν
1
α
2
ρ,δ,π
4
ϕ
3
ρ,δ,π
ρ,δ
5
χ
ν xterm vulnerability found
α file appended
ϕ file created
χ altered file corrected
ρ remove xterm privledges
δ disable logging facilities in xterm
π path xterm
0 Secure State
1 xterm vulnerability available
2 System compromised by file append
3 System compromised by file create
4 System remains compromised (not vulnerable)
5 System no longer vulerable to xterm attack
6 System cannot be patched

Figure 6.3. A small security policy for using an unpatched version of *xterm* when an unknown attack disables the utility of patching *xterm*.

6.5 Extension to DPDA Example

In this section, we extend our *xterm* example to use a DPDA to count the number of file alterations that occurred because of a system intrusion. This section does not appear in [GMT05].

Consider the DPDA shown in Figure 6.4. This DPDA will make a note in the stack each time a file is created or modified as a result of an intrusion. Further,

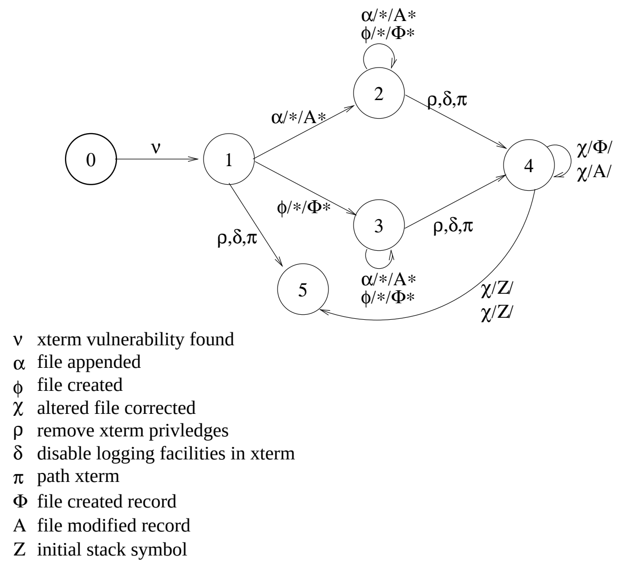


Figure 6.4. System that can count the number of files modified by a system intrusion and correct each file.

using the transitions defined at State 4 the policy enforces a complete system clean up before arrival in the secured state. Using a plant model with the safe (or vulnerable) states marked, we could verify this policy for controllability and hence determine whether the policy was secure or salvageable.

Note, that in Chapter 5 we do not provided a method for computing the supremal controllable sublanguage in the case when the controller is modeled by DPDA and the plant is modeled by an FSM but neither generate prefix closed languages. In this case, if the security policy were not secure, we could define securability or salvageability as a parametric control problem as we defined it in the last section of Chapter 5. Let M_K be the model of the security policy (i.e., a DPDA as in Figure 6.4. In this case, the policy would be securable (salvageable) if there were a controllable submachine M'_K of M_K that is controllable with respect to the given system security model. Thus the work presented in [GMT05] and summarized in this chapter extends easily to the extended notion of discrete event control we provide in this thesis.

6.6 Conclusion

In this chapter, we have seen how to apply the theory of discrete event control to modeling and analyzing security policies. We have also seen how our pushdown automaton models of control policies can enhance the modeling power of security policies, by allowing them to count arbitrary numbers of access violations (as in the final example). This extended the original models shown in [GMT05].

Chapter 7

Optimal Parametric Discrete Event Control

7.1 Introduction

In this chapter, we define and study an optimal parametric control problem in the Supervisory Control Theory (see Chapter 2). As usual, let Σ be a discrete event alphabet with $\Sigma = \Sigma_u \cup \Sigma_c$. As defined in Chapter 2, Σ_c denotes the set of controllable events, while Σ_u is the set of uncontrollable events and $\Sigma_c \cap \Sigma_u = \emptyset$.

Let $X_1, \ldots, X_n$ be binary variables; we write this as $X_1, \ldots, X_n \in \mathcal{B} = \{0, 1\}$. Let G be a trim plant model with marked language $L = \mathcal{L}_{\mathcal{M}}(G)$ and language $\overline{L} = \mathcal{L}(G)$. We assume that G is given as part of the problem statement.

Let $M_K(X_1, \ldots, X_n)$ be a function taking variables $X_1, \ldots, X_n$ and returning a machine (e.g., FSM, DPDA, PDA, TM) that will define a language $K \subseteq L$. The language K is the target language–the language that describes desirable event traces to be emitted by the plant. Recall that $\mathcal{C}(L)$ denotes controllable sublanguages of L (see Chapter 2). The statement $M_K(X_1, \ldots, X_n) \models \Pi$ indicates that the machine $M_K(X_1, \ldots, X_n)$ makes true a set of logical sentences Π that describe structural properties of the machine $M_K(X_1, \ldots, X_n)$. For the remainder of this chapter, we will assume that Π is given. Let $C(w)$ denote the cost associated with a string w. This too will be defined explicitly in Section 7.2. We concentrate on the following optimal discrete event control problem:

$$\begin{aligned}
&\min_{X_1,\ldots,X_n} \max_{w\in K} C(w)\\
&\text{s.t. } \mathcal{L}_{\mathcal{M}}(M_K(X_1,\ldots,X_n)) \subseteq L\\
&\qquad K = \mathcal{L}_{\mathcal{M}}(M_K(X_1,\ldots,X_n))\\
&\qquad M_K(X_1,\ldots,X_n) \models \Pi\\
&\qquad K \in \mathcal{C}(L)\\
&\qquad (X_1,\ldots,X_n) \in \mathbb{B}
\end{aligned} \tag{7.1.1}$$

This is *similar* to the optimal control problem studied by LaFortune et al. in [SL98] however in this chapter:

1. We consider $M_K(X_1,\ldots,X_n)$ to have range consisting of deterministic pushdown automata; i.e., for each possible value assignment to the variables $X_1,\ldots,X_n$, M_K is in $DPDA$. (Since DPDA are generalizations of FSM, we may also consider the case when $M_K(X_1,\ldots,X_N)$ contains FSM.)

2. Instead of using a dynamic programming approach to the optimal control problem (as in [SL98]), we apply a *branch-and-bound* algorithm driven by the constraints. We compare and contrast the problem we are solving with the problem investigated in [SL98] in Section 7.5 and show why we do not use a dynamic programming approach in our solution.

3. We will provide an example where elements in the range of $M_K(X_1,\ldots,X_n)$ will have specific structure (See Chapter 8).

The approach proposed is like a classical optimal parametric control problem phrased for discrete event systems. Recall, classical parametrized control asks the question, "Assuming a PID controller is to be used what are the optimal tuning parameters to use in order to minimize a cost function." In the case of statistical parametric control, the cost function may be the mean square error leading to MMSE parametric control or a cost function involving the gradient of the control function if large deviations in the controllable vector are undesirable [Cas02].

Our problem is similar to this. We specify a mapping $M_K(X_1,\ldots,X_n)$. The output of this function must have certain properties (satisfy Π) and further it must give rise to an objective language K such that $K \in \mathcal{C}(L)$; this is another implicit property of the output of $M_K(X_1, \ldots, X_N)$. We then attempt to find the optimal control structure satisfying these properties that minimizes a given cost function.

Using the results in Chapter 3, we have the following theorem:

Theorem 7.1.1. *There is no algorithm to solve Problem 7.1.1 when the range of $M_K(X_1, \ldots, X_N)$ contains machines whose accepted languages are not properly contained in the class* **DCFL** *over a given alphabet Σ. Further, there is only an algorithm to solve Problem 7.1.1 only if Π is decidable.*

Proof. The proof is clear from Theorem 3.6.2. If the languages of machines in the range of $M_K(X_1, \ldots, X_N)$ are not contained in **DCFL**, then there is no algorithm in general to check controllability. Hence we cannot analyze our constraints for feasibility. The statement on the decidability of Π follows from a similar argument. □

For the remainder of this chapter, we will assume that logical sentences may use two types of constants (nouns)

1. Constants from: $Q = \{q_0, \ldots, q_n\}$, $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ and $\Gamma = \{Z_0, \ldots, Z_k\}$ and

2. Constants from the sets Q_f and δ

They may relate these two together using the $\in$ relation. An example sentence,

$$\exists p_1((q_0, \sigma_1, Z_0, p, Z_0) \in \delta \wedge (q_0, \sigma_2, Z_0, p, Z_0) \notin \delta)$$

is a simple sentence saying there is a transition from the start state q_0 on input σ_1 with top stack symbol Z_0 to state p, but no such transition on input symbol σ_2.

Clearly, if we restrict our attention to this class of sentences Π, then for all functions $M_K(X_1, \ldots, X_n)$ whose range consists of DPDA, $M_K(X_1, \ldots, X_n) \models \Pi$ is decidable for fixed $X_1, \ldots, X_n$.

7.2 Optimal Control

Recall Problem 7.1.1 defined above. We are now in a position to define precisely what our objective function is, what we mean by the function $M_K(X_1, \ldots, X_n)$

and the constraints $K \in \mathcal{C}(L)$ and $M_K(X_1, \ldots, X_n) \models \Pi$. Let G be a fixed plant model with $L = \mathcal{L}_{\mathcal{M}}(G)$.

7.2.1 Objective Function

In [SL98] Sengupta and LaFortune proposed the minimax objective function for a regular language L. For each event, Sengupta and LaFortune assumed a given enabling cost $\kappa_\alpha(a)$ and a given execution cost for each event $a \in \Sigma$, $\kappa_\rho(a)$. Both of these costs may be negative, thus making them rewards. Given a string w accepted by a machine M_K, the cost of the string w could be computed by inductively determining which events were enabled during string execution and which events in w had actually been executed.

Let $w = a_1 \cdots a_n$ be a string. Let $w(i-1) = a_1 a_2 \cdots a_{i-1}$ for $1 \leq i \leq n$. Also, let $w(0) = \epsilon$. Define

$$\kappa_\alpha(M_K|w(i-1)) = \sum_{a \in \Sigma, w(i-1)a \in \mathcal{L}(M_K)} \kappa_\alpha(a)$$

This is the cost associated with enabling events in machine M_K assuming that $w(i-1)$ has already been observed. This is because if $w(i-1)a \in \mathcal{L}(M_K)$, then after string $w(i-1)$ has been observed it is possible to see event a, hence $w(i-1)a \in \mathcal{L}(M_K)$. Thus, even a must be enabled.

Then the cost of string w is:

$$C(w) = \sum_{i=1}^{n} \beta^{i-1}(\kappa_\alpha(M_K|w(i-1)) + \kappa_\rho(a_i)) \tag{7.2.1}$$

Trivially for $\beta \in [0, 1)$, there is a supremal value C^* such that for all $w \in L$, $C(w) \leq C^*$. Hence we need not concern ourselves with questions of convergence.

Remark 7.2.1. The cost function defined in [SL98] is identical to the one given above except [SL98] assume $\beta = 1$. In order to guarantee a convergent optimization algorithm, Sengupta and LaFortune assumed that all closed loops in the machine M_K had zero net cost. (Otherwise, when considering arbitrarily long loops, the controller could easily produce an infinite cost.) We do not make this restriction. Other differences between our approach and that in [SL98] are discussed in Chapter

7.5.

Remark 7.2.2. One *negative* point about abandoning Sengupta and LaFortune's approach is the affect a short term time horizon has on the overall objective function. We appreciate that this violates the principle of infinite operation embedded in the controllability predicate, however we counter that many real world optimal control problems often contain either a finite time horizon, (in which case strict controllability is not needed) or they use a depreciation constant in an infinite time horizon. Therefore, we do *not* feel that this is in anyway detrimental to our theory. Further, we believe it opens an new avenue of research for those interested in pursuing a theory of optimal parametric discrete event control in which $\beta = 1$.

7.2.2 Computing the Objective Function

In general, it is simpler to approximate the objective using a numerical technique. This is because computing a closed form expression for the objective function requires us to compute a specific representation of the strings that are generated by $M_K(X_1, \ldots, X_n)$ for all values of $X_1, \ldots, X_n$. This is simple in some instances (e.g., Example 2.3.2), but may be highly complex for arbitrary DPDA. In our research, we could not identify a straight-forward way to solve this problem, and hence developed an approximation method.

To approximate the objective numerically, we chose an ζ value. Let $C_{\max}$ be the largest combined event and enabling cost. The maximal enabling cost may be computed by looking at the enabling costs associated with each state of the controller M_K. That is, given a state q in M_K, the enabling cost of state q can be computed by summing the enabling cost associated with each transition defined at q.

It follows that any string of length n must have total cost less than

$$\sum_{k=0}^{n} \beta^k C_{\max}.$$

More importantly, there is some value n so that

$$\sum_{k=n}^{\infty} \beta^k C_{\max} < \zeta.$$

It follows that:

$$n = \left\lceil \frac{\log\left(\frac{\epsilon(1-\beta)}{C_{\max}}\right)}{\log(\beta)} \right\rceil \tag{7.2.2}$$

is the minimum integer value of n for which this is true. Hence, we may compute the objective function to ζ precision by exploring all strings up to length n. This can be done in a secondary branch-and-bound framework. This problem is like a traveling salesman problem [CLRS01]. In this case however, we are looking for the longest route of n steps. We can also compute $\min_{w\in\mathcal{L}_{\mathcal{M}}(M_K')} C(w)$ in this way as well. To do so, compute the minimum cost of all strings up to length n and also any string with length less than n that is accepted by M_K' that has *no* continuation (i.e., accepting strings that cause deadlock in both the plant and the controller).

