ONLINE DISCRETE EVENT CONTROL OF HYBRID SYSTEMS

JAMES P. MILLAN

NOTE TO USERS

Online Discrete Event Control of Hybrid Systems

by

©James P. Millan

B.Eng., Memorial University of Newfoundland, 1984

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Faculty of Engineering and Applied Science
MEMORIAL UNIVERSITY OF NEWFOUNDLAND

October 2006

ST.JOHN'S NEWFOUNDLAND

Your file Votre référence
ISBN: 978-0-494-30434-1
Our file Notre référence
ISBN: 978-0-494-30434-1

To my parents,

Steven and Brenda,

and for Les.

Online Discrete Event Control of Hybrid Systems

by

James P. Millan

Abstract

The increasing proliferation of automatic control systems in embedded and distributed applications has lead to increasingly complex systems. These systems manifest a mixture of continuous and discrete dynamics due to the interaction of the computer controlled or logical decision-making subsystems interacting with the real world, and are thus referred to as hybrid systems. The inherent complexity of such hybrid systems makes them difficult to model, analyze and design. As such, industrial application of hybrid system theory has yet to gain widespread acceptance.

This thesis presents an approach to the modeling, synthesis and implementation of automatic controllers for hybrid systems. This work centers on a flexible hybrid system modeling framework that permits automated synthesis of controllers for hybrid systems, based on safety and performance design specifications. This hybrid modeling framework is the switched continuous model (SCM), based on discrete switching between continuous system models (CSM). Discrete abstractions of the CSM dynamics enable the controller actions to be simple discrete decisions at appropriate points in the state space of the controlled system. The SCM communicates with external discrete event systems (DES) through sets of shared discrete events, thus allowing

the techniques of DES supervisory control synthesis to be employed. The resulting controllers are model-based, and safe by design, since they encapsulate the continuous and discrete event models that together model the plant and specification dynamics. Due to the inherently uncountable state space of the hybrid system model, the controller computation is performed online, and is limited to a finite time horizon in order to preserve the finite state properties of the discrete abstraction.

The details of the modeling framework, controller synthesis, and online implementation are developed, including a computational approach, architecture, and algorithms. A software package that implements these control concepts was developed. Two detailed modeling and control synthesis applications are presented: a simple benchmark hybrid control example and a realistic industrial example.

Acknowledgements

I would like to take this opportunity to thank my thesis supervisor, Dr. Siu O'Young for his untiring enthusiasm, optimism and guidance. Siu was instrumental in encouraging me to enter graduate studies and later to pursue my Ph.D.; a somewhat non-trivial task, considering the length of time that had elapsed since I had left university (over 15 years). Although it has been challenging, I have enjoyed the learning and the discovery that comes with graduate work.

I would like to thank my employer, the National Research Council, Institute for Ocean Technology, for giving me the opportunity to change my career direction from an instrumentation and control systems engineer to that of a researcher. I thank Mr. David Murdey and Dr. Bruce Colbourne in that respect. Thanks to my co-worker Dr. Wayne Raman-Nair for many discussions and advice on a wide range of topics that proved to be useful to my thesis work, including solutions to ODEs, numerical modeling, teaching, and how to write a Ph.D. dissertation.

I would like to thank my thesis committee, in particular Dr. Theo Norvell, for their careful reading, and helpful corrections and suggestions.

Finally, and most importantly, I would like to thank my family; my wife Roxanne, daughter Kelsey, and son Jonathan, for their patience and understanding while I spent many hours working, especially in the past few months, as I completed this document.

Contents

List of Tables

List of Figures

List of Abbreviations

CSM Continuous System Model

DES Discrete Event System

DP Dynamic Positioning

ESD Emergency Shut Down

FSM Finite State Machine

FSA Finite State Automaton

FPSO Floating Production Storage and Offloading Vessel

HIL Human In the Loop

HPA Hybrid Product Automaton

HTG Hybrid Transition Graph

IVP Initial Value Problem

JOM Joint Operations Manual

ODE Ordinary Differential Equation

PMS Power Management System

SCM Switched Continuous Model

SCT Switched Continuous Trajectory

Chapter 1

Introduction

1.1 Background

Mathematical models are approximations of the physical world. These models allow us to understand, analyze, predict, and control the physical processes that surround us. The latter task, control, is the subject of this thesis. Traditionally, mathematical models take the form of continuous linear or nonlinear differential equations; this is because the physical processes they model tend to vary in a smooth, continuous manner. Consequently, the vast majority of control theory has been developed for the control of continuous dynamical systems.

With the increasing proliferation of automatic control, and the corresponding increase in the complexity of controlled systems, high-level control functions such as supervision and coordination have become a necessity. As a result of this, an important class of systems, known as hybrid systems, have grown increasingly important. These are systems that cannot be described easily by continuous dynamical models only, and require a model that also incorporates discrete changes of state. Hybrid dynamics are often the manifestation of a discrete decision-making process (i.e. digital control) interacting with a continuous dynamical system. Hybrid behaviour may also arise autonomously if a system switches discretely between multiple modes of

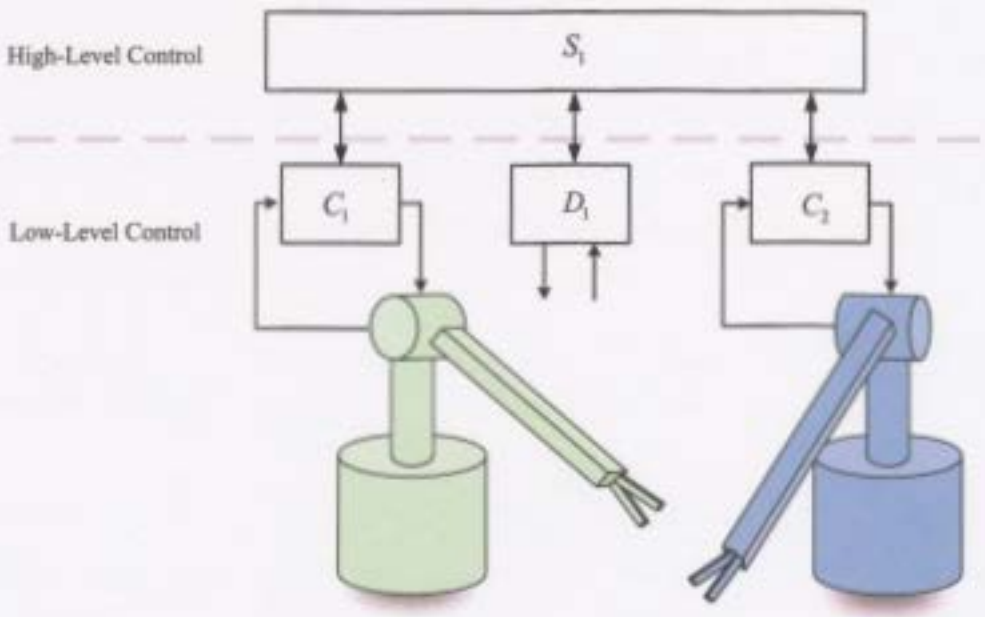


Figure 1-1: A typical complex control system having hybrid dynamics. Low-level continuous control tasks, C_1, C_2 and digital control D_1, are coordinated and supervised by a high-level controller, S_1.

operation.

Many practical control problems lie somewhere in this hybrid "spectrum" – somewhere between continuous and discrete dynamics. Examples include: robotics, process control, autonomous vehicles and reconfigurable manufacturing. The common thread in all of these applications is that there are both continuous and discrete control tasks involved. For example, a continuous dynamical task for a robotic arm may be to precisely follow a motion profile that specifies both a velocity and position through time. A parallel discrete event goal may be that the arm repeat the motion profile a specific number of times, synchronize its actions with a neighbouring machine to avoid collision (mutual exclusion), and avoid a deadlock condition with the neighbouring machine. This situation is illustrated in the example of Fig. 1-1, in which the continuous controllers C_1 and C_2 control the motion of two robot arms. A discrete control system D_1 may be responsible for control of discrete processes such as

opening or closing valves. The supervisory controller S_1 must coordinate and enforce certain behaviours amongst the low-level systems. The mixture of discrete and continuous dynamics makes this a hybrid system. Now suppose the robots are handling a hazardous material that cannot be dropped: this adds a safety-critical aspect to the control task, focusing the need for formalized control design procedures that can be proven to be safe, or error-free.

1.2 Problem Discussion

The modeling, analysis and control of hybrid systems is an open and active area of research. The intent of this research is to develop theory and techniques that can be applied by control system practitioners. As control system designers, the objective is to design provably safe controllers for hybrid systems such as the one described above. In the domain of discrete-event systems, it is possible to exhaustively search very large system state spaces, removing trajectories that lead to unsafe states. And, in the continuous systems domain, it is possible to ensure the stability of controlled systems under a variety of disturbances and uncertainties. Finding a balance between these two disparate, but mutually desirable approaches to hybrid system control is the task at hand. Exhaustive reachability of hybrid state spaces is in general, not possible, due to the uncountable state space. Likewise, input-to-state, and input-to-output stabilization is problematic for even the most simple hybrid systems. The current approaches to the hybrid design problem involve various combinations of continuous and discrete-event modeling, simulation and analysis strategies. In either case, the usual approach is to place more emphasis on one or the other of the types of dynamics; e.g. approximated continuous dynamics combined with discrete switching, or abstracted switching combined with higher-fidelity continuous models. At this time, hybrid analytical and synthesis tools are at a primitive state in comparison to the tools of typical industrial practice. A detailed survey of the theoretical results

for automatic control systems in general, and hybrid systems in particular, is given in Chapter 2. Even if the serious theoretical hurdles of hybrid system control can be reasonably dealt with, a major barrier to adoption of hybrid control system design techniques remains: design tools must be user-friendly and have sufficient utility that designers will choose to use them. Simulation is currently the most widely utilized technique for hybrid control system design. Controllers are simulated in many "ad-hoc" test scenarios to identify and correct failure points in the design. This approach relies on heuristics – the designer's skill, and knowledge of the system, to ensure safety.

1.3 Contributions

To solve the problem described in the previous section, it was necessary to take an approach that was balanced between theoretical and practical considerations. This thesis documents the technique and supporting theory that enables the automated synthesis of supervisory controllers for systems with hybrid dynamics. The contributions are as follows:

Modeling The modeling framework developed in this thesis accommodates embedded continuous simulations, thus enabling control system designers to utilize existing simulation tools. The model, which is based on discrete switching of continuous dynamics, is simple to use and is very expressive for capturing hybrid dynamics. These features are an important step towards gaining acceptance of this technique in industry.

Control Synthesis The controller synthesis technique described in this thesis uses a hybrid system plant model and a discrete event specification to produce a discrete event supervisory controller that is safe by design. Because the controller is implemented online, it can accommodate time-varying plants, and has reduced computational complexity compared to offline controllers, since it is

computed on a limited horizon. This controller can be guaranteed to be safe (i.e. failsafe) always, by inclusion of emergency shutdown states, allowing this technique to be utilized in safety-critical applications.

Computation A software package called HySynth was developed that implements the control theory concepts of this thesis. The software can be used to model, design, synthesize, and simulate online discrete event supervisory controllers, and it helps to demonstrate the various contributions of this thesis including: automated control synthesis for hybrid systems, online operation, failsafe control, embedded simulation, controller complexity reduction, and human in the loop control.

Application The ship control application presented in this thesis marks the first time that hybrid system control synthesis techniques have been described for control of marine vessels. This controller is unique in that it is suited to the incorporation of human in the loop control. This inclusion of the human operator may make this control technique more attractive to implement from an operational and liability standpoint.

1.4 Organization

This document is organized as follows: Chapter 2 contains a review of the literature that is relevant to the topics of discrete event and hybrid systems modeling, simulation and control. Chapter 3 develops a general continuous system modeling framework. Particular attention is paid to the partitioning framework that will be used to produce the discrete abstractions of the continuous dynamics. Chapter 4 introduces the switched continuous model framework, and its discrete graph representation, the hybrid transition graph. In Chapter 5 there is brief review of discrete event controller synthesis. Developed next is the theory to support synthesis of a fail-safe discrete-event controller for a hybrid system. This is based on the synchro-

nization of a switched continuous model of the plant with a discrete event model of a specification. Chapter 6 is an overview of the computational framework that is used to support the modeling, design and online controller synthesis. Chapter 7 examines two applications of the theory; the first is a benchmark hybrid control problem. This simple example serves to illustrate the modeling environment, and through simulation, gives benchmark run time complexity results. The second example demonstrates the control design process for a realistic, industrial control problem. It also illustrates the capacity of the control framework to incorporate heuristics (i.e. human-in-the-loop) control. Finally, Chapter 8 again summarizes the contributions that this thesis makes to hybrid control systems research, and suggests directions for future work.

Chapter 2

Background and Related Work

This thesis is concerned with the control of complex dynamical systems in real time. As such, the background material contained in this chapter is of a diverse nature, encompassing elements of control system design and applications, continuous control system theory, discrete event control theory, hybrid system theory, and the modeling, analysis and simulation of these systems. This chapter is a brief overview of the models, methods and theory developed to support control system design and analysis in these areas, and which are relevant to the results of this thesis.

2.1 Continuous System Modeling and Control

Continuous system modeling has been the dominant paradigm for theoretical and practical developments in control systems during the 20th century. Initially, controllers themselves were mechanical, then electromechanical and finally electronic (excluding the actuators) (Michel 1996). The "classical era" in control theory and practice was developed around frequency domain stability techniques combined with transient response performance analysis. Control system models were based on linear time invariant (LTI) models in a single input/single output (SISO) modeling framework, and control design practitioners had many semi-automated procedures

for synthesizing controllers. Many of these techniques were developed by practicing engineers and the theoretical explanations followed afterwards (Bernstein 2002).

With the advent of the 1960s came the state-space modeling approach of the so-called "modern era" and the ability to model, analyze and design controllers for multivariable or multiple input multiple output (MIMO) systems. The fundamental concepts of state controllability and observability were formally identified by Kalman (Kalman 1960). The state space approach lends itself well to algorithmic (and hence digital computer) implementation. Given an LTI plant model, a Linear Quadratic Gaussian (LQG) controller can be synthesized for the system that is optimal in a least squares sense. Furthermore, the controller is formulated for a stochastically disturbed modeling and measurement environment, so it lends itself well to practical application. In fact, the optimal estimator (the Kalman filter) is widely credited with making possible the first lunar landing of 1969 (p.14 (Grewal and Andrews 1993)).

Initially, there were serious drawbacks with the state-space approach since there was no way to specify stability; and modeling errors could lead to control instability. With $\mathcal{H}_\infty$ control design (Francis, Henton and Zames 1984), the frequency domain approach of the classical control design techniques and notions of input to output stability were developed for multivariable systems; see (Skogestad and Postelthwaite 1993) for an overview. Multivariable control design was further extended to include controller robustness to parametric and structured modeling uncertainty with the advent of μ-synthesis techniques (Williams 1990), (Balas and Packard 1996).

Up to this point we have been dealing with linear system models. With nonlinear system models, the familiar control system tools no longer apply. Nonlinear models exhibit certain phenomena that do not arise in linear systems, including finite escape time, multiple equilibria, limit cycles, deterministic chaotic behaviour, and multiple modes of operation (Khalil 2002). Typically the approach is to linearize the nonlinear system model about some operating point, if this is possible, in order to use the familiar and powerful linear system tools. Unfortunately, there are many classes of

system for which the locally linearized approximate model cannot be used; e.g. this situation might exist if a system by necessity has more than one operating point. For systems like this, gain scheduling (Leith and Leithead 2000) and sliding mode control techniques have seen extensive use in industry (Kaynak, Erbatur and Ertugrul 2001).

2.2 Discrete Event Systems

Discrete event dynamical systems (DES) are characterized by having a state space that is a discrete set and a state transition mechanism that is event driven. Usually DES models take the form of automata or petri nets. Supervisory control theory for DES was developed by Ramadge and Wonham, (Ramadge and Wonham 1987) and (Wonham and Ramadge 1987). Aspects of control that are not possible to specify in the traditional continuous control theory, such as the ordering of events, coordination of multiple processes and enforcement of safety properties became possible with this technique. Specification and plant are both DES and modeled as finite state automata (FSA). Large models can be conveniently constructed by synchronous composition of multiple FSA. Control optimality is achieved by designing a controller that minimizes interference with the plant (minimizing plant event disablement), while enforcing the specification.

Many extensions to the basic supervisory control theory have been developed including limited observation (Lin and Wonham 1988), decentralized supervisory control (Rudie and Wonham 1992), and robustness (Bourdon, Lawford and Wonham 2005). While technically DES have no sense of time, since they are event driven, by addition of integer clocks and special event called *tick*, specifications and plant models can incorporate coarse timing (O'Young 1991) and (Brandin and Wonham 1992).

DES supervisory controllers are amenable to automated computation, and a number of educational and academic packages have been developed for supervisory controller design, including TTCT (Meder 1997), OTCT (O'Young 1992), and UMDES

(*UMDES Software Library* 2006), which has recently added a graphical user interface.

More detail on DES supervisory control is given in Appendix D; and for a thorough treatment of DES modeling and supervisory control theory, refer to (Cassandras and Lafortune 1999) and (Kumar and Garg 1995).

2.3 Hybrid System Modeling

An early hybrid system model was proposed by Witsenhausen (Witsenhausen 1966), baring a striking resemblance to the definition used today. A hybrid system was described as:

> *"A class of continuous time systems with part continuous, part discrete state is described by differential equations combined with multistable elements."*

With any hybrid model, the goal is to capture the mixture of continuous and discrete dynamics that are the characteristic of what we know today as hybrid systems. Generally speaking, the various hybrid models differ primarily in their intended purpose and in the expressiveness of the continuous dynamics that are admitted by the model. Furthermore, hybrid modeling tools reflect the community from which they arise; we divide these into the computer science community and the control engineering community. In general, the computer science community's approach has been centered around proving correctness of a system with respect to a given specification (verification), while the controls community seeks parallels to traditional control system theory, such as stability, controllability and observability. The modeling paradigms for computer science have traditionally centered around automaton based methods, while those of the controls community have centered around switched systems. This being said, there is considerable overlap between these communities; each have made significant contributions to the understanding of hybrid systems and the control of hybrid systems.

We now examine some hybrid system models.

2.3.1 Timed Automata

The abstraction level of the coarse-timed FSM lacks the desired timing expressiveness that is necessary for real-time control. The abstraction of the discrete-time DES supervisory control approach is deemed to be unsuitable when reasoning about systems that act (or react) directly with physical processes. The (dense) timed automaton of (Alur and Dill 1994) is a finite state automaton having a finite set of real-valued clocks. These clocks may be reset to zero upon the state transitions of the automaton in order to keep track of time between events. Timed automata theory allows for algorithmic analysis and verification of real time systems (Alur, Courcoubetis and Dill 1993). This approach proves useful when performing model checking on systems that are naturally specified as elapsed times, or time delays. Dense time models are still essentially an abstraction of the underlying physical processes (i.e. continuous variables) that give rise to the discrete events.

Automatic verification tools have been developed for this class of system, notably UPPAAL (Bengtsson, Larsen, Larsson, Pettersson and Yi 1995) and KRONOS (Bozga, Daws, Maler, Olivero, Tripakis and Yovine 1998). These packages have both been applied to the verification of communication protocols; problems that contain "hard" timing constraints (Daws, Kwiatkowska and Norman 2004) (David and Yi 2000). However, owing to the complexity of these protocols, these examples have been carried out only on some portion of the protocol, and were formulated with simplified models of the protocol software code.

2.3.2 Hybrid Automata

This is a finite state graph, in which each state has some continuous dynamics (not necessarily constant rate) specified as differential equations. The switching between states is instantaneous and is governed by guards (or invariants) based on the con-

tinuous variables (Henzinger and Ho 1995). The hybrid automaton is an intuitive and expressive model since it uses the familiar finite state automaton paradigm. An execution of a hybrid model then consists of the continuous states varying according to the currently specified dynamics, followed by a discrete jump to a new state and so on. A natural extension of the timed automaton is the so-called "linear" hybrid automaton (Henzinger 2000), a special case of hybrid automaton that requires the continuous dynamics to be constant rate. Essentially, the LHA is a special case of a timed automaton in which the clocks may run at different rates with respect to each other. This extension of the timed automaton takes the model one step closer to the physical variables, since now the variable rate clocks may model a variety of real-valued continuous variables instead of time.

In general however, the algorithmic verification of the hybrid automaton models is undecidable, since model checking is based ultimately on computing the reachability of an infinite state space. Algorithmic verification of system properties for LHAs are only semi-decidable. When the model is based on a special sub-classes of the linear hybrid automaton; i.e. the rectangular automaton, verification is known to be decidable (Henzinger, Kopke, Puri and Varaiya 1998). A software package that implements hybrid system verification for LHAs called HyTech (Henzinger, Ho and Wong-Toi 1997), (Henzinger, Ho and Wong-Toi 1996) was developed and has found considerable use primarily as a teaching tool and for academic research. HyTech has been reportedly used to verify and parameterize properties in a variety of simplified applications including (to name a few), a steam boiler control (Henzinger and Wong-Toi 1995b), a distributed sensor network (Coleri, Ergen and Koo 2002), ship coordination and control system (Millan and O'Young 2000) and a pneumatic automotive suspension control system (T. Stauner, O. Mueller and M. Fuchs 1997). Unfortunately, the main shortcoming of these applications is that the nonlinear continuous dynamics must be approximated by constant rate dynamics (Henzinger and Wong-Toi 1995a). If a system is meant to be safety critical, then incorrect approx-

imation of the nonlinear dynamics could lead to safety violations. Furthermore, for the control examples, HyTech assumes that a controller exists already for the hybrid system; it verifies the design or parameterizes it; in general designing the controller for a complex system is an important part of the problem.

The hybrid I/O automaton (HIOA) framework was intended to support description and analysis of hybrid systems, adding a complex input/output interface to the basic HA (Lynch, Segala and Vaandrager 2003). Composition operations amongst HIOA models accommodate more complex modeling of hybrid systems. Unfortunately there is no computational tool to support this modeling framework, so the composition and verification is carried out by hand using mathematical proofs thus limiting applications to simple laboratory-based demonstrations (Fehnker, Vaandrager and Zhang 2003) and (Mitra, Wang, Lynch and Feron 2003).

2.3.3 Quantized I/O (Discrete Event Abstraction)

Another approach to hybrid systems modeling has centered around discrete abstractions of continuous systems. This approach is characterized by a control theoretical approach, centered around leveraging the "correct-by-design" results of DES supervisory control theory. In (Raisch and O'Young 1998), discrete abstractions based on the truncated time history of discrete-time LTI continuous models were used to synthesize DES supervisory controllers. In a behavioural sense, if the behaviour of the discrete abstraction contains that of the continuous system, then the safety properties of a DES controller based on the abstraction are ensured (Raisch 2000), (Moor, Raisch and Davoren 2001). The controller is a discrete-event controller, while the plant exists in the continuous domain, so from an I/O point of view, there are A/D and D/A interfaces between the two (Lemmon, He and Markovsky 1999), (Koutsoukos, Antsaklis, Stiver and Lemmon 2000). In (Su, Abdelwahed, Karsai and Biswas 2003), (Abdelwahed, Su and Neema 2005), discrete abstractions of continuous dynamics were adapted in a limited horizon to synthesize DES supervisors.

2.3.4 Switched Systems

Many approaches to hybrid modeling fall into the category of switched systems. The switched system approach is characterized by the high fidelity modeling of the continuous dynamics, with less attention paid to the logic; these are generally non-automaton based representations of hybrid systems.

The emphasis of the switched system approach to hybrid systems is primarily on control system stability and optimality. Typically there are a collection of continuous system dynamics amongst which a controller may switch; conditions are sought under which the switched (or hybrid) system is stable. Worth noting is the fact that even if each individual system is stable, unconstrained switching may actually destabilize the overall system. Conversely, switching may be used to stabilize the overall system even if the individual subsystems are themselves unstable (Hespanha and Morse 2002). For arbitrary switching by the supervisory controller, the hybrid system will be stable if a common Lyapunov function can be found for each of the continuous dynamics. Under state based switching conditions, stability may be guaranteed if multiple Lyapunov functions can be found for each of the switched systems (Branicky 1998).

Many special subclasses of switched system models have been proposed that use approximated continuous dynamics to achieve improved computational complexity at the expense of verification and control conservatism. These models include mixed logical dynamical (MLD), piecewise affine (PWA) and others; each has been shown to be input-state-output equivalent under certain assumptions (Heemels, de Schutter and Bemporad 2001). Closed loop model predictive control (MPC) has also been shown to be equivalent to these other forms of linear switched systems under certain assumptions (Bemporad, Heemels and de Schutter 2002), meaning that switched system results can also be applied to MPC by translating them into MLD or PWA problems.

Software has been developed for analyzing, simulating and even synthesizing controllers for systems modeled by PWA and MLD models (Torrisi and Bemporad 2004),

(Torrisi and Bemporad 2001) in discrete time. Based on the package HYSDEL (Hybrid Description Language) and implemented in the Matlab®/Simulink® environment, PWA models can be interfaced to finite state automata. The software is capable of generating linear and hybrid MPC (receding horizon) control laws in piecewise affine form. Another software tool, *CheckMate*, has been developed in the Matlab/Simulink environment for hybrid system verification (Chutinan and Krogh 2003). Beginning with a polyhedral set of initial continuous states and continuous ranges of parameter values, this package can verify that all trajectories of the model meet some specification.

Typical applications that have been looked at are synthesizing an engine idle speed controller (Balluchi, Natale, Sangiovanni-Vincentelli and van Schuppen 2004) using PWA hybrid models, air traffic control routing problem optimized by using mixed integer linear programming (MILP) (Bayen and Tomlin 2003) and a chemical batch processing system using PWA and MLP (Potocnik, Bemporad, Torrisi, Music and Zupancic 2004). A survey of automotive applications of the switched system control approach are contained in (Balluchi, Benvenuti and Sangiovanni-Vincentelli 2005).

General references for switched systems control and stability can be found in (Liberzon 2003), (Hespanha 2004), and for a short overview, see (Lin and Antsaklis 2005).

2.4 Hybrid System Simulation

When designing control systems for hybrid systems, simulation is without a doubt the most heavily utilized tool by designers. Typically, controllers are tested under a variety of conditions by simulation to evaluate the safety and correctness of a particular design. However, due to the ad-hoc choice of these test conditions, this technique may miss the particular combination of conditions that leads to design failure. In spite of this, hybrid simulation is still an important tool.

The statechart modeling formalism was originally developed by (Harel 1987) to encapsulate the notions of hierarchy, concurrency, and communication for discrete event system models. Statecharts have been widely used and were subsequently extended to include continuous dynamics; an example of a commercial simulation tool using statecharts is the Matlab StateFlow® toolbox for Simulink . Various packages have also been developed for academic use, including CHARON (Alur, Dang, Esposito, Hur, Ivancic, Kumar, Lee, Mishra, Pappas and Sokolsky 2003) a language for describing hybrid and timed systems. Ptolemy is a general-purpose modeling package with a graphical user interface (Lee 2003). HyVisual, based on Ptolemy, is also a visual modeling package, but is designed specifically to model hybrid systems (Brooks, Cataldo, Lee, Liu, Liu, Neuendorffer and Zheng 2005). HYBRSIM is an object-oriented hybrid simulation tool based on bond graph models of hybrid systems (Mosterman 2002). Another hierarchical hybrid simulation tool called YAHMST (Yet Another Hybrid Modeling and Simulation Tool) has also been reported (Thevenon and Flaus 2000).

A comprehensive overview of these and other hybrid modeling, simulation and verification tools is given in (Carloni, DiBenedetto, Pinto and Sangiovanni-Vincentelli 2004).

2.5 Complexity

A common thread in the control problems formulated with the models presented here is that most are either undecidable or computationally intractable (Blondel and Tsitsiklis 2000). Undecidable problems are ones for which a suitable algorithm cannot be constructed to: a) terminate, and b) return a correct answer. Computationally intractable problems are considered to be those for which a polynomial-time algorithm cannot be found, and thus they are not amenable to computation; these are known as $\mathcal{NP}$-hard problems.

It has been shown for simple hybrid systems consisting of switched continuous

systems that verifying properties such as stability and controllability are either undecidable or $\mathcal{NP}$-hard (Blondel and Tsitsiklis 1999). Verification of properties for systems modeled by simple linear hybrid automata (and even for some timed automata), have been shown to be undecidable (Henzinger et al. 1998). In DES supervisory control, the modular supervisor control synthesis is $\mathcal{NP}$-hard due to the familiar "state explosion" problem (Gohari and Wonham 2000). Even in the area of robust control, the calculation of the structured singular value μ, has been shown to be $\mathcal{NP}$-hard (Braatz, Young, Doyle and Morari 1994).

Clearly, the quest for verification and optimality in "real" hybrid or DES control problems is unlikely to be successful. Hence, control solutions will likely have to be sub-optimal or "fit for purpose", and thus new control theory has to be driven by the applications.

2.6 Assessment of Relevant Work

The work presented in this thesis is inspired by the industrial control problems encountered with the safety critical control and coordination and manoeuvring of multiple marine vessels. Practicing controls engineers need design techniques and tools that are easy to use and understand.

2.6.1 Model Formulation

The switched continuous model (SCM) that is developed in Chapter 4 is a blend of the switched system and discrete abstraction approaches to hybrid modeling. We use a flexible state space partitioning based on continuously differentiable functionals as in (Koutsoukos et al. 2000). However, instead of switching piecewise constant inputs, we switch the entire continuous dynamic as is done in the switched system approach. This admits a very expressive continuous modeling to be utilized. The vast majority of switched system approaches emphasize global stability or optimality,

and therefore must use linear approximations of continuous dynamics in order to make the computation more tractable. Because we use a finite time horizon, we can relax the goal of stability, which is traditionally defined on infinite time. In addition, because we deal with a discrete abstraction, optimality is relaxed to merely a safety requirement in the sense of state avoidance. These tradeoffs permit us to admit a larger class of nonlinear continuous dynamics ((Millan and O'Young 2006)).

2.6.2 Discrete Abstraction

Previous discrete abstraction work has focussed on obtaining a single offline discrete event model, with the added requirement that the model be deterministic. This desire leads to state space partitioning regimes that attempt to match the flow of the continuous dynamics (Koutsoukos and Antsaklis 2001). In (Su et al. 2003), the partitioning is based on refinements of polyhedral partitions until the model's nondeterminism is reduced to some satisfactory measure. Since our technique involves abstracting the model repeatedly in an online fashion, no single discrete abstraction is required. And having full-state information, a deterministic model is not required, since we have cast our DES supervisor synthesis as a state avoidance problem. As a result, the main consideration of the partitioning is to generate discrete events (symbols) in order to synchronize with other processes that make up the plant or specification. Furthermore, (Raisch and O'Young 1998) showed that enforcing safety of the discrete abstraction guarantees the safety of the corresponding continuous model if the discrete abstraction is a conservative approximation of the continuous model.

2.6.3 Controller Synthesis

Similar to our work is (Stursberg 2004), in which the nonlinear continuous dynamics are retained as embedded simulations. Working with a finite set of control actions, an acyclic graph branching in discrete time intervals, with hybrid nodes (states) is constructed. The search of this graph is steered by optimality constraints using a

combination of depth and breadth first reachability. Our technique differs in that we construct a finite state graph which is pruned in a maximally permissive sense with respect to a safety specification, in accordance with optimal DES supervisory control theory. Furthermore, our approach admits both state and time dependent switching of dynamics.

2.6.4 Computation

In the work of (Stursberg, Fehnker, Han and Krogh 2003), it was noted that a reduction in computational complexity may be realized by including the specification when calculating reachable sets for hybrid verification problems. Most hybrid reach set computations simply expand the reach set incrementally in all directions without regard to the specification. In our controller synthesis technique, the inclusion of the specification during synthesis allows for a reduction in computational complexity due to the fact that illegal traces may be eliminated as soon as an illegal state is reached; i.e. before it is added to the reach set.

We utilize a limited lookahead scheme similar to that initially explored in (Chung, Lafortune and Lin 1992), in which DES supervisors are computed for a limited lookahead event horizon. This technique was intended to reduce computational complexity for DES control synthesis and to allow time-varying plants to be handled, since it is an online technique. In limited lookahead control, safety and nonblocking properties can only be guaranteed by adopting a conservative approach with regard to the extension of traces beyond the lookahead horizon; that is, they assume that all traces continue to unsafe or blocking states. Our approach is also conservative, and we define the notion of emergency shutdown states, specially marked states to ensure system safety ((Millan 2006)).

In (Giorgetti, Pappas and Bemporad 2005), a finite-time discrete transition system is extracted from the linear continuous dynamics of a discrete-time hybrid automaton (DHA) model on a limited horizon. A technique known as bounded model checking

(BMC) is then used to verify the system against a specification, which is expressed as a temporal logic formula. Instead of verifying a controller design, as in this offline approach , we repeatedly construct controllers online by the synchronous product connection of the plant and specification. Our finite state graph (called a hybrid transition graph) represents the controller and is correct by design because it represents the (exhaustive) reachable state space, on a limited horizon, of the plant pruned by a safety specification.

2.7 Summary

In this chapter we have examined some common approaches to hybrid system modeling that have been reported in the literature. The various techniques and tools for simulation, verification, and control synthesis have developed from two communities with backgrounds of control systems (electrical engineering) and computer science. Both of these research approaches have had some successes, but no hybrid system control techniques have yet seen any widespread acceptance by industry. Simulation still seems to be the dominant approach to hybrid system control design. The promise of the definitive verification, optimality and provably stable hybrid system controller appears to be an elusive goal; many of these have been shown to be either undecidable problems or computationally intractable.

A comparison of the techniques developed in this thesis with those of the literature has been presented. In the following chapters, these modeling and computational and control synthesis techniques are developed in further detail.

Chapter 3

Abstraction of Continuous System Dynamics

3.1 Introduction

The goal of this chapter is to develop a discrete event abstraction of a continuous model that ultimately will be suitable for discrete event supervisory control. The approach taken is to select a natural and expressive continuous modeling framework and then to overlay it with a discrete event, input/output (I/O) interface. For now, we consider the output aspects of the interface, or the conversion of the continuous dynamics to that of discrete event dynamics.

The continuous dynamics of a system may be described by a nonlinear ordinary differential equation (ODE),

$$\dot{x}(t) = f(x, t) \tag{3.1}$$

In general, the objective of the discrete abstraction is to achieve a single, preferably deterministic, automaton representation of the continuous dynamics. Based on this discrete abstraction, standard DES supervisory control techniques can be used to develop a DES controller. The discrete abstraction is intended to capture only the important dynamics (those that matter to the DES controller), thereby reducing the

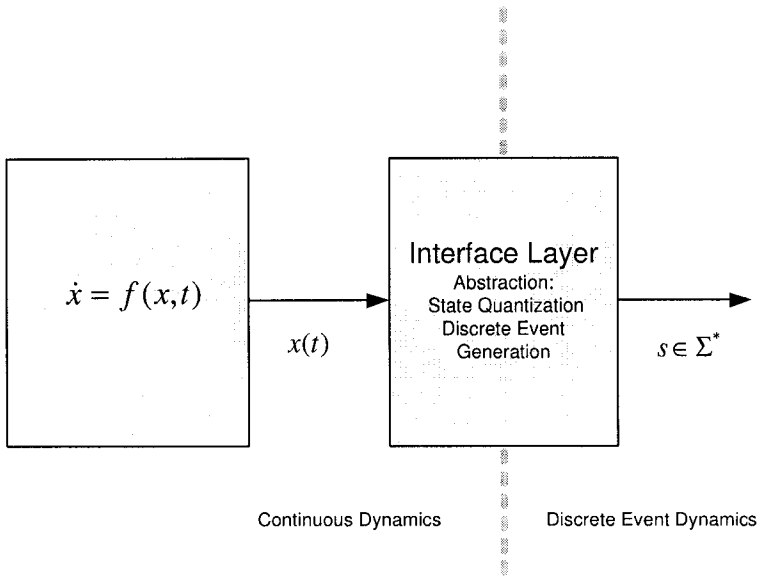


Figure 3-1: An interface layer between the continuous dynamics and the discrete event dynamics is used to develop the discrete abstraction.

model complexity. The choice of an appropriate state quantization technique must be considered carefully with this approach since it directly affects the complexity, the determinism, and the fidelity of the model. There is a trade-off between modeling complexity and the behavioural fidelity of the discrete abstraction.

One can think of state quantization as observing the continuous system's state space through a sort of "compound lens", in the sense that it partitions the continuous state space into multiple disjoint discrete states, approximations of the continuous states. Continuous trajectories traversing across this quantized continuous state space generate discrete events (or symbols) as the trajectory crosses boundaries between the states. These output events drive or synchronize external discrete event systems. The continuous system along with its interface layer can be considered to be a discrete event generator, as pictured in Fig. 3.1.

For another view of the relationship between a continuous model and its discrete event abstraction, refer to Fig. 3-2. In the upper left is the phase portrait of a

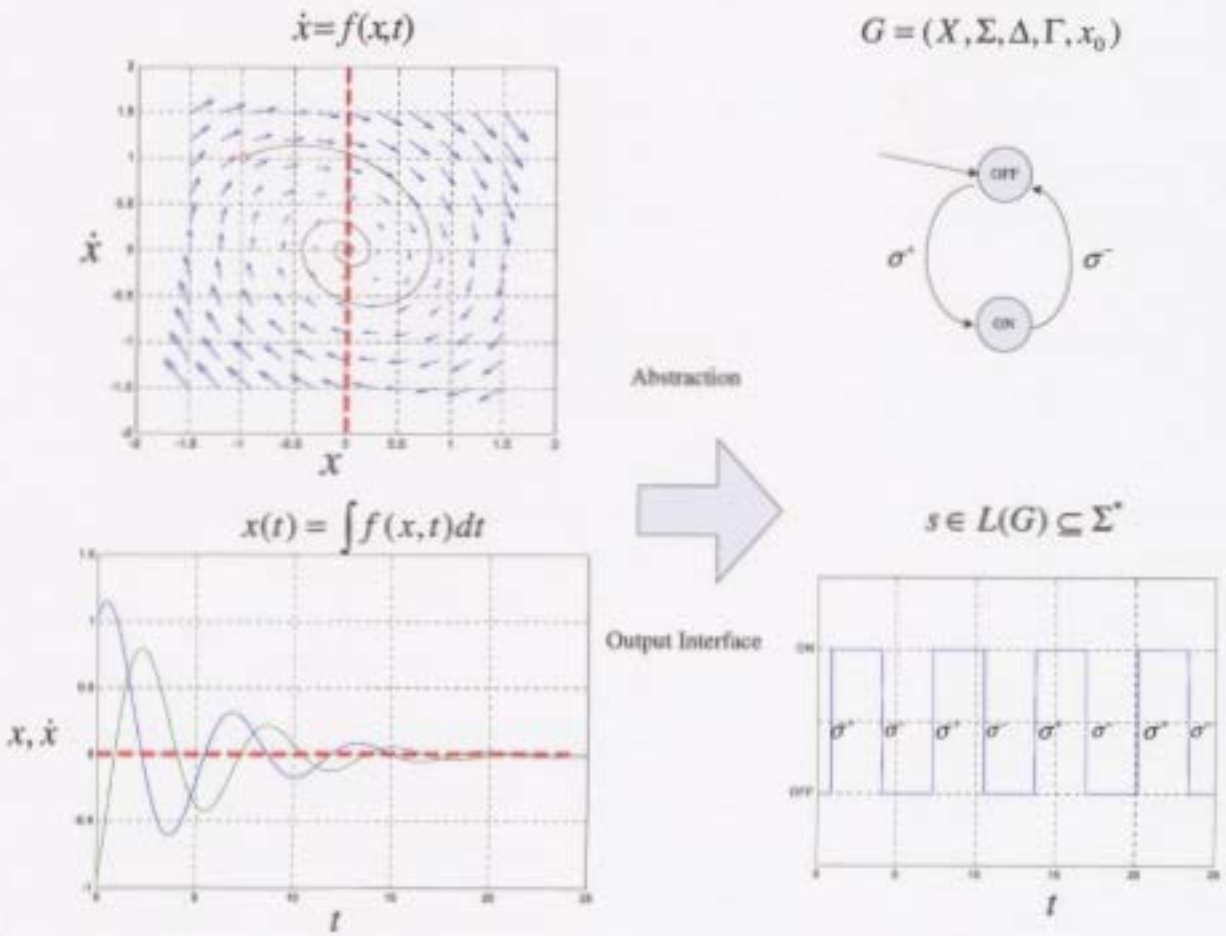


Figure 3-2: A comparison of continuous modeling (left) and discrete modeling (right).

continuous system, essentially a graphical representation of the continuous dynamics of a modeled system (Eq. 3.1). Superimposed on the phase portrait is a trajectory $x(t)$, the continuous behaviour, and the phase plane has been partitioned into two regions. In the lower left, the state variables of $x(t)$ are plotted against time. The upper right of the diagram is the graphical representation of the DES model of the same continuous system, a finite state machine graph. In the lower right, the discrete event behaviour of the FSM for the same continuous trajectory. Comparisons may be drawn between the continuous state space approach and its counterpart, the automaton representation. Likewise, there is a parallel between the continuous input/output model and languages of automata (Boel, Cao, Cohen, Giua, Wonham and van Schuppen 2002).

A discrete abstraction of a continuous model is defined by the state quantization and the event generation processes. This chapter examines the discrete abstraction of a generalized continuous model on a finite time horizon. The discrete abstraction of the continuous system can be viewed as an autonomous generator of discrete events. In this context, we examine one particular state abstraction technique that utilizes continuous functionals to partition the state space of a given system model. This technique was developed extensively in (Stiver, Koutsoukos and Antsaklis 2000) and (Koutsoukos et al. 2000). In these works, functionals $F : \mathbb{R}^n \rightarrow \mathbb{R}$ are used to partition the state space of a continuous system. For purposes of supervisory control, the null space of these functionals are designed to be invariant manifolds with respect to the vector field of the continuous dynamics. The resulting partitions have common entry and exit boundaries, thus permitting deterministic DES models to be extracted.

In this chapter, we expand on the work of (Koutsoukos et al. 2000) by developing bounds on the cardinality of the state label set and event label set of a discrete abstraction due to a general family of partitioning functionals. We relax the requirement that the resulting partitions be invariant with respect to the continuous flow field, since without loss of generality, we do not require a deterministic DES model.

The emphasis is to develop a practical and flexible mechanism for obtaining discrete abstractions of continuous dynamical systems, from which an algorithmic implementation can be developed. Finally, this chapter outlines the conditions that will be required for a generalized discrete abstraction in the following chapters.

3.2 State Quantization

This section outlines the quantization of the state space of a continuous model. Smooth functionals of the continuous state variables are a powerful way of producing state partitions, since they can be designed around the discrete event information that we wish to extract from a continuous model. A functional-based quantization allows for quantizations based on the entire continuous state vector.

Definition 3.2.1 (Functional) *A functional $F : \mathbb{R}^n \to \mathbb{R}$, is a real-valued function on a vector space. For the purposes of this work, F is smooth, i.e. continuously differentiable.*

Definition 3.2.2 (Gradient Operator) *The gradient operator ∇ returns a gradient vector*

$$\nabla F(x) = \left(\frac{\partial F}{\partial x_1}, \frac{\partial F}{\partial x_2}, \cdots \frac{\partial F}{\partial x_n} \right)^T$$

Definition 3.2.3 (Hypersurface) *Let $\mathcal{N}(F)$ be the null space of a smooth functional F,*

$$\mathcal{N}(F) = \{x \in \mathbb{R}^n : F(x) = 0\}$$

such that

$$\nabla F(\xi) \neq 0, \forall \xi \in \mathcal{N}(F)$$

then $\mathcal{N}(F)$ is a smooth hypersurface of codimension one, that is, $\dim(F) - \dim(\mathcal{N}(F)) = 1$

Definition 3.2.4 (Set Partition) *A hypersurface $\mathcal{N}(F)$, forms a partition of a set $Q \subseteq \mathbb{R}^n$, into exactly two subsets, $Q' = \{x : F(x) \geq 0\}, Q'' = \{x : F(x) < 0\}$, provided that $\mathcal{N}(F) \cap Q \neq \emptyset$. If a partition is created, then there exists $Q', Q'' \subset \mathbb{R}^n$ such that $Q' \cup Q'' = Q$*

Note that if a partition is created, Q', Q'' are pairwise disjoint sets. Thus, the intersection of a single smooth functional with a set produces a partition of the set into two subsets. We examine two operations that will be used to further develop the partitioning mechanism.

Definition 3.2.5 (Set Partition Operation (I)) *Let $\mathcal{N}(F)$ be a hypersurface formed by a functional F, and let $Q \subseteq \mathbb{R}^n$, then the set partition operation P_s is defined as*

$$P_s(Q, \mathcal{N}(F)) = \begin{cases} \{Q\}, & \text{if } \mathcal{N}(F) \cap Q = \emptyset. \\ \{Q', Q''\}, & \text{if } \mathcal{N}(F) \cap Q \neq \emptyset. \end{cases}$$

where Q', Q'' are as per Def. 3.2.4.

We will now define a partitioning operator that operates on families of sets, so that it can be used in recursive definitions.

Definition 3.2.6 (Set Family Partition Operation) *Let $H = \{Q_j \subseteq \mathbb{R}^n | 1 \leq j \leq M\}$ be a family of sets Q_j that are pairwise disjoint. Let $\mathcal{N}(F)$ be a hypersurface arising from a functional F, then the set family partition operation, $P_f(H, \mathcal{N}(F))$, returns a family, H' of sets which is the result of the set partition operation applied to each element of H such that*

$$H' = P_f(H, \mathcal{N}(F)) = \bigcup_{j=1}^{M} P_s(Q_j, \mathcal{N}(F)) \tag{3.2}$$

The union of the elements of the post-operation family, H', is equal to the union of

the elements of the pre-operation family,

$$\bigcup_{Q'_j \in H'} Q'_j = \bigcup_{Q_j \in H} Q_j$$

A simple example of the set family partitioning operation follows,

Example 3.2.1 *Let* $H = \{Q_1, Q_2, \ldots, Q_M\}$ *if for all* j, $Q_j \cap \mathcal{N}(F) \neq \emptyset$ *then*

$$\begin{aligned} H' &= P_f(H, \mathcal{N}(F)) = P_s(Q_1, \mathcal{N}(F)) \cup P_s(Q_2, \mathcal{N}(F)) \cup \\ &\quad \ldots \cup P_s(Q_M, \mathcal{N}(F)) \\ H' &= \{Q'_1, Q''_1\} \cup \{Q'_2, Q''_2\} \cup \ldots \cup \{Q'_M, Q''_M\} \\ &= \{Q'_1, Q''_1, Q'_2, Q''_2, \ldots, Q'_M, Q''_M\} \end{aligned}$$

Now $\bigcup_j (Q'_j \cup Q''_j) = \bigcup_j (Q_j)$.

Now for repetitive partitioning operations, it is necessary to prove some properties of the set family partitioning operation.

Lemma 3.2.1 *Let* $H = \{Q_j \subseteq \mathbb{R}^n | 1 \leq j \leq M\}$ *and* $\mathcal{N}(F) \cap Q_j = \emptyset$, $\forall Q_j \in H$ *and* $\bigcap_{j=1}^{M} Q_j = \emptyset$, *then the number of sets in the resulting family,* $|H'| = |P_f(H, \mathcal{N}(F))| = M$ *and moreover,* $H = H'$.

Proof. Since $\mathcal{N}(F) \cap Q_j = \emptyset$ for all $Q_j \in H$, then it follows from the definition of the set family partition operation P_f (Def. 3.2.6), that

$$\begin{aligned} |H'| &= \left| \bigcup_{j=1}^{M} P_s(Q_j, \mathcal{N}(F)) \right| \\ &= |P_s(Q_1, \mathcal{N}(F))| + |P_s(Q_2, \mathcal{N}(F))| + \ldots + |P_s(Q_M, \mathcal{N}(F))| \\ &= \underbrace{1 + 1 + \ldots + 1}_{M} \\ &= M \end{aligned}$$

■

Now for case that each partitioning operation results in a non-empty hypersurface intersection,

Lemma 3.2.2 *Let* $H = \{Q_j \subseteq \mathbb{R}^n | 1 \leq j \leq M\}$ *and* $\mathcal{N}(F) \cap Q_j \neq \emptyset$, $\forall Q_j \in H$ *and if* $\bigcap_{j=1}^{M} Q_j = \emptyset$ *then the number of sets in the resulting family,* $|H'| = |P_f(H, \mathcal{N}(F))| = 2M$.

Proof. Since $\mathcal{N}(F) \cap Q_j \neq \emptyset$ for all $Q_j \in H$, then it follows directly from the definition of the set family partition operation P_f (Def. 3.2.6) that

$$\begin{aligned} |H'| &= \left| \bigcup_{j=1}^{M} P_s(Q_j, \mathcal{N}(F)) \right| \\ &= |P_s(Q_1, \mathcal{N}(F))| + |P_s(Q_2, \mathcal{N}(F))| + \ldots + |P_s(Q_M, \mathcal{N}(F))| \\ &= \underbrace{2 + 2 + \ldots + 2}_{M} \\ &= 2M. \end{aligned}$$

■

A further example will illustrate the successive partitioning operations, given Lemma 3.2.1 and Lemma 3.2.2.

Example 3.2.2 *Let* $H = \{Q_1, Q_2\}$ *be a family of sets and let* F *be a functional such that* $\mathcal{N}(F) \cap Q_1 \neq \emptyset$ *and* $\mathcal{N}(F) \cap Q_2 \neq \emptyset$. *Then* $H' = S_f(H, \mathcal{N}(F_a)) = \{Q'_1, Q''_1, Q'_2, Q''_2\}$ *and* $|H'| = 2\,|H| = 2 \cdot 2 = 4$. *Likewise, if there are no set intersections with the hypersurface, then the operation returns the original family of sets unaltered* $H' = S_f(H, \mathcal{N}(F_a)) = \{Q_1, Q_2\}$, *i.e.* $|H'| = |H| = 2$.

Up to this point, only a single functional has been used to partition a single set or family of sets. We will now look at the effect a family of partitioning functionals has upon a set, by applying the set family partitioning operator recursively

Definition 3.2.7 (Set Partition Operation (II)) *Let $Q \subseteq \mathbb{R}^n$ and let Ψ be a family of smooth functionals, $\{F_i : \mathbb{R}^n \to \mathbb{R},\ 1 \leq i \leq N\}$. The set family partition operator recursively partitions Q into a family of sets*

$$H' = \underbrace{P_f(\ldots P_f(P_f}_{N \text{ times}}(\{Q\}, \mathcal{N}(F_1)), \mathcal{N}(F_2)), \ldots, \mathcal{N}(F_N)) \tag{3.3}$$

The entire state space of a system, $Q = \mathbb{R}^n$, can be separated into a family of subsets using the operator described in Def. 3.2.7. Given a family Ψ, of N functionals, $\{F_i : \mathbb{R}^n \to \mathbb{R}, 1 \leq i \leq N\}$ the corresponding hypersurfaces, $\mathcal{N}(F_i)$ separate the state space of a system into a family of sets.

How does the family of sets $Q \in H'$ relate to the discrete states of the DES model? It can be shown that the family of partitioning functionals establishes an equivalence relation on the system state space.

Definition 3.2.8 (Equivalence relation) *Let $\Psi = \{F_i : \mathbb{R}^n \to \mathbb{R}, 1 \leq i \leq N\}$ be a family of functionals defined on the state space of the system described by $\dot{x}(t) = f(x,t)$, $x(t) \in \mathbb{R}^n$, then an equivalence relation is defined on $\mathbb{R}^n$ by the partitioning functionals*

$$x_1 \sim_p x_2 \iff (\mathsf{sign}(F_i(x_1)) \times \mathsf{sign}(F_i(x_2)) = 1, \text{ for all } i,\ 1 \leq i \leq N) \tag{3.4}$$

Definition 3.2.9 (Equivalence Class) *Each set $Q_j \subset \mathbb{R}^n$ is an equivalence class created by the equivalence relation $\sim_p$of Eq. 3.4.*

Definition 3.2.10 (Quotient Set) *The set of all equivalence classes $\mathcal{X}$, given the equivalence relation $\sim_p$, is known as the quotient set $\mathcal{X} = \mathbb{R}^n / \sim_p$.*

The members of the quotient set are the subsets Q_j resulting from a state space partitioning operation. These subsets (or equivalence classes) will be associated with discrete system states through a state labeling function that assigns a unique state

label to each of the discrete states corresponding to the subsets $Q_j \in \mathcal{X}$.

Definition 3.2.11 (Unit Step Function) *We define a unit step function as:*

$$h(\xi) = \begin{cases} 1 & \xi \geq 0 \\ 0 & \xi < 0 \end{cases}$$

Definition 3.2.12 (State Labeling Function) *Let Ψ be a family of N functionals partitioning a state space, then let $V : \mathbb{R}^n \to \{0,1\}^N$, be a function that identifies the system state $x \in \mathbb{R}^n$ with a labeling vector as follows:*

$$V(x) = \begin{bmatrix} h(F_1(x)) & h(F_2(x)) & \cdots & h(F_N(x)) \end{bmatrix}^T$$

Thus each member of the quotient set $Q_j \in \mathcal{X}$ is associated with a unique label vector generated by the state labeling function.

We will establish bounds on the cardinality of the resulting family of sets due to this state partitioning operation. Let H'_i be the ith family of sets returned by the ith nested set family partition operation, $P_f^i(H_i, \mathcal{N}(F_i))$, then for the next recursive operation, P_f^{i+1}, $H_{i+1} = H'_i$. If each functional F_i intersects with only one set $Q_j \in H_i$, for each $P_f^i(H_i, \mathcal{N}(F_i))$ operation in Eq. 3.3 then this will be termed as *minimal intersection.* Conversely, if each of the functionals, F_i intersects with all $Q_j \in H_i$ for each $P_f^i(H_i, \mathcal{N}(F_i))$ operation this is termed *maximal intersection.*

Maximal intersection determines the upper bound of cardinality resulting from the state space partitioning operation.

Lemma 3.2.3 (State Space Partition Upper Bound) *Let $Q = \mathbb{R}^n$ be the state space of a system, and let $\Psi = \{F_i : \mathbb{R}^n \to \mathbb{R},\ 1 \leq i \leq N\}$ be a family of functionals. The set partition operation (II) of Def. 3.2.7, will produce a family of sets, H', such that the upper bound on the number of sets is $|H'| = 2^N$.*

Proof. Proof of the upper bound is made by assuming maximal intersection and by induction. For the (base) case of one functional $N = 1$, the set family partition operation $P_f^1(\{Q\}, \mathcal{N}(F_1))$ reduces to the set partition operation $P_s(Q, \mathcal{N}(F))$ because there is only one set in the family $\Pi_1 = \{Q\}$ and only one functional. Since $Q = \mathbb{R}^n$, thus $Q \cap \mathcal{N}(F_1) \neq \emptyset$. It follows from Def. 3.2.5 that the cardinality of the returned family is (by Lemma 3.2.2) $|H'_1| = 2^N = 2^1 = 2$. The inductive hypothesis is that $|H'_N| = 2^N$. It remains to show $|H'_{N+1}| = 2^{N+1}$. Suppose there exists F_{N+1} that has maximal intersection for all $Q_j \in H'_N$ and since by definition

$$\begin{aligned} H_{N+1} &= H'_N \\ H'_{N+1} &= P_f^{N+1}(H_{N+1}, \mathcal{N}(F_{N+1})) = \bigcup_{j=1}^{M} P_s(Q_j, \mathcal{N}(F_{N+1})) \end{aligned}$$

Therefore,

$$\begin{aligned} |H'_{N+1}| &= \left| \bigcup_{j=1}^{M} P_s(Q_j, \mathcal{N}(F)) \right| \\ &= |P_s(Q_1, \mathcal{N}(F_{N+1}))| + |P_s(Q_2, \mathcal{N}(F_{N+1}))| + \ldots + |P_s(Q_M, \mathcal{N}(F_{N+1}))| \\ &= \underbrace{2 + 2 \ldots + 2}_{M} \\ &= 2M \end{aligned}$$

where $|H'_N| = M = 2^N$. It follows then that $|H'_{N+1}| = 2M = 2 \times 2^N = 2^{N+1}$, thus proving the inductive hypothesis to be correct. ■

The lower bound of the state space partitioning operation is determined by the minimal intersection of the partitioning functionals.

Lemma 3.2.4 (State Space Partition Lower Bound) *Let $Q = \mathbb{R}^n$ be the state space of a system, and let $\Psi = \{F_i : \mathbb{R}^n \to \mathbb{R},\ 1 \leq i \leq N\}$ be a family of functionals, such that there is minimal intersection. The set partition operation (II) of Def. 3.2.7 will produce a family of sets H', such that the lower bound on the number of sets is*

$|H'| = N + 1$.

Proof. Prove the lower bound by induction. The base case is $N = 1$, or one functional F_1, so

$$H_1 = P_f^1(\{Q\}, \mathcal{N}(F_1)) = P_s(Q, \mathcal{N}(F_1)),$$

and clearly $Q \cap \mathcal{N}(F_1) \neq \emptyset$, therefore $|H_1'| = 2$ by definition. The inductive hypothesis is that $|H_N'| = N + 1$. It remains to show that for $N + 1$ functionals $|H_{N+1}'| = (N + 1) + 1 = N + 2$. Suppose there exists F_{N+1} and only one set $Q_1 \in H_N'$ (minimal intersection), such that $Q_1 \cap \mathcal{N}(F_{N+1}) \neq \emptyset$ and recall that $H_N' = H_{N+1}$. Then, by definition

$$H_{N+1}' = P_f^{N+1}(H_{N+1}, \mathcal{N}(F_{N+1})) = \bigcup_{j=1}^{M} P_s(Q_j, \mathcal{N}(F_{N+1}))$$

where $|H_N'| = |H_{N+1}| = M = N + 1$. Expanding the cardinality expression

$$\begin{aligned}
|H_{N+1}'| &= \bigcup_{j=1}^{M} P_s(Q_j, \mathcal{N}(F_{N+1})) \\
&= |P_s(Q_1, \mathcal{N}(F_{N+1}))| + \underbrace{|P_s(Q_2, \mathcal{N}(F_{N+1}))| + \ldots + |P_s(Q_M, \mathcal{N}(F_{N+1}))|}_{M-1 \text{ terms}} \\
&= 2 + \underbrace{1 + \ldots + 1}_{M-1} \\
&= 2 + M - 1 \\
&= N + 2.
\end{aligned}$$

■

Having established upper and lower cardinality bounds on the state space partitioning operation, it is possible to develop a general theorem on the range of cardinality for the general result of this operation.