We provide Algorithm $\mathfrak{E}$, a subroutine that will approximate the solution to the problem $\max_{w\in\mathcal{L}_{\mathcal{M}}(M_K(X_1,\dots,X_n))} C(w)$, when there is not a substructure of $M_K(X_1,\dots,X_n)$ that allows an infinite number of ϵ-transitions to occur.

Algorithm $\mathfrak{E}$ operates as follows: We will simulate the operation of the DPDA (or FSM) $M_K(X_1,\dots,X_n)$ as it generates strings. We initialize data structures to store the state and stack of $M_K(X_1,\dots,X_n)$. We initialize the incumbent solution to $-\infty$ (since we are attempting to identify $\max_{w\in\mathcal{L}_{\mathcal{M}}(M_K(X_1,\dots,X_n))} C(w)$). We determine and upper bound on n using Equation 7.2.2. This will allow us to bound our search at some point.

We then choose an enabled transition and simulate a move of $M_K(X_1,\dots,X_n)$, updating the state and stack. We record the cost associated with this move using the objective function defined in Equation 7.2.1. We repeat this process until one of two things happens:

1. The string we have simulated has length greater than n. In this case, we know the computed cost will not grow significantly greater. If the cost is greater than the incumbent solution, then the incumbent is replaced.

2. It is shown that the cost of the string cannot be any larger than an incumbent solution.

When either of these two events occur, we remove the last symbol (going back to the previously recorded state and stack combination) and move on to the next enabled transition that has not been explored.

It is easy to see that this algorithm will exhaustively explore all strings generated by $M_K(X_1, \ldots, X_n)$ with length less than n. It is also clear that we could modify Algorithm $\mathfrak{C}$ to find the minimum cost string generated by $M_K(X_1, \ldots, X_n)$. The change to do this is trivial, and we do not discuss this in detail. To do this, we simply explore minimal cost strings and use a branch-and-bound on the lowest cost string, not the highest cost one.

Algorithm Description 7.2.1 – Algorithm $\mathfrak{C}$

Initialization: Suppose we have machine M_K and we want to approximate the objective function with strings of length n. Enter the search step with the start state of M_K and set $I = \emptyset$, $m = 0$, and $\gamma_{\max} = -\infty$ and $\gamma = 0$. Let $C_{\max}$ be the largest event and enabling cost.

Search Step: We enter the search step with state q of M_K, a set I and counter m.

1. If $m > n$, then stop. If $\gamma > \gamma_{\max}$, then set $\gamma_{\max} = \gamma$ and return;
2. Find the first transition τ defined at q in M_K with appropriate stack symbol if necessary and that is not contained in I. If no such transition exists, return.
3. Set $\gamma' = \beta^n(C(q) + C(t))$, where $C(q)$ is the enabling cost associated with state q and $C(t)$ is the event cost of the transition t.
4. If $\gamma' + \beta^{n+1} \sum_{k=0}^{m-1} \beta^k C_{\max} = \gamma' + \beta^{n+1} \frac{C_{\max}(1-\beta^m)}{1-\beta} < \gamma_{\max}$ return;
5. Add τ to I;
6. If τ is not an ϵ-transition, then transition M_K on τ and find the new state q'; store stack information as appropriate. GOTO STEP 1 on M_K, q', γ', $m+1$ and set $I' = \emptyset$.
7. If τ is an ϵ-transition, then transition M_K on τ and find the new state q'; store stack information as appropriate. GOTO STEP 1 on M_K, q', γ', m and set $I' = \emptyset$.

Theorem 7.2.3. *Algorithm $\mathfrak{C}$ converges iff there is no substructure of M_K in which an infinite number of ϵ-transitions can occur.*

Proof. Step 1 of Algorithm $\mathfrak{C}$ will return when $m > n$. Hence, no path of length larger than n can ever be explored. The value of m is incremented at each level of descent in Step 6, however if Step 7 is executed instead of Step 6, then m is not incremented because ϵ-transitions do not increase string length, but may modify the stack structure of a DPDA. Hence, if an infinite number of ϵ-transitions is executed, then Algorithm $\mathfrak{C}$ may never terminate. The converse is clear by a similar argument. □

Lemma 7.2.4. *Algorithm $\mathfrak{E}$ will find an ζ approximation of the true objective value. Furthermore, the problem of objective function approximation is contained in* NP*.*

Proof. The fact that Algorithm $\mathfrak{E}$ can find an approximation of the objective function is clear from its structure. It will continue searching through the space of strings of length $< n$ until a maximal one is discovered. Other smaller strings will be fathomed by Step 4. Further, it is clear from the nature of the problem that approximating the objective function is in NP, since this problem can be phrased as a pure integer programming problem with variables x_{ij} defined so that $i = 0, \ldots, n$ and j is defined over the transitions of M_K. The transition structure of M_K must then be coded as a series of constraints. Since Integer Programming is NP-complete [HU79, CLRS01] it follows that this problem is contained in NP. □

7.2.3 The Function $M_K(X_1, \ldots, X_n)$

In our formulation, let M be a pushdown machine providing a specification language S. For the fixed plant model G, compute a machine $M_K = M \cap G$. This machine accepts the language $S \cap L$. Suppose that M_K has controllable transitions $\tau_1, \ldots, \tau_n$. For binary variable X_i, we have $X_i = 1$ if and only if τ_i is enabled in M_K. Hence, we may define $M_K(X_1, \ldots, X_n)$ to be the machine obtained from M_K by enabling or disabling the appropriate transitions. Clearly, $M_K = M_K(1, 1, \ldots, 1)$. Thus, our principal constraint $K = \mathcal{L}_{\mathcal{M}}(M_K(X_1, \ldots, X_n))$ enforces the parametric control notions we defined in Section 5.4. Note the slight abuse of notation we have introduced: here $M_K(X_1, \ldots, X_n)$ is the function returning a submachine of M_K, the intersection of M and G; and we have $M_K = M_K(1, 1, \ldots, 1)$.

7.2.4 Constraints: $K \in \mathcal{C}(L)$ and $M_K(X_1, \ldots, X_n) \models \Pi$

By $K \in \mathcal{C}(L)$ we simply require that the resulting language of the supervised system must be controllable with respect to the plant language $L = \mathcal{L}(G)$.

Let $\psi_1, \ldots, \psi_m$ be a set of statements in Π. We will say that $M_K(X_1, \ldots, X_n) \models \Pi$ for fixed values of $X_1, \ldots, X_n$ if and only if each sentence in Π is true in the automaton $M_K(X_1, \ldots, X_n)$. That is, the graph structure of M_K satisfies sentences in Π [CLRS01].

Suppose that ψ is a sentence Π and fix values for $X_1, \ldots, X_n$ Suppose that $M_K(X_1, \ldots, X_n) \not\models \psi$; that is, ψ is not true in $M_K(X_1, \ldots, X_n)$. We will say that ψ is *reachable* from $M_K(X_1, \ldots, X_n)$ if there is a finite set of variables $X_{i_1} = \cdots = X_{i_k} = 1$ so that when we set $X_{i_1} = \cdots = X_{i_k} = 0$, then $M_K(X_1, \ldots, X_n) \models \psi$. Alternatively, ψ is reachable if there is a finite set of controllable transitions $\tau_1, \ldots, \tau_k$ in $M_K(X_1, \ldots, X_n)$ that can be disabled and thus make the new structure satisfy ψ. The concept of reachability will be important in our branch-and-bound optimization algorithm.

7.2.5 Branch-and-Bound Algorithm for Optimization

We provide a depth first search branch-and-bound algorithm below. At each node of the branch-and-bound tree, we will have a set of fixed variable indexes, $\mathfrak{I}$ (for immutable). These are variables that are fixed in value and cannot be modified.

The algorithm we provide operates as follows. Initially, all variables are initialized to 1. This means that every controllable transition in M_K is enabled. We test to see whether the parametric control problem is solvable with $M_K(X_1, \ldots, X_n)$. If not, then we stop. We also test to see if Π is reachable; if not we stop. In each case, we stop because we've determined we can never find a substructure of $M_K(X_1, \ldots, X_n)$ that is controllable with respect to the plant model and that satisfies Π. We use Algorithm $\mathfrak{D}$ to solve the parametric control problem. If Algorithm $\mathfrak{D}$ disables some transitions in $M_K(X_1, \ldots, X_n)$, then we identify the new values for $X_1, \ldots, X_n$. If possible, we determine an incumbent solution. At this point, we are ready to descend. We choose a (non-zero) variable (say X_1) and set it to zero. This *disables* a transition in the base structure M_K. We add 1 to the immutable set $\mathfrak{I}$. With 1 in $\mathfrak{I}$ this means we cannot change the value of X_1 at any node *below* level 1 of the tree.

We descend down the tree repeating the testing process above and also checking for feasible solutions that yield an objective function value that is smaller than the current incumbent solution. When we cannot set any more variables to 0, we climb back up the tree and change the X_i variables that were set to 0 to 1; we then descend with these values fixed at 1. Ultimately, we will return to the root node and X_1 will be changed from 0 to 1. The process then repeats with $X_1 = 1$ (and immutable). A completely expanded search tree is shown in Figure 7.1

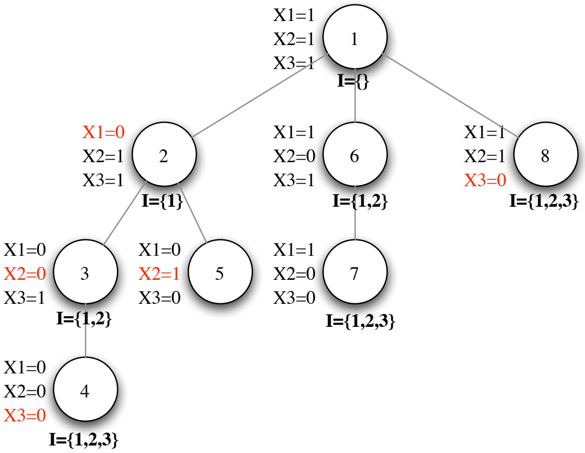


Figure 7.1. A fully expanded tree; nodes are labeled in order they will be investigated. The immutable set is shown below each node.

In the main body of the algorithm, there are a few caveats. First, if Algorithm $\mathfrak{D}$ attempts to disable a transition corresponding to a variable with index in the immutable set, then search along this branch is discontinued. This is because we can be certain we have already explored this structure–otherwise, the index would not be both in the immutable set and have its corresponding variable set to 1. Second, we have devised a cost bound based on the fact we are solving a minimax problem that allows us to stop exploring a branch when we are certain no improved solution can be obtained. (This is explained completely in Lemma 7.2.5.) Finally, we can stop descending along a branch when we are certain Π is not reachable with the current variable set $X_1, \dots, X_n$.

In short, the algorithm (which we call Algorithm $\mathfrak{C}$, shown on pages 100-101) is an exhaustive search algorithm that implicitly enumerates some of the nodes in the search when it is clear they will not yield an improved solution. This fact is shown in Theorem 7.2.6.

Lemma 7.2.5. *Let N_1 and N_2 be two nodes in the branch-and-bound tree produced by Algorithm $\mathfrak{C}$ and suppose that M_K^1 and M_K^2 are the controllable substructures of M_K generated after Step 2 for these two nodes respectively. Finally, suppose that N_1 is the parent of N_2. Then:*

1. $\min_{w \in \mathcal{L}_{\mathcal{M}}(M_K^1)} C(w) \leq \min_{w \in \mathcal{L}_{\mathcal{M}}(M_K^2)} C(w)$ *and*

Algorithm Description 7.2.2 – Algorithm $\mathfrak{C}$: Initialization

Given $M_K(X_1, \dots, X_n)$ and G, initialize $X_1 = X_2 = \cdots = X_n = 1$.

1. Use Algorithm $\mathfrak{A}$ to test whether $\mathfrak{L}_{\mathcal{M}}(M_K(X_1, \dots, X_n))$ is controllable with respect to $\mathfrak{L}_{\mathcal{M}}(G)$.
    (a) If $\mathfrak{L}_{\mathcal{M}}(M_K(X_1, \dots, X_n))$ is not controllable with respect to $\mathfrak{L}_{\mathcal{M}}(G)$, then GOTO 2.
    (b) Otherwise use the Tableau Method from [Sim00] to test whether Π is reachable from $M_K(X_1, \dots, X_n)$.
        i. If Π is not reachable. STOP, the problem is infeasible.
        ii. Otherwise GOTO 3.
2. $\mathfrak{L}_{\mathcal{M}}(M_K(X_1, \dots, X_n))$ is not controllable with respect to $\mathfrak{L}_{\mathcal{M}}(G)$; apply Algorithm $\mathfrak{D}$.
    (a) If Algorithm $\mathfrak{D}$ returns $\emptyset$, STOP, the problem is infeasible.
    (b) Otherwise, let A be the machine output from Algorithm $\mathfrak{D}$. Use A to determine current values of $X_1, \dots, X_n$.
    (c) Use the Tableau Method from [Sim00] to test whether Π is reachable from $M_K(X_1, \dots, X_n)$.
        i. If Π is reachable from $M_K(X_1, \dots, X_n)$, then GOTO 3.
        ii. Otherwise, STOP, the problem is infeasible.
3. If $M_K(X_1, \dots, X_n)$ satisfies Π, then set the incumbent objective value C_0 to the objective value of $\mathfrak{L}_{\mathcal{M}}(M_K)$ computed using Algorithm $\mathfrak{E}$. Declare the incumbent solution to be $X_1^0 = X_1, \dots, X_n^0 = X_n$. Otherwise, set $C_0 = \infty$ and set the incumbent solution to NULL.
4. Initialize $\mathfrak{I}$ to the set of variable indexes of variables that are currently zero.