Theorem 3.2.1 (State Space Partition Boundedness) *Let $Q = \mathbb{R}^n$ be the state space of system and let $\Psi = \{F_i : \mathbb{R}^n \to \mathbb{R},\ 1 \leq i \leq N\}$ be a family of functionals, with N finite. Then the state space shall be partitioned into a family of sets H', such*

that $|H'|$ is finite and furthermore that

$$N + 1 \leq |H'| \leq 2^N.$$

Proof. Let $|\Psi| = N$. It follows from Lemma 3.2.4 that the cardinality of a resulting state space partition is bounded below and is $|H'| \geq N + 1$. Lemma 3.2.3 ensures that there is an upper bound on the cardinality of the resulting state space partition, $H \leq 2^{N+1}$. Hence the result follows that any finite family of functionals induces a finite partition of sets on $\mathbb{R}^n$. ■

See Appendix B for further results on set partitioning using functionals.

3.3 Discrete Event Generation

A state quantization scheme has been established using families of functionals that produces a finite set of discrete states and corresponding state labels. There is an equivalence between these discrete states and subsets of the continuous state space. We now define the event generation mechanism for our discrete abstraction. To do this, we examine in more detail the continuous trajectories crossing the state space of the continuous model, and how these trajectories are manifested in the discrete abstraction.

Definition 3.3.1 (Continuous Trajectory) *A continuous trajectory of a system is defined as a solution $x(t)$, to an IVP for $\dot{x} = f(x,t)$, and an initial condition $x_0 = x(t_0)$, over some finite time interval $[t_0, t_f]$.*

A transition occurs upon the traversal of a continuous trajectory between a pair of regions or sets. For further definitions and discussion of continuous solutions to ordinary differential equations, refer to Appendix A. In general, solutions of ODEs require an assumption that the function f be Lipschitz continuous in order for the solution to exist and be unique.

Definition 3.3.2 (Transition) *Let Q_1,$Q_2 \subset \mathbb{R}^n$ be a pair of sets such that $Q_1 \cap Q_2 = \emptyset$. Let $x(t) \in \mathbb{R}^n$, a solution to an IVP on time interval $[t_0, t_f]$ such that $x(t) \in Q_1 \cup Q_2$ for all $t \in [t_0, t_f]$. If the continuous trajectory crosses a hypersurface, and then enters the state, then the system is said to have undergone a transition.*

$$\exists x_1 = x(t_1), x_2 = x(t_2),\ t_1, t_2 \in [t_0, t_f] \text{ and } t_1 < t_2 \text{ such that } x_1 \in Q_1 \text{ and } x_2 \in Q_2$$

The transition of a continuous trajectory from Q_1 to Q_2 will be indicated by the following notation

$$Q_1 \rightsquigarrow Q_2$$

Formally, the transition occurs at the moment when the trajectory $x(t)$ first enters the region Q_2.

Definition 3.3.3 (Discrete Event) *An atomic discrete system event σ is generated when a transition occurs.*

For the purpose of event timing, it is important to have a consistent definition of exactly *when* the discrete event occurs.

Definition 3.3.4 (Discrete Event Time) *Let Q_1,$Q_2 \subset \mathbb{R}^n$. Let $x(t) \in \mathbb{R}^n$ be a solution to an IVP for $\dot{x} = f(x,t)$, and an initial condition $x_0 = x(t_0)$, over some finite time interval $[t_0, t_f]$ such that $x(t) \in Q_1 \cup Q_2$ for all $t \in [t_0, t_f]$. There is a corresponding transition, as in Def. 3.3.2. Let $x'(t) \in Q_1$ be the solution on the time interval $[t_0, t_e)$ and let $x''(t) \in Q_2$ be the solution on the time interval $[t_e, t_f]$. The atomic discrete event corresponding to the state transition is said to occur at time t_e, the atomic moment at which the trajectory $x(t)$ enters Q_2 (Fig. 3-3).*

In this definition, the partitioning hypersurface may be included in the trajectory's originating region Q_1 or in the terminal region, Q_2. So according to our definition, a transition may occur when a trajectory leaves, or when it lies on the partitioning hypersurface.

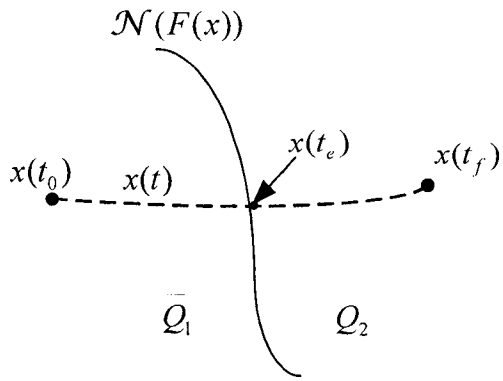


Figure 3-3: A continuous trajectory (dashed line) crossing a hypersurface $\mathcal{N}(f(x))$ generates discrete event at time t_e.

With the state space of a system partitioned into possibly many subsets or regions, it is necessary to have a way of uniquely identifying each of the continuous system's transitions with the discrete abstraction's state transitions.

Definition 3.3.5 (Output Event Set) *The output event set Σ_{out}, is a set of discrete event labels that identify the discrete abstraction's state transitions uniquely. The event labels are associated with the continuous system's transitions.*

There are two event labels for each pair of adjacent states, since the direction in which the state transition occurs must be distinguished. Thus a distinct event label is reserved for each pair of regions to preserve the transition direction information:

$$Q_1 \rightsquigarrow Q_2 \text{ generates event } \sigma_{1,2}$$
$$Q_2 \rightsquigarrow Q_1 \text{ generates event } \sigma_{2,1}$$

Definition 3.3.6 (Output Event Function) *The output event function, $\Lambda : \mathcal{X} \times \mathcal{X} \rightarrow \Sigma_{out}$ is a map from an adjacent pair of states (equivalence classes) to an output event.*

When a continuous trajectory crosses a hypersurface, the output event corresponding to the state transition is given by the output event function. For example, the output event function returns the pair of complementary output events σ^+, σ^-, for adjacent states separated by the hypersurface $\mathcal{N}(F(x))$,

$$\begin{aligned} \sigma^+ &= \Lambda\left[V\left(\lim_{F(x)\to 0^-} x\right), V\left(\lim_{F(x)\to 0^+} x\right)\right] \in \Sigma_{out} \\ \text{and } \sigma^- &= \Lambda\left[V\left(\lim_{F(x)\to 0^+} x\right), V\left(\lim_{F(x)\to 0^-} x\right)\right] \in \Sigma_{out} \end{aligned}$$

The following example illustrates the state partitioning, labeling and the output event functions.

Example 3.3.1 *Let $\Psi = \{F_1, F_2\}$ be the partitioning functionals. Functional F_1 partitions the state space into $H = \{Q_1', Q_1''\}$, where $Q_1' = \{x \in \mathbb{R}^n | F_1(x) \geq 0\}$ and $Q_1'' = \{x \in \mathbb{R}^n | F_1(x) < 0\}$. Since there is one pair of states, the event set has two events, $\Sigma_{out} = \{\sigma_{12}, \sigma_{21}\}$. Now, to continue the example, if we partition H with the second functional F_2, (refer to Fig. 3-4) then*

$$\begin{aligned} H' &= P_f(H, F_2) = P_f(\{Q_1', Q_1''\}, F_2) \\ &= \{(Q_1')_2', (Q_1'')_2'', (Q_1')_2'', (Q_1'')_2'\} \end{aligned}$$

where

$$\begin{aligned} Q_1 &= (Q_1')_2' = \{x \in \mathbb{R}^n : F_1(x) \geq 0 \wedge F_2(x) \geq 0\}, \\ Q_2 &= (Q_1'')_2'' = \{x \in \mathbb{R}^n : F_1(x) < 0 \wedge F_2(x) < 0\}, \\ Q_3 &= (Q_1')_2'' = \{x \in \mathbb{R}^n : F_1(x) \geq 0 \wedge F_2(x) < 0\}, \\ Q_4 &= (Q_1'')_2' = \{x \in \mathbb{R}^n : F_1(x) > 0 \wedge F_2(x) \geq 0\}. \end{aligned}$$

The discrete state labels are:

$$\begin{aligned}
\forall x &\in Q_1\ V(x) = \begin{bmatrix} 1 & 1 \end{bmatrix}^T \\
\forall x &\in Q_2,\ V(x) = \begin{bmatrix} 0 & 0 \end{bmatrix}^T \\
\forall x &\in Q_3,\ V(x) = \begin{bmatrix} 1 & 0 \end{bmatrix}^T \\
\forall x &\in Q_4,\ V(x) = \begin{bmatrix} 0 & 1 \end{bmatrix}^T
\end{aligned}$$

Each of the states is adjacent to one another, so there are 6 pairs of sets in the state space partition:

$$(Q_1, Q_2), (Q_2, Q_3), (Q_3, Q_4), (Q_4, Q_1), (Q_1, Q_3), (Q_2, Q_4)$$

therefore the cardinality of the output event set

$$|\Sigma_{out}| = \frac{M!}{(M-2)!} = \frac{4!}{(4-2)!} = \frac{24}{2} = 12$$

and assigning labels to each transition gives the output event set is

$$\Sigma_{out} = \{\sigma_{1,2}, \sigma_{2,1}, \sigma_{2,3}, \sigma_{3,2}, \sigma_{3,4}, \sigma_{4,3}, \sigma_{4,1}, \sigma_{1,4}, \sigma_{1,3}, \sigma_{3,1}, \sigma_{2,4}, \sigma_{4,2}\}$$

We will now develop the cardinality of the output event set given a family of subsets.

Theorem 3.3.1 *Let $H = \{Q_j \subseteq \mathbb{R}^n | 1 \leq j \leq M\}$ be a family of subsets. The corresponding output event set Σ_{out} has finite cardinality and an upper bound of $|\Sigma_{out}| = \frac{M!}{(M-2)!}$.*

Proof. The upper bound occurs if every Q_j is adjacent to every other Q_j. This is a permutation, since the order in which the open sets are paired matters. A pairwise permutation for M objects is given by the formula for a so called *r-permutation* where

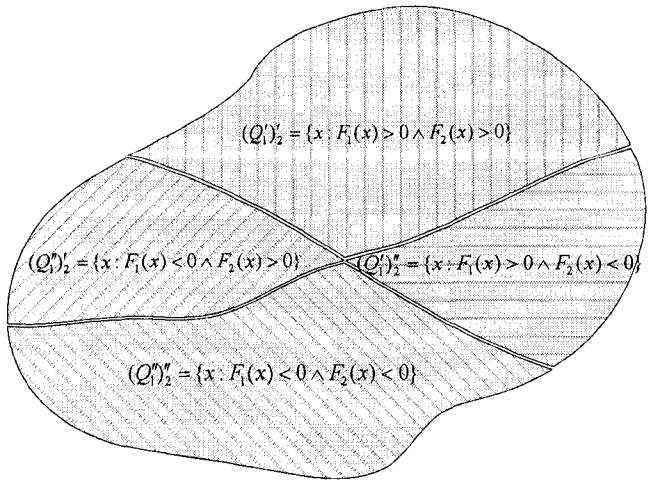


Figure 3-4: Example of state space partitioning.

r is the number objects to be arranged in each permutation

$$\begin{aligned} P(M,r) &= M(M-1)(M-2)\ldots(M-r+1) \\ &= \frac{M!}{(M-r)!} \end{aligned}$$

and since a pair of sets gives rise to an event, $r = 2$ and

$$P(M,2) = \frac{M!}{(M-2)!}$$

■

3.4 Examples

The discrete abstraction framework based on functionals allows the extraction of discrete event information based on a function of any combination of the state variables, providing that the function is continuous. The following examples will illustrate the functional-based discrete state abstraction technique.

Example 3.4.1 *Consider an example of a simple system having continuous dynamics described by an ODE with two state variables*

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

For simplicity, no particular dynamics will be assigned, but suppose that the state variables represent the voltage and current respectively of an electric motor armature. If we wish to be notified (by discrete event) that the instantaneous power of the system has passed through some threshold, then we can capture this information in the discrete abstraction by appropriate selection of the partitioning functionals. For a 100 *Watt positive power threshold (i.e. forward and reverse motoring)*

$$F_1(x) = x_1 x_2 - 100$$

and for a 100 *Watt negative power threshold (forward and reverse motor regeneration)*

$$F_2(x) = x_1 x_2 + 100$$

These functionals are depicted in Fig. 3-5, where they appear as two saddle-like surfaces (shaded grey). Where the surfaces intersect the $x_1 x_2$ plane (the phase plane) are the hypersurfaces $\mathcal{N}(F_1))$ and $\mathcal{N}(F_2))$ which are marked in the figure as heavy grey lines. In Fig. 3-6 the view has been changed so that we are looking directly down at the phase plane. For illustrative purposes, some arbitrary dynamics have been included for the phase portrait, and a trajectory $x(t)$ originating at initial condition x_0 is included as a dotted line. The functionals partition the state space into three

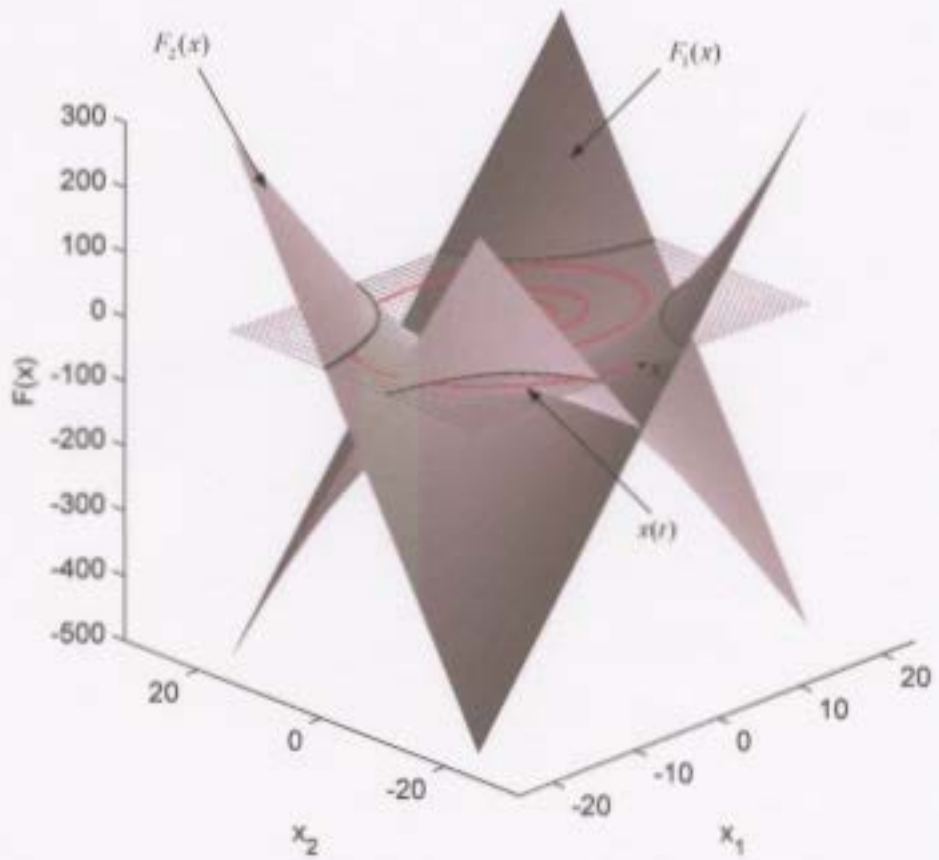


Figure 3-5: Equal power surfaces for discrete event generation on power threshold.

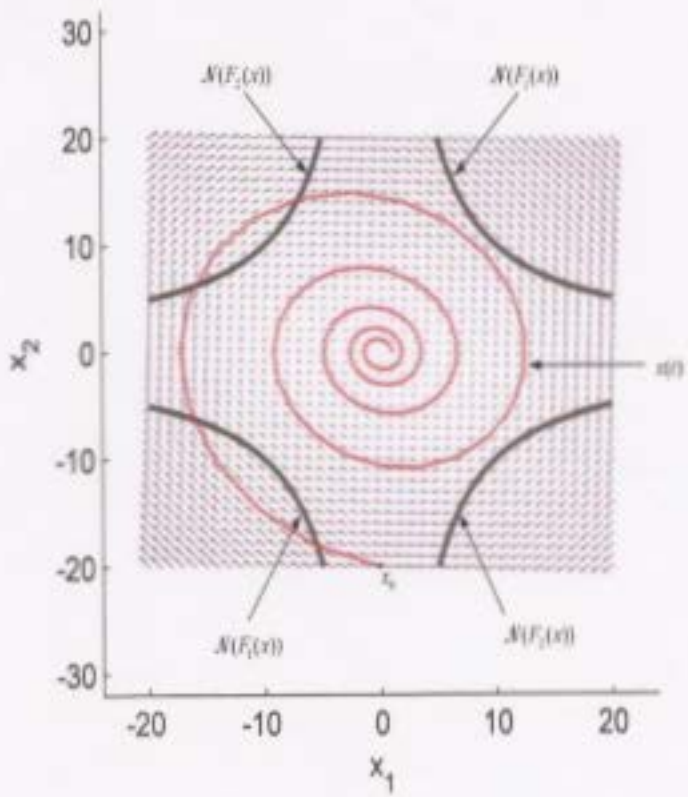


Figure 3-6: View of the phase plane, with hypersurfaces $\mathcal{N}(F_1)), \mathcal{N}(F_2))$, and an illustrative trajectory $x(t)$.

states,

$$\begin{aligned} Q_1 &= \{x : F_1(x) < 0 \wedge F_2(x) < 0\} \\ Q_2 &= \{x : F_1(x) > 0 \wedge F_2(x) < 0\} \\ Q_3 &= \{x : F_1(x) < 0 \wedge F_2(x) > 0\} \end{aligned}$$

Clearly, the set $\{x : F_1(x) > 0 \wedge F_2(x) > 0\} = \emptyset$. *This continuous trajectory generates four discrete events from the following state transitions*

$$\begin{aligned} Q_1 &\rightsquigarrow Q_2 \\ Q_2 &\rightsquigarrow Q_1 \\ Q_1 &\rightsquigarrow Q_3 \\ Q_3 &\rightsquigarrow Q_1 \end{aligned}$$

The discrete event labels may be evaluated using the event labeling function

$$\begin{aligned} \sigma_{1,2} &= \Lambda(Q_1, Q_2) \\ \sigma_{2,1} &= \Lambda(Q_2, Q_1) \\ \sigma_{1,3} &= \Lambda(Q_1, Q_3) \\ \sigma_{3,1} &= \Lambda(Q_3, Q_1) \end{aligned}$$

Example 3.4.2 *A further example will illustrate the natural expressiveness of the functional partitioning technique, using a simple nonlinear ODE model of pendulum with damping. The dynamics are described by*

$$ml\frac{d^2\theta}{dt^2} + bl\frac{d\theta}{dt} + mg\sin\theta = 0$$

where m *is the mass of the pendulum bob and* l *its length,* b *is a friction term,* g *is*

the acceleration due to gravity, and, θ is the pendulum angle . The state vector shall be defined as

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}$$

and then the state equations are

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\frac{g}{l}\sin x_1 - \frac{b}{m}x_2 \end{aligned}$$

The total energy for this system is defined as the sum of the kinetic energy K and potential energy P, which we will define in the rotational coordinate frame.

$$E = K + P$$

where kinetic energy is

$$K = \frac{1}{2}m(lx_2)^2$$

and the potential energy is

$$P = -mgl(\cos x_1)$$

The total energy expression E represents an equal energy surface when plotted as a function of the state variables. If we wish to partition the state space such that the states represent energy levels, we use the following functional

$$F(x) = \frac{1}{2}m(lx_2)^2 + mgl(1 - \cos x_1) - 9.81$$

this gives a hypersurface that partitions the state space into two discrete states; system energy greater than 9.81 J and system energy less than 9.81 J. The potential energy term in the expression has been modified to change datum (zero energy when the bob is at the bottom).This functional is depicted in Fig. 3-7 as a shaded grey surface. A

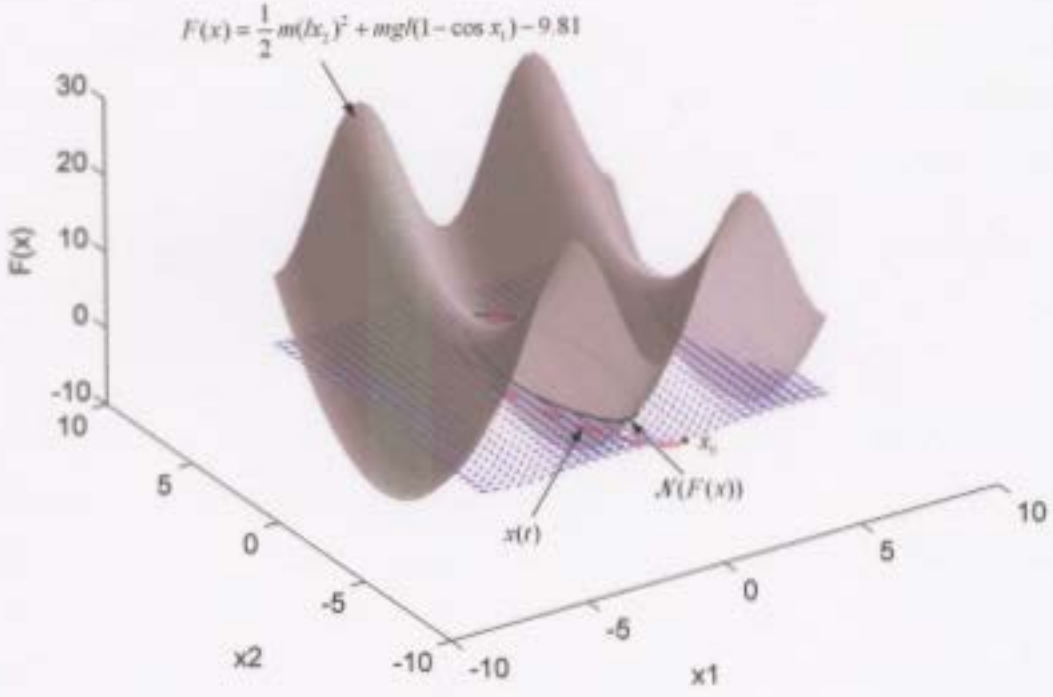


Figure 3-7: A partitioning functional $F(x)$ based on the total energy in the pendulum system.

trajectory $x(t)$ originating with initial condition x_0 is depicted as a dashed line in the phase plane. As expected, this functional partitions the state space into two states $Q \in \mathcal{X}$ as follows

$$\begin{aligned} Q_1 &= \{x : F(x) > 0\} \\ Q_2 &= \{x : F(x) < 0\} \end{aligned}$$

In Fig. 3-8, the view has been changed to examine the phase plane and the functional surface has been removed. The shaded area in the figure indicates Q_2, a state representing energy less than 9.81 J. A trajectory of the pendulum system has been plotted in a dotted line. The hypersurface $N(F(x))$, indicated by the ellipse-like line represents an equal-energy contour on the phase plane of 9.81 *J. This particular trajectory will generate a single event due to the state transition $Q_1 \rightsquigarrow Q_2$*

3.5 Conclusions

The discrete abstraction technique presented in this chapter, based on a family of partitioning functionals, is flexible and presents a natural way to extract discrete event information from a continuous model. However, this is only one possible means of inducing a partition on a system state space. For example, the familiar grid-like quantization that an A/D convertor produces is actually a specific case of a functional-based partition. It is important to discuss the detailed mechanics of how a partition is induced on the state space and how discrete events are generated within this framework. However, the details of partitioning should not influence the generality of the control theory that is developed in the following chapters.

It is preferable to be more general than this before proceeding; ultimately, the most general requirement is that if a system is viewed as a sort of "black box" – an event generator (as in Figure 3.1), we wish this generator to produce a finite number of events, for a finite time window. Alternatively, the requirement can be restated so

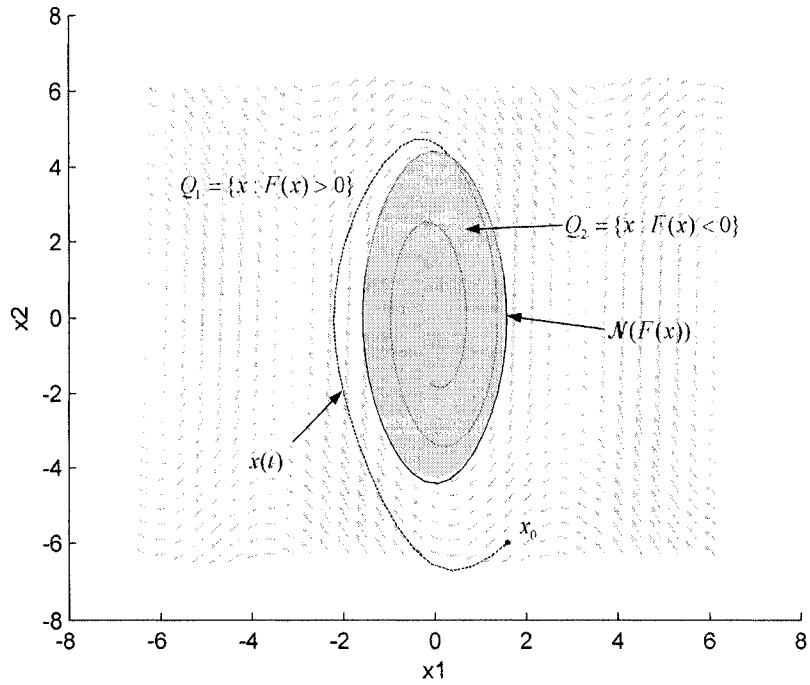


Figure 3-8: Pendulum phase plane with hypersurface $\mathcal{N}(F(x))$ representing an equal energy contour. A single event will be generated by the example trajectory $x(t)$ (dotted line).

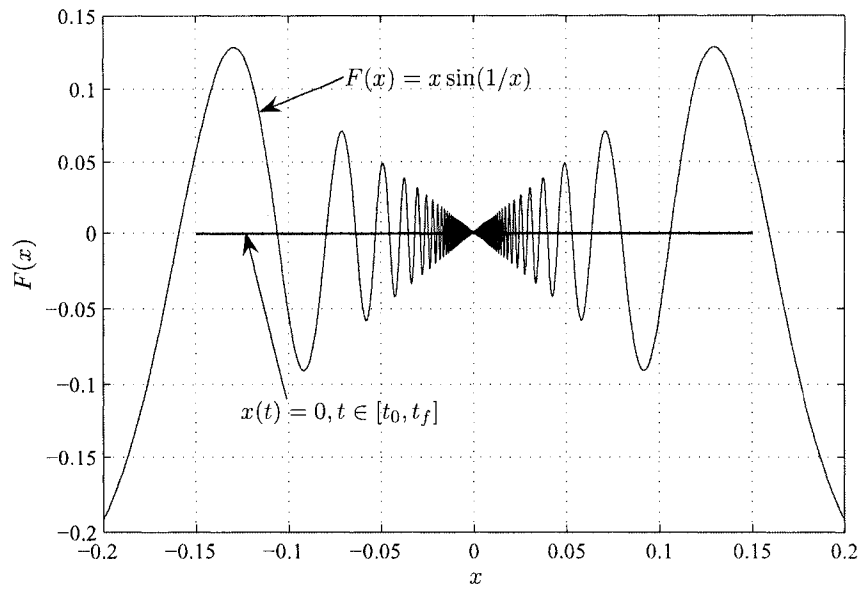


Figure 3-9: A continuous functional that produces a finite partition $F(x) = xsin(1/x)$, and a trajectory x on a finite interval, may generate a infinite number of transitions.

that the system transitions a finite number of times amongst its discrete states in a finite time window. A variety of "pathological" conditions may cause this condition to be violated, including: a) infinite "ripples" in the partitioning functionals, b) infinite ripples in the continuous trajectories and c) zeno switching behaviour. For now, we will confine the discussion to (a) and (b), since switching behavior will be considered in a later chapter. To demonstrate how these conditions can lead to infinite behaviour, we will examine two examples that use the functional partitioning framework of §3.2–§3.3.

We have shown that a finite set of functionals induces a finite set of partitioned regions and therefore corresponding finite sets of discrete state and event labels for the discrete abstraction. However, there is no guarantee that a given continuous trajectory traversing this partition will produce a finite string of events, as we will show in the following example.

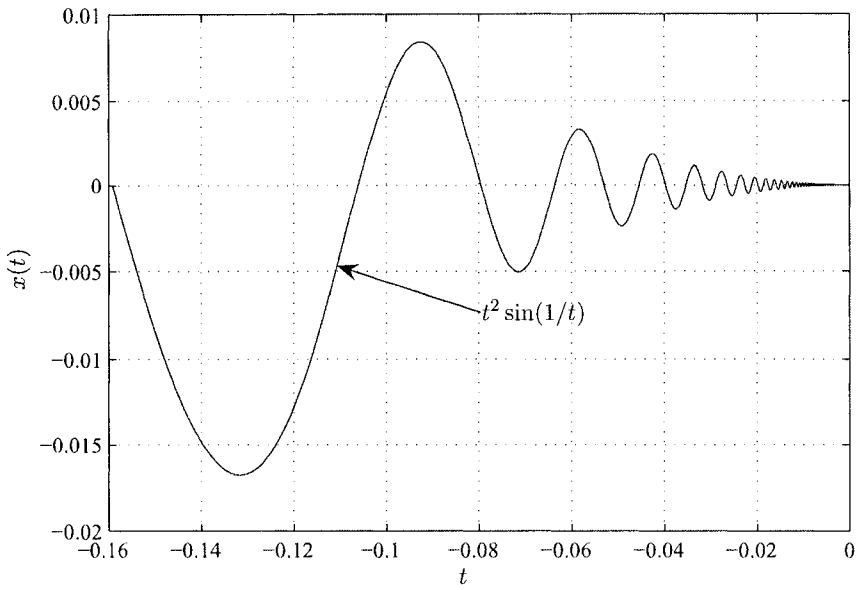


Figure 3-10: The trajectory $x(t) = t^2 sin(1/t)$ on a finite interval, produces an infinite number of transitions.