2. $\max_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^1)} C(w) \geq \max_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^2)} C(w)$.

Proof. In obtaining M_K^2 from M_K^1, we remove a transition. This effectively removes a collection of strings from $\mathfrak{L}_{\mathcal{M}}(M_K^1)$ to obtain $\mathfrak{L}_{\mathcal{M}}(M_K^2)$. If the strings producing a maximal cost were removed, then it follows that $\max_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^1)} C(w) > \max_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^1)} C(w)$; otherwise $\max_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^1)} C(w) = \max_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^1)} C(w)$. Likewise, if the strings producing a minimal cost were removed, then it follows that $\min_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^1)} C(w) < \min_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^1)} C(w)$; otherwise it is immediately clear that $\min_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^1)} C(w) = \min_{w \in \mathfrak{L}_{\mathcal{M}}(M_K^1)} C(w)$. □

Algorithm Description 7.2.3 – Algorithm $\mathfrak{C}$: Branching Step

Suppose we have arrived at node N with $X_1, \dots, X_n$ and immutable set $\mathfrak{I}$. Store a reference copy of $X_1, \dots, X_n$ at this node.

1. Let $Y_1 = X_1, \dots, Y_n = X_n$. Choose i from 1 to n such that i is not in $\mathfrak{I}$ and set $Y_i = 0$. (Effectively, we will delete a transition τ from $M_K(X_1, \dots, X_n)$.) Move i to $\mathfrak{I}$. If no such variable is available, then fathom this node.
2. Test that $M_K(Y_1, \dots, Y_n)$ is controllable with respect to $\mathfrak{L}_{\mathfrak{M}}(G)$ using Algorithm $\mathfrak{A}$.
    - (a) If $M_K(Y_1, \dots, Y_n)$ is controllable with respect to $\mathfrak{L}_{\mathfrak{M}}(G)$ then GOTO 4.
    - (b) Otherwise GOTO 3.
3. Apply Algorithm $\mathfrak{D}$ (From Chapter 5) to $M_K(Y_1, \dots, Y_n)$.
    - (a) If Algorithm $\mathfrak{D}$ returns $\emptyset$, then GOTO 1.
    - (b) Otherwise, let A be the machine output from Algorithm $\mathfrak{D}$. Use A to determine current values of $Y_1, \dots, Y_n$.
        - i. If any variable Y_j has been set to zero, but j is in $\mathfrak{I}$ then GOTO 1.
        - ii. Otherwise, add the indexes of any of the $Y_1, \dots, Y_n$ that were set to zero by Algorithm $\mathfrak{D}$ to $\mathfrak{I}$. GOTO 4.
4. Test that Π is reachable from $M_K(Y_1, \dots, Y_n)$ using the Tableau Method from [Sim00].
    - (a) If Π is reachable, GOTO 5.
    - (b) If not, then GOTO 1.
5. Compute $\min_{w \in \mathfrak{L}_{\mathfrak{M}}(M_K(Y_1,\dots,Y_n))} C(w)$ using Algorithm $\mathfrak{E}$ for minimum cost.
    - (a) If $\min_{w \in \mathfrak{L}_{\mathfrak{M}}(M_K(Y_1,\dots,Y_n))} C(w) > C_0$, then GOTO 1.
    - (b) Otherwise GOTO 6.
6. If Π is satisfied by $M_K(Y_1, \dots, Y_n)$ as computed using the Tableau Method from [Sim00], and $C_0 > \max_{w \in \mathfrak{L}_{\mathfrak{M}}(M_K(Y_1,\dots,Y_n))} C(w)$, computed using Algorithm $\mathfrak{E}$, then set $C_0 = \max_{w \in \mathfrak{L}_{\mathfrak{M}}(M_K(Y_1,\dots,Y_n))} C(w)$ and set $X_1^0 = Y_1, \dots, X_n^0 = Y_n$. GOTO 7.
7. Create a branch with $X_1 = Y_1, \dots, X_n = Y_n$ and a **copy** of the current set $\mathfrak{I}$ from node N. GOTO 1 on $X_1, \dots, X_n$ and $\mathfrak{I}'$.

<u>Convergence:</u> When the algorithm is finished execution, either $X_1^0, \dots, X_n^0$ is the optimal solution or the problem will have been identified as infeasible.

Theorem 7.2.6. *Given M, G and $M_K = M \cap G$, if there is a substructure M_K' of M_K that is controllable, then Algorithm $\mathfrak{C}$ will find it. Further, Algorithm $\mathfrak{C}$ will*

find the values of $X_1, \ldots, X_n$ *that solve Problem 7.1.1.*

Proof. It suffices to show that Algorithm $\mathfrak{C}$ will implicitly enumerate all possible substructures of M_K that can be obtained by disabling controllable transitions. Trivially, in Step 1 of the branching process, we remove a transition and add this transition to the immutable transitions at node N–we do this by adding index i to the immutable set. If we return to Step 1 at Node N, this transition will never be removed again in any child node of N because the immutable set propagates down the tree as shown in Step 7. Hence, at least we know that all branches are independent and if we only executed these two operations we would eventually consider all possible ways of disabling controllable transitions in M_K.

In Steps 2 and 3, if neither $M_K(Y_1, \ldots, Y_n)$ nor A is controllable, then by Theorem 5.4.1, we know that no substructure of $M_K(Y_1, \ldots, Y_n)$ can ever be controllable. Hence, we may implicitly enumerate all child nodes that occur below this node by fathoming this node. On the other hand, suppose that an index in I (the immutable set) is removed in creating A, then by the independence of the branches, this structure has already been enumerated in an earlier branch and we may ignore it.

Finally, if A is controllable and no immutable variables have been changed, then we know by Theorem 5.4.1 that we have removed the minimum number of transitions from $M_K(Y_1, \ldots, Y_n)$ to achieve controllability and we may continue.

In Step 4, if Π is unreachable from $M_K(Y_1, \ldots, Y_n)$, then there is no combination of transition disabling that can force our decidable predicate constraints to be true along the remainder of the current branch. Hence, we may fathom away all further branches resulting from this node.

We have shown in Lemma 7.2.5 that $\min_{w \in \mathcal{L}_{\mathcal{M}}(M_K(Y_1,\ldots,Y_n))} C(w)$ increases as the tree descends. Hence, if the minimal cost of a string in $M_K(Y_1, \ldots, Y_n)$ is greater then the current incumbent cost, this branch can yield no better solution and all subsequent nodes can be immediately fathomed. This depends on the uniqueness statement discussed in Remarks 5.4.2 and 5.4.3. The second remark is *so* important that we reiterate it here:

Reiteration of Remark 5.4.3

Suppose that we apply Algorithm $\mathfrak{D}$ to machines M_K and G. Recall, we are operating on a machine that accepts the language $\overline{\mathcal{L}_{\mathcal{M}}(M_K)}^c \cap \overline{\mathcal{L}_{\mathcal{M}}(G)}$. Suppose that we remove a set of transitions $\mathcal{T}$ to obtain M_K' from M_K. It follows that if any of the transitions in $\mathcal{T}$ were not removed, then the resulting M_K' would produce a language K' uncontrollable with respect to $\mathcal{L}_{\mathcal{M}}(G)$.

Suppose that we remove another controllable transition τ' from M_K'. Suppose further that after removing element τ' from M_K' we replace some transition $\tau \in \mathcal{T}$. We now have a machine M_K'' with some transition $\tau \in \mathcal{T}$ enabled and another transition $\tau' \notin \mathcal{T}$ disabled. Then there are two possibilities:

1. The language $\mathcal{L}_{\mathcal{M}}(M_K'')$ is still controllable with respect to $\mathcal{L}_{\mathcal{M}}(G)$. If this is the case, then we showed that τ must be effectively disabled by the removal of τ'. To see this, recall that transition τ is only added to $\mathcal{T}$ *because* it leads to a Σ_u-reverse path (defined in Chapter 3.3); i.e., τ now becomes a useless transition in the machine M_K''. If it were not useless, then τ could still fire and a Σ_u-reverse path could be reached and $\mathcal{L}_{\mathcal{M}}(M_K'')$ would be uncontrollable with respect to $\mathcal{L}_{\mathcal{M}}(G)$. Hence, disabling τ' has the effect of also disabling τ.

2. The language $\mathcal{L}_{\mathcal{M}}(M_K'')$ is not controllable with respect to $\mathcal{L}_{\mathcal{M}}(G)$. In this case at one Σ_u-reverse path still exists that creates the condition of uncontrollability.

As we said in Remark 5.4.3, Case 1 above shows that $\mathcal{T}$ is *the unique* set of transitions that must be removed to obtain machine M_K' from M_K by effectively disabling the fewest number of transitions. That is, there is *no alternate set* of transitions that can still be disabled to achieve controllability.

Let us consider the impact this information has on the value of the objective function and the algorithm. When we disable τ' in Case 1 above, the resulting objective value must be no greater than the objective value obtained for M_K'. This follows from Lemma 7.2.5 and the fact that if removing τ' leads to a controllable machine M_K'', then effectively transition τ is also removed and hence cannot affect the objective function value. On the other hand, if τ leads to an uncontrollable system, then we may apply Algorithm $\mathfrak{D}$ again and the result follows inductively.

Removal of the transition τ' will occur on a descendant node–assuming that the index corresponding to transition τ is not an element of the immutable set at this node. The lower bound computed at a node in Step 5 is valid only for descendants of this node and is a value below which the objective function cannot fall while proceeding along this branch. We know this to be true by the uniqueness shown in Remark 5.4.3 and above) and by Lemma 7.2.5.

Finally, in Step 6, we identify new incumbent solutions. Hence, whenever a new controllable and feasible substructure $M_K(Y_1, \ldots, Y_n)$ is identified with lower maximal cost than the current incumbent, it is deemed the new incumbent solution.

Thus we have shown that in the absence of Steps 2-4 of the branching process in Algorithm $\mathfrak{C}$ we would enumerate all possible substructures of M_K. We then showed that Steps 2-5 allow us to implicitly enumerate a number of branching steps, thus reducing the algorithm running time. Hence, it follows that if Algorithm $\mathfrak{C}$ returns a solution it is optimal and controllable and satisfies Π. Thus, it must be the solution to Problem 7.1.1. □

7.3 Computational Analysis of the Algorithms

We show the computational complexity of evaluating a single branching step in the branch-and-bound tree is in NP. We also provide two tables showing our computational experience with this algorithm, indicating that where possible, parallelization for exploring the branch-and-bound tree would be required to tackle large scale problems.

7.3.1 Computational Complexity Upper Bound

Theorem 7.3.1. *Each problem solved at any node of the branch-and-bound tree of Algorithm* $\mathfrak{C}$ *is at least in* NP*.*

Proof. We know that the Boolean satisfiability problem is NP-complete. Hence, if at any time we must check whether or not there is a substructure of M_K that satisfies a set of Boolean constraints, this is a Boolean satisfiability problem and hence for arbitrary structures M_K, it is NP-complete. We may choose any arbitrary structure for M_K by setting $L = \mathfrak{L}_{\mathcal{M}}(G) = \Sigma^*$ and $M = M_K$. Hence, we may code an arbitrary graph structure in M_K. It follows at once that we may code an arbitrary Boolean satisfiability problem using this structure.

We have already shown that the sub-algorithms used in this algorithm

1. Algorithm $\mathfrak{A}$ (Theorem 3.5.1)

No. States	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Mean
3	7	6	9	12	12	8	7	8.7
4	38	42	28	28	41	39	44	37.13
5	173	132	256	118	40	165	146	147.5
6	613	636	686	447	747	714	321	594.86

Table 7.1. Running times (seconds) for Java Application executing Algorithm $\mathfrak{C}$.

2. Algorithm $\mathfrak{B}$ (or Algorithm $\mathfrak{D}$) (Theorem 5.2.7)

3. Algorithm $\mathfrak{E}$ (7.2.4)

are either P-complete or at worst NP-complete. Hence, each branching step at is at worst an NP-complete problem. □

7.3.2 Computational Experience

We explored executing this algorithm in two contexts. In the first case, we used a trim finite state plant model G that was randomly generated. We used a randomly generated deterministic pushdown machine M with $\mathfrak{L}_{\mathcal{M}}(M) = \mathfrak{L}_{\mathcal{M}}(G)$. Hence, $M_K = M$ at the root of the branch-and-bound tree. In this case, the number of states of M was identical to the number of states of G. Algorithm $\mathfrak{C}$ was coded in the Java programming language. Each state of G had three transitions defined at it. We varied the number of states in the finite state machine model and used the branch-and-bound algorithm to identify the optimal substructure of M. The running times for varying numbers of states in the plant model G is shown in Table 7.1. In all cases, we used no graph logic constraints. That is, $\Pi = \emptyset$.