Example 3.5.1 *Consider a continuous dynamical system modeled by* $\dot{x} = f(x,t)$ *with system state space* $x \in \mathbb{R}$ *and let* F *be a smooth continuous functional*

$$F(x) = \begin{cases} x \sin(\frac{1}{x}) & x \neq 0 \\ 0 & x = 0 \end{cases}$$

Then suppose there exists a continuous trajectory x *which is a solution of* f*, for a finite time interval* $[t_0, t_f]$ *such that* $x(t) = 0$*, for all* $t \in [t_0, t_f]$ *and the initial condition* $x_0 = x(t_0) = -0.15$ *(Fig. 3-9). Although this partition induces a pair of regions, or discrete states, on a finite time interval this trajectory can clearly generate an infinite number of events.*

Likewise, a similar example can be contrived to show that certain continuous trajectories may also lead to infinite events.

Example 3.5.2 *Suppose the dynamics of a system are modeled by the Lipschitz-*

continuous ODE, with initial condition $x_0 = 0$

$$\dot{x} = f(x,t) = \begin{cases} 2t\sin(\frac{1}{t}) - \cos(\frac{1}{t}) & t \neq 0, \forall x \\ 0 & t = 0, \forall x \end{cases}$$

The generalized solution of such a system is

$$x(t) = \begin{cases} t^2\sin(\frac{1}{t}) & t \neq 0 \\ 0 & t = 0 \end{cases}$$

which is plotted in Fig. 3-10 for a time interval $t = [t_0, t_f) = [-0.1, 0)$. *Now, if we define a functional* F *as follows*

$$F(x) = 0, \text{ for all } x$$

the trajectory will generate an infinite number of events as $t \rightarrow t_f = 0$.

Example 3.5.1 illustrates the situation in which the functional is responsible for generating infinite events in an finite time interval. This is a somewhat contrived example, and can generally be avoided since the specification of the functional is under the control of the designer. However, the case where the solution itself gives rise to the infinite behaviour, as in example 3.5.2, is more difficult to avoid. On the other hand, if the solution is the result of an ordinary differential equation solver, since the limitations of numerical precision of the computer will prevent an infinite solution from occurring in the first place. For results in future chapters, the primary assumption is that the partitioning of the state space does not lead to any of the pathological conditions just outlined.

Chapter 4

Switched Continuous Model

4.1 Introduction

In the previous chapter, the techniques of state quantization and discrete event generation were developed for the discrete abstraction of a single continuous dynamical model. This continuous system model has an output interface, behaving as an event generator to the external discrete-event world. In this chapter, we will develop a hybrid model based on a collection of embedded continuous system models. The complete model is a form of hybrid model, called a switched continuous model (SCM). In (Koutsoukos et al. 2000), input events were linked to corresponding actuator actions in the continuous model. The SCM broadens this scope by linking the discrete input events to a complete change (or switch) in continuous dynamics. An approach similar to this was taken in (Abdelwahed et al. 2005).

Graphically, the switched continuous model is depicted in Fig. 4-1. The switched continuous model has a collection of continuous system models, each with a DE output abstraction layer. The *input* connection controls the switching between these embedded continuous dynamical models.

The objective is to provide a discrete event modeling environment that can ultimately be used to construct a supervisory controller. As a result, the switching

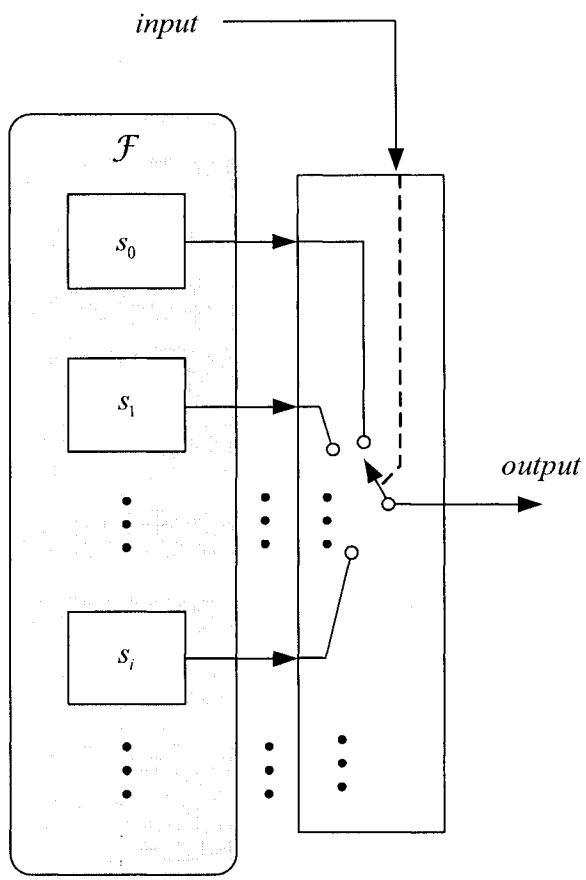


Figure 4-1: Graphical representation of a switched continuous model.

behaviour due to the *input* is determined by the need to exercise control and to maintain finite discrete event behaviour. Two types of switching will be considered, Case I, in which the switching between continuous dynamics is permitted at some time interval Δt only; and Case II, in which the continuous dynamics are permitted to switch either at a time interval, or upon a state transition. In general, the focus of the thesis will be on the Case II model, but Case I is developed since it is instructive to consider.

4.2 Switched Continuous Model

We define a switched continuous model and two possible switching methodologies. We begin by formalizing the definition of a continuous system model, as described in Chapter 3, in which we include the partitioning functionals.

Definition 4.2.1 (Continuous System Model) *Let a continuous system model, s, be defined as a five-tuple:*

$$s = (f, \Psi, \Lambda, V, x_0)$$

where:

f *is a Lipschitz-continuous ordinary differential equation,* $\dot{x} = f(x, t)$,

Ψ *is a family of partitioning functionals,*

Λ *is the output event function,*

V *is a state labeling function,*

x_0 *is the initial condition,* $x(t_0)$.

The continuous system model (CSM) is wrapped in a discrete abstraction layer (Fig. 4-2), allowing the embedded continuous model to exhibit the behaviour of a discrete event system model.

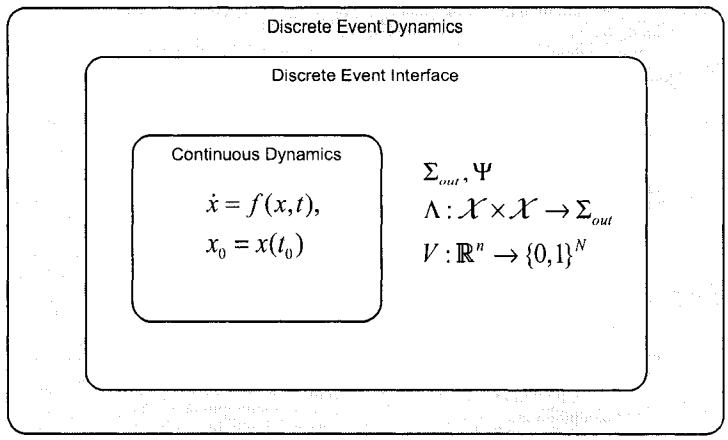


Figure 4-2: Block diagram for the continuous system model of Chapter 3.

The switched continuous model is an automaton-like model composed of a family (set) of CSMs. It is desirable for this model to have the ability to exhibit hybrid behaviour and to allow for control synthesis. Additionally, the model framework is deliberately simple in order to expedite the proof of the theoretical properties. Later in Chapter 6, the implementation-specific details of the model will be given.

Definition 4.2.2 (Family of continuous system models) *Let* $\mathcal{F} = \{s_0, s_1, \ldots s_i \ldots\} = \{s_i\}$ *be a family of continuous system models of possibly infinite cardinality. Each element* $s \in \mathcal{F}$ *is a continuous system model with discrete abstraction layer as in Def. 4.2.1.*

Definition 4.2.3 (Enabled System Function) *Let* $\mathcal{A} = \{a \subset \mathcal{F} : 1 \leq |a| < \infty\}$ *be the set of non-empty finite subsets of* $\mathcal{F}$*, then an* ***enabled system function*** Γ *is defined as a function*

$$\Gamma : \mathcal{F} \rightarrow \mathcal{A}$$

Definition 4.2.4 (Switched Continuous Model) *Let a switched continuous model* G *be defined as an automaton-like triple*

$$G = (\mathcal{F}, \Gamma, s_0)$$

where:

$\mathcal{F}$ *is a family of continuous system models (Def. 4.2.2)*

Γ *is the enabled system function (Def. 4.2.3)*

s_0 *is the initial continuous system model*

The switched continuous model has behaviour similar to a multiplexor (Fig. 4-1). The switch function permits only one continuous system model to be selected at any instant in time.

Definition 4.2.5 (Execution) *An* ***execution*** *of a switched continuous model is defined as a sequence v, of selected continuous system models*

$$v = \{s_0, s_1, \ldots s_\alpha \ldots\}$$

An execution of a system modeled by a SCM starts with the selection of the initial continuous system model s_0. At some point in time, the system will switch to, or select, another continuous system model.

Definition 4.2.6 (Choice Point) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a SCM, and let s' be the currently selected CSM. The point in time at which the SCM switches execution to another continuous system model s'' is known as a* ***choice point.*** *The switches occur either due to a time event (tick, or t) or due to a state transition within the currently selected CSM, s'.*

A choice point can be thought of as the moment a controller exercises control. In Case I switching, choice points occur at some time interval, not necessarily with a constant time spacing. Case I switching is analogous to that of a PLC in industrial practice, in which the controller polls it's inputs and updates control outputs on some time schedule.

The enabled system function Γ, of Def. 4.2.3, is an implementation-dependent map. It returns a finite set of continuous system models and is invoked at the choice points. The function is a convenient method of defining the future execution of the SCM recursively, and is an abstraction of the actual control algorithm. Since the switched continuous model is an abstraction of a real system, there must always be a system eligible for execution, hence the requirement that the set of enabled system models be non-empty $|\Gamma(s)| \geq 1$.

Definition 4.2.7 (Successor Continuous System Model) *Let $s \in \mathcal{F}$ be a continuous system model, then any element of the family, $s' \in \Gamma(s)$ at some choice point is a **successor CSM** of s.*

Definition 4.2.8 (Control) *Let the currently selected model be $s \in \mathcal{F}$. The selection of a single successor continuous system model s' from the set of eligible successor CSMs $\Gamma(s)$, referred to as **control** of the modeled system.*

In Fig. 4-1 the SCM is illustrated as a multiplexor with a switching (or control) input. We will associate the selection of continuous system models by a controller with discrete (input) events. At each choice point, the controller may select a continuous system model to execute. To ensure the finiteness of the switching behaviour in the SCM, the control choice must always be finite. Recall from Def. 4.2.3, $\Gamma : \mathcal{F} \rightarrow \mathcal{A}$, where $\mathcal{A} = \{\alpha \subset \mathcal{F} : 1 \leq |\alpha| < \infty\}$ represents the finite set of enabled continuous system model for a particular choice point. The SCM input control interface associates each element $s \in \alpha$, with a unique input event label.

Definition 4.2.9 (Input Event Set) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be an SCM. Then let Σ_{in} be a set of input event labels such that there exists a unique input event label $\sigma_i \in \Sigma_{in}$ for every enabled continuous system model at each choice point $\forall s_i \in \Gamma(s)$.*

The historical (past) execution of a CSM is clearly a simple sequence. However, due to the fact that $|\Gamma(s)|$ at a choice point may be greater than one, then there are

a number of possible future executions of the SCM, and a branch in the future (or predicted) execution occurs.

Example 4.2.1 *In Fig. 4-3, a future execution tree (prediction) is illustrated. Note that the CSM subscripts for the diagram do not necessarily indicate any sort of sequential order, they are simply distinguishing labels. Beginning with the initial CSM* s_0*, a choice point occurs, indicated on the time axis as a tick (of the universal timebase,* t*). Evaluation of the enabled system function at the conclusion of the* s_0*,*

$$\Gamma(s_0) = \{s_1, s_2, s_3\} \subset \mathcal{F}$$

then, projecting forward in time, each of the systems represents a branch in the future execution of the SCM. Again, following the top branch, s_1*, at the next system choice point we get,*

$$\Gamma(s_1) = \{s_4, s_5, s_6\} \subset \mathcal{F}$$

To summarize, Γ reduces the infinite possible continuous system models $\mathcal{F}$, to a finite subset of continuous models which are eligible for execution at each choice point. Therefore, a prediction of the future CSM selection, is a branching tree. When the SCM is executed, only a single CSM $s' \in \Gamma(s) \subset A$ is selected at any choice point, and the execution of the model is a sequence of continuous system models.

4.3 Prediction - Case I Switching

We will now examine the future execution of the SCM. It is desirable to establish a bound on the number of systems in the future execution; that is, the total reachable systems under the time switching regime (Case I). To facilitate this, we begin by defining an n-ary version of the enabled system function.

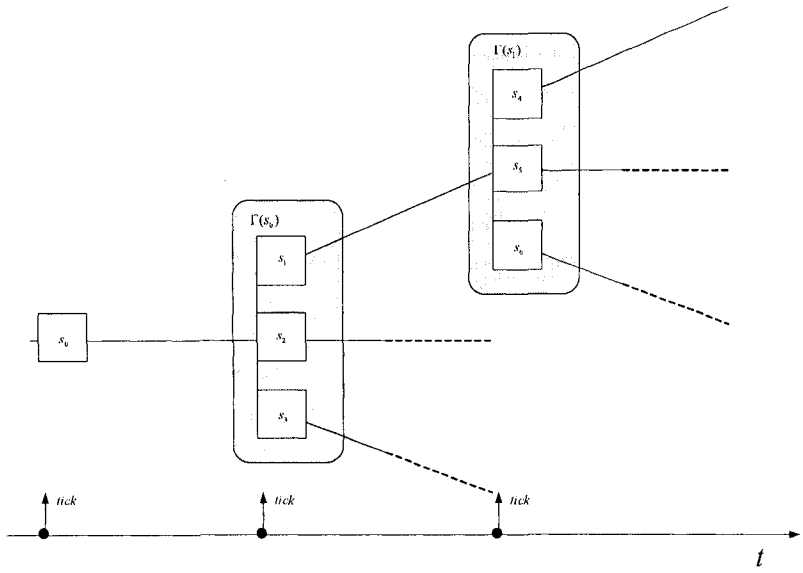


Figure 4-3: Diagram illustrates the branching of an SCM future execution.

Definition 4.3.1 (Enabled System Operator) *Let G be an SCM with enabled system function Γ, and let $\mathcal{S} = \{s_i\}$ be a family of continuous system models, then the* ***enabled system operator*** *Γ_f : is defined as follows*

$$\Gamma_f(\mathcal{S}) = \bigcup_{\forall s_i \in \mathcal{S}} \Gamma(s_i), \tag{4.1}$$

Clearly, for any system, $s_i \in \mathcal{S}$, then $\Gamma(s_i) \subseteq \Gamma_f(\mathcal{S})$. The distinction between Γ and Γ_f is that at each choice point, Γ returns the family of enabled continuous system models, while for each time interval, Γ_f returns the family of continuous system models for all the choice points at a particular time step. An execution of the switched continuous system can be described recursively in terms of the enabled system operator, forming a tree-like graph with choice (branch) points occurring at some time interval, Δt (Fig. 4-4).

Any branch in this figure is a possible future execution of the system, taking the system from the currently selected system model to one on the time horizon, $t_0 + k\Delta t$.

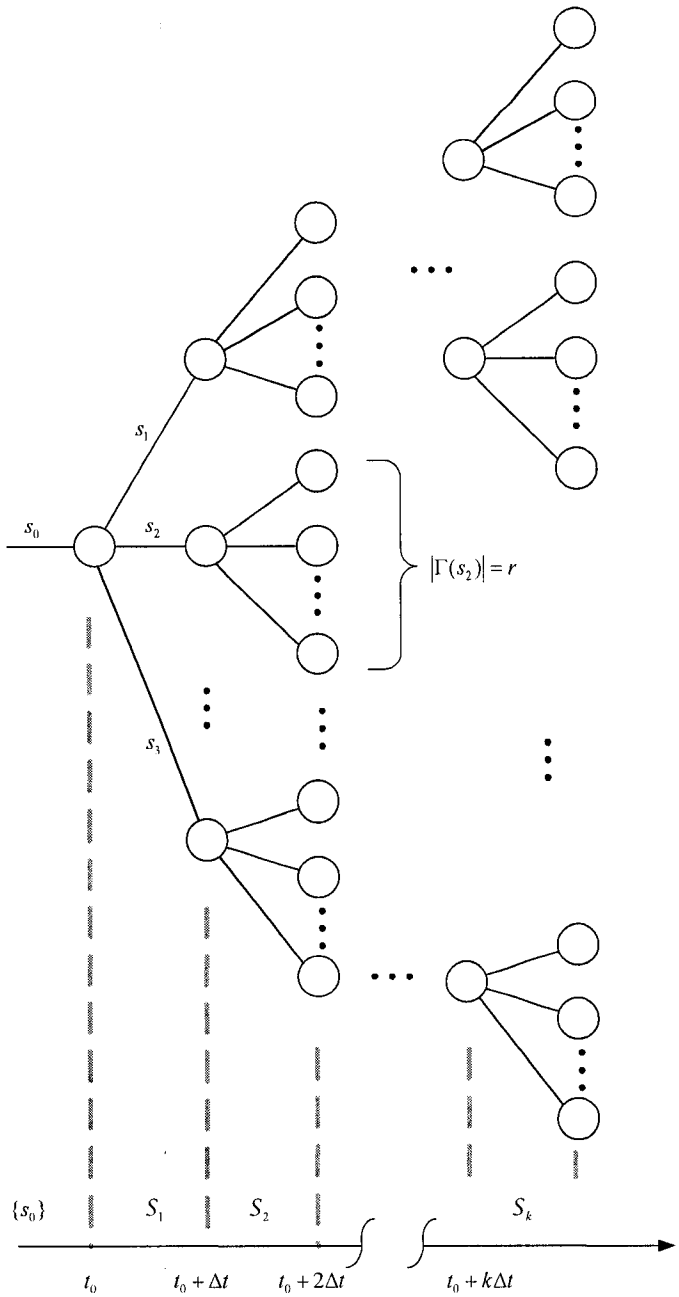


Figure 4-4: An execution of a switched continuous model with choice points at time interval Δt.

Definition 4.3.2 (Families of Continuous System Models) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model and let p be the number of time intervals, Δt, over which the model will be executed. Let $\mathcal{S}_k$ be the kth family of continuous system models, at integer $k \geq 1$ time steps from the selection of the initial system, s_0. This family can be expressed in terms of recursive enabled system operations:*

$$\mathcal{S}_k = \underbrace{\Gamma_f(\ldots\Gamma_f}_{k \text{ times}}(\{s_0\})\ldots) \tag{4.2}$$

Time is implicit in this equation and in Fig. 4-4, with each nested Γ_f operator being evaluated at a choice point. Each choice point occurs due to the passage of an interval of time, Δt.

We now show that the number of eligible continuous system models for future execution may grow exponentially with time. The following proof establishes the size of Γ_f at any time step.

Lemma 4.3.1 *For a switched continuous model, the cardinality of the family of enabled continuous system models at the kth time step for integer $k \geq 1$, as per Eq. 4.2, has upper bound of $|\mathcal{S}_k| = r^k$, provided that at each choice point there are at most r possible continuous system models to switch amongst.*

Proof. The lemma can be proven by assuming maximal switching, and using induction on k. For the base case, $k = 1$, the cardinality of the first family of enabled continuous system models, $|\mathcal{S}_1| \leq |\Gamma(\{s_0\})| = r$, which is consistent with r^k since $r^1 = r$. The inductive hypothesis is that $|\mathcal{S}_k| \leq r^k$. The family of continuous system models enabled at k switches from the initial system, s_0 is $\mathcal{S}_k$ and the cardinality is

$|\mathcal{S}_k| \leq r^k$. By Eq. 4.2 then:

$$\begin{aligned}
|\mathcal{S}_{k+1}| &= |\Gamma_f(\mathcal{S}_k)| \\
&= \left| \bigcup_{\forall s_i \in \mathcal{S}_k} \Gamma(s_i) \right| \\
&= |\Gamma(s_1)| + |\Gamma(s_2)| + \ldots + |\Gamma(s_{r^k})| \\
&\leq \underbrace{r + r + \ldots + r}_{r^k \text{ times}} \\
&= r^k \times r \\
|\mathcal{S}_{k+1}| &\leq r^{k+1}
\end{aligned}$$

thus proving the inductive hypothesis to be correct. ■

Definition 4.3.3 (Reachable CSM) *Let $(\mathcal{F}, \Gamma, s_0)$ be a SCM. A continuous system model $s' \in \mathcal{F}$ is said to be* ***reachable*** *from s_0 if there is a future execution v such that $s' \in v$*

$$v = \{s_0, \ldots, s', \ldots\}$$

Now we wish to get an expression for the size of the set of reachable continuous system models $\mathcal{S}_R$ for some arbitrary number of time steps.

Definition 4.3.4 (Reachable Set of Continuous System Models) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model, then the family of continuous system models that is reachable from s_0 (including s_0 itself) in p time steps, $\mathcal{S}_R$, is defined recursively as the union for $k = 1, 2, \ldots, p$, of each of the families in Eq. 4.2 , as follows:*

$$\mathcal{S}_R = \{s_0\} \bigcup \left(\bigcup_{k=1}^{p} \underbrace{\Gamma_f(\ldots \Gamma_f}_{k \text{ times}}(\{s_0\}) \ldots) \right) \tag{4.3}$$

Lemma 4.3.2 *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model executed on a finite number of time intervals, $p \geq 1$. Let the cardinality of the family of enabled continuous system models at any choice point be $|\Gamma(s_i)| \leq r$ for all $s_i \in \mathcal{F}$, then the family of*

continuous system models, $\mathcal{S}_R$, reachable from the initial continuous system model, s_0, as defined in Eq. 4.3, has the following cardinality:

$$|\mathcal{S}_R| \leq \frac{r^{p+1}-1}{r-1} \tag{4.4}$$

Proof. *By Lemma 4.3.1, the cardinality of the kth family of enabled continuous system models is given by $|\mathcal{S}_k| = r^k$. And the reachable set of continuous system models is, by Eq. 4.3,*

$$\begin{aligned} |\mathcal{S}_R| &= \left| \{s_0\} \bigcup \left[\bigcup_{k=1}^{p} \underbrace{\Gamma_f(\ldots\Gamma_f}_{k}(\{s_0\})\ldots) \right] \right| \\ &= 1 + |\mathcal{S}_1| + |\mathcal{S}_2| + \ldots + |\mathcal{S}_p| \\ &\leq r^0 + r^1 + r^2 + \ldots r^p \end{aligned}$$

the sum of this geometric series is

$$\sum_{k=0}^{p} r^k = \begin{cases} \frac{r^{p+1}-1}{r-1} & r \neq 1 \\ p+1 & r = 1 \end{cases}$$

■

Thus we have proven a general closed form expression for the upper bound for the cardinality of the reachable continuous system models. Since the number of possible enabled systems at any choice point is unpredictable, it is possible only to establish an expression for the upper bound on the cardinality of the set of reachable continuous system models.

Definition 4.3.5 *Let **maximal switching** be defined as: for all choice points, there are exactly $|\Gamma(s_i)| = r$ continuous system models to switch between.*

Definition 4.3.6 *Let **minimal switching** be defined as for all choice points, there is one choice of continuous system model to switch to, $|\Gamma(s_i)| = 1$, for all $s_i \in \mathcal{S}_R$.*

Theorem 4.3.1 (Continuous System Reachability) *Let $G_c = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model. For a switching time interval of Δt, a finite number of time intervals, $p > 0$, and the maximum enabled systems at any choice point finite $1 \leq |\Gamma(s_i)| \leq r$, for all $s_i \in \mathcal{F}$, then the family of continuous system models, $\mathcal{S}_R$, reachable from the initial continuous system model, s_0, as defined in Eq. 4.2, is finite, and furthermore,*

$$p + 1 \leq |\mathcal{S}_R| \leq \frac{r^{p+1} - 1}{r - 1}$$

Proof. The reachable family of continuous system models is bounded above by maximal switching, $|\Gamma(s_i)| = r$, for all $s_i \in \mathcal{S}_R$. By Lemma 4.3.2, if both p and r are finite, the cardinality of summation of Eq. 4.4, $|\mathcal{S}_R|$, must also be finite. The lower bound is for the condition of minimal switching,

$$\begin{aligned} |\mathcal{S}_R| &= 1 + |\mathcal{S}_1| + |\mathcal{S}_2| + \ldots + |\mathcal{S}_p| \\ &= 1 + \underbrace{1 + \ldots + 1}_{p} \\ &= p + 1 \end{aligned}$$

■

4.4 Prediction - Case II Switching

In §4.2, the switched continuous model was constrained to switch between models on some time interval, Δt. It will now be extended to permit switching of continuous dynamics when a continuous trajectory transitions a partition boundary. This implies that additional choice points (branching points) may occur between time intervals. The goal is to show that the finite properties of this model are maintained under partition switching conditions, and to establish an upper bound on the cardinality of the reachable state space.

For purposes of controlling the model, it was stated earlier that time switching

(Case I) was analogous to the update cycle of an industrial PLC, occurring at some regular (or possibly multirate) scan time. Partition switching can be viewed as giving the controller the opportunity to react to an unscheduled alarm, similar to an interrupt-driven control action of a real-time control task. Thus the Case II SCM switching framework models the control of hybrid plants when a controller is permitted to perform both synchronous and asynchronous control actions on the plant.

Definition 4.4.1 (Partition Switching) *A choice point occurs due to a state transition occurring within the currently selected continuous system model.*

As was stated earlier, any complete branch from left to right in Fig. 4-5, is a possible future execution v of the SC system model.

Definition 4.4.2 (Execution Cardinality) *The cardinality of a finite execution v is the number of elements (continuous system models) in the sequence, and is indicated with the notation of set cardinality $|v|$.*

First we will determine the cardinality of the reachable set of continuous models for a single time interval Δt, then extend it to multiple time intervals.

Lemma 4.4.1 *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model with partition switching, and an upper bound on branching at each choice point, $|\Gamma(s_i)| \leq r$. Let $\mathcal{S}_{\Delta t}$ be the reachable continuous system models in one time interval Δt. Let v_m be an execution with the most partition switches in time interval, Δt, such that $|v_m| = q$. If every execution v has finite cardinality $0 \leq |v| \leq q$, then the family of reachable continuous system models from s_0 in one Δt is of finite cardinality:*

$$1 \leq |\mathcal{S}_{\Delta t}| \leq \frac{r^{q+1} - 1}{r - 1} \tag{4.5}$$

Proof. The proof is identical to that of Lemma 4.3.2. The family of enabled continuous system models at k switches from s_0 is denoted as $\mathcal{S}_k$, and its cardinality

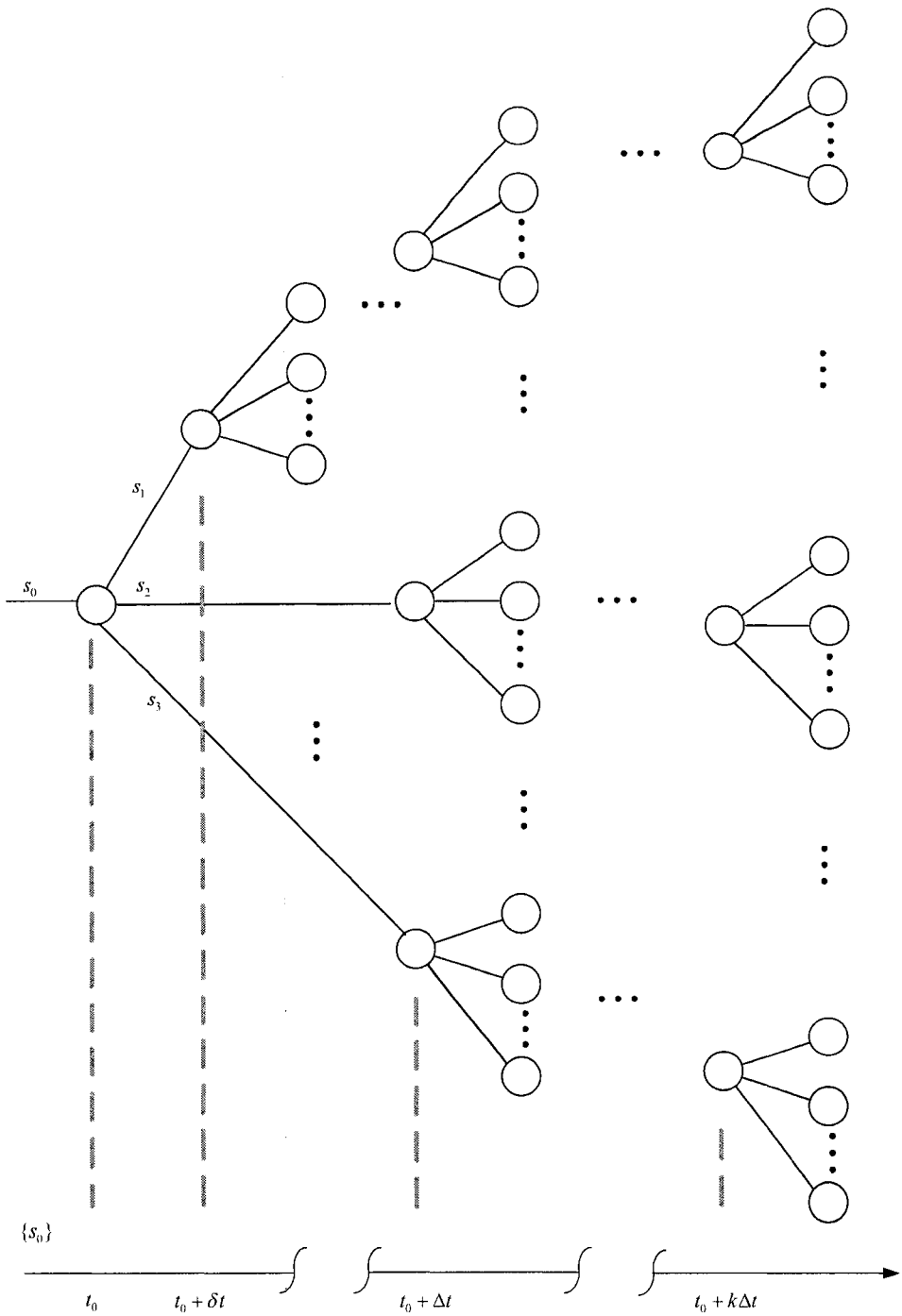


Figure 4-5: Case II switching structure, showing that event may occur at some time $\delta t \leq \Delta t$ that is within the controller switching interval due to state dependent switching.