We also implemented Algorithm $\mathfrak{C}$ in C++ only focusing on finite state structures. In this case, the running times are significantly faster than those for Java. This is most likely do the simpler control checking process for finite state machines and the fact that C++ is a compiled language. Table 7.2 shows the results for this case.

As is clear from the results, problem specific streamlining of Algorithm $\mathfrak{C}$ may be necessary to solve large-scale problems.

No. States	Run 1	Run 2	Run 3	Mean
5	0	1	0	0.33
7	0	0	1	0.33
9	1	2	4	2.33
11	14	9	9	10.67
13	49	54	50	51
15	197	202	206	201.667
17	826	796	791	804.33

Table 7.2. Running times (seconds) for C++ Application executing Algorithm $\mathfrak{C}$ using only finite state machines for M_K.

7.3.3 Streamlining Operations for Executing Algorithm $\mathfrak{C}$ in the Large Scale

The running times shown in Tables 7.1 and 7.2 indicate that algorithm streamlining may be necessary to tackle large-scale problems. In our experience, the most complex and time consuming part of the algorithm was controllability checking, even in the discrete event case. We provide three recommendations for enhancing the speed of Algorithm $\mathfrak{C}$ if no specific knowledge of G and M_K is available.

1. (Streamlining Cost Computation): Since we must re-compute the cost at each branch-and-bound node, it is useful to store information on the transitions used by the maximal (and minimal) cost strings. When branching, if these transitions are disabled, then recomputing maximum and minimum costs is necessary. If not, then the costs need not be recomputed. Even though, computing cost did not dominate the time in our computational experience, for large values of β and small values of ϵ, this can become a concern.

2. (Streamlining Boolean Constraint Checking) Since we solve a satisfiability problem at each branch-and-bound node, we can store the solution to this problem. If, at any branch, we do not remove a transition that was enabled in the solution to this problem, then clearly, the solution is still valid and we may skip Branching Step 4 of Algorithm $\mathfrak{C}$.

3. (Streamlining Controllability Checking): The major subroutines used in checking controllability are intersection and checking the uselessness of states and transitions. The problem of intersection is quadratic in the number

of states of the plant and controller machine and is difficult to streamline. Checking for useless states and transitions can be sped up by implementing a parallel checking system. That is, using a multiple processor architecture with shared memory, we can check multiple states and transitions simultaneously. We attempted to simulate this with a multi-threaded algorithm and in our experience this decreased the time to check controllability by between 10% and 50%. Since control checking and analysis was *the most* time consuming process in our tests, we expect that parallelization of this step may be the only way to attack large scale real-world problems.

7.4 Example

In this section, we show a simple example of the branch-and-bound tree produced when analyzing a small controller. For simplicity, we assume the plant language $L = \mathfrak{L}_{\mathcal{M}}(M_K(1, 1, \ldots, 1))$, the specification. That is, we do not have to construct $M_K(1, 1, \ldots, 1)$ from some initial specification M as in Algorithm $\mathfrak{D}$. This also ensures controllability at the root node of the branch-and-bound tree. We assume $\Sigma_c = \{b_1, b_2, c_1, c_2\}$.

The base controller $M_K(1, 1, \ldots, 1)$ is shown in Figure 7.2 We have appended cost information to the transitions in M_K. There are four variables $X_1, \ldots, X_4$ where X_1 corresponds to the transition labeled b_1 connecting State 1 and State 0; X_2 corresponds to the transition labeled c_1 connecting State 1 with State 2; X_3 corresponds to the transition connecting State 4 with State 3 labeled b_2; and X_4 corresponds to the transition connecting State 4 with State 5 labeled c_2. The resulting branch-and-bound tree is shown in Figure 7.3. In this problem, we assume that $\Pi = \emptyset$. In the branch-and-bound tree, we show the variable values to the left

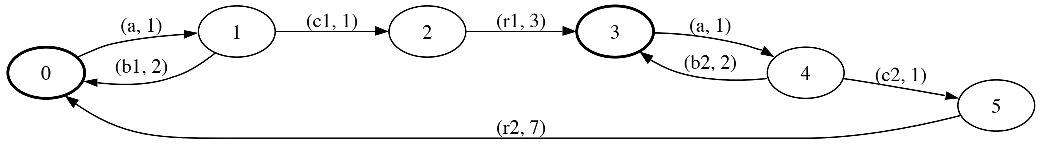


Figure 7.2. The controller M_K that begins the branch-and-bound process.

of the tree nodes and we show the corresponding transition that is removed along the edges of the tree. The shade of the node indicates whether the node is feasible or fathomed. If it is feasible, the shade indicates whether a new incumbent solution is defined at this point. If the node is infeasible, the shade indicates why the node was fathomed. The resulting solution is shown in Figure 7.4

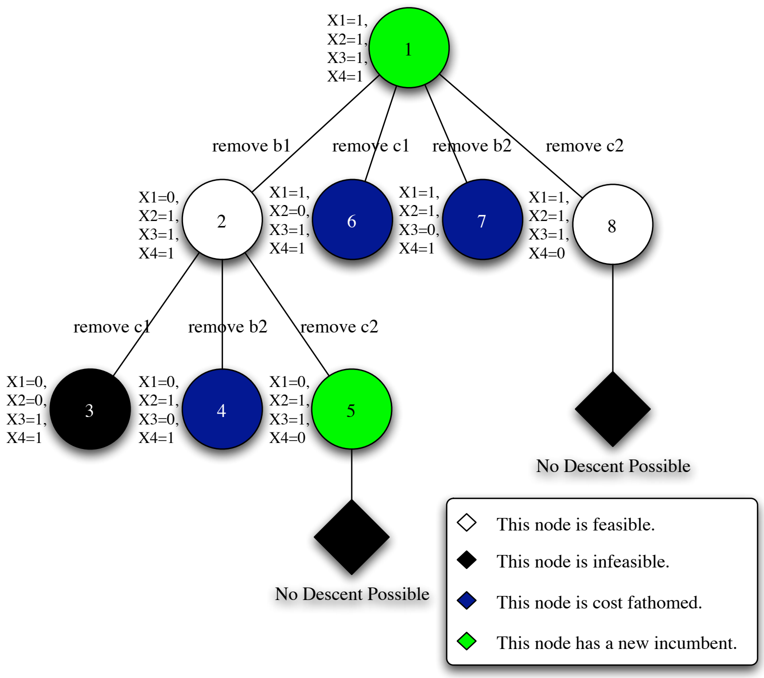


Figure 7.3. The branch-and-bound tree resulting from Algorithm $\mathfrak{C}$ when running on the problem given above.

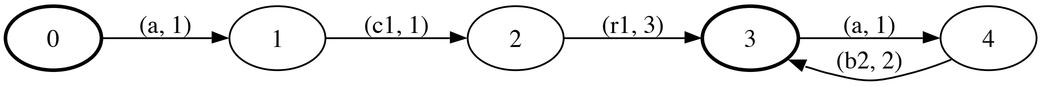


Figure 7.4. Solution to problem given above.

7.5 Relation to the Work of Sengupta and LaFortune

The problem Sengupta and LaFortune investigated was fundamentally different than the one we investigate.

1. [SL98] assumes a plant model G is provided. The objective of the algorithms presented in [SL98] is to identify the optimal substructure of G that can act as a *non-blocking* controller. A controller is non-blocking if it does not prevent the plant from reaching a final state during its execution. In the case they study, the language of the identified substructure is controllable if and only if it is non-blocking.

2. [SL98] does not assume any a priori controller design problem; i.e., they do not assume the existence of a controller machine M_K. Nor do they impose a second set of logical constraints (i.e., there is no set Π).

It is clear that our problem reduces to the problem studied in [SL98] when we make the following simplifying assumptions.

1. The plant model (G) is a finite state machine and M_K is a finite state machine.

2. The base structure M_K is identical to the plant model G.

3. $\Pi = \emptyset$

Though Sengupta and LaFortune discuss the possibility that they can extend their approach to infinite state machines, it is not clear how their solution method would work in this case.

We mentioned that [SL98] uses a dynamic programming (DP) approach to solve an optimal discrete event control problem. We do not use a DP approach for two reasons:

1. [SL98] assumes no a priori controller structure and operates solely on substructures of the plant. As we have already mentioned this is not the problem we study in this chapter.

2. A DP analysis relies on a knowledge of the termination states. In this case, the potential termination states are given by the final states *and* all possible stack configurations and hence form an infinite discrete (hence disconnected) set. The branch-and-bound method proposed here seems to be a more straight-forward approach to solving this problem. However, an open question that results from this research is whether DP can be applied to solve the optimal parametric control problem we've posed.

7.6 Conclusions

In this chapter we defined the optimal parametric control problem for discrete event control. We provided a branch-and-bound algorithm that solves this problem when the parametric control function $M_K(X_1, \dots, X_n)$ has range properly contained in a set of DPDA and the depreciation constant $\beta < 1$. We showed that this algorithm was convergent and that each branching action was at worst an NP-complete problem.

There are a number of potential future directions for this research. The problem of optimal control when $\beta = 1$ is not considered in this work. Further, we've only provided a numerical approximation method for computing the objective function at each node in the branch-and-bound tree. There are conditions where an exact function can be computed. An analysis of these cases would significantly enhance the speed of computation. Finally, we've also discussed many ways of speeding up the branch-and-bound process using multi-threading or multi-processor systems. A computational analysis of this approach would show the potential for this branch-and-bound problem to be used in large scale problems.

Chapter 8

Exception Handling Control and an Application to Robotic Controller Design

8.1 Introduction

Ramadge and Wonham style discrete event control is the study of the fight against uncontrollable events. At its very core, a specification $S \subseteq \Sigma^*$ should simply be a collection of rules for dealing with uncontrollable events in an attempt to drive a plant model to produce a *desirable* sequence of symbols by enabling specific controllable events. Finite state models of specifications have limited memory, which is determined by the number of states present in the system. Hence, when a number of uncontrollable events occur sequentially, responses to each must be stored by encoding the information in the state space of the specification S.

Example 8.1.1. Suppose that an uncontrollable event $u \in \Sigma_u$ indicates that an order has been placed in a manufacturing system. Each order requires a sequence of events (in Σ_c) to occur on an assembly line. If multiple orders are placed simultaneously, we must record this fact in the states and transitions of a state machine describing a specified behavior. The number of states in the state machine limits the maximum number of simultaneous orders that can be recorded. If too few states are available an order is lost. If too many states are available, the model suffers from state explosion and is not parsimonious.

The use of a DPDA can reconcile many of these problems. Using the stack as a storage mechanism, we can record uncontrollable events as they occur. The control mechanism can then respond to these events by enabling subsets of controllable events. Once a response is complete, the next uncontrollable event can be dealt with. We formalize the concept of using the stack of a DPDA to remember and handle long strings of uncontrollable events as an *exception handling controller* (EHC).

8.2 Exception Handling Controllers

An EHC is a DPDA with the following state types:

1. Execution States (required),
2. Response States (required),
3. Stack Modification States (optional),
4. A single Start State (required) and
5. A single Stop State (optional).

If no stop state is given, the controller must execute indefinitely.

For each controllable event $c \in \Sigma_c$ there is a corresponding stack symbol $C \in \Gamma$. Additionally, there may be a finite, but arbitrarily large, number of auxiliary stack symbols $Z_0, \dots, Z_n$. Then $\Gamma = \bigcup_i \{Z_i\} \cup \bigcup_{c \in \Sigma_c} \{C\}$. The transition structure of an EHC has the following properties

1. If q_0 is a start state, then there is a transition with the form $(q_0, \epsilon, q_E, Z_0, \gamma Z_0)$ in the set δ for some executor state q_E.
2. If q_E is an executor state, then for some (possibly empty) set of controllable events $E \subseteq \Sigma_c$, $\delta(q_E, c, C) = (q'_E, \epsilon)$ or $\delta(q_E, c, C) = (q'_E, C')$ for some other executor state (or the Stop State) q'_E and for each $c \in E$.

3. If q_E is an executor state, then for some $u \in \Sigma_u$ $\delta(q_E, u, Z)$ is defined for some $Z \in \Gamma$ and further, $\delta(q_E, u, Z) = (q_R, Z)$ where q_R is a response state. Further, for any Z_i, if $\delta(q_E, \epsilon, Z_i, q'_E, \gamma)$ is defined, then there are no other transitions from q_E with top stack symbol Z_i and q_E is an executor state.

4. If q_R is a response state then there is at least one transition of the form $(q_R, \epsilon, Z, q', \gamma)$ where q' is a stack modification state *or* an execution state.

5. If q_M is a stack modification state, then there is at least one transition of there form $(q_M, \epsilon, Z, q', \gamma)$ for some $Z \in \Gamma$ and q' is either *different* stack modification state *or* an execution state.