$|\mathcal{S}_k| = r^k$ as before:

$$\begin{aligned} |\mathcal{S}_{\Delta t}| &= \left| \{s_0\} \bigcup \left[\bigcup_{k=1}^{q} \underbrace{\Gamma_f(\ldots \Gamma_f}_{k \text{ times}}(\{s_0(t)\})\ldots) \right] \right| \\ &\leq 1 + r^1 + r^2 + \ldots + r^q \\ &= \sum_{k=0}^{q} r^k \\ &= \frac{r^{q+1} - 1}{r - 1} \end{aligned}$$

Which is the upper bound of the family of reachable continuous system models. The lower bound of 1 comes from $|\{s_0\}| = 1$. ■

Now we show that the set of reachable continuous system models for an arbitrary number of time steps is also finite.

Theorem 4.4.1 (Reachable Continuous System Models(II)) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model with a switching time interval of Δt, having a finite integer multiple of time intervals, $p > 0$, and an upper bound on branching at each choice point, $|\Gamma(s_i)| \leq r$. If the maximum number of partition switches in any clock interval Δt has a maximum such that all executions $|v_i| \leq q$, where $q \geq 0$ is an integer, then the family of continuous system models reachable from s_0 in p time intervals,Δt, is finite:*

$$p + 1 \leq |\mathcal{S}_R| \leq \frac{r^{pq+1} - 1}{r - 1} \tag{4.6}$$

Proof. Let $v_m \subseteq \mathcal{S}$ be a an execution on interval $[t_0, t_0 + p\Delta t)$. Let $v_1 \subset v_m$ be an execution on time interval $[t_0, t_0 + \Delta t)$, $v_2 \subset v_m$ be an execution for time interval $[t_0 + \Delta t, t_0 + 2\Delta t)$ and so on. If for all v_i, $|v_i| = q$, then

$$\begin{aligned} |v_m| &= |v_1| + |v_2| + \ldots + |v_p| \\ &= \underbrace{q + q + \ldots + q}_{p \text{ times}} \\ &= pq \end{aligned}$$

The largest reachable set occurs if every switched continuous trajectory produces pq switches with maximal switching, $|\Gamma(\cdot)| = r$, at each of the corresponding choice points. The proof follows directly from Lemma 4.3.2:

$$\begin{aligned} |\mathcal{S}_R| &= \left| \{s_0\} \bigcup \left[\bigcup_{k=1}^{pq} \underbrace{\Gamma_f(\ldots\Gamma_f}_{k}(\{s_0\})\ldots) \right] \right| \\ &\leq 1 + r^1 + r^2 + \ldots + r^{pq} \\ &= \sum_{k=0}^{pq} r^k \\ &= \frac{r^{pq+1} - 1}{r - 1}, \text{ for } r > 0 \end{aligned}$$

and the lower bound is for minimal switching, $r = |\Gamma(\cdot)| = 1$, and no partition switches within each time interval, $q = 0$ for all choice points. The lower bound reduces to that of Case I as in theorem 4.3.1:

$$\begin{aligned} |\mathcal{S}_R| &= 1 + |\mathcal{S}_1| + |\mathcal{S}_2| + \ldots + |\mathcal{S}_p| \\ &\geq 1 + \underbrace{1 + 1 + \ldots + 1}_{p \text{ times}} \\ &= p + 1 \end{aligned}$$

■

So, for a maximum finite number of partition switches within one time interval q, a maximum finite number of eligible CSMs at each switch r, and a finite number of time intervals p, the reachable set of continuous system models is also finite.

4.5 Continuous Dynamics

Up to this point, we have deliberately ignored the discrete and continuous dynamics of the SCM, as we have dealt with the qualities of execution and reachability of the continuous system models within the two switching frameworks. The details of the

underlying CSMs (time, continuous dynamics and discrete event dynamics) and their contribution to the switching was hinted at in the definition of choice points. We now examine the reachability properties of the continuous state space of the SCM, in the context of the switching frameworks.

Definition 4.5.1 (Solution of Continuous System Model) *Let* $s_i = (f, \Psi, x_0)$ *be a continuous system model for the time interval* $[t_0, t_f)$ *then the* ***solution*** *to the IVP thus posed is*

$$x_i(t) = x_0 + \int_{t_0}^{t_f} f(x, \tau) d\tau, \text{ for } t \in [t_0, t_f)$$

exists and is unique.

So for the period of time while a continuous state model is selected, there is a continuous state vector that is uniquely determined by the CSM's initial condition and dynamics. Since an execution v of the SCM consists of a sequence of selected CSMs, then the continuous state of the SCM is easily defined in terms of the corresponding sequence of solutions (or continuous trajectories).

Notation 4.5.1 *Care should be taken to distinguish a point in a solution from a solution on an interval. In general, a point of a solution will be denoted as a solution evaluated at a point in time*

$$x(t_a) \in \mathbb{R}^n \text{ is the point of a solution evaluated at time } t_a$$

For a solution, $x_a(t)$ *the reference to time will be omitted to reduce notational complexity*

$$x_a \text{ is a solution as a function of time, } a \text{ is an index}$$

An exception to these notation conventions is the initial condition of a continuous system model, $x_0 \in \mathbb{R}^n$ *which is a point. If there is likely to be confusion, the solution will be referred to as a function of time, while a point is a solution explicitly evaluated*

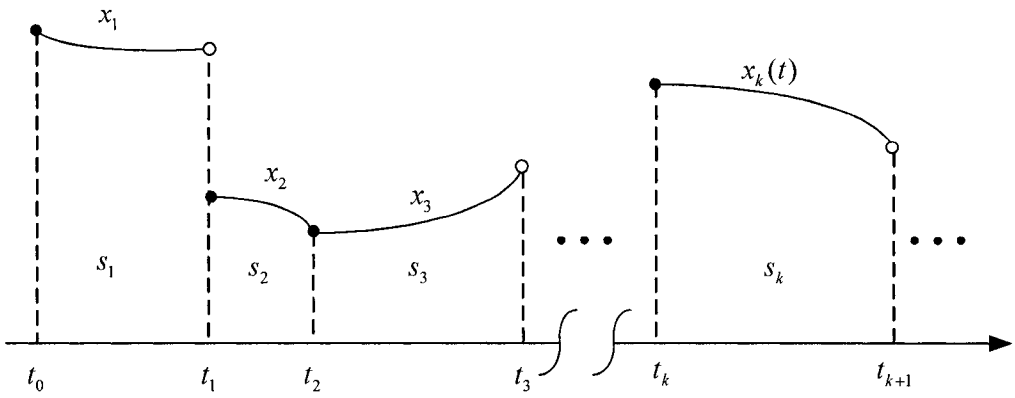


Figure 4-6: Switched continuous model execution $v = \{s_1, s_2, s_3 \ldots, s_k, \ldots\}$ and its corresponding switched continuous trajectory $\xi = \{x_1, x_2, x_3, \ldots, x_k, \ldots\}$.

at some point in time.

Definition 4.5.2 (Switched Continuous Trajectory) *Let* $v = \{s_0, s_1, \ldots s_k \ldots\}$ *be an execution of an SCM, then the* ***switched continuous trajectory*** ξ*, is the sequence of matching solutions to the IVPs posed by each continuous system model on the respective time intervals*

$$\xi = \{x_0, x_1, \ldots x_k \ldots\}$$

Given this definition, Fig. 4-6 illustrates a hypothetical SCM execution and its corresponding switched continuous trajectory (Def. 4.5.2). This is a typical hybrid system trajectory, having continuous runs interspersed with discrete changes in state and/or dynamics.

For Case I switching, the choice points are due to a time-related event, a *tick*. As pictured in Fig. 4-5, the choice points are not equally spaced. For Case II switching, the choice points are due either to state transitions (discrete output events) or to time related *tick* events. Either way, the choice points originate from within the continuous system model.

Similarly to the definition for a successor CSM, we may define a successor continuous trajectory.

Definition 4.5.3 (Successor Trajectory) *Let s_b be a successor continuous system model of s_a. Let x_a be the solution to the IVP posed by s_a on the time interval $[t_0, t_1)$. Then x_b is the solution to the IVP posed by $s_b \in \Gamma(s_a)$ on the time interval $[t_1, t_2)$., where $t_0 < t_1 < t_2$, and x_b is a successor continuous trajectory (or, alternately, the successor solution) of x_a. Notationally, we can say*

$$x_b = succ(x_a)$$

Definition 4.5.4 (Predecessor Trajectory) *If x_b is a successor continuous trajectory of x_a, then x_a is the predecessor continuous trajectory of x_b.*

$$x_a = pre(x_b)$$

The successor function can be used to form an alternative definition of the switched continuous trajectory using recursion.

Definition 4.5.5 (Switched Continuous Trajectory) *A switched continuous trajectory ξ, is a set of continuous trajectories:*

$$\xi = \{x_i : x_{i+1} = succ(x_i)\},\ i = 1, 2, ...$$

The definition of a successor trajectory (Def. 4.5.3) ensures that any switched continuous trajectory ξ has no "gaps" in time, nor does it have any "overlaps" in time.

4.6 Continuous State Reachability

In §4.3 and §4.4, we examined the reachable continuous system models, and in the previous section (§4.5), the relationship between a continuous system model and its corresponding continuous state space was detailed. The reachable continuous state space of the SCM can be defined in terms of switched continuous trajectories.

4.6.1 Case I Switching

Given a finite prediction horizon in time, it is desirable to find an expression for the reachable continuous state space. Let $G = (\mathcal{F}, \Gamma, s_0)$ be a SCM and let the state prediction be defined for the time interval $T = [t_0, t_f)$, where t_0 is the initial execution (or simulation) time and t_f be the time horizon relative to t_0.

Definition 4.6.1 (Complete Switched Continuous Trajectory) *A switched continuous trajectory, $\xi = \{x_1, x_2, \ldots, x_\alpha\}$ is said to be* complete *on some time interval (t_0, t_f) if x_1 is a solution to an IVP over a time interval starting at time $t > t_0$, and x_α is a solution to an IVP over a time interval ending at t_f.*

A single complete SCT, ξ, on some interval of time is a depth-first reach, and represents the reachable continuous state space corresponding to an execution v of the SCM, G.

Definition 4.6.2 (Reachable Continuous Solutions) *Let G be a switched continuous model, then the state space reachable from x_0 (the initial condition specified by CSM s_0) on some time interval, $T = [t_0, t_f)$ is defined as:*

$$\mathcal{R} = \bigcup_{\forall \xi_i} \xi_i, \ \xi_i \text{ are complete with respect to } T$$

This definition indicates that the union of all complete switched continuous trajectories for some time interval is the reachable state space. Computationally, the

reachable state space can be assembled by the union of all depth-first reaches.

Alternatively, the reachable state space can be defined in terms of the reachable continuous system models.

Definition 4.6.3 (Reachable State Space) *Let $\mathcal{S}_k$ be the family of continuous system models reachable from s_0 in k time steps, as in Eq. 4.2. Let the time horizon be a finite number, p, of time steps, Δt. The reachable state space of a switched continuous model is defined as the set of all solutions to the IVP's posed by the reachable sets of continuous system models*

$$\mathcal{R} = \{x \mid \exists k : 1 \leq k \leq p,\ \exists s \in \mathcal{S}_k,\ x \text{ is a solution to } s\}$$

Lemma 4.6.1 (Finite Reachable State Space (I)) *The cardinality of the continuous solutions in the reachable state space $\mathcal{R}$ is finite:*

$$p + 1 \leq |\mathcal{R}| \leq \frac{1 - r^{p+1}}{1 - r} \tag{4.7}$$

Proof. Due to Def. 4.5.1, there exists a direct correspondence between elements the reach set $\mathcal{R}$, and the reachable state space such that for all $s \in \mathcal{S}_R$ there exists a unique solution $x \in \mathcal{R}$, by the earlier assumption of Lipschitz continuity of continuous dynamics. Therefore, the cardinality result of Theorem 4.3.1 also holds for the reachable state space $\mathcal{R}$. ■

4.6.2 Case II Switching

We will establish the cardinality bounds for the reachable state space for Case II switching. An important issue with a partition-switched model is the potential for zeno execution. Models that have instantaneous switching of dynamics have the potential for zeno execution. Zeno execution is technically an artifact of modeling, since no real system can be zeno (Zhang, Johansson, Lygeros and Sastry 2000). However, it

is an undesirable condition in a model or simulation since it leads to infinite switching in a finite period of time. In the case of the switched continuous model, only Case II is prone to exhibit zeno executions.

Definition 4.6.4 (Non-Zeno Switched Continuous Trajectory) *A switched continuous trajectory, ξ on some finite time interval, $\Delta t = [t_0, t_f)$, is nonzeno if $|\xi| \leq \infty$.*

Assumption of nonzeno characteristics is problematic, since it may be difficult to predict in advance that a model will exhibit zeno executions (Heymann, Lin, Meyer and Resmerita 2002). The assumption is that for Case II switching, all executions are nonzeno. We base this assumption on the premise that zeno execution can be avoided through the use of modified models, or implementation-specific modeling techniques, including temporal or spatial regularization (Johansson, Egerstedt, Lygeros and Sastry 1999) or other zeno solution extension techniques. Indeed, in Theorem 4.4.1, which claimed finiteness of the reachable set of continuous system models, there was an implicit assumption of nonzeno execution[1].

Lemma 4.6.2 (Finite Reachable State Space (II)) *The cardinality of the continuous solutions in the reachable state space $\mathcal{R}$ is finite and bounded above:*

$$|\mathcal{R}| \leq \frac{r^{pq+1} - 1}{r - 1}, \; r > 1 \tag{4.8}$$

Proof. For every continuous system model, $s_i \in \mathcal{S}_R$ there is a corresponding solution to the IVP, $x_i \in \mathcal{R}$, the cardinality of the reachable state space is identical to Eq. 4.6, the cardinality of the reachable continuous system models for Case II switching. ■

[1]Later, nonzenoness will be a necessary condition for existence of a controller (since the controller is model-based).

4.7 Discrete Event Dynamics

The previous sections have examined the reachable continuous properties of the SCM. This section will examine the discrete event properties of the SCM. Recall from §3.5 that a CSM under certain special conditions may generate infinitely many transitions on its partitioned state space within a finite time interval. For the results of this section, we must assume that the contrary condition is true, that is, no continuous trajectory on a finite time interval will generate infinite transitions. This assumption is justified by the fact that no real system can behave in this way either by design, or due to practical limitations such as finite precision of calculations and finite machine cycle times.

4.7.1 Case I Switching

Proposition 4.7.1 (Finite Events (I)) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model, with switching time interval of Δt, a finite number of system switches, $p > 0$, and an upper bound on switching at each choice point exists, $|\Gamma(s_i)| \leq r$, for all $s_i \in \mathcal{F}$. If for all $x_i \in \mathcal{R}$, the number of transitions generated n_i for each solution is finite, then the reachable state space $\mathcal{R}$ will generate a finite number of events due to discrete state transitions.*

Proof. Lemma 4.6.1 established that the cardinality of $\mathcal{R}$ is finite, with upper bound

$$\frac{r^{p+1} - 1}{r - 1}$$

Let n_i denote the number of transitions generated by trajectory $x_i \in \mathcal{R}$. The total number of transitions, N_e, generated by G in p time steps is

$$N_e = \sum_{i=1}^{\frac{r^{p+1}-1}{r-1}} n_i$$

Since the limit of this sum is finite and all n_i are finite, then $N_e$is finite. ■

4.7.2 Case II Switching

Proposition 4.7.2 (Finite Events (II)) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model, with a switching time interval of Δt (clock), a finite number of time switches, $p > 0$. If an upper bound on switching at each choice point exists, $|\Gamma(s_i)| \leq r$, for all $s_i \in \mathcal{F}$, if all switched continuous trajectories, ξ_i, are nonzeno, and the maximum number of partition switches in any clock interval Δt has a maximum, $|\xi_i| \leq q$, the reachable state space, $\mathcal{R}$, will generate a finite number of events due to discrete state transitions (i.e. crossing partition boundaries).*

Proof. By Lemma 4.6.2, the cardinality of the reachable state space is finite, with upper and lower bounds as indicated by Eq. 4.8. Since an event occurs at every partition crossing, or choice point, the number of events generated must also be finite. ■

4.8 Hybrid Transition Graph

Thus far, the properties of the SCM have been developed without the explicit intervention of a controller. In this section, we develop one possible discrete event representation of the SCM that introduces control input. This model, called a hybrid transition automaton or hybrid transition graph, will be used in the development of the discrete event supervisory controller synthesis technique that is the subject of the following chapter.

From here on, without loss of generality, Case II switching is assumed for all results involving SC models. Once again, in Fig. 4-7, the predicted execution of a SCM is illustrated. Since this is Case II switching, the alignment of the states with each other does not represent the time of occurrence of the choice point, but merely the ordering. As in the earlier figures representing the SCM execution, the choice

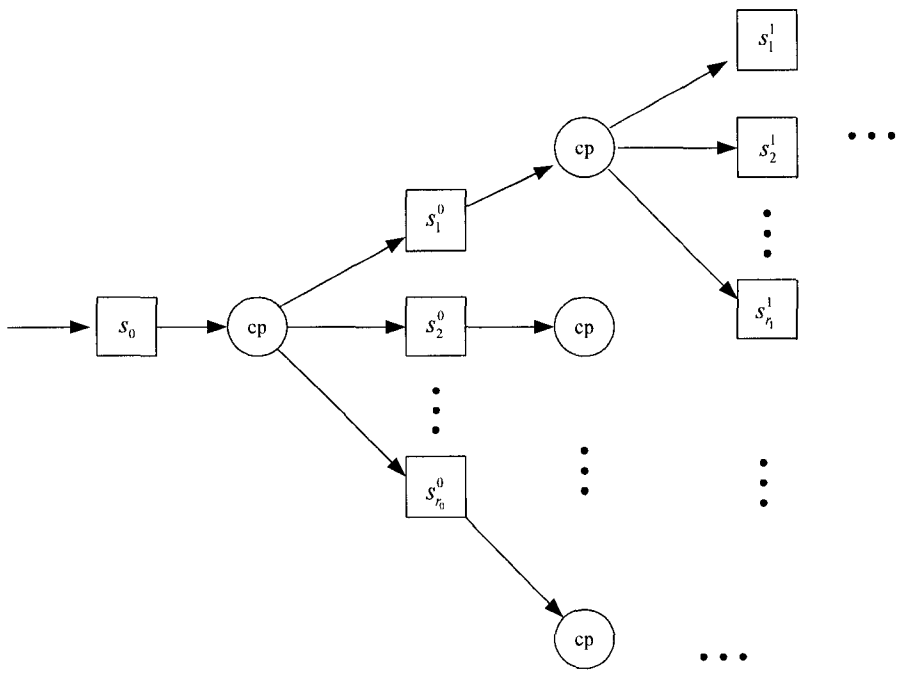


Figure 4-7: A predicted execution set of continuous system models.

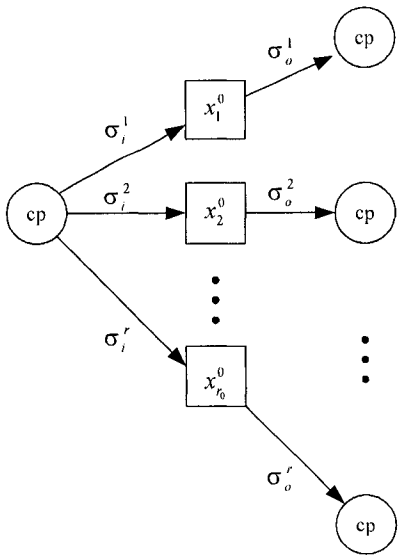


Figure 4-8: Prediction of continuous dynamics due to SCM execution.

points are indicated as circles, and the enabled systems $s_j^k \in \mathcal{F}$ are indicated as boxes. In this case, the system superscript k, indicates the predecessor system, and the subscript j, indicates the k^{th} element of $\Gamma(s_0)$. The subscript $1 \leq j \leq r_0$, where $r_0 = |\Gamma(s_0)|$. For purposes of exposition, we will not explicitly specify the type of lookahead horizon; it can be either time or events. Provided that the number of choice points is finite, and the number of branches at each choice point is finite, then the set of reachable continuous systems will also be finite.

In Fig. 4-8, the continuous system models have been replaced by their equivalent solutions, x_k^j, where superscript j, and the subscript k, are each derived from the matching system model, solved on the matching time interval. Thus, the set of all reachable continuous system solutions is $\mathcal{R}$. The output events $\sigma_{out}^i \in \Sigma_{out}$ occur as the result of each of the continuous solutions crossing some partition boundary, signaling that a change of discrete state has occurred, and thus initiating a new choice point. The input events $\sigma_{in}^i \in \Sigma_{in}$ of Fig. 4-8 (Def. 4.2.9) are representative of the connection of a discrete event supervisory controller to the system. At any choice

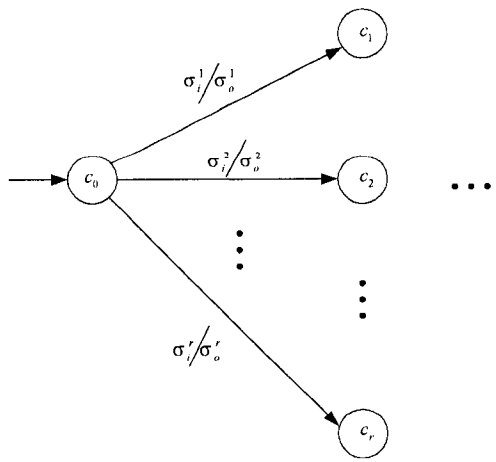


Figure 4-9: Hybrid Transition graph based on Fig. 4-7 and Fig. 4-8.

point there is a (finite) set of input event labels that may be used to select the desired continuous system dynamics that will be executed. After the appropriate continuous system dynamics have been evaluated, then an output event and a new choice point occur.

An alternative representation of the predicted behaviour of an SCM is a hybrid transition graph (HTG) (Fig. 4-9). The HTG brings together the discrete event input and output interface of the SCM in a directed graph that has continuous states for the nodes and discrete event transitions as edges. The HTG is the basis for the graph exploration algorithms (Chapter 5) upon which discrete event supervisory controller synthesis is based. We begin by defining the nodes of the graph.

Definition 4.8.1 (Timed Stamped Continuous State) *Let $x_a \in \mathbb{R}^n$ be a solution on a time interval $[t_0, t_1)$ to the IVP posed by a CSM $s_a \in \mathcal{F}$, $s_a = (f, \Psi, x_0)$. The timed-stamped continuous state evaluated at time $t' \in [t_0, t_1)$ is defined as*

$$c = (t', x_a(t')) \in \mathbb{R} \times \mathbb{R}^n$$

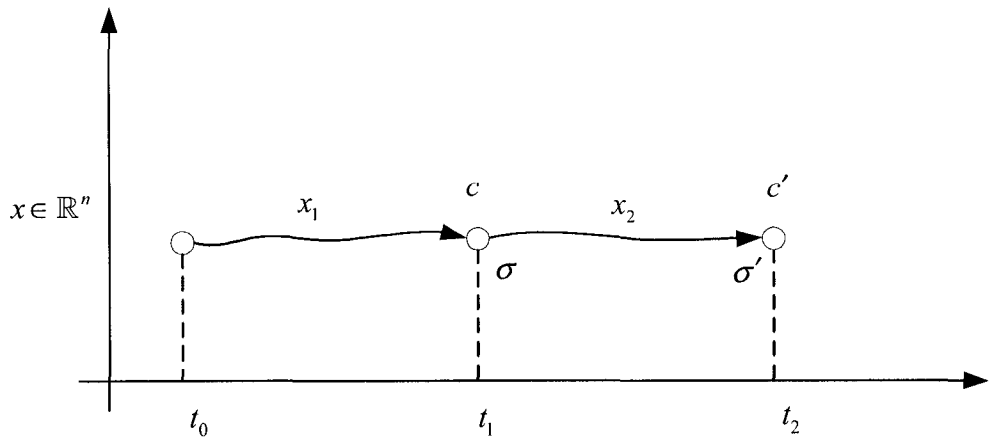


Figure 4-10: A pair of continuous trajectories give rise to an equivalent transition.

a point in the Cartesian product of time and the continuous solution domain. For future notation convenience,

$$\mathcal{C} \subset \mathbb{R} \times \mathbb{R}^n$$

In the HTG, nodes (timed stamped continuous states) are associated with the choice points of a SCM execution. Connecting the timed continuous states together are transitions. The continuous solutions can be discarded, since this information is unnecessary to discrete event processes. The solutions are replaced by a labeled transition with only the essential discrete event information remaining.

Definition 4.8.2 (Discrete Event Equivalent Transition) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a SCM and let $\xi = \{x_1, x_2\}$, a switched continuous trajectory. Let $x_1 \in \mathbb{R}^n$ be a solution to an IVP on time interval $t \in [t_0, t_1)$ and let $x_2 \in \mathbb{R}^n$ be the successor solution on time interval $t \in [t_1, t_2)$, that is, $x_2 = succ(x_1)$ (Fig. 4-10). Then the discrete event equivalent transition for the solution pair is defined as*

$$\tau = (c, \sigma, \sigma', c')$$

where $c = (t_1, x_1(t_1))$, $c' = (t_2, x_2(t_2)) \in \mathbb{R} \times \mathbb{R}^n$ are timed stamped continuous states,

the endpoints of the solutions x_1 and x_2 respectively, and $\sigma \in \Sigma_{in}$ and $\sigma' \in \Sigma_{out}$ are discrete events.

The input event $\sigma \in \Sigma_{in}$ is the control or input event for the transition. The output event $\sigma' \in \Sigma_{out}$ occurs as a result of the transition of the continuous solution into another region (crossing a hypersurface), or as a result of reaching the end of the designated simulation time interval, Δt, in which case the output event is *tick*. Thus, the input event can be seen as initiating the occurrence of the output event.

Definition 4.8.3 (Transition Set) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a SCM with $\mathcal{R}$ the set of all reachable continuous solutions for some finite lookahead horizon. The transition set is the set of all equivalent transitions $\tau \in \mathcal{T}_R$ (Def. 4.8.2, above) corresponding to all successor pairs of $x \in \mathcal{R}$*

$$\mathcal{T}_R = \{\tau : \tau \text{ is an equivalent transition for } \xi = \{x, x'\},\ x, x' \in \mathcal{R} \text{ and } x' = succ(x)\}$$

and furthermore, $\mathcal{T}_R \subset \mathcal{C} \times \Sigma_{in} \times \Sigma_{out} \times \mathcal{C}$ such that

$$\forall c \in \mathcal{C}, \sigma \in \Sigma_{in},\ \ |\{(\sigma', c') | (c, \sigma, \sigma', c') \in \mathcal{T}_R\}| \leq 1$$

The transition function and enabled events function may be defined in terms of the transition set.

Definition 4.8.4 (Transition Function) *The transition function $\delta_h : \mathcal{C} \times \Sigma_{in} \rightarrow \mathcal{C}$ is defined in terms of the transition set $\mathcal{T}_R$ for $c \in \mathcal{C}$ and $\sigma \in \Sigma_{in}$:*

$$\delta_h(c, \sigma) = \begin{cases} c' & \text{if } \exists \tau = (c, \sigma, *, c') \in \mathcal{T}_R \\ undefined & otherwise \end{cases}$$

where the symbol $*$ indicates a wildcard or "don't care" event in Σ_{out}. The domain of

δ_h may be extended to $\mathcal{C} \times \Sigma_{in}^*$ as follows

$$\begin{aligned} \delta_h(c, \epsilon) &= c \\ \delta_h(c, \sigma w) &= \delta_h(\delta_h(c, \sigma), w) \text{ where } w \in \Sigma_{in}^* \end{aligned}$$

Output events from the hybrid transition graph are the result of the output function.

Definition 4.8.5 (Output Function) *The output function $\omega : \mathcal{C} \times \Sigma_{in} \rightarrow \Sigma_{out}$ is defined as follows for $c \in \mathcal{C}$ and $\sigma \in \Sigma_{in}$*

$$\omega(c, \sigma) = \begin{cases} \sigma' & \text{if } \exists \tau = (c, \sigma, \sigma', *) \in \mathcal{T}_R \\ \text{undefined} & \text{otherwise} \end{cases}$$

where the symbol $*$ indicates a wildcard or "don't care" time stamped continuous state in $\mathcal{C}$.

Definition 4.8.6 (Enabled Events Function) *The enabled events function $\Gamma_h : \mathcal{C} \rightarrow 2^{\Sigma_{out}}$ and is defined as*

$$\Gamma_h(c) = \{\sigma' \in \Sigma_{out} : \exists \sigma \in \Sigma_{in},\ \sigma' = \omega(c, \sigma) \text{ is defined}\}$$

In Fig. 4-11, the execution of Fig. 4-6 has been replaced by its equivalent hybrid transition structure. The transition structure forms a tree-like directed graph representing the predicted discrete event behaviour of a SCM.

Definition 4.8.7 (Hybrid Transition Graph) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a SCM with $\mathcal{S}_R$ the set of all reachable continuous system models and $\mathcal{R}$ the reachable continuous solutions. A hybrid transition graph is a tuple*

$$H = (\mathcal{C}, \Sigma, \delta_h, \Gamma_h, \omega, c_0)$$

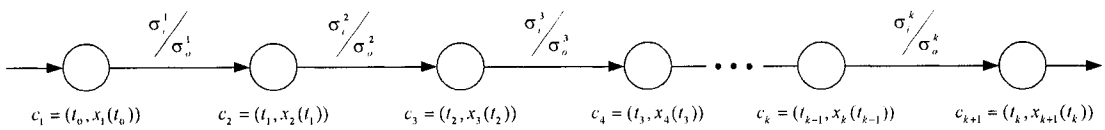


Figure 4-11: Hybrid transition equivalent to the execution $v = \{s_1, s_2, s_3 \ldots, s_k, \ldots\}$ and its corresponding switched continuous trajectory $\xi = \{x_1, x_2, x_3, \ldots, x_k, \ldots\}$ of Figure 4-6.

where:

$\mathcal{C}$ is a set of time stamped states, or graph nodes, $\mathcal{C} \subset \mathbb{R} \times \mathbb{R}^n$,

Σ is a set of input and output event labels, $\Sigma = \Sigma_{in} \cup \Sigma_{out}$,

Γ_h is the enabled events function,

δ_h is the state transition function,

ω is the output function,

c_0 is the initial time-stamped state of the graph.

So for a SCM G, there exists a hybrid transition graph H, that is based upon the predicted behaviour of the SCM on a particular lookahead horizon (either in time or events).

The input/output event pairs of the HTG are similar to a Mealy implementation of a finite state automaton. Note the similarity between Γ_h of the HTG automaton and Γ of the SCM. In the former, the output events are explicitly a function of system state and time $c \in \mathcal{C} \subset \mathbb{R} \times \mathbb{R}^n$. In the latter, the enabled systems function Γ, the enabled systems are a function of the currently executing system $s \in \mathcal{F}$, evaluated at the choice point. Clearly, c is derived from s by evaluating the continuous solution at the choice point.

An algorithmic implementation of δ_h and Γ_h will be presented later in Chapter 6.

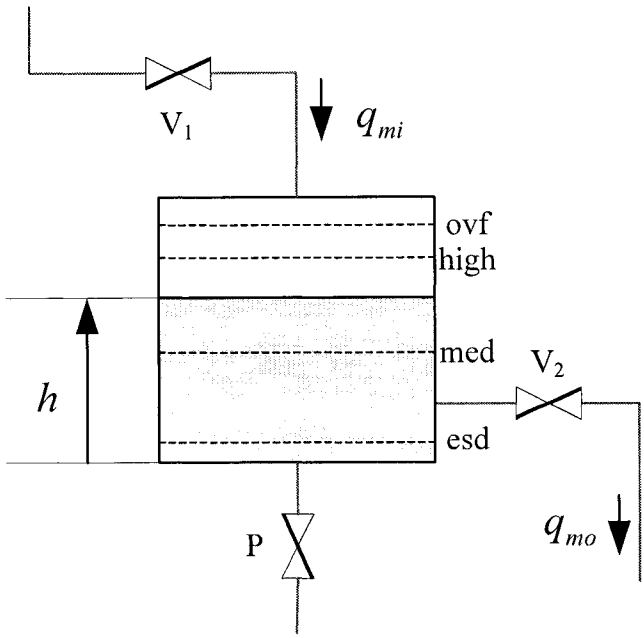


Figure 4-12: Schematic for tank system model.

4.9 SCM Example

An example will be presented to illustrate the properties of the SCM and its application to a modeling problem.

For this example, the modeled system is a tank of fluid (Fig. 4-12). It is desired to control the level of this tank through the opening and closing of valves. While this is a trivial example, it is a useful system to study since it has discrete dynamics (valves opening and closing) and nonlinear continuous dynamics.

The controls available for the tank are valves V_1, the fill valve; V_2, the drain valve; and P, the purge valve. The purge valve is a "use once" emergency shutdown control that is invoked by the system in the event of emergency. Table 4.1 lists the combinations of valve positions and associates these actuator combinations with the input event set $\Sigma_{in} = \{i_1, i_2, i_3, i_4, sd\}$. For example, the input event i_2 is associated with the actuator control vector $u_c = [0, 1, 0]^T$ which corresponds to valve positions

Table 4.1: Valve control structure

	Control, u_c		
σ_{in}	V_1	V_2	P
i_1	0^a	0	0
i_2	0	1	0
i_3	1	0	0
i_4	1	1	0
sd	0	0	1

[a]Valve open = 1, closed = 0

$[V_1, V_2, P]$ = [closed, open, closed]. The *shutdown* operation is initiated by the input event sd, which opens only the purge valve to drain the tank. The completion of this operation is indicated by the esd output event.

The continuous dynamics for the tank can be described by a nonlinear differential equation. Assume that opening V_1 causes a constant mass flow into the tank q_{mi}, while opening of V_2 or P causes turbulent flow from the tank. then the general expression for the tank dynamics (Palm III 2000) is

$$\dot{h} = \begin{bmatrix} q_{mi} & -\frac{R_{t1}^{-1}\sqrt{\rho g h}}{\rho A} & -\frac{R_{t2}^{-1}\sqrt{\rho g h}}{\rho A} \end{bmatrix} u_c \tag{4.9}$$

where h is the liquid level, ρ is the density of the liquid, and A is the cross sectional area of the tank. Turbulent resistances for valves V_2 and P are R_{t1} and R_{t2} respectively. The five different actuator control vectors u_c, with Eq. 4.9 yield five distinct dynamical models for the tank. Each of these actuator settings along with a set of state partitioning functionals, forms a separate CSM which will be embedded in the switched continuous model. For this example, the CSMs corresponding to input events i_1 to i_4 share the same set of functionals $\Psi_1 = \{F_1, F_2, F_3, F_4\}$ and the CSM for emergency shutdown operation (purge valve P open) has $\Psi_2 = \{F_5\}$. The functionals are defined in Table 4.2 along with the associated output events.

If the SCM is executed or predicted (Case II switching) on a 90 second time interval, the set of predicted continuous trajectories S_R reachable from initial condition

Table 4.2: Output events, with associated functionals and hypersurface crossing directions.

σ_{out}	Functional	Zero-crossing	Alarm
ovf	$F_1(h) = h - 33$	$\uparrow$	over fill
hi	$F_2(h) = h - 31$	$\uparrow$	high
med	$F_3(h) = h - 18$	$\downarrow$	medium
unf	$F_4(h) = h - 15$	$\downarrow$	under fill
esd	$F_5(h) = h - 0.5$	$\downarrow$	emergency shutdown

$x_0 = 26$ are pictured in Fig. 4-13. The figure shows the branching of the trajectories on detection of events. Note that since the *sd* input event and corresponding dynamics do not share the same partitioning functionals as the other dynamics, there is no event detected (and consequently no branching).

The hybrid transition graph matching the predicted SCM behaviour is pictured in Fig. 4-14. Based on the graph, the plant language generated for the lookahead horizon of 90 seconds is

$$L = \{med\ unf\ tick,\ med\ tick,\ hi\ tick,\ tick\}$$

Note the apparent nondeterminism of this HTG model when only the output events are considered. However, the model is deterministic, since each transition at any node is guarded by a unique input event (control action). This is an unconstrained plant model, so at each choice point there is a choice of $|\Gamma(s)| = 5$ control actions. We will see in the next chapter that when this plant model is combined with a specification to form a controller, the branching will be constrained due to the requirement of the models to synchronize on output events.

4.10 Conclusions

The SCM is a flexible hybrid model based on discrete switching between various embedded continuous system models. Switching of dynamics and hence input/output discrete event synchronization, are designed to occur at state boundaries and/or on

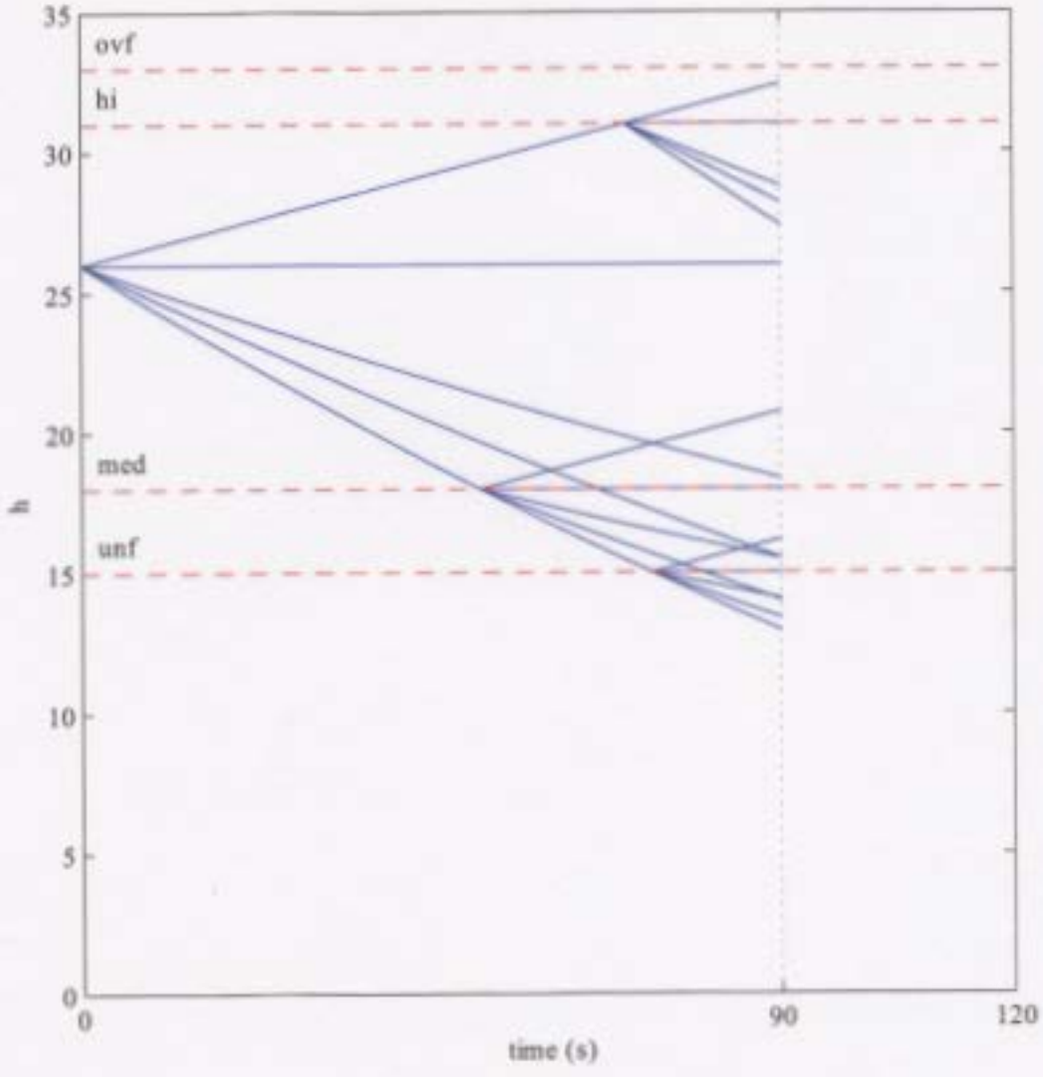


Figure 4-13: The predicted state trajectories of the tank for intial condition $x_0 = 26$ and prediction time $t = [0, 90)$, with Case II switching.

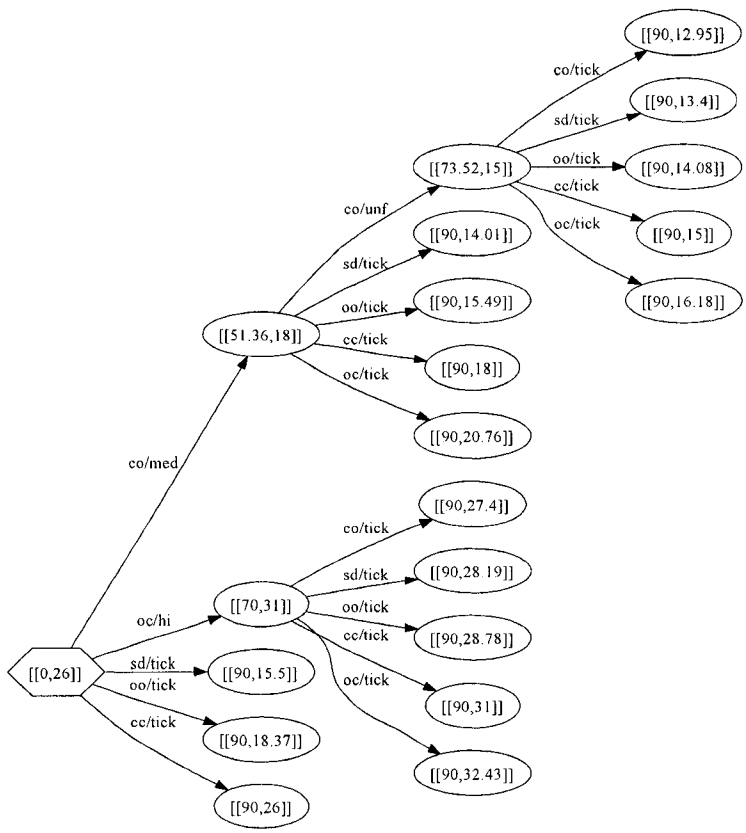


Figure 4-14: Hybrid transition graph equivalent for continuous reachable state space of Fig. 4-13

a synchronous or time-based schedule. Synchronization occurs in both coarse (`tick`) and dense time. This is analogous to the operation of industrial control systems in which a synchronous control cycle is augmented by interrupt-driven control. The point at which the system switches dynamics is known as a choice point. A finite set of enabled systems is available for the system to go forward in time, and one system is selected to execute. Predicting the future execution of an SCM $G = (\mathcal{F}, \Gamma, s_0)$ consists of considering all such enabled systems, extending simulations (solutions) for each, creating new choice points, and again extending the simulations. Reachability of the SCM can be expressed in the set of reachable continuous state space solutions $\mathcal{R}$. The hybrid transition graph captures the discrete-event behaviour of the SCM on this reachable set. Providing that the SCM has nonzeno behaviour, then the set, and the HTG will be finite, and therefore computable.

In the next section, we will examine the process of synthesizing DES supervisory controllers based on the HTG, which is in essence a discrete event model of the SCM.

Chapter 5

Control of Hybrid Systems

The switched continuous model is a class of hybrid system model. The hybrid dynamics embodied by this model are characterized by instantaneous switching between the various embedded continuous dynamical models within the SCM. In order to control such a process, the controller must act either in the discrete event or the continuous domain. A DES supervisor has been selected as the most appropriate tool, since the control objective is to coordinate and sequence the actions of low-level continuous and discrete event systems that themselves may be controlled or uncontrolled.

The switching framework of the SCM model has been designed so that it mimics the intervention of a discrete event supervisory controller, while the output framework is designed to communicate, and thus synchronize, with external discrete event processes. For purposes of modeling, analysis, and synthesis, it is desirable that the plant modeled as a SCM may be treated as any other discrete event process.

More complex models may be constructed by forming synchronous products of switched continuous and finite state models (FSM). The resulting synchronous product behaviour of the product model is similar to the product connection of finite state models (FSM). The main difference is that the state space of the SCM is infinite on an infinite horizon, so that the product connection of FSMs with a SCM has

an infinite state space. If the product behaviour is developed on a limited lookahead horizon, then the infinite state space becomes finite, due to the truncation of all future trajectories of the SCM.

This chapter develops the foundations of DES controller synthesis for plants modeled by the SCM.

5.1 Discrete Event Controller Synthesis

In the DES supervisory control theory developed by Ramadge and Wonham (Ramadge and Wonham 1987),(Ramadge and Wonham 1989), a DES supervisory controller is synthesized by forming the product of finite state models of plant P and specification S. The optimal supervisory controller C is the closed-loop controller that permits the largest set of joint behaviour in the concurrent connection of P and S, denoted $P \parallel S$. This controller is known as a *maximally permissive* controller. In languages of automata, the legal language K (Fig. 5-1) is the joint behaviour of the plant and the specification automata

$$K = L(P) \cap L(S)$$

The notation $L(P)$ is used for the language of automaton P and it may also be denoted as L_P. The controller, once connected to the plant, enforces the largest subset of the legal language, also known as the *supremal controllable sublanguage* (Wonham and Ramadge 1987). Based on the assumption that not all plant behaviour is controllable, control is exercised by disabling only the *controllable* events in the plant. The optimal DES supervisor is the controller that enforces legal behaviour with the least amount of plant intervention; i.e. minimal disablement. In this linguistic paradigm, the controller can be defined as a function from the plant language to the power set of Σ (the plant event set):

$$C : L(P) \rightarrow 2^{\Sigma}$$

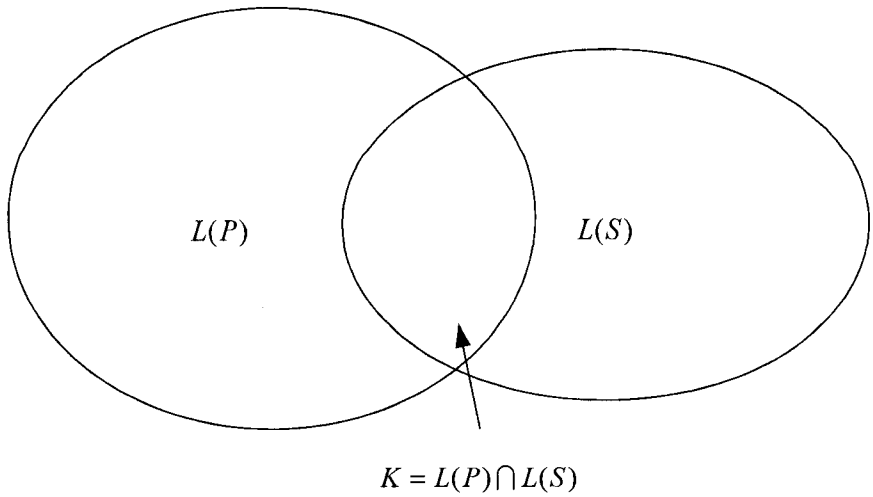


Figure 5-1: Intersection of plant and specification languages.

Fig. 5-2 illustrates this closed-loop control connection of the plant and controller. For a more detailed treatment of DES control synthesis, the reader is referred to Appendix D.

Controller synthesis may also be approached from a state exploration and avoidance point of view. For example, let P be a finite state automaton model of a plant in which one or more states may be deemed to be illegal. Removal of all the illegal states from the automaton (or its graph representation) then constitutes the admissible (with respect to some specification) transition structure of the plant. A supervisory controller is then based on this trimmed transition structure. This approach is well-suited to algorithmic implementation, since typically, the constituent models that are used for controller synthesis are represented explicitly as automata. As a result, the treatment of control synthesis in this chapter will focus on graph-based search techniques. Therefore, a typical task is to construct products of synchronously communicating automata; known also as the synchronous product or parallel composition operation, it is denoted by the $\|$ symbol.

Ignoring the issue of event controllability for now, the controller for a plant can

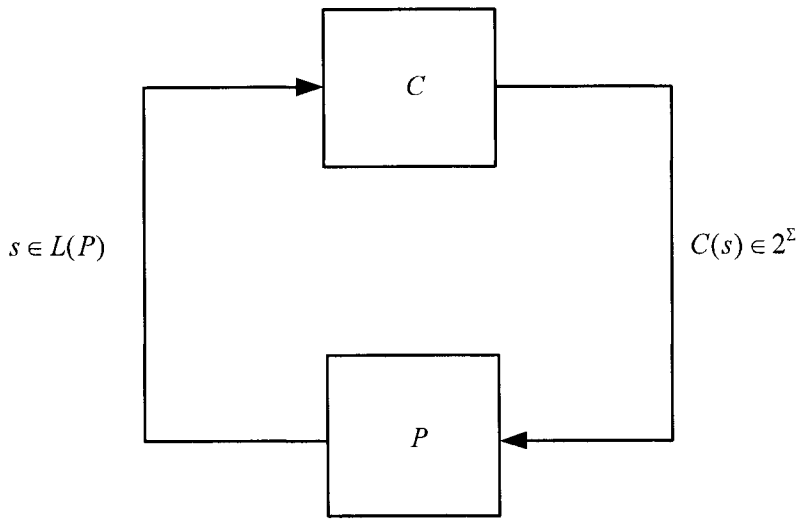


Figure 5-2: Closed loop connection of DES supervisor C to a plant P.

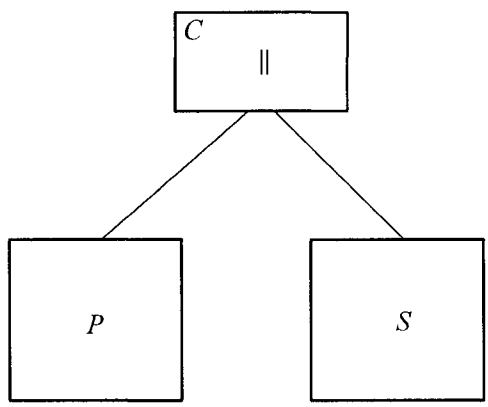


Figure 5-3: Controller synthesis as the concurrent (parallel) connection of two processes.

be constructed from the synchronous product of an automaton model P and the automaton model of the specification S, $C = P \parallel S$, illustrated in Fig. 5-3. Before defining the product of automaton models, the formal definition of a deterministic finite state automaton is as follows:

Definition 5.1.1 (Deterministic Finite State Automaton) *A deterministic finite state automaton is a tuple:*

$$G = (Q, \Sigma, \Delta, q_0) \tag{5.1}$$

where:

Q is a finite set of states,

Σ is a finite set of events,

Δ is a finite set of transitions (Def. 5.1.2 below),

q_0 is the initial state.

Definition 5.1.2 (Transition Set) *The transition set is a finite set of transitions* $\Delta \subseteq Q \times \Sigma \times Q$ *such that*

$$\forall q \in Q, \forall \sigma \in \Sigma, \ \ |\{q' | (q, \sigma, q') \in \Delta\}| \leq 1$$

The transition function will be used extensively for graph exploration, and so will be defined in more detail.

Definition 5.1.3 (Transition Function) *A transition function* $\delta : Q \times \Sigma \rightarrow Q$ *may be a partial function on its domain. For* $q \in Q$ *and* $\sigma \in \Sigma$,

$$\delta(q, \sigma) = \begin{cases} q' & \text{if } \exists (q' \in Q) \text{ and } \exists \text{ a transition } (q, \sigma, q') \in \Delta \\ \text{undefined} & \text{otherwise} \end{cases} \tag{5.2}$$

The domain of δ may also be extended to $Q \times \Sigma^*$. For example,

$$\begin{aligned} \delta(q, \epsilon) &= q \\ \delta(q, \sigma w) &= \delta(\delta(q, \sigma), w) \end{aligned}$$

For clarity, the function δ has domain $Q \times \Sigma$ and range Q while Eq. 5.2 defines the element to element map. Forthwith, the same convention will be utilized in function definitions.

A second function, known as the enabled events function, is also useful for graph exploration.

Definition 5.1.4 (Enabled Events Function) *The enabled events function $\Gamma : Q \rightarrow 2^{\Sigma}$ is defined as follows*

$$\Gamma(q) := \{\sigma \in \Sigma : \delta(q, \sigma) \text{ is defined}\}$$

While inclusion of both δ and Γ in the definition of a FSM is redundant, in the sense that they are already defined in terms of the transition set Δ, they are useful when defining synchronization of multiple automata. In future notation, and for ease of exposition, finite state automata will be defined in an expanded form as follows:

$$G = (Q, \Sigma, \delta, \Gamma, q_0)$$

Henceforth, the transition set Δ, will not be explicitly included in the definition of the FSA, since it can be derived from δ.

We define the reach operation, which returns the subautomaton that represents the portion of an automaton that is reachable from its initial state q_0.

Definition 5.1.5 (Reach Operation) *Let $G = (Q, \Sigma, \delta, \Gamma, q_0)$ be a finite state au-*

tomaton, then the **reach** *operation is defined as follows*

$$\begin{aligned} reach(G) &: = (Q_r, \Sigma, \delta_r, \Gamma, q_0) \text{ where} \\ Q_r &= \{q \in Q : \exists s \in \Sigma^*,\ \delta(q_0, s) = q\} \\ \delta_r &= \delta \text{ on a restricted domain such that } Q_r \times \Sigma \rightarrow Q_r \end{aligned}$$

Conceptually, the controller of 5-3, C, is a model-based controller, with models of the plant and the specification embedded within it. Thus, C is the synchronous product of finite state automata (Def. 5.1.1). Throughout this document, the term "product" will be synonymous with "synchronous product". Formally, the definition of the synchronous product of two finite state automata, is as follows:

Definition 5.1.6 (Product Automaton) *Let* $G_1 = (Q_1, \Sigma_1, \delta_1, \Gamma_1, q_{01})$ *and* $G_2 = (Q_2, \Sigma_2, \delta_2, \Gamma_2, q_{02})$ *be finite state automata (see Appendix C for some examples). The (synchronous) product automaton* $G_1 \parallel G_2$ *is defined as*

$$G_{1\|2} = reach(Q_{1\|2}, \Sigma_{1\|2}, \delta_{1\|2}, \Gamma_{1\|2}, (q_{01}, q_{02})) \tag{5.3}$$

where $Q_{1\|2} = Q_1 \times Q_2$ *is the Cartesian product of the state sets,* $\Sigma_{1\|2} = \Sigma_1 \cup \Sigma_2$ *is the union of the event sets, and the product transition function is defined in the following definition.*

The *reach* operation in Eq. 5.3 is useful since some parts of the product automaton may not be reachable from the initial product state (q_{01}, q_{02}), and these portions are not of interest.

Definition 5.1.7 (Product Transition Function) *Let* G_1 *and* G_2 *be finite state automata (Def. 5.1.6), then their product transition function*

$$\delta_{1\|2} : (Q_1 \times Q_2) \times (\Sigma_1 \cup \Sigma_2) \rightarrow (Q_1 \times Q_2)$$

$$\delta_{1\|2}((q_1,q_2),\sigma) = \begin{cases} (\delta_1(q_1,\sigma),\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_1(q_1) \cap \Gamma_2(q_2) \\ (\delta_1(q_1,\sigma),q_2) & \text{if } \sigma \in \Gamma_1(q_1) \backslash \Sigma_2 \\ (q_1,\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_2(q_2) \backslash \Sigma_1 \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Note that $\delta_{1\|2}$ like δ_1 and δ_2, is a partial function.

Definition 5.1.8 *The product enabled events function*

$$\Gamma_{1\|2} : (Q_1 \times Q_2) \rightarrow 2^{(\Sigma_1 \cap \Sigma_2)}$$

$$\Gamma_{1\|2}(q_1,q_2) = [\Gamma_1(q_1) \cap \Gamma_2(q_2)] \cup [\Gamma_1(q_1) \cap (\Sigma_1 \setminus \Sigma_2)] \cup [\Gamma_2(q_2) \cap (\Sigma_2 \setminus \Sigma_1)] \quad (5.4)$$

The first term of Eq. 5.4 requires that events common to both enabled events functions Γ_1 and Γ_2 (and hence common to the event sets of both automata) must be synchronized. The latter two terms are the sets of enabled events that are private to each of the automata; there is no requirement for these to be synchronized, so they will always be included. The synchronous product operation of automata is associative (Cassandras and Lafortune 1999). The associative property permits the product automaton operation of Def. 5.1.6 to be extended easily to an n-ary product, in which n automata are synchronized. Associativity also permits the hierarchical nesting of synchronous product operations. This modularity lends itself naturally, as we shall see later, to an object oriented programming implementation. Hierarchical modeling also allows for a flexible modeling scheme in which automata may be grouped by their functional relevance to each other; e.g. the sub-automata that form a specification and the sub-automata that form a plant. Fig. 5-4 is an example of a hierarchical model of the discrete event behaviour of a ship's power and propulsion systems. In this example, each of the various subsystem models are grouped as functionally related components. For example, the power system behaviour is modeled as a product of the two product automata, Generator #1 and Generator #2. A "flat" equivalent to

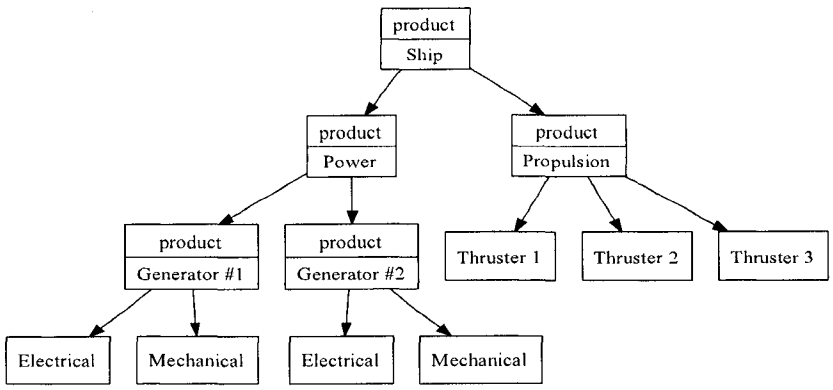


Figure 5-4: Hypothetical hierarchical automaton model for a ship's propulsion and power generation subsystems..

this hierarchical model may be constructed by taking the lowermost (non-product) components of the tree and placing them in a single-level product automaton.