6. If q_S is a stop state, then there are a set of looping ϵ-transitions at q_S that empties the stack.

In an EHC, every executor state is final and Z_0 is the initial stack symbol.

Figure 8.1 shows the general structure of an EHC.

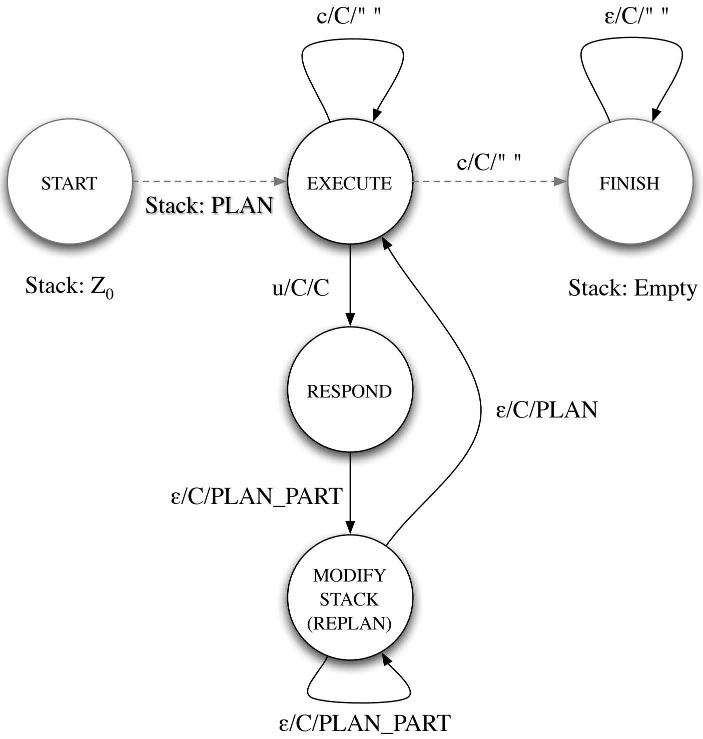


Figure 8.1. Exception Handling Controller–General Structure

Remark 8.2.1. We will implicitly assume that the ϵ-transitions associated with the stack modification states produce a consistent path to return to an execution state. That is, we assume that there is no chance of deadlocking in one of these states because of improperly formed stack operations. This fact is easily checkable by verifying that the push and pop operators defined in the ϵ-transitions provide a path entering and leaving these states and by ensuring that if the top stack symbol is unknown, all possible top stack symbols are considered in the transitions.

8.3 Properties of Exception Handling Controllers

Theorem 8.3.1. *If M is an exception handling controller accepting $\mathcal{L}_{\mathcal{M}}(M)$, then $\mathcal{L}_{\mathcal{M}}(M)$ is prefix closed.*

Proof. Let $w \in \mathcal{L}_{\mathcal{M}}(M)$. By definition, the only accepting states of M are execution states. Therefore, there is a series of ϵ transitions taking w from some state that is not an execution state to a state that is an execution state. Suppose that $w = sa$, where $a \in \Sigma$. Suppose that $(q_0, s, Z_0) \vdash^* (q, \epsilon, \gamma)$. If q is an executing state, then s is accepted. Suppose that q is not an executing state. If it is a response state, then there is a series of ϵ-transitions leading back to an accepting state, and hence s is accepted. If, q is a stop state, then there is no string extending s and hence w does not exist. By assumption, we know that if q is a stack modification state, then there is a series of ϵ-transitions leading back to an accepting execution state. Hence s is accepted. Trivially, $\epsilon \in \mathcal{L}_{\mathcal{M}}(M)$ by the definition of exception handling controller. This completes the proof. □

Theorem 8.3.2. *If M is an exception handling controller, then a machine M^c accepting $\mathcal{L}_{\mathcal{M}}(M)^c$ is derived from M by:*

1. *Adding a single state (*DUMP*).*
2. *If q_E is an executor state and $a \in \Sigma$ is not defined for a stack symbol $A \in \Gamma$, then define $(q_E, a, A, \text{DUMP}, A) \in \delta$, the transition function.*
3. *For each executor state q_E declare q_E non-final and*
4. *Declare* DUMP *final.*

5. *For all* $a \in \Sigma$, $A \in \Gamma$ *define* $(\mathrm{DUMP}, a, A, \mathrm{DUMP}, A) \in \delta$.

Proof. Let $L = \mathcal{L}_{\mathcal{M}}(M)$. We will show $L^c \subseteq \mathcal{L}_{\mathcal{M}}(M^c)$. Choose $w \in L^c$. There is a shortest prefix of w such that $w \in L$. Let $w = st$, where s is this prefix. We have already shown that L is prefix closed and further that any accepted string must lead to an executing state. Since the structure of M^c is identical to that of M except when M is undefined at executing states, it follows that $(q_0, s, Z_0) \vdash^* (q_E, \epsilon, \gamma)$ where q_E is an executor state. Let $t = av$, $a \in \Sigma$ and let $\gamma = A\gamma'$. Trivially, if a transition at a were defined at q_E with stack symbol A, then s would not be the shortest prefix of w accepted by M. Hence, when reading w with M^c, there will be a transition to DUMP on a. The definition of the transitions at the dump state ensures that w will be accepted by M^c. Hence $L^c \subseteq \mathcal{L}_{\mathcal{M}}(M^c)$.

We will now show that $\mathcal{L}_{\mathcal{M}}(M^c) \subseteq L^c$. Choose $w \in \mathcal{L}_{\mathcal{M}}(M^c)$. By definition, w is accepted at DUMP and there is a longest prefix $w = st$ such that $s \in L$; without loss of generality, let $t = a$, $a \in \Sigma$. If $sa \in L$, then there is a transition in M at some state q_E defined for a and an appropriate stack symbol. Clearly, this contradicts our definition for M^c, which is still deterministic. Thus, it follows that $w \notin L$. Thus we have shown that $\mathcal{L}(M^c) = L^c$. □

Definition 8.3.3. A *quasi-exception handling controller* is a deterministic pushdown machine with all the properties of an exception handling controller *except* not all execution states need be final.

Theorem 8.3.4. *If* M *is an exception handling controller accepting* $S = \mathcal{L}_{\mathcal{M}}(M)$ *and* G *is a finite state machine (with no non-deterministic) transitions accepting* $L = \mathcal{L}_{\mathcal{M}}(G)$*. Then* $S \cap L$ *is accepted by a quasi-exception handling controller.*

Proof. The proof is a straight-forward consequence of intersecting a deterministic pushdown machine and a finite state machine. (See Chapter 2, Section 2.7.6.) □

Remark 8.3.5. The prefix closure of a quasi-exception handling controller is obtained by marking every execution state. Hence, this fact combined with Theorems 8.3.2 and 8.3.4 will be important for efficiently computing the controllability predicate.

Furthermore, taking complements in quasi-exception handling controllers is precisely like taking complements in exception handling controllers except we reverse the marking on the executor states; i.e., some executor states may be non-final. These become final in the complement.

Remark 8.3.6. In a quasi-EHC, there is no substructure that can cause an infinite number of ϵ-transitions to occur. This is clear since ϵ-transitions are only used to modify the stack after uncontrollable events have occurred.

Theorem 8.3.7. *Let G be a plant model, M an exception handling controller and M_K the result of intersecting M and G. Let $K = \mathcal{L}_{\mathcal{M}}(M_K)$ and $L = \mathcal{L}_{\mathcal{M}}(G)$. Let N be a machine accepting $\overline{L} \cap \overline{K}^c$. Then K is controllable with respect to L if and only if there is no uncontrollable transition leading from a non-final state of N to a final state of N.*

Proof. ($\Rightarrow$): Necessity is a trivial consequence of Theorem 3.3.2.

($\Leftarrow$): Sufficiency is established by showing that whenever M_K is uncontrollable there is an uncontrollable transition connecting a non-final state of N to a final state of N by an uncontrollable transition. By Theorem 8.3.2 when computing the complement of an EHC, it follows that no ϵ-transitions will lead from the non-final states to the final states. Intersecting such a complement with a finite state machine will not change this fact. Thus, it follows at once that if there is any Σ_u reverse path in N, it must be a single uncontrollable transition. □

8.4 Types of Stack Modification Operations

Stack modification occurs in the stack modification states and is in response to uncontrollable events. (It is possible to use the stack markers $Z_0, \dots, Z_n$ to force re-planning as well using an ϵ-transition from the execute state however, we will not discuss this in detail.) There are three types of modification that can occur in response to an uncontrollable event:

1. Erasure: In an erasure operation, part of the existing stack is irretrievably erased. The "planned" controllable events corresponding to this portion of the stack may not be executed.

2. Write: In this case, new stack symbols are written directly above the existing stack symbols. Any plans currently in operation are suspended and a new plan begins execution.

3. Shuffle Write: Using finite state storage, a portion (whose length is proportional to the amount of finite state storage available) is removed from the stack. A new stack is appended below this plan and then the previously executing plan is rewritten to the stack.

We give an example of conditions where each of these operations is rational:

Erasure The controller is directing the top-level activities of an air campaign. The uncontrollable event indicates that a target is no longer of importance. Existing plans to attack it are erased.

Write Your controller is directing high-level behavior of a robotic rover. A message is received from an operator indicating that a soil sample must be taken at present coordinates immediately. All plans currently in operation are suspended and a new plan is executed.

Shuffle Your controller is directing the behavior of a manufacturing system. Requests must be kept in relative order, but all requests must be processed (hence infinite storage memory may be required to avoid state explosion). New orders are placed as far down on the stack as possible to simulate a FIFO queue.

Note in the Shuffle example, a true FIFO queue *cannot* be simulated with a single pushdown stack and a finite number of states. Two pushdown stacks are required. A machine with two pushdown stacks is equivalent to a Turing machine and hence by the undecidability results given in Theorem 3.6.2, arbitrary control systems of this type cannot be verified for controllability.

8.5 Application of the EHC Parametrization to Robot Control

The properties of quasi-exception handling controllers make them ideal candidates for optimal parametric discrete event control:

1. Taking their complements is simpler than for arbitrary deterministic pushdown machines. In a sense they are almost regular.

2. Uncontrollability can be checked without creating a predicting machine [HU79] as shown in Theorem 8.3.7.

3. Executing Algorithm $\mathfrak{B}$ ($\mathfrak{D}$) is also simpler as a result of Theorem 8.3.7.

4. It is clear from the algorithms presented so far, the class of quasi-EHC structures is closed under Algorithms $\mathfrak{B}$, $\mathfrak{D}$ and thus $\mathfrak{C}$.

8.6 Livingstone Models and Robotic Software Verification

We now show a very simple example of Algorithm $\mathfrak{C}$ for obtaining an optimal quasi-EHC structure for robotic control. This quasi-EHC is checked for controllability against a finite state model of robotic operation. Checking for controllability is a form of *model validation* [CGP99].

Static model validation is the process by which a formal system model is checked for certain properties. Models having necessary properties are validated. Model validation is usually carried out by means of an automated theorem prover (ATP) operating on a pre-specified logic– sometimes Computational Tree Logic (CTL) [GHR99] though in the case of the Prototype Verification System (PVS), a higher order logic (HOL) [OSRSC01] is used. Livingstone is a labeled-transition-system-based modeling language use by the National Aeronautics and Space Administration (NASA) for modeling and validating critical systems used on spacecraft. Livingstone specifications have been used to model several NASA applications including e.g. the propulsion system for the now discontinued X-34 space plane. Labeled transitions systems are finite state machines and hence Livingstone models may be considered plant models in the Discrete Event Control Paradigm.

There are well known translations of Livingstone Models to SMV models [NP02, Pec00] . SMV [CCO$^+$05] is an ATP for use with CTL and its derivative Linear Temporal Logic (LTL) [GHR99]. Most Livingstone models are evaluated by translation to SMV.

We can use discrete event control formalisms for model validation without requiring the use of a cumbersome and generic ATP. We use the underlying automata theory inherent in CTL to test safety properties of system models. We have shown that safety properties provable in CTL are equivalent to controllability properties of discrete event control systems [GMT05]. Specifically, we show that the predicate formula AG(¬unsafe), which says that for all paths starting from a given state, and for each state in the path, no state is unsafe, is equivalent to a specialized formulation of the controllability predicate. This sentence is used most often in security validation [PD02, SW03, JSW02a, SHJ$^+$02]. We have shown in [GMT05, DGP$^+$06] that this approach to model checking can be used for verifying both software/network systems and communications protocols. We have also shown that for certain safety properties we can verify properties of systems that can be modeled as pushdown automata–a class of infinite state automata [Gri06, Gri07]. Types of systems that can be modeled as pushdown automata include many programming languages and exception handling systems. This makes them ideal for validating artificial intelligence systems that execute exception handling.

As a cost savings mechanism, a space agency will reduce the support staff for its interplanetary rovers. A control system on the rover will handle requests that are transmitted from an orbiter. These requests are received from Earth and are produced by mission scientists using a web interface. The control system on the rover must be validated to ensure that no requests will lead to deadlock (thus disabling the rover); controllability is essential in this example. The mission engineers designing the controller wish to optimize it in terms of power use. This scenario is shown in Figure 8.2.