Synthesis of a DES controller in the SCM environment requires the construction of synchronous product structures of SCMs and finite state models. The discrete event interface that was developed for the SCM in the previous chapter allows these product structures to be constructed. This is the basis of DES supervisory controller synthesis for hybrid systems.

5.2 SCM and Control Synthesis

Suppose that we have a SCM P, that models the dynamics of a plant and a timed FSM, S that models a specification. In our framework, controller synthesis consists of the synchronous product of the plant and specification models. In §4.8, the hybrid transition graph was introduced as a representation of the discrete event dynamics for a SCM. In addition, on a finite event or time horizon, the HTG is of finite size.

Now we define a new transition structure which encapsulates the synchronous product of an HTG and a FSM where the state of this structure is the product state

of the respective states of the HTG and the FSM. The product of a hybrid transition graph and a finite state automaton is defined as follows.

Definition 5.2.1 (Hybrid Product Automaton) *Let* $H = (\mathcal{C}, \Sigma_h, \delta_h, \Gamma_h, \omega, c_0)$ *be a hybrid transition graph and* $G = (Q, \Sigma_g, \delta_g, \Gamma_g, q_0)$ *be a finite state automaton, the product automaton is defined as*

$$H \parallel G = reach(\mathcal{C} \times Q, \Sigma_h \cup \Sigma_g, \delta_{h\|g}, \Gamma_{h\|g}, \omega, (c_0, q_0))$$

with $\delta_{h\|g}$ *and* $\Gamma_{h\|g}$ *as defined below.*

Note that the transition set $\mathcal{T}_R$ can be omitted from the HTG definition, since δ_h and Γ_h can provide the same information set. States of a hybrid product automaton (HPA) are truly hybrid by the usual definitions of a hybrid system, since the state has both a continuous and discrete state component. Note also that the product states of the HPA $H \parallel G$ inherit the time stamps of the HTG. The inclusion of time within the state of the product automaton also ensures that the resulting product graph will be acyclic.

The objective of the modeling framework is to achieve a finite state representation. By making the assumption that the event set of G, $\Sigma_g \subseteq \Sigma_{out}$, then the set of events $\Gamma_g(q) \backslash \Sigma_{out} = \emptyset$. This means that events are generated by the SCM only, and the finite state machine G acts as an acceptor. For control synthesis, H is the plant model and G is either a model of the specification, or part of the plant model, so this is a reasonable assumption. Since H generates the output events via the output event function, only the shaded set(s) illustrated in Fig. 5-5 are necessary to consider for the transition function.

Definition 5.2.2 (HPA Transition Function) *Let* $\Sigma_h = \Sigma_{in} \cup \Sigma_{out}$, *then*

$$\delta_{h\|g} : (\mathcal{C} \times Q) \times \Sigma_{in} \rightarrow (\mathcal{C} \times Q)$$

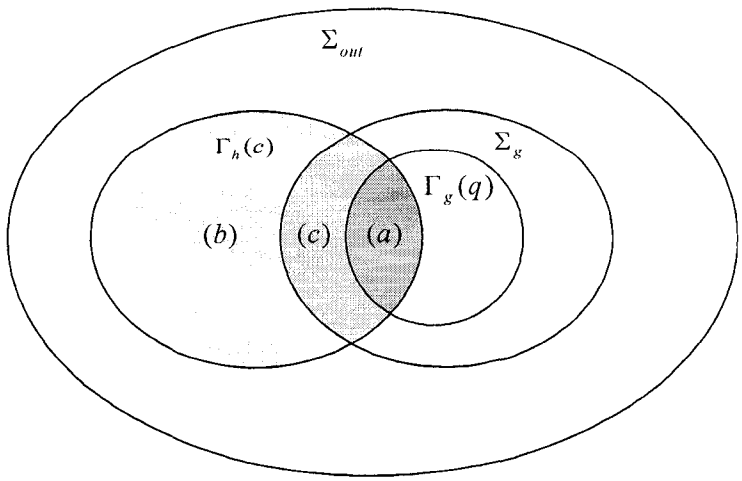


Figure 5-5: Set definitions for the HPA transition function.

is the product transition function, a partial function. Let $\sigma \in \Sigma_{in}$, $(c,q) \in (\mathcal{C} \times Q)$ *and* $\eta = \omega(c,\sigma) \in \Sigma_{out}$, $\Sigma_g \subseteq \Sigma_{out}$, *then* $\delta_{h\|g}$ *is defined as*

$$\delta_{h\|g}((c,q),\sigma) = \begin{cases} (\delta_h(c,\sigma), \delta(q,\eta)) & \text{if } \eta \in \Gamma_g(q) \quad (a) \\ (\delta_h(c,\sigma), q) & \text{if } \eta \notin \Sigma_g \quad (b) \\ \textit{undefined} & \textit{otherwise} \quad (c) \end{cases}$$

The domain of the transition function $\delta_{h\|g}$ *may be extended to* $\mathcal{C} \times \Sigma_{in}^*$ *as follows*

$$\begin{aligned} \delta_{h\|g}((c,q),\epsilon) &= (c,q) \\ \delta_{h\|g}((c,q),\sigma w) &= \delta_{h\|g}(\delta_{h\|g}((c,q),\sigma),w) \text{ where } w \in \Sigma_{in}^* \end{aligned}$$

Now the definition of the enabled events function,

Definition 5.2.3 (HPA Enabled Events Function) *Let the enabled events function*

$$\Gamma_{h\|g} : (\mathcal{C} \times Q) \rightarrow 2^{(\Sigma_{out})}$$

If $c \in \mathcal{C}$ *and* $q \in Q$, *then*

$$\Gamma_{h\|g}(c,q) = [\Gamma_h(c) \backslash \Sigma_g] \cup [\Gamma_h(c) \cap \Gamma_g(q)]$$

From these definitions, it is apparent that the HTG automaton synchronizes only its output events with the events of the finite state automaton. The intent here is to mimic the standard synchronization technique that is used in DES supervisory synthesis. The focus is placed on the plant's discrete event behaviour which is communicated by the output events. Specifications are normally written in terms of the desired (output) dynamics, thus the synchronization of output events is a practical modeling decision.

Input events are not synchronized; these represent the actions that are available to the controller. Later, we will see that a *control choice mechanism* selects one $\sigma \in \Sigma_{in}$ as the control action. The result of the selection of a particular input event leads to the generation of an output event, when the underlying continuous system dynamics transition to a new discrete state. So in effect, synchronization of output events causes certain input events to be ineligible implicitly. An improvement to the SCM framework would be to include explicit input synchronization. For example, SC models could have a finite state "front-end", permitting the set of enabled input events (viable control actions) to be a function of the state of the front-end automaton.

Finally, we will look at the synchronization of SCMs with each other. It may be desirable to approach the modeling of the continuous dynamics of a system in a modular fashion. If the continuous dynamics are separated into multiple SCMs, they may be synchronized at the discrete event level. Since dense time information is available from each of the HTG processes, it is possible to detect the "earliest" event that occurs amongst them; this becomes a new state in the product by evaluating the solutions of each of the other systems at this event time.

An alternative approach to synchronization of SCMs at the discrete event level, is to lump all continuous state variables into a single set of continuous models, and

embed these into a single SCM. The choice is left up to the designer; if the continuous dynamics have significant coupling, then communicating state variables are best lumped together. If the continuous dynamics are coupled through indirect discrete event communication, then they may be modeled more effectively as separate SCMs. An algorithm that implements multiple SCM object synchronization will be presented in chapter 6.

5.2.1 Example: Product of SCM and FSM

To illustrate the synchronous product operation, we will revisit the SCM tank modeling example of §4.9. The HTG that results from this example models the discrete event behaviour of the uncontrolled tank for a 90 second lookahead horizon (recall Fig.4-14, p. 86). Let the finite state model of the specification

$$\begin{aligned} S &= (Q, \Sigma_s, \delta, \Gamma, q_0) \\ \Sigma_s &= \{hi,\ tick,\ unf\}. \end{aligned}$$

The plant graph is presented again in Fig.5-6, along with the specification The plant model "inherits" its output event set from the switched continuous model. In Table 4.2 (p. 84), recall that the SCM output events are

$$\Sigma_{out} = \{ovf,\ hi,\ med,\ unf,\ esd\} \cup \{tick\}$$

and the input event set is

$$\Sigma_{in} = \{oo,\ oc,\ co,\ cc,\ sd\}$$

The resulting product $P \parallel S$, illustrates how the original plant graph is modified by the product connection of the specification (Fig. 5-6, bottom). Starting with the initial state of the plant, $c_0 = [0, 26] \in (\mathbb{R} \times \mathbb{R})$, $\Gamma_P(c_0) = \{tick,\ hi,\ med\}$, all transitions

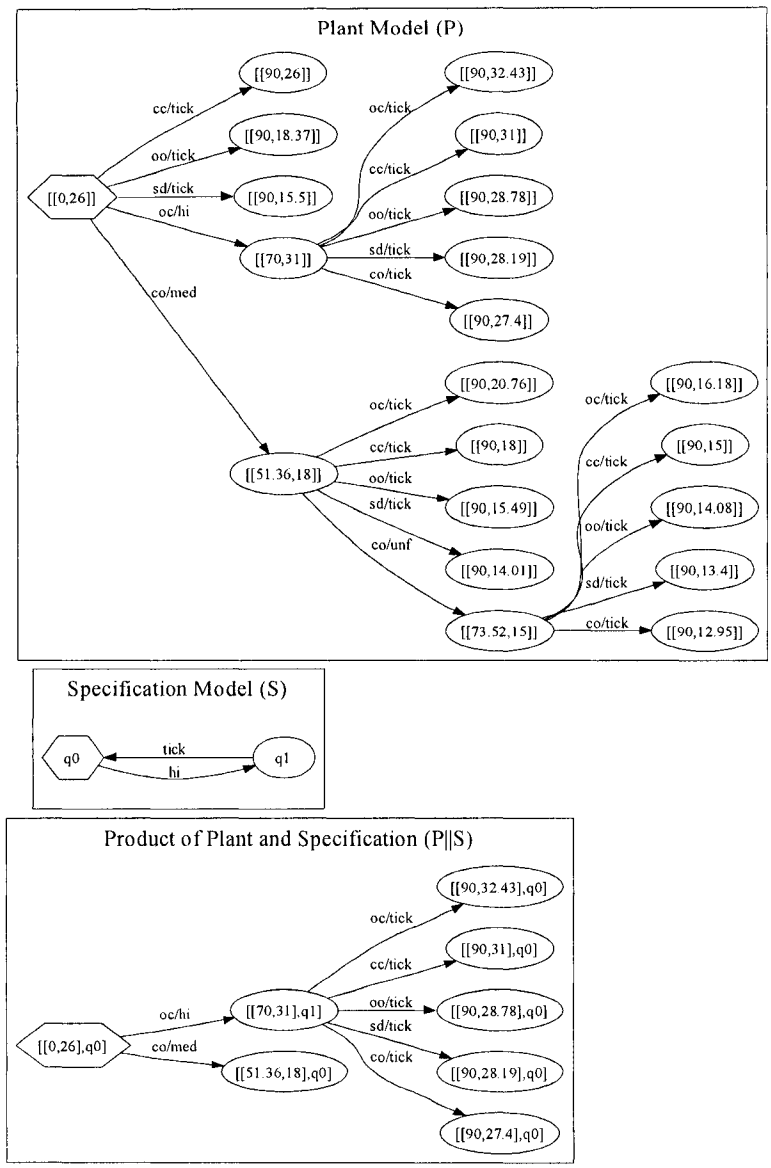


Figure 5-6: Product of plant modeled by a SCM and specification modeled as a FSM.

with *tick* output events are removed from the graph because ($tick \notin \Gamma_S(q_0)$). The transition having output event *hi* remains because it synchronizes with the specification. The transition with output event *med* remains in the product because ($med \notin \Sigma_s$). The process of trimming continues until each remaining (and reachable) state in the product graph has been visited.

Starting from the initial state of the product automaton (the root of the tree), there exist 6 unique branches. Of these six branches, one ends in a product state $c = [[51.36, 18], q_0] \in ((\mathbb{R} \times \mathbb{R}) \times Q)$. The time of the continuous product state is $t = 51.36$, which is less than the lookahead horizon. Since this branch does not take the system to the simulation horizon, it is not a viable choice for a controller to make, since the system cannot safely continue on this trajectory to the horizon. This concept will be elaborated upon in the next section. The viable controller actions for this plant, at this time, are the set of input strings

$$L(C) = \{oc\ oc,\ oc\ cc,\ oc\ oo,\ oc\ sd,\ oc\ co\}$$

5.3 Blocking

In standard DES supervisory control theory, the concept of controller blocking is defined in the context that certain states have a special status; i.e. marked states. In the DES framework, if an automaton reaches a state that is not marked, and $\Gamma = \emptyset$, it is said to be deadlocked or blocked. Supervisory controllers are designed to be both safe and nonblocking.

In a system model based on a SCM, the concept of blocking is defined differently from the typical DES definition. In the SCM framework there is a richer set of information, particularly the fact that dense-time state (and event) information is available. Having the knowledge of the time at which the system has entered a state (the time stamp of the HTG states) allows us to define blocking in terms of the terminal state time. The system has the extra dimension of time to define the progress

(or lack thereof) of the system, in addition to state information. Alternatively, the number of events in a trajectory, and whether that trajectory reaches the lookahead horizon may also be a determination of the blocking.

A plant modeled by a SCM, on a finite horizon, synchronized with a specification modeled as an FSM, forms the basis for a simulation. This simulation captures the control interaction of a discrete event supervisor with a real system. Transitions that will violate the safety of the system and carry the plant to an illegal state are prevented from occurring, via event disablement. Recall the definitions of $\mathcal{R}$ (Def. 4.6.2 and 4.6.3), the set of reachable continuous trajectories of a SCM. For some lookahead horizon, the set $\mathcal{R}$ collectively represents a simulations (or prediction) of the uncontrolled future plant behaviour of the system up to some future time or event horizon. Refining the definition of the reachable state space:

Definition 5.3.1 (Continuous Reachable State Space (Events)) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model. The state space reachable from $x_0 \in s_0$ in exactly p events is denoted as $\mathcal{R}^p$.*

Definition 5.3.2 (Continuous Reachable State Space (Time)) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model. The state space reachable from $x_0 \in s_0$ in the time interval $T = (t_0, t_f]$ is denoted as $\mathcal{R}^T$.*

When a controller or other agent disables events in the discrete event behaviour of an SCM, it results in truncated, or incomplete, switched continuous trajectories $\xi \in \mathcal{R}$, that do not reach the lookahead horizon. This is the continuous behaviour of the SCM as described by the synchronous product of the HTG and a FSM. We define the switched continuous trajectory (SCT) as follows

Definition 5.3.3 (Incomplete SCT (Events)) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model, and let $\mathcal{R}^p$ denote the continuous state space reachable in p events. An SC trajectory $\xi \in \mathcal{R}^p$ is incomplete if $|\xi| < p$.*

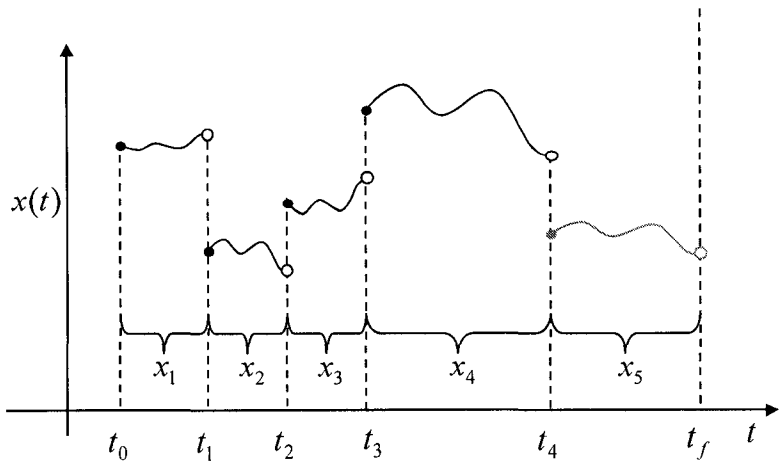


Figure 5-7: Incomplete trajectory of example 5-7.

Definition 5.3.4 (Incomplete SCT (Time)) *Let $G = (\mathcal{F}, \Gamma, s_0)$ be a switched continuous model and let $\mathcal{R}^T$ be the continuous state space reachable in the time interval $T = [t_0, t_f)$. An SC trajectory $\xi = \{x_1, x_2, \ldots x_\alpha\} \in \mathcal{R}^T$ is incomplete if x_α is a continuous solution of for a time interval $[t_{\alpha-1}, t_\alpha)$, such that $t_\alpha < t_f$.*

Example 5.3.1 (Incomplete SCT (Time)) *Let $G_p = (\mathcal{F}, \Gamma, s_0)$ be an SC model of a plant and let the switched continuous trajectory*

$$\xi_p = \{x_1, x_2, x_3, x_4, x_5\} \in \mathcal{R}^T$$

Suppose that the continuous trajectory x_5 terminates in an illegal state, then the controller must prevent x_5 from occurring, and the modified trajectory becomes $\xi = \{x_1, x_2, x_3, x_4\}$ (Fig. 5-7). In this case, the trajectory is incomplete because x_4 is a solution on the time interval $[t_3, t_4)$ and $t_4 < t_f$.

Returning to the discrete event behaviour now, this truncation of switched continuous trajectories leads to a HTG representation with some "stub" (truncated) branches in the graph. These branches in the HTG are considered to be blocking,

since it is not possible to find a path (trajectory) from the initial state to the lookahead horizon. In the following definitions, for clarity, we consider the only the legal behaviour of HTGs, assuming that synchronization with an external process has already enforced legal behaviour (a specification for example).

Definition 5.3.5 (Blocking & Nonblocking States (Time Horizon)) *Let H be a hybrid transition graph based on a time $T = [t_0, t_f)$ reach of the SCM G. Let H have initial state $c_0 = (t_0, x(t_0)) \in \mathbb{R} \times \mathbb{R}^n$ and let $c' = (t', x(t')) \in \mathbb{R} \times \mathbb{R}^n$ be a time stamped continuous state in the graph H such that $\Gamma_h(c') = \emptyset$. If there exists a string $u \in \Sigma_{in}^*$ such that $\delta_h(c_0, u) = c'$, and if $t' < t_f$, then state c' is blocking. Conversely, if $t' = t_f$, then state c' is nonblocking.*

Definition 5.3.6 (Blocking & Nonblocking States (Event Horizon)) *Let H be a hybrid transition graph based on a reach of p events of the SCM G. Let H have initial time stamped state $c_0 = (t_0, x(t_0)) \in \mathbb{R} \times \mathbb{R}^n$ and let $c' = (t', x(t')) \in \mathbb{R} \times \mathbb{R}^n$ be a state such that $\Gamma_h(c') = \emptyset$. If there exists a string $u \in \Sigma_{in}^*$ such that $\delta_h(c_0, u) = c'$ and $|u| < p$, then c' is a blocking state. Conversely, if $|u| = p$, then state c' is nonblocking.*

To illustrate blocking, we start with the HTG H of Fig. 5-8, a plant model of a system. For purposes of exposition, the graph is the unrestricted (uncontrolled) SCM behaviour. We will assume that there exists a FSA S, as an acceptor, that models the specification. It supplies the state marking by labeling the continuous states as illegal if the discrete-event language of the plant H, falls outside of the legal behaviour specified by S. Therefore, all continuous trajectories of this system are complete, either in time or events; that is, states c_1 through c_8 are at the lookahead horizon. In the continuous state of the SCM, any path traversing the graph from the initial state c_0 to one of these end states corresponds to a complete switched continuous trajectory. Illegal states are indicated as grey-coloured nodes, c_3, c_4, c_5, and c_9 (marked by the specification acceptor automaton). To enforce safety, these nodes must be removed from the graph, with the result indicated in Fig. 5-9.

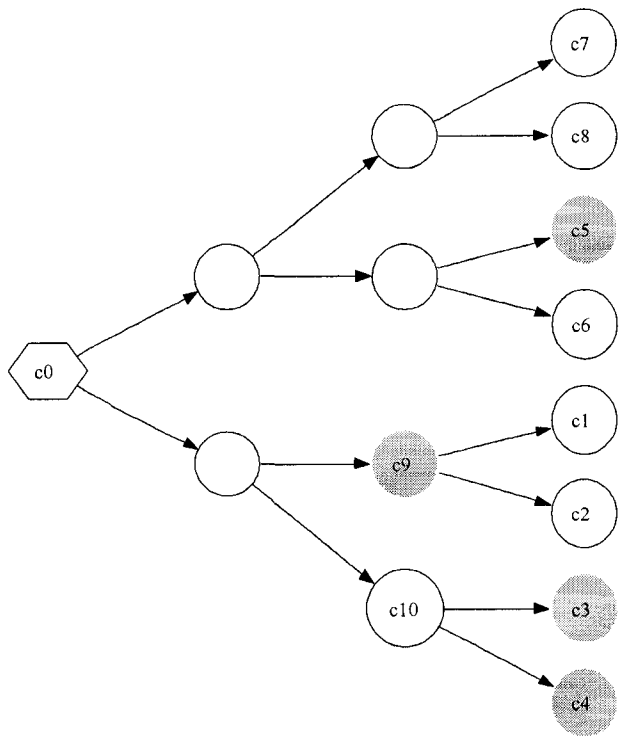


Figure 5-8: Hybrid transition stucture with illegal states identified in grey.

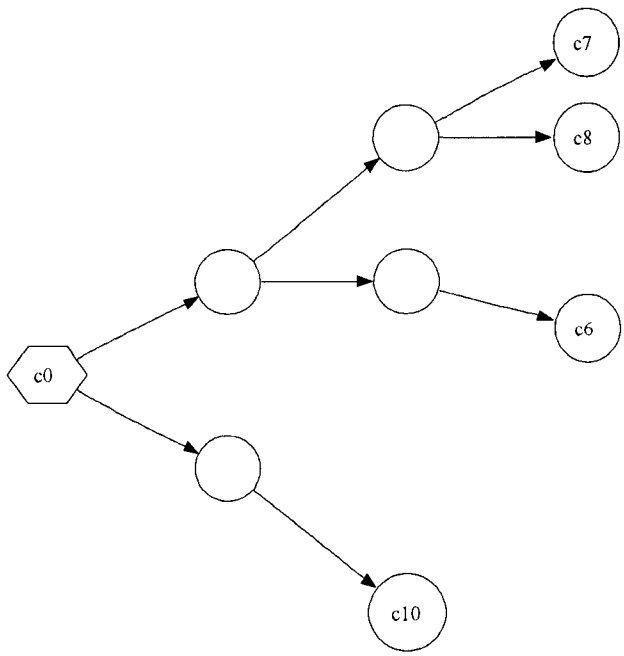


Figure 5-9: Hybrid transition stucture of Fig. 5-8 with illegal states and related transitions deleted.

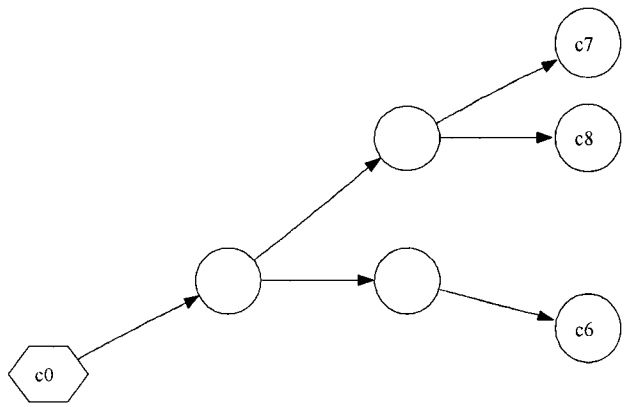


Figure 5-10: Non-blocking and legal HTG for example of Fig. 5-8.

In the case of states c_1 and c_2, they cannot be reached without traversing an illegal state, c_9 first. Removing c_9 and the transition that enters it, makes c_1 and c_2 unreachable, and therefore they are not viable. States c_3 and c_4, although on the lookahead horizon, are illegal. Removing these states leaves a stub branch in the HTG at state c_{10}, which is a blocking state. The blocking trajectory that ends with c_{10} must be removed from the graph. By examination, all states and transitions of this branch will have to be pruned back to the initial state in order to avoid blocking. The result is illustrated in Fig. 5-10, which retains three possible legal and nonblocking trajectories, taking the system from c_0 to c_6, c_7 and c_8.

5.4 Fail-safe Controller Operation

As we have seen in the previous sections, a product of a HTG and one or more finite state models is the basis for the DES controller. Appropriate trimming of states and transitions from the HTG yields a controller graph that represents the safe and nonblocking controller actions, given the limited horizon of knowledge that is available. In previous limited lookahead work (Chung et al. 1992) the set of safe actions or trajectories, are known as *pending* traces. This set of legal control actions is further refined by taking either an *optimistic* or *conservative* policy with respect to the expected behaviour of the system beyond the current lookahead horizon. With an optimistic policy (or outlook), all pending traces are assumed to have continuations beyond the lookahead horizon that are both legal and marked. In the case of the conservative policy however, all trajectories are assumed to continue uncontrollably into illegal or blocking conditions. These attitudes condition how the set of pending traces is further refined. The farther the lookahead horizon is extended, the less ambiguity there is about the pending traces.

An online controller algorithm must have a means of selecting the next control action. The ultimate objective of this controller is to drive the system from the initial

state to the lookahead horizon without violating the safety.

Definition 5.4.1 (Nonblocking Controller) *Let H be the HTG of a system modeled by SCM G having only legal states, and an initial time stamped state $c_0 = (t_0, x(t_0)) \in \mathbb{R} \times \mathbb{R}^n$. The system has a nonblocking safe controller if there exists at least one nonblocking state c' and there exists a control string $u \in \Sigma_{in}^*$, such that $c' = \delta_h(c_0, u)$.*

Again, we have examined H in isolation. The assumption is that SCM G is synchronized with an external process, ensuring only legal states exist.

But what happens if the system arrives at a state from which only blocking trajectories exist? In the absence of a legal control choice, the system must continue (since time cannot be stopped), and since only illegal choices remain, the controller will be forced to proceed with a control action that ultimately violates the system safety. Hence, nonblocking is equivalent to safety. Unfortunately, there can never be a guarantee that just beyond the lookahead horizon, the controller might block, and a safety violation will be forced. We would like to design a controller for this online discrete event environment which can be guaranteed to be free of this sort of forced safety violations.

5.4.1 Emergency Shutdown

A standard design practice in safety critical industrial control is to incorporate an emergency shutdown (ESD) mechanism into the control system. It is generally considered good design practice to include some sort of fail-safe subsystem in controlled systems at design time. Examples of industries that utilize such fail-safe mechanisms as part of the control infrastructure include:

- Oil and Gas Processing
- Nuclear Power Generation

- Chemical Manufacturing
- Transportation (Rail Transit)
- Motion Control

Terms for fail-safe control procedures vary by industry, but examples are emergency shut down (ESD), from Oil and Gas processing applications, emergency stop (E-Stop), from motion control applications and the SCRAM procedure for nuclear reactors. In order to model the emergency shutdown behaviour, it is necessary to define what is meant in a discrete-event sense as an ESD.

Definition 5.4.2 (Emergency Shutdown State Set) *Let* $H = (\mathcal{C}, \Sigma, \delta_h, \Gamma_h, \omega, c_0)$ *be a HTG of SCM* $G = (\mathcal{F}, \Gamma, s_0)$. *The emergency shutdown state set* $E \subseteq \mathcal{C}$ *such that*

$$E = \{c : \Gamma(c) = \{tick\} \text{ and } \delta(c, \sigma) \in E, \forall \sigma \in \Sigma_{in}\}$$

An element $c \in E$ *is an emergency shutdown state.*

Obviously, an ESD state should be considered by a DES controller as a safe state; the ESD states can be identified (or marked) by synchronization of additional FSM plant model when designing the controller. In the context of the continuous behaviour of the switched continuous model, an ESD state might be abstractly interpreted as a Lyapunov-stable equilibrium (see Appendix A) within one or more of the CSMs. If a continuous trajectory of the system can enter such a Lyapunov stable region, it will be "captured", remaining within a defined region. If the region has been partitioned appropriately, any trajectory entering the region will fail to generate events (other than *tick*), since no partition boundaries are crossed. Thus, an emergency shutdown region is defined in terms of (stable) dynamics and an appropriate partitioning boundary.

While it is undesirable to enter an ESD state from the point of view that it performs no useful work, it is preferable to a potentially catastrophic safety violation.

Therefore, the ESD state is a way of gracefully handling a potentially disastrous controller block.