Consider the plant model shown in Figure 8.3

This plant model describes the functioning of a robotic rover that can receive requests for operations from a web-service. The rover can execute the following

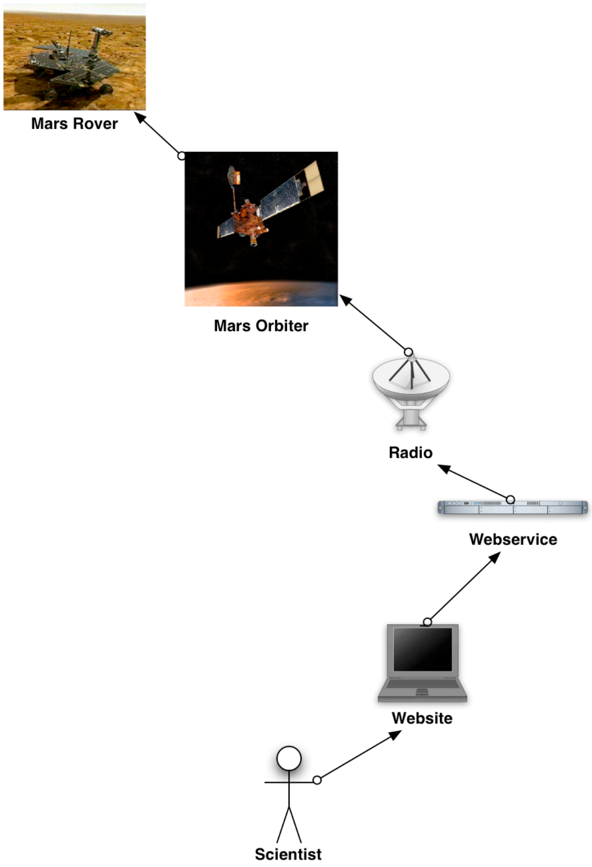


Figure 8.2. The flow of information in our scenario.

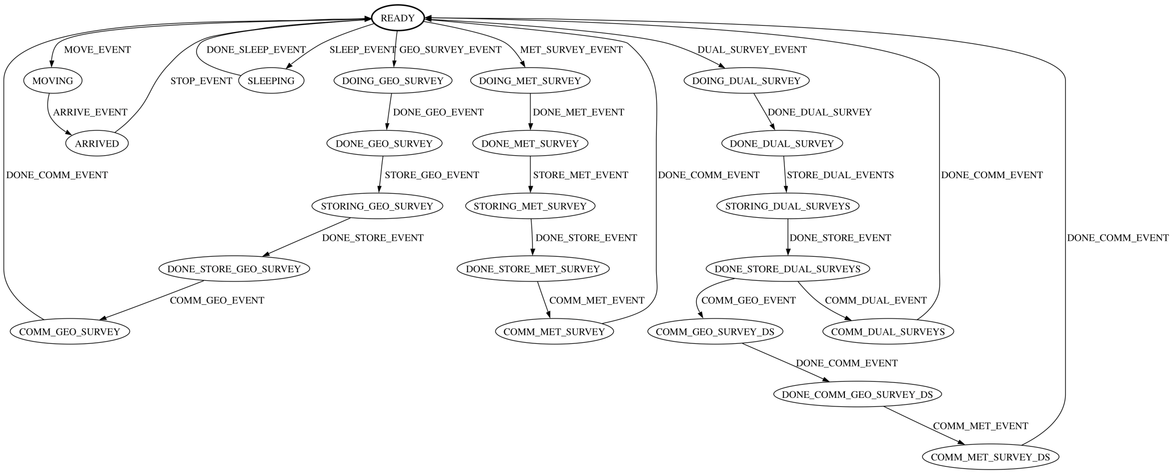


Figure 8.3. Robotic rover plant model.

behaviors:

1. Move to a specific position
2. Execute a geologic survey
3. Execute a meteorologic survey
4. Sleep and recharge its battery,
5. Store and communicate the data that has been collected.

Several requests can be made, including a request for both a meteorological and geological survey. This can be satisfied either by simultaneously executing the surveys, which uses more power but takes less time, or by executing them in series. Similarly, the results of a dual survey can either be communicated simultaneously or in series.

To store an arbitrary number of requests, a stack structure must be used, otherwise it is possible that some request will not be processed. The exception handling mechanism is shown in the abstract in Figure 8.4.

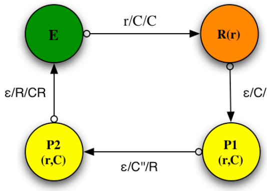


Figure 8.4. Exception handlers recognize specific requests and store them on the stack below the currently executing command.

The quasi-EHC controller (shown in Figure 8.5 has a number of executor states (all states shown). The response and stack modification states have been suppressed for readability, but have the form shown in Figure 8.4.

From any executor state a number of uncontrollable *request events* may be fired. When this occurs, the top stack symbol is removed (transition $\epsilon/C/\epsilon$) and replaced by the symbol CR, where R is the command corresponding to the request r. In Figure 8.4 separate transitions would have to be provided for each of the possible values of C.

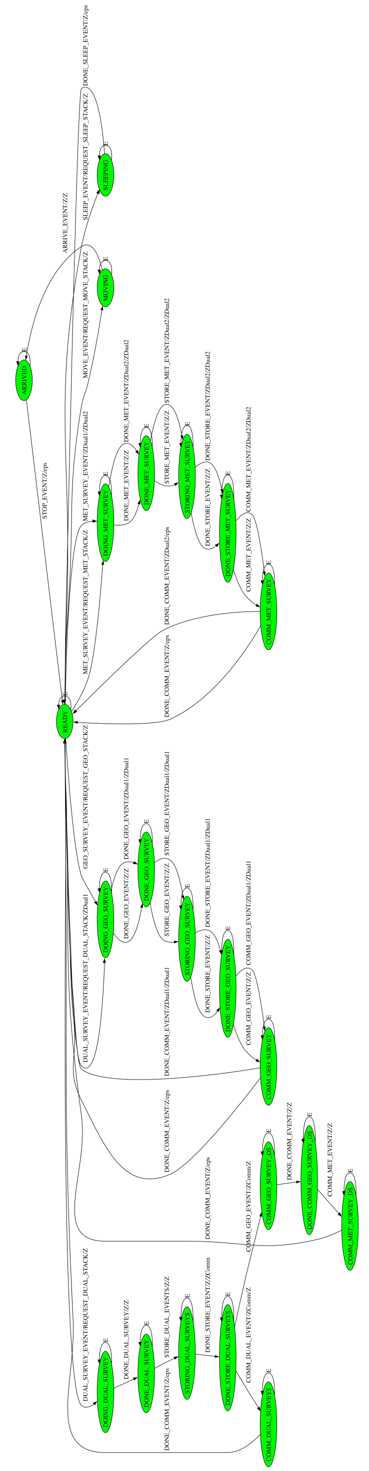


Figure 8.5. An quasi-EHC with looping exception handling transitions suppressed for space.

In this quasi-EHC, we allow the state to be defined by the requests that have come in (the stack) and by the action currently being performed by the controller (e.g., geologic survey, move, store data etc.). We use a top stack symbol (Z) to indicate an existing mission. Hence from the `READY` state the transition

```
GEO_SURVEY_EVENT/REQUEST_GEO_SURVEY_EVENT/Z
```

indicates that mission mode is activated and the controller will transition to states corresponding to executing a geologic survey. The next appropriate mission controllable events are determined by state structure. This makes the EHC easier present and restricts the stack to recording requests only. During a dual task mission, communications activity is the auxiliary stack symbol $ZComm$. indicates that a communications mission is to be performed. The stack symbols $ZDual1$ and $ZDual2$ are used to indicate that dual survey mission is to be performed. The symbol $ZDual1$ indicates the surveys are performed simultaneously (more power consumed, but less time). The symbol $ZDual2$ indicates the surveys are performed in serial (longer time, but more power consumed).

The current mode (e.g., performing a survey, ready for next command, etc.) is given by the top stack symbol. The EHC maintains this top stack symbol until a specific mission is complete. In the case of a geologic survey, completion occurs when communication of the geologic survey data is transmitted and the Z symbol is popped off the top of the stack. The next request can then be read in and operation will continue. This effectively allows the controller to block on any new missions until the current mission is complete, instead storing the mission requests in its stack. Note, when using the storage structure shown in Figure 8.4, the requested missions will always be processed in reverse order. Adding a more complex exception handling mechanism could allow the system to behave more like a queue.

The branch-and-bound algorithm must now decide whether or not transitions

```
DUAL_SURVEY_EVENT/REQUEST_DUAL_STACK/ZDual1
COMM_DUAL_EVENT/ZComm/Z
```

should be enabled or disabled. Clearly, a branch-and-bound algorithm is *not* needed to decide so trivial a thing, however it helps illustrate the purpose. We have the following constraints that can be written formally and make up the set Π:

- Request geologic survey events are processed by a geological survey event.
- Request Meteorological Survey Events are processes by a meteorological survey event.
- Request dual survey events are processed either by a dual survey event or serial survey events.
- Request move events are processed by a move event.
- Power warn events are processed by a sleep event.

An example of logical expression for Requirement 1 is: *There exists a transition labeled by GEO_SURVEY_EVENT defined at the state READY*, or if q_E is the ready state, then we may write:

$$\exists q \in Q \exists t \in \delta \left(\delta(q_E, \text{GEO\_SURVEY\_EVENT}, \text{REQUEST\_GEO\_STACK}) = (q, Z)\right)$$

where Q is the state set of the machine in question and δ is the transition relation.

We assign the following costs:

Event Name:	**Cost**
GEO_SURVEY_EVENT	2.5
STORE_GEO_EVENT	1
COMM_GEO_EVENT	1
MET_SURVEY_EVENT	1.5
STORE_MET_EVENT	1
COMM_MET_EVENT	1
DUAL_SURVEY_EVENT	5
STORE_DUAL_EVENT	2
COMM_DUAL_EVENT	3

with all other event costs being zero. Clearly moving power should be considered in a real application, but for our example, we need only consider the impact of executing a dual survey request as a series of geologic and meteorological requests or not.

8.6.1 Running the Algorithm

Without loss of generality, we will consider the impact of running the algorithm and consider four specific nodes. (See Figure 8.6.) At the "Root" node there is an incumbent cost of 33.167 because the initial quasi-EHC (shown in Figure 8.5 is controllable and satisfies the logical specifications given in Π. Hence $X_1 = X_2 = \cdots = X_n = 1$ is a feasible solution (but not optimal). This value can be computed using Algorithm $\mathfrak{E}$.

Assume that we assign the following binary variables to enabling or disabling transitions as follows:

1. $X_1 = 0$ corresponds to disabling the event

   ```
   MET_SURVEY_EVENT/REQUEST_MET_STACK/Z
   ```

2. $X_2 = 0$ corresponds to disabling the event

   ```
   STORE_MET_EVENT/Z/Z
   ```

3. $X_3 = 0$ corresponds to disabling the event

   ```
   COMM_DUAL_EVENT/ZComm/Z
   ```

In the branch-and-bound framework, we first investigate the solution $X_1 = 0$, $X_2 = X_3 = \ldots, X_n = 1$. In this case, when we disable the event corresponding to X_1, the control structure $M_K(X_1, \ldots, X_n)$ is controllable, but violates one of the constraints described in Π. (Specifically, it violates Constraint 2.) Hence, this node is fathomed.

We next attempt solution $X_1 = 1$, $X_2 = 0$, $X_3 = X_4 = \ldots, X_n = 1$. In this case, the control structure $M_K(X_1, \ldots, X_n)$ is uncontrollable and Algorithm $\mathfrak{D}$ will return $\emptyset$. Hence, we fathom this node as well.

We next attempt the solution $X_1 = X_2 = 1$, $X_3 = 0$, $X_4 = X_5 = \cdots = X_n = 1$. In this case, the control structure $M_K(X_1, \ldots, X_n)$ is controllable and no constraints are violated. A new incumbent solution is found with objective value 21.465. We will continue branching on this node until we set the variable corresponding to the event

DUAL_SURVEY_EVENT/REQUEST_DUAL_SURVEY_EVENT/ZDual1

to zero. At this point, we will identify a new incumbent solution with objective value 15.123. The algorithm will continue other branching, but every other node will be fathomed because of uncontrollability or inability to satisfy the constraints given in Π. Hence, for the prescribed event costs, the optimal solution disables both of the events in question. The final branch-and-bound may look something like the tree shown in Figure 8.6.

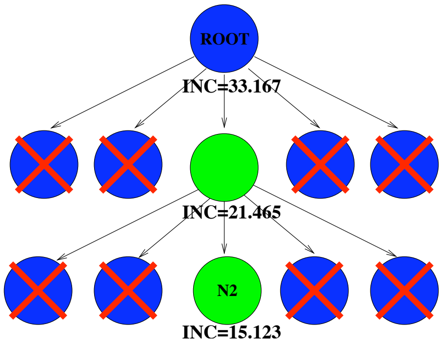


Figure 8.6. The branch-and-bound tree with various fathomed nodes.