Definition 5.4.3 *A failsafe controller is one that can be guaranteed to always operate safely (without violating specifications) and is nonblocking.*

Proposition 5.4.1 (Failsafe Controller Existence) *Let H be the HTG of a system modeled by SCM G, and with initial time stamped state $c_0 = (t_0, x(t_0)) \in \mathbb{R} \times \mathbb{R}^n$. A fail-safe controller exists for this system if there exists at least one ESD state $c' = (t', x(t'))$ in HTG H and if there exists a control string $u \in \Sigma_{in}^*$, such that $c' = \delta_h(c_0, u)$.*

Proof (Event Lookahead). Controller existence hinges on there being a control string $y \in \Sigma_{in}^*$ such that y causes the system to reach a nonblocking state. By definition, $\Gamma(c') = \{tick\}$, which implies that the control string u can be extended by an event $\sigma' \in \Sigma_{in}$

$$c'' = \delta(c', \sigma') = \delta_h(\delta_h(c_0, u), \sigma')$$

and by definition, c'' is also an emergency shutdown state. Similarly, an arbitrarily long sequence of input events may always be chained together starting at the initial ESD state c', to form input control string w because all successive states of c' are also ESD states by definition. In the case of a finite p event lookahead horizon, w can be finitely extended so that $|u| + |w| = p$. Thus the state

$$c_{nb} = \delta(c', w) = \delta(c_0, uw)$$

is nonblocking because it is at the event horizon, and is reachable from the initial state c_0 by a control string $y = uw$. ■

Proof (Time Lookahead). This proof is for the case that graph H is due to a finite time lookahead $T = [t_0, t_f)$, where the time stamp of the ESD state is $t_0 < t' \leq t_f$. As in the previous proof, an arbitrarily long control string w may be

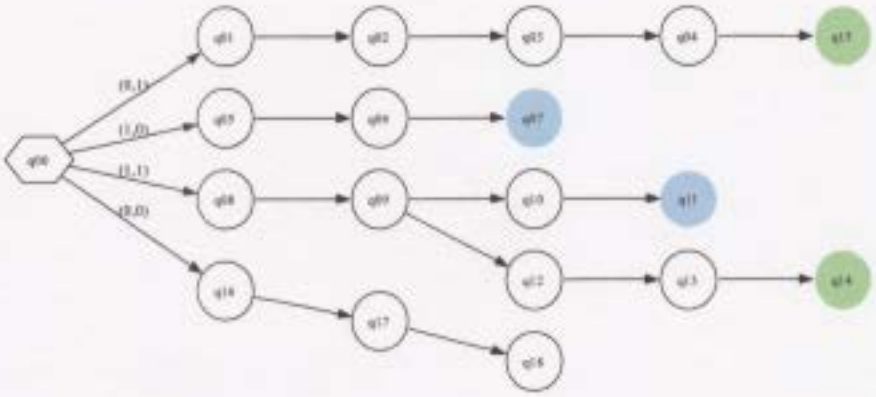


Figure 5-11: Control choice with emergency shutdown states.

extended from c'

$$c_{nb} = (t'', x(t'')) = \delta(c', w) = \delta(c_0, uw)$$

For c_{nb} to be nonblocking, $t'' \geq t_f$. Let $t'' = t' + k\Delta t$,where $\Delta t > 0$ is the control sample time associated with *tick*, and $k \geq 1$ is an integer. Since for each ESD state $\Gamma(c') = \{tick\}$, then the length of the control string must be $|w| \geq k$. Assuming nonzeno execution, w can always be finitely extended so that there are k *tick* events in w, and ensuring that c_{nb} is nonblocking. As before, $c_{nb} = \delta(c_0, uw)$ is reachable from the initial state c_0. ■

The requirement that the controller must include ESD states is similar to the conservative lookahead policy of (Chung et al. 1992), in the sense that the outlook assumes that just beyond the lookahead horizon, the system will block (or be unsafe), and thus the system must be prepared to shut down. Having an ESD state within reach (i.e. within the lookahead horizon), guarantees that the system can at least shut down without violating safety. The fail-safe controller existence hinges on *ESD reachability*.

Let Fig. 5-11 represent the graph of legal control actions, with the present (initial) controller state denoted by a hexagonal-shaped node. ESD states are indicated by

Table 5.1: Choice of control action

Subgraph Status		
ESD Reachable	Nonblocking	Transition
F	F	exclude
F	T	exclude
T	F	admit[b]
T	T	admit[a]

[a]First priority, [b]second priority.

the blue coloured nodes, and nonblocking nodes (those on the lookahead horizon) are coloured green. By Def. 5.4.2, we know that any emergency shutdown states ($q07$ and $q11$) may be extended arbitrarily to the horizon; these extensions are omitted for clarity. Table 5.1 summarizes the control actions based on the subgraph reachability of each immediate control choice. Clearly, subgraphs that are not emergency shutdown reachable will be excluded form the controller, so branches labeled $(0,0)$ and $(0,1)$ are removed from consideration. In any case, branch $(0,0)$ would have been deleted, since state q_{18} is blocking. Of the remaining two branches ($(1,0)$ and $(1,1)$), the branch that is ESD reachable and nonblocking receives priority for control selection over branches that lead only to ESD; this matter will be dealt with in the following section since it pertains to selection of a control action.

5.5 Controller Propagation

A hybrid transition graph is a representation of the timed discrete event behaviour of a switched continuous model at a particular time and state. Furthermore, it models this behaviour for a particular prediction horizon. No attempt has been made to produce a closed-form FSM that represents the discrete event behaviour of the modeled system for all time. Thus, the controller HTG should be considered to be a temporary data structure that will be used to choose the control action at a particular point in time and space. After the information contained in a particular controller HTG is no longer useful, a new one must be created from the basic SCM information – the "source"

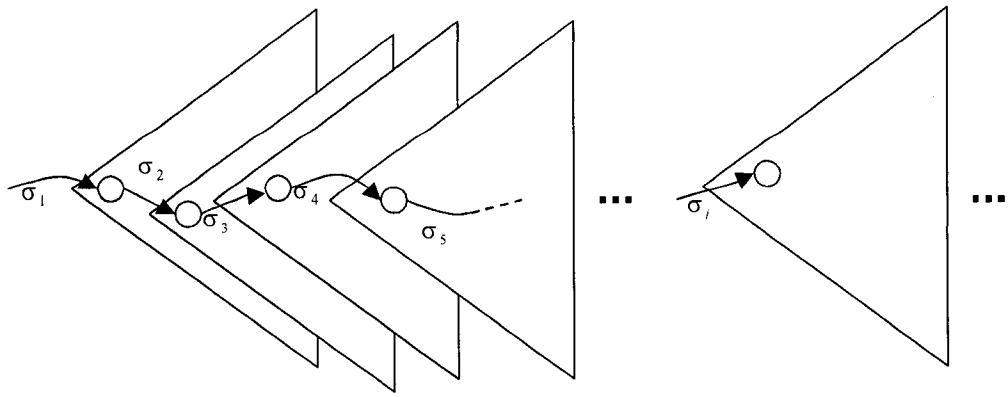


Figure 5-12: Propagation of controller graph through time and space.

model. The new model is now relevant for the current state and time of the system. This process continues, extending the controller forward in time. If at each control update step, there is a suitable control choice, then the control system effectively guides the system along, constructing the safe trajectory one event (control decision) at a time, based on a tree of predicted future behaviour. This concept is illustrated in Fig. 5-12, in which each controller graph is represented as a "pie-shaped" wedge.

The controller as described so far, does not implement control in itself; it actually models the future safe discrete event behaviour of the plant, including only the unambiguously safe, nonblocking and ESD reachable control trajectories. This implies that there is still a choice to be made, since there may be more than one safe and nonblocking trajectory available. To complete the full closed-loop control implementation (Fig. 5-13), some sort of choice mechanism must be added to the controller.

Let C be a controller connected to a hybrid plant P. At each time-stamped hybrid state $h = (c, q) \in (C \times Q)$, a controller graph is produced, a prediction of the controlled plant behaviour on a limited lookahead horizon. The graph is computed according to the requirements of nonblocking (legal) behaviour and ESD reachability. The changes

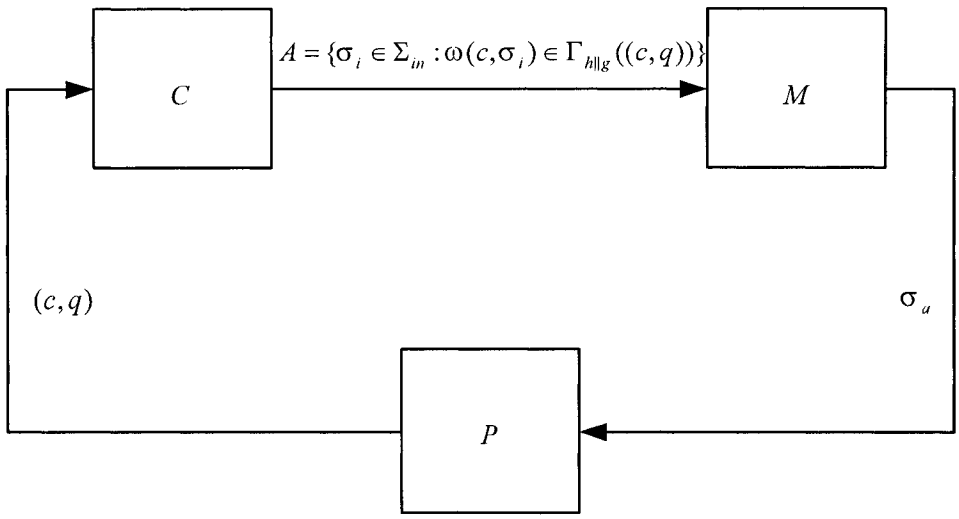


Figure 5-13: Closed loop connection of controller C and choice mechanism M with the hybrid plant P.

in the plant state are signalled by discrete events occurring in the plant and by the universal *tick* event; these are the choice points that have been encapsulated in the SC modeling framework. At any product state of the graph, there is a set of legal control choices that may be made, the actuator event set:

Definition 5.5.1 (Actuator Event Set) *Let C be a controller, a product of an HTG and a FSA, with hybrid state $(c,q) \in (\mathcal{C} \times Q)$, then the actuator event set is defined as*

$$A = \{\sigma_i \in \Sigma_{in} : \omega(c,\sigma_i) \in \Gamma_{h||g}((c,q))\}$$

Definition 5.5.2 (Choice Mechanism) *The choice mechanism "chooses" a single actuator event σ_a from set A. The choice mechanism is modeled as the function M*

$$M : A \rightarrow A$$

What possible mechanisms could be used to select actuator event σ_a from the set

of legal actuator events A? Choice mechanisms M, can be classified as either manual or automatic techniques. In the case of manual choice technique, a human operator picks an actuator event $\sigma_a \in A$ using some heuristic to select "the best" action. This is known as "human-in-the-loop" (HIL) control. Provided that $A \neq \emptyset$, the operator's actions are guaranteed always to be safe, since the controller presents only actuator events that lead to nonblocking and ESD-reachable states.

Although nonblocking and ESD states are essentially indistinguishable from each other, the choice mechanism must discriminate between ESD reachable and "pure" nonblocking choices, as in Table 5.1. Actuator events fall into two sets

$$A = A_{en} \cup A_e$$

where A_{en} is the set of control events that lead to ESD reachable and nonblocking trajectories, and the set A_e of events that lead only to an emergency shutdown. It is preferable to choose actions that have some chance of continuing without invoking an emergency shutdown.

Automatic control choice mechanisms, lacking human system knowledge on which to base a choice of σ_a may operate in variety of ways including (but are not limited to):

1. **Minimum switching**: minimize the number of times the actuator settings are changed.

2. **Optimal**: Attempt to minimize some sort of cost/performance functional not already captured in the specification. Such a scheme might involve assigning weighting factors to states in the plant model and assigning cost factors to input (control) events for example. The control action considered to be optimal by these rules would then be selected by M, which is essentially a dynamic programming problem.

3. **Stochastic**: a choice mechanism based on probabilities of outcomes, a Markov decision problem.

4. **Other**: Choose control action based on the number of states of the controller subgraphs. A greater number of states may improve the likelihood of a non-blocking continuation beyond the lookahead horizon.

As just indicated, there are many possible techniques to implement M, but they will not be covered in any further detail. The behaviour of M does not affect our analysis, since all previous sections have been based on the assumption that some event will be selected and the state of the plant will be advanced. Exactly how the event is selected is not central to further discussion.

5.5.1 Online Operation: Controller Update Cycle

In any real-time control process, there are a set of scheduled actions that must be repeatedly executed, usually at some regular time interval Δt called the *scan time.* For a control system, this set of actions will be called the *control kernel,* with the term kernel used in the context of software engineering and not its mathematical meaning. For the real-time process to proceed without failure, each of the scheduled actions within the kernel must complete within a predictable and finite time. Additionally, the sum of these times must not exceed a finite time interval Δt. A digital control system is such a real-time process, with one example being that of an industrial programmable logic controller (PLC).

Example 5.5.1 (Digital Control (motor speed)) *An example of a digital controller is a DC motor speed control system. The actions to be executed in the control kernel is as follows:*

1. *Measurement: sample motor speed*

2. *Compute control solution: Proportional, integral differential control solution*

3. Output control action: send control solution to amplifier

Example 5.5.2 (Digital Control with State Estimation) *An example of control and the comparison to optimal control with state estimation.*

1. Output control action: send control solution to actuator

2. Measurement: sample feedback signal

3. State estimate: predict future value of system state

4. Compute control solution: using state estimate and controller gain matrix

With an online control scheme, the control kernel is executed repeatedly as the system moves forward in (real) time. An outline of the scheduled actions within the control kernel is, after initialization,

1. choose the control action (σ_a),

2. synchronize the controller (state/event synchronization)

3. recompute controller

The controller is computed for the first time, starting at the initial product state of the plant and specification models. If the controller exists for the initial condition (initialization is successful), then the controller can be "enabled" or allowed to execute physically in connection with the plant. Referring to Fig. 5-13, the choice mechanism selects the event to apply to the plant from the subset of events that are enabled by the controller A. At this point, the plant dynamics of the selected continuous system model will begin the progression of the plant state towards the next choice point. Immediately, the synthesis of a controller must be initiated, starting from the next state which has been selected by the operator (the human in the loop) or by some other process. This process continues as long as the controller is online and enabled.

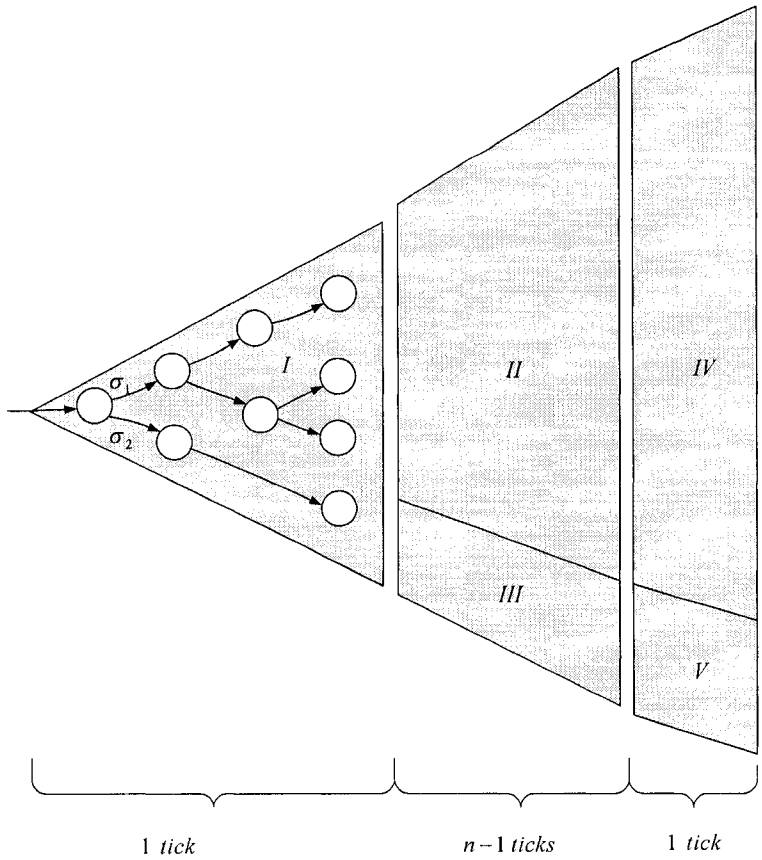


Figure 5-14: Controller propagation with time.

It is interesting to note the similarities of the online control kernel to the digital controller with state estimation in example 5.5.2. In particular, step 2 (controller synchronization) of the controller is equivalent to the measurement, and step 3 (controller computation) is equivalent to the state estimation or prediction step. Step 1 (actuator event choice) is roughly equivalent to steps 1 and 4 of the digital controller.

5.5.2 Horizon Extension

The fundamental action of the control kernel is the controller map propagation step. To ensure that the controller map continues to coincide as an approximate map of the plant's local dynamics, the lookahead must be extended or advanced with each time step. In Fig. 5-14, the diagram illustrates a controller graph starting at the initial state on the left, with time flowing from left to right. Suppose the graph is due to a p event lookahead and the controller graph consists of sections I, II and III (detail has been omitted from sections other than section I for clarity). The system will eventually execute one trajectory within this graph $u \in \Sigma_{in}^*, |u| = p$ in the next p events. The exact trajectory that is executed depends upon the sequence of input control events that will be selected. Controller sections IV and V together represent the breadth-first extension of the lookahead horizon by one *tick*. Indeed, to maintain an n-tick time lookahead horizon, as at initialization, it is necessary to extend the horizon of the controller map by one event after each input control event is executed.

Example 5.5.3 *In the simple example of Fig. 5-14, initially there are only two choices for controller action, σ_1 and σ_2. Assuming σ_2 is taken by the choice mechanism, only successor trajectories of σ_2 are valid, thus controller map sections II and IV can be discarded immediately. Conversely, if σ_1 were taken, sections III and V would be discarded.*

5.5.3 Example: Controller Propagation

As a concrete example of the preceding discussion, we will revisit the tank level control example of §4.9, looking at the propagation of the controller at one event intervals. Fig. 5-15 illustrates six consecutive control maps represented by the graphs in subfigures $(a) - (f)$. These graphs have been specially formatted to reveal the controller structure: the states are scaled down to dots and the event and state labels have been removed. The red dot in each graph indicates the initial state, c_0. The past

controller history is indicated by the light blue trace in each graph. The graph layout engine[1] that was used to make these plots uses a spring model layout technique, which tends to create a fan-shaped graph surrounding the initial condition. The growth of the ends of the fans by one event is evident from graph to graph, as the lookahead horizon is extended by one event each in each instance. Note also that there are a constant number of events from the initial condition to the branch ends on the horizon (11 events). Finally, note the large sections of the graph that are discarded as the controller state advances past certain control points. This graph was generated with a specification that eliminated most of the control (input) events, so that most nodes have only one possible control action. Typically, controller graphs are more complex than this example.

5.6 Summary

A hybrid modeling framework has been described based on a synchronous product of switched continuous model and finite state models. The product connection of a SCM and a FSM at the discrete event level, results in a hybrid system model. The expansion of the discrete event behaviour of these models on a limited horizon, produces a finite graph. If the product model consists of plant and specification models, then the limited horizon graph embodies the legal discrete event behaviour of the plant. Synthesizing an online DES supervisor based on this graph requires that incomplete (blocking) trajectories (those that do not carry the system from the initial state to the horizon), be removed from the control graph. Since this controller is intended to be implemented online, and no legal traces are available (controller is blocking), there will be a resulting safety violation. Thus it is not sufficient to ensure legal behaviour within the lookahead horizon. Potentially blocking (unavoidable safety violations) may always exist beyond the lookahead horizon.

[1] The *neato* layout engine is a module of the Graphviz suite of programs from AT&T (Gansner, Koustofios and North 2002)

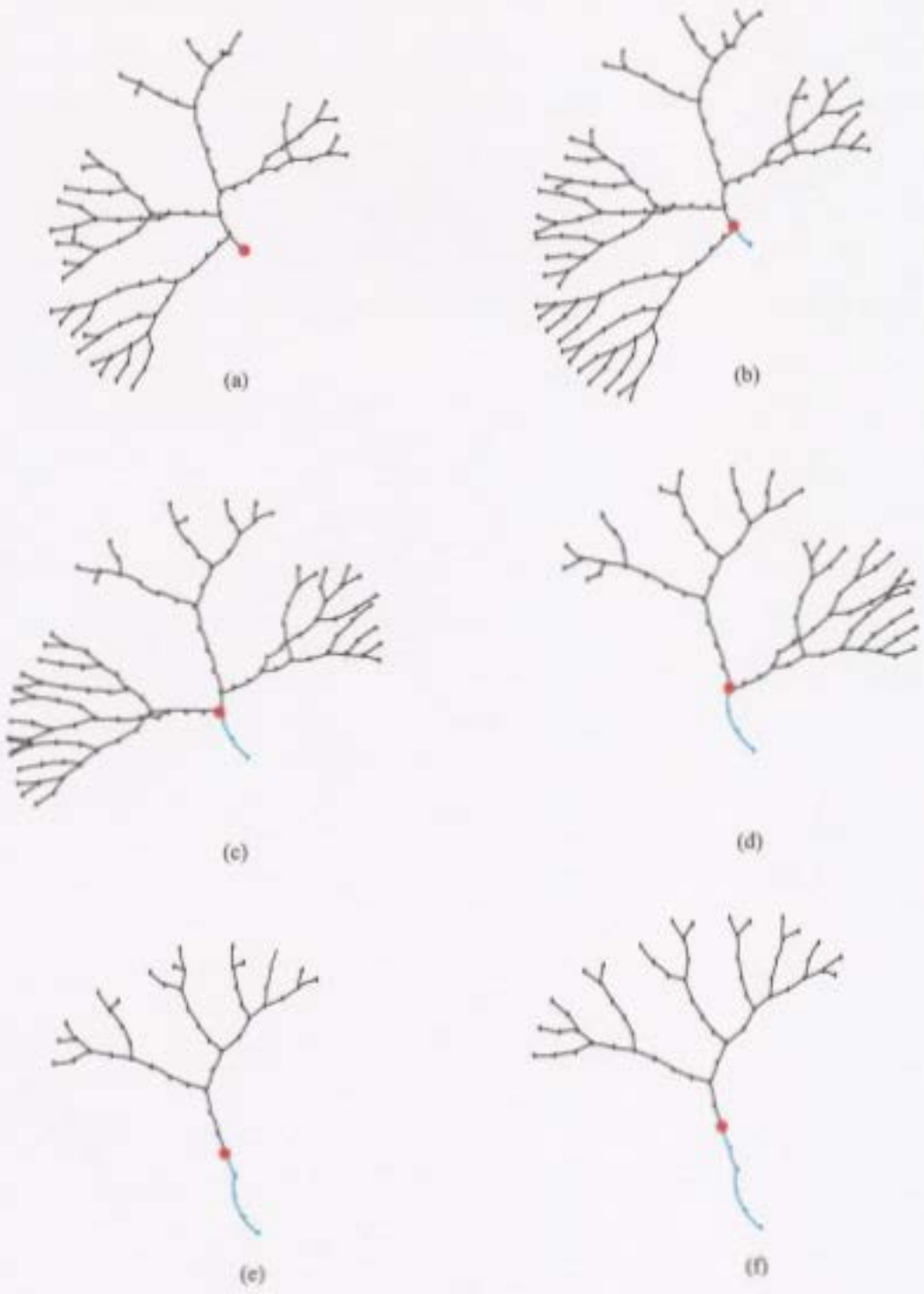


Figure 5-15: Controller graph propagation through six events.

By establishing a special state called an emergency shutdown state, it is possible to always guarantee the safety of our system. The so-called fail-safe controller always requires that valid trajectories be ESD state reachable. This conservative control policy assumes that beyond the horizon, there will be a safety violation. Therefore, even safe nonblocking trajectories will be eliminated because they cannot be guaranteed to be safe beyond the horizon, the cost of the safety guarantee. Within the described modeling framework, conditions for the existence of the fail-safe controller have been established.

The controller is, by necessity an online controller, because of the need to reduce complexity and to respond to time varying modeling conditions. Complexity reduction forces a limited lookahead horizon model; this model represents a "snapshot" of the system state space. The controller designed for this snapshot must be moved along in time and state as the controlled system (plant) progresses.

The next chapter will take the concepts of this and the previous chapters and develop the implementation-specific details for the fail-safe online controller.

Chapter 6

Computation: From Theory to Implementation

In the previous chapter, a framework was designed that enables an SCM model to be synchronized with multiple FSM models in a discrete event fashion. The resulting synchronized (product) model is a form of hybrid model since it has hybrid states. This framework also makes it possible to synthesize supervisory discrete event controllers for the modeled hybrid system. Control is exercised as a sequence of discrete event control actions, applied by the controller at specified system states in dense-time. These control actions are planned by predicting the behaviour of the modeled system on a limited lookahead horizon. Through the incorporation of a specially marked state known as an emergency shutdown state, the controlled system can be guaranteed to be safe (once the controller successfully initializes). The controller model assumes that illegal states are just beyond the lookahead horizon. This pessimistic outlook leads to controller conservatism: the controller requires that at least one emergency shutdown trajectory must always be present in the control structure.

This chapter details the general computational approach, the algorithmic implementation of the modeling concepts, and the specific implementation of a controller

synthesis and modeling software package.

6.1 Style

In the previous chapter, the theoretical considerations of the DES controller for a hybrid system were outlined. This section will examine the basic computational approach that has been adopted. There are three fundamental concepts that are necessary to the control implementation, these are: lazy computing, limited lookahead, and online implementation. These techniques will be leveraged to develop an efficient computational framework for modeling, controller synthesis, and control. Reduction of computational complexity is of utmost importance to our controller implementation, since it is to be implemented as an online controller. First, we will examine the concept of lazy computing,and its applicability to computing DES supervisors.

6.1.1 Lazy Computing Model

The term 'lazy" is used here in the sense of expending the least effort necessary to accomplish a job. It is a technique that helps to reduce computational complexity in space and time. Essentially, we are contrasting a hierarchical product model with that of its flattened equivalent model. This hierarchical approach to finite state modeling has been exploited before to avoid the state explosion problem that occurs with multiple product machines. In (Brave and Heymann 1991) a hierarchical statechart approach is taken for modeling while (Gaudin and Marchand 2005) uses the hierarchical approach to synthesize supervisors in systems without shared events. This hierarchical technique lends itself well to the state-based techniques that are used to construct the controller. In this section, a simple example in discrete event supervisory control synthesis using the lazy technique will serve to illustrate the lazy computing advantages over a "keen" computing approach. While the example is somewhat simplistic, it is a useful exercise to motivate the succeeding sections. The

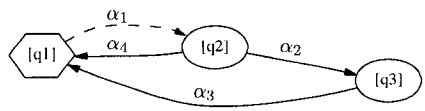


Figure 6-1: Finte state machine model M1 of the plant.

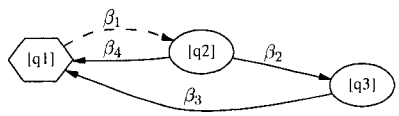


Figure 6-2: Finte state machine model M2 of the plant.

example also serves to introduce some of the terms used to describe the controller synthesis algorithms.

6.1.1.1 Example: Product DES Model

Suppose we have a pair of simple machines that share a common work area. For this example, the plant is modeled by a pair of finite state machines, $M_1 = (Q_1, \Sigma_1, \delta_1 \Gamma_1, q_{1,0})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, \Gamma_2, q_{2,0})$, illustrated in Fig. 6-1 and Fig. 6-2. Each machine has three states and four events as follows:

$$M_1 \; : \begin{cases} Q_1 = \{q_1, q_2, q_3\} \\ \Sigma_c = \{\alpha_1\} \\ \Sigma_u = \{\alpha_2, \alpha_3, \alpha_4\} \\ q_{1,0} = q_1 \end{cases}$$

$$M_2 \; : \begin{cases} Q_2 = \{q_1, q_2, q_3\} \\ \Sigma_c = \{\beta_1\} \\ \Sigma_u = \{\beta_2, \beta_3, \beta_4\} \\ q_{2,0} = q_1 \end{cases}$$

The controllable transitions are indicated by the graph edge with a small line segment, and the initial states of the models are indicated by the hexagonal-shaped graph

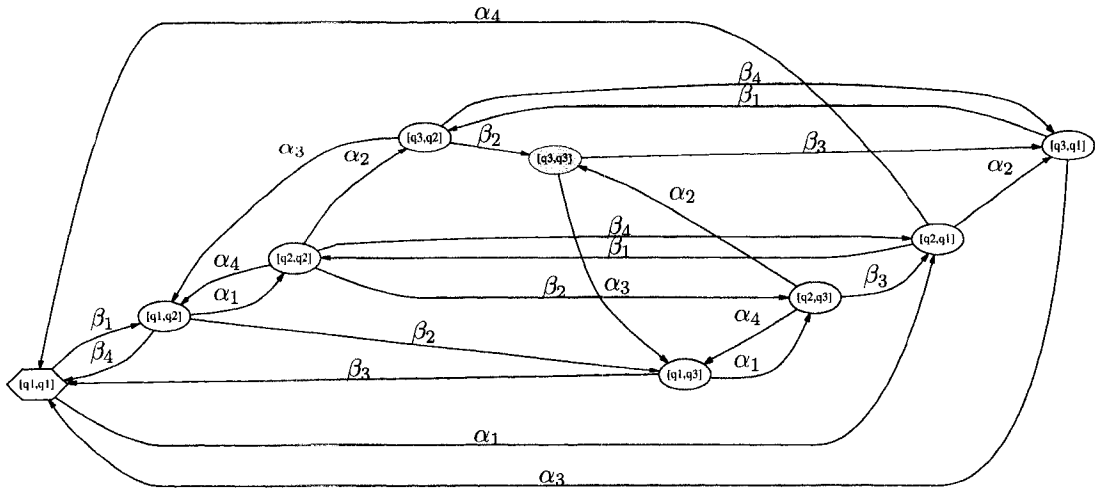


Figure 6-3: Product finite state model of the plant, $M1 \parallel M2$.

nodes. In this example, the state labels have been chosen to indicate the locations of the machines. If both M_1 and M_2 are in state q_2 (indicated by the product state $[q_2, q_2]$), then