The resulting control structure is shown in Figure 8.7. Note the both events have been disabled, leading to a smaller structure.

8.7 Induction Proof of Queuing Controllability

The EHC defined in Figure 8.7 with exception handler 8.4 is clearly controllable. It is easy to see that every execution path ultimately leads back to the ready state where the next requested action can be removed from the stack and acted upon. Clearly, this is true regardless of the order in which we push the requests onto the stack; i.e., suppose we have the exception handling structure shown in Figure 8.8: then the resulting system is still controllable, but we keep the top two stack commands in the correct order before recording a new request. Clearly, for any n we could construct an EHC with appropriate structure that retained the appropriate order for the first n commands on the stack. We now have the following theorem:

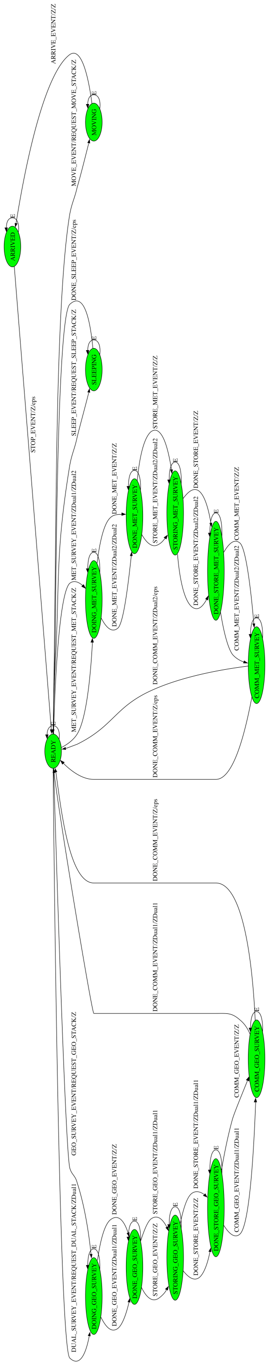

ARRIVE_EVENT/Z/Z
MOVE_EVENT/REQUEST_MOVE_STACK/Z
MOVING
E
ARRIVED
E
DONE_SLEEP_EVENT/Z/eps
SLEEP_EVENT/REQUEST_SLEEP_STACK/Z
SLEEPING
E
STOP_EVENT/Z/eps
MET_SURVEY_EVENT/ZDual1/ZDual2
MET_SURVEY_EVENT/REQUEST_MET_STACK/Z
DOING_MET_SURVEY
E
DONE_MET_EVENT/Z/Z
DONE_MET_EVENT/ZDual2/ZDual2
DONE_MET_SURVEY
E
STORE_MET_EVENT/Z/Z
STORE_MET_EVENT/ZDual2/ZDual2
STORING_MET_SURVEY
E
DONE_STORE_EVENT/Z/Z
DONE_STORE_EVENT/ZDual2/ZDual2
DONE_STORE_MET_SURVEY
E
COMM_MET_EVENT/Z/Z
COMM_MET_EVENT/ZDual2/ZDual2
COMM_MET_SURVEY
E
DONE_COMM_EVENT/ZDual2/eps
DONE_COMM_EVENT/Z/eps
READY
E
DONE_COMM_EVENT/Z/eps
DONE_COMM_EVENT/ZDual1/ZDual1
GEO_SURVEY_EVENT/REQUEST_GEO_STACK/Z
DUAL_SURVEY_EVENT/REQUEST_DUAL_STACK/ZDual1
DOING_GEO_SURVEY
E
DONE_GEO_EVENT/Z/Z
DONE_GEO_EVENT/ZDual1/ZDual1
DONE_GEO_SURVEY
E
STORE_GEO_EVENT/ZDual1/ZDual1
STORE_GEO_EVENT/Z/Z
STORING_GEO_SURVEY
E
DONE_STORE_EVENT/ZDual1/ZDual1
DONE_STORE_EVENT/Z/Z
DONE_STORE_GEO_SURVEY
E
COMM_GEO_EVENT/Z/Z
COMM_GEO_EVENT/ZDual1/ZDual1
COMM_GEO_SURVEY
E

Figure 8.7. The minimum cost quasi-exception handling control system for robotic control.

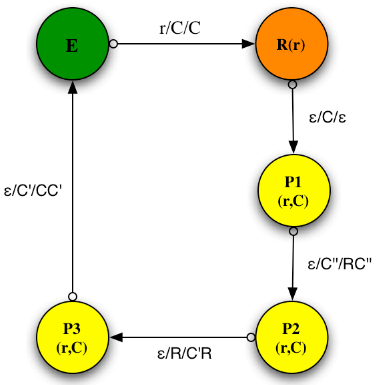


Figure 8.8. Exception handler that preserves the top two stack commands.

Theorem 8.7.1. *For arbitrary n the quasi-EHC with the structure given in Figure 8.7, capable of keeping the top n stack symbols in order is controllable with respect to the plant model given in 8.3.*

Proof. We have argued already that this fact holds for $n = 1$. Suppose it is true for all $n \leq N$, we show it holds true for $N + 1$. Suppose that the stack consists of N symbols and a new request comes in and is placed on the stack. Then this is indistinguishable from the case when there are N symbols on the stack that execute in proper order, clearing the stack and a new request is entered on the stack. By the induction hypothesis, we know that each of these cases cannot produce uncontrollability. Hence, the theorem holds for $N + 1$ and is proved inductively. □

Thus, we can in fact implement a true queue structure with no fear of uncontrollability. This fact, is interesting. A queue machine is strictly more powerful than a stack machine and hence, the undecidability theorem in [Gri06] implies that no general algorithm exists to verify controllability in this case. However, it may be the case that induction methods will allow controllability verification for some classes of queue machines when used with exception handling control structures.

8.8 Conclusions and Future Directions

In this chapter we have defined the concept of exception handling controller. Exception handling controllers are deterministic pushdown machines that are designed to store an executable plan on the pushdown stack. They have computational properties that make them similar to but more powerful than finite state machines. This makes them ideal for use in the branch-and-bound algorithm defined in Chapter 7. We showed an example of their use as a natural extension to the Livingstone models used by NASA for modeling critical systems on deep-space hardware.

In this chapter we also demonstrated how an induction proof could be used to show that a "queue" machine can be proved controllable with respect to a finite state machine plant. This proof shows that it is possible to check controllability for structures more complex than just deterministic pushdown machines and raises the natural question, when can an induction method be used to explore queue machines?

Chapter 9

Conclusion and Future Directions

9.1 Summary

In this thesis we have generated four new algorithms for use in discrete event control. In Chapters 1 and 2 we introduced our main topics and provided key preliminary results needed to motivate the remainder of the thesis.

In Chapter 3 We showed that it is not decidable whether arbitrary specifications generated by PDA are controllable against regular plant models. We have demonstrated the positive result that controllability can be tested for specifications generated by DPDA and regular plant models.

Deterministic pushdown automata can be used to form specifications that are more complicated than regular specifications. In particular, the pushdown stack can be used as a to-do list for future action. In this case, uncontrollable events can be seen to alter the composition of the list causing the controller to re-plan future actions. Furthermore, nested response to uncontrollable events can be encoded using DPDA. For example, suppose that a string w must be enabled each time an uncontrollable event u is observed. Then the resulting behavior $\{u^n w^n | n \geq 1\} \subseteq K$ cannot be specified by a finite state automaton and hence a DPDA must be used. All of these results were published in [Gri06].

In Chapter 4 we demonstrated how discrete event control theory can be used in the design and verification of wireless network protocols, even in the case where certain uncontrollable events cannot be identified until the protocol is deployed. For example, this is the case with unanticipated strategies for attacking the network.

We have also provided a method for determining whether or not a given protocol specified by automata models can cope with all known uncontrollable hazards, and a method of estimating the confidence level for controllability of a protocol in the case where this fails. This approach can be used for designing new protocols for wireless networks that are robust to attack.

In Chapter 5 we developed an algorithm for computing the supremal controllable sublanguage of a language K when K is accepted by a DPDA and the plant language L is accepted by a FSM and both are prefix closed. We further showed that this supremal controllable sublanguage was *also* accepted by DPDA's. As a complement to this, we showed that when K is accepted by arbitrary non-deterministic PDA, it is undecidable whether $\sup_{\mathrm{C}}(K) = \emptyset$ with respect to an arbitrary regular plant language L. When taken with the results of Chapter 3 or [Gri06] this shows that languages accepted by arbitrary PDA are not practical as controller target languages. On the contrary, these results show promise for using languages accepted DPDA and in particular prefix closed languages accepted by DPDA's as control specifications. Finally, we have introduced the concept of Parametric Control in discrete event systems.

In Chapter 6 we showed how to apply the theory of discrete event control to modeling and analyzing security policies. We also showed how our pushdown automaton models of control policies can enhance the modeling power of security policies, by allowing them to count arbitrary numbers of access violations (as in the final example). This extended the original models shown in [GMT05].

In Chapter 7 we defined the optimal parametric control problem for discrete event control. We provided a branch-and-bound algorithm that solves this problem when the parametric control function $M_K(X_1, \ldots, X_n)$ has range properly contained in a set of DPDA and the depreciation constant $\beta < 1$. We showed that this algorithm was convergent and that each branching action was at worst an NP-complete problem.

Finally, in Chapter 8 we defined the concept of exception handling controller. Exception handling controllers are deterministic pushdown machines that are designed to store an executable plan on the pushdown stack. They have computational properties that make them similar to but more powerful than finite state machines. This makes them ideal for use in the branch-and-bound algorithm defined in Chapter 7. We showed an example of their use as a natural extension to the Livingstone models used by NASA for modeling critical systems on deep-space hardware. We also demonstrated how an induction proof could be used to show that a "queue" machine can be proved controllable with respect to a finite state machine plant. This proof shows that it is possible to check controllability for structures more complex than just deterministic pushdown machines and raises the natural question, when can an induction method be used to explore queue machines?

9.2 Future Directions

Plants and Controllers that are DPDA: Neither in this thesis, nor in the published literature, have we said anything about the case when both the specification and plant are given by DPDA. We have reasons to suspect that in this case the controllability predicate is undecidable. However, we are not aware of any proof of this statement.

Supremal Controllable Sublanguage of DPDA Control Languages that are not Prefix Closed: We have left several open questions worth investigation. First, we have not developed an algorithm for computing $\sup_{\mathrm{C}}(K)$ when K is accepted by DPDA and not prefix closed. We have also not discussed any closure properties; i.e., in this case, it may not be the case that $\sup_{\mathrm{C}}(K)$ is accepted by DPDA. We believe that these questions are worth further investigation. Extending this theory would go along way to enhancing the theory of discrete event control.

Partial Observability in DPDA Controlled Systems: There is a good deal of study in partially observable discrete event systems [KGM91]. In such a system, there is a mask on the events generated by the plant so that only some of them are observable. A good deal of work has been done on showing conditions under which controllability can be decided when such an event mask is in place. Naturally, this work only exists for finite state systems and there is no work on deterministic pushdown machine control systems. Thus, a new area of study that extends this work is to investigate partial observability in DPDA controlled systems.

Timed DPDA Systems In [BW92, BW94] Brandin and Wonham introduced the notion of timed discrete event control. In this study, transitions have temporal predicates that prevent them from firing. [CL99] has a detailed description of timed automata. Brandin and Wonham were able to prove an analogous controllability result to the seminal result of Ramadge and Wonham [RW87c]. Extending DPDA based control to timed systems introduces a new level of complexity. Evaluating controllability requires finding the complement of a formal language. Introducing temporal information into the discrete event system makes this process more difficult. A study of timed discrete event pushdown systems as controllers is essential to complete the study begun in this thesis.

Hierarchical Discrete Event Control In [ZW90], Zhang and Wonham introduced the notion of hierarchical discrete event control. In this study, a single plant model is controlled by a supervisor, while the supervisor is itself controlled by a second supervisor. The supervisor's supervisor does not receive a complete picture of the events in the plant, but instead receives an aggregate picture. Aggregation is accomplished by transforming longer symbol strings into single (aggregated symbols). There is a deep theory of hierarchical consistency in multi-layer finite state machines. In this case, plant models and supervisors are modeled as finite state machines. Supervisor supervisors are also modeled as finite state machines. Extension of the DPDA framework to hierarchical discrete event control is difficult. If both supervisors are modeled as DPDA, then the combined system may not be verifiable for controllability. An important area of extension for this work is to hierarchical discrete event control models.

Improvements to the Optimal Control Analysis: There are a number of potential future directions for this research. The problem of optimal control when $\beta = 1$ is not considered in this work. Further, we've only provided a numerical approximation method for computing the objective function at each node in the branch-and-bound tree. There are conditions where an exact function can be computed. An analysis of these cases would significantly enhance the speed of computation. Finally, we've also discussed many ways of speeding up the branch-and-bound process using multi-threading or multi-processor systems. A computational analysis of this approach would show the potential for this branch-and-bound problem to be used in large scale problems.

Two Player Discrete Event Control Games: We have considered the case when there is a single controller acting on a plant. Consider the case when two controllers are acting on a plant simultaneously. The controllable events for Player 1 are uncontrollable for Player 2. Then we can define a natural two player game just as we defined an optimal control problem. A necessary condition for equilibrium is that both players have controllable control languages with respect to the plant model, when it is controlled by the opposite player (a constrained plant model). An interesting and unexplored question is whether or not a player will ever have an advantage if they use a DPDA controller. To solve this problem, we must analyze the various cost structures that could be imposed on the events and the plant structure. It may be that optimal responses to finite state controller strategies are also finite state strategies. It may also be possible that a DPDA response will provide a better response.

Bibliography

[Axe00] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Department of Computer Engineering, Chalmers University of Technology, March 2000.

[BCV90] Charles A. Brooks, Randy Cieslak, and Pravin Varaiya. A method for specifying, implementing, and verifying media access control protocols. *IEEE Control Systems Magazine*, 10(4):87–94, June 1990.

[Bel57] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[BGK$^+$90] R. D. Brandt, V. K. Garg, R. Kumar, F. Lin, S. I. Marcus, and W. M. Wonham. Formulas for calculating supremal controllable and normal sublanguages. *Systems Control Lett.*, 15:111–117, 1990.

[BH93] Y. Brave and M. Heymann. On optimal attraction of discrete-event processes. *Inform. Sci.*, 67:245–276, 1993.

[BJS04] M. Bazaraa, J. Jarvis, and H. Sherali. *Linear Programming and Network Flows.* Wiley-Interscience, 3 edition, 2004.

[BW92] B. A. Brandin and W. M. Wonham. The supervisory control of timed discrete-event systems. In *Proc. 31st IEEE Conference on Decision and Control*, volume 1, pages 1–6, Dec 16-18 1992.

[BW94] B. A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. *IEEE Trans. Automatic Control*, 39(2):329–342, Feb. 1994.

[Cas02] E. Del Castillo. *Statistical Process Adjustment for Quality Control.* Wiley-Interscience, New York, NY, 2002.

[CCO$^+$05] R. Cavada, A. Cimatti, E. Olivetti, G. Keighren, M. Pistore, M. Roveri, S. Semprini, and A. Tchaltsev. Nusmv 2.3 user manual. Technical report, IRST, Povo (Trento), Italy, 2005.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, 1999.

[CL99] C. G. Cassandras and S. LaFortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Boston, MA, USA, 1999.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[Cor03] Microsoft Corporation. SQL server notification services books online: Building notification services applications, February 23 2003.

[Den87] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[DGP03] Sarah Damiani, Christopher Griffin, and Shashi Phoha. Automated generation of discrete event controllers for dynamic reconfiguration of autonomous sensor networks. In *Proceedings of the 48*th *Annual SPIE Conference*, San Diego, CA, USA, August 2003.

[DGP$^+$06] Sarah Damiani, Christopher Griffin, Shashi Phoha, Stephen Racunas, and Christopher Rogan. Verification of secure network protocols in uncertain environments. *International Journal of Wireless Information Networks*, 13(3):221–228, July 2006. Accepted for publication (with revisions).

[FRL02] J. Fu, A. Ray, and C. Lagoa. Unconstrained optimal control of regular languages. In *Proc. 41st IEEE Conference on Decision and Control*, pages 799–804, Las Vegas, NV, USA, December 2002.

[GHR99] Dov Gabby, Ian Hodkinson, and Mark Reynolds. *Temporal logic : mathematical foundations and computational aspects*. Oxford University Press, New York, NY, 1999.

[GMT05] Christopher Griffin, Bharat Madan, and Kishor Trivedi. State space techniques for software security quantification. In *Second International Workshop on Software Cybernetics (in conjunction with COMPSAC 2005)*, pages 83–88, Edinburgh, Scotland, July 25-28 2005.

[Gri06] Christopher Griffin. A note on deciding controllability in pushdown systems. *IEEE Trans. Automatic Control*, 51(2):334–337, February 2006.

[Gri07] Christopher Griffin. A note on the properties of the supremal controllable sublanguage in pushdown systems. *IEEE Trans. Automatic Control*, Accepted April 2007.

[Har80] Harrison. *Introduction to Formal Language Theory*. John Wiley and Sons, Reading Massachusetts, 1980.

[HHL$^+$93] Chung-Ming Huang, Jenq-Muh Hsu, Huei-Yang Lai, Jao-Chiang Pong, and Duen-Tay Huang. An estelle interpreter for incremental protocol verification. In *Proceedings of the International Conference on Network Protocols*, pages 326–333, San Francisco CA, USA, October 19-22 1993. IEEE.

[HL98] J. D. Howard and T. A. Longstaff. A common language for computer security incidents. Technical Report SAND98-8667, Sandia National Laboratories, Livermore, CA, October 1998.

[HU79] Jeffrey Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 2 edition, 1979.

[IR74] H. B. Hunt III and D. J. Rosenkrantz. Computational parallels between the regular and context-free languages. In *Proceedings of the Sixth Annual ACM Symposium on the Theory of Computing*, pages 64–74, 1974.

[Jac85] N. Jacobson. *Basic Algebra I*. W. H. Freeman, 1985.

[JSW02a] Somesh Jha, Oleg Sheyner, and Jeannette Wing. Two formal analyses of attack graphs. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 49–63, 2002.

[JSW02b] Somesh Jha, Oleg Sheyner, and Jeannette M. Wing. Minimization and reliability analyses of attack graphs. Technical Report TR02-109, Department of Computer Science, 2002.

[KG95a] R. Kumar and V. Garg. Optimal supervisory control of discrete event dynamical systems. *SIAM J. Control Optim.*, 33:419–439, 1995.

[KG95b] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[KGM91] R. Kumar, V. K. Garg, and S. I. Marcus. On controllability and normality of discrete event dynamic systems. *Systems Control Lett.*, 17(3):157–168, 1991.

[KH96] Ratnesh Kumar and Lawrence E. Holloway. Supervisory control of deterministic petri nets with regular specification languages. *IEEE Trans. Automatic Control*, 41(2):245–249, February 1996.

[LBMC93] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws, with examples. Technical Report NRL/FR/5542-93-9591, Naval Research Laboratory, September 1993. ACM Computing Surveys, vol. 26, no. 3, pp. 211-254, September 1994.

[Lin03] Tancred Lindholm. Setting up a bluetooth packet transport link, February 23 2003.

[LKJ98] U. Lindqvist, P. Kaijser, and E. Jonsson. The remedy dimension of vulnerability analysis. In *Proceedings of the 21st National Information Systems Security Conference*, pages 91–98, Arlington, VA, USA, October 1998.

[MBL00] H. Marchand, O. Boivineau, and S. LaFortune. On the synthesis of optimal schedulers in discrete event control problems with multiple goals. *SIAM J. Control Optim.*, 39(2):512–532, 2000.

[NP02] S. Nelson and C. Pecheur. Formal verification for a next-generation space shuttle. In *FAABS*, 2002.

[OSRSC01] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. Pvs system guide. Technical Report Version 2.4, SRI International, 2001.

[PA89] K. M. Passino and P. J. Antsaklis. On the optimal control of discrete event systems. In *Proc. 28th IEEE Conf. on Decision and Control*, pages 2713–2718, Tampa, FL, USA, 1989.

[PD02] Jean Philippe Pouzol and Mireille Ducassé. Formal specification of intrusion signatures and detection rules. In *CSFW*, pages 64–76, 2002.

[Pec00] C. Pecheur. Verification and validation of autonomy software at nasa. Technical Report 2000-209602, NASA/TM, 2000.

[PS98] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *NSPW '98: Proceedings of the 1998 workshop on New security paradigms*, pages 71–79. ACM Press, 1998.

[Rab89] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[Rog87] Hartley Rogers. *Theory of recursive functions and effective computability.* MIT Press, Cambridge MA, USA, 1st mit press edition edition, 1987.

[RP02] A. Ray and S. Phoha. A language measure for discrete event automata. In *International Federation of Automatic Control World Congress*, Barcelona, Spain, 2002.

[RW87a] P. J. Ramadge and W. M. Wonham. Modular feedback logic for discrete event systems. *SIAM J. Control Optim.*, 25(5):1202–1218, 1987.

[RW87b] P. J. Ramadge and W. M. Wonham. On the supremal controllable sublanguage of a given language. *SIAM J. Control Optim.*, 25(3):637–659, 1987.

[RW87c] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.

[S9́7] Géraud Sénizergues. L(A) = L(B)? In *Infinity'97: Second International Workshop on Verification of Infinite State Systems*, Bologna, 1997. Elsevier.

[SAFK$^+$03] Tao Song, Jim Alves-Foss, Calvin Ko, Cui Zhang, and Karl Levitt. Using acl2 to verify security properties of specification-based intrusion detection systems. In *Proc. ACL2-2003*, 2003.

[SCS98] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 338–347, 1998.

[SHJ$^+$02] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 273. IEEE Computer Society, 2002.

[Sim00] S. Simpson. Mathematical logic. Available at: http://www.math.psu.edu/simpson/courses/math557/logic.pdf, 2000.

[SL98] R. Sengupta and S. Lafortune. An optimal control theory for discrete event systems. *SIAM J. Control Optim.*, 36(2):488–541, 1998.

[Sre93] R.S. Sreenivas. A note on deciding the controllability of a language k with respect to a language l. *IEEE Trans. Automatic Control*, 38(4):658–662, April 1993.

[SS04] C. R. Shalizi and K. L. Shalizi. Blind construction of optimal nonlinear recursive predictors for discrete sequences. Technical report, arXiv:cs.LG/0406011v1, 2004.

[SSC02a] C. R. Shalizi, K. L. Shalizi, and J. P. Crutchfield. An algorithm for pattern discovery in time series. Technical report, arXiv:cs.LG/0210025 v3, 2002.

[SSC02b] C. R. Shalizi, K. L. Shalizi, and J. P. Crutchfield. Pattern discover in time series, part i: Theory, algorithm, analysis, and convergence. Technical Report Working Paper 02-10-060, Santa Fe Institute, 2002.

[Sti02] Colin Stirling. Deciding dpda equivalence is primitive recursive. In *Proceedings of the 3st International Colloquium on Automata, Languages and Programming*, pages 821–832. Springer-Verlag, 2002.

[SW03] Oleg Sheyner and Jeannette Wing. Tools for generating and analyzing attack graphs. In *Proceeding of the Second International Symposium on Formal Modeling for Components and Objects*, pages 344–371, 2003.

[Tur36] A. M. Turing. On computable number, with an application to the entscheidungs problem. *Journal of the London Mathematical Society, Series 2*, 42:230–265, 1936.

[WR02] X. Wang and A. Ray. Signed real measure of regular languages. In *Proc. American Control Conference*, Anchorage AK, USA, 2002.

[ZW90] Hao Zhong and W. Murray Wonham. On the consistency of hierarchical supervision in discrete event systems. *IEEE Trans. Automatic Control*, 35(10):1125–1134, October 1990.

Vita

Christopher Griffin

Education

- Ph.D. Industrial Engineering and Operations Research, The Pennsylvania State University, (Graduating December 2007)
- M.A. Mathematics, The Pennsylvania State University, (2001-2004)
- B.S. (with High Distinction) Mathematics, The Pennsylvania State University (1997-2000)

Significant Publications (26 publications, 4 pre-prints total)

1. C. Griffin. A Note on the Properties of the Supremal Controllable Sublanguage in Pushdown Systems. *IEEE Trans. Automatic Control*, Accepted April 2007.
2. C. Griffin. A note on deciding controllability in pushdown systems. *IEEE Transactions on Automatic Control*, 51(2):334337, February 2006
3. C. Griffin and R. R. Brooks. A note on the spread of worms in scale-free networks. *IEEE Transactions on Systems, Man and Cybernetics, Part B*, 36(1):198-202, 2006.
4. S. Phoha, T. LaPorta, and C. Griffin, editors. Sensor Network Operations. IEEE Press, (ISBN 0-471-71976-5), May 2006.

Summary of Qualifications Mr. Griffin has a Masters Degree in Mathematics from the Pennsylvania State University and is completing (part-time) a Ph.D. in Industrial Engineering and Operations Research. He is also a full-time Research Engineer with the Penn State Applied Research Laboratory, the largest independent research lab on Penn States campus. Mr. Griffin has worked with the Applied Research Laboratory for almost seven years. He is currently the co-Principal Investigator of the Learning and Prediction for Enhanced Readiness Program an Office of Naval Research Sensor Fusion Program (PM: Wendy Martinez/Gary Toth). He is the Principal Investigator for the IR&D Program Optimal Control Model of Task Execution with Personality. He was the Mathematical Modeling Coordinator for the Critical Infrastructure Protection University Research Initiative (CIP/URI), Mobile Ubiquitous Security Environment (MUSE). The MUSE program created survivable network infrastructures by combining peer-to-peer and mobile code technology. Mr. Griffin also served as the Test Bed Coordinator for the Multi-University Research Initiative Emergent Surveillance Plexus (ESP). ESP developed distributed power aware sensor technologies for target tracking and recognition in noisy environments. He has been a reviewer for IEEE Transactions on Mobile Computers, the International Journal of Distributed Sensor Networks, the International Conference on Distributed Computer Systems, and Elseviers Information Fusion. He is also the co-editor of the book Sensor Network Operations published by IEEE Press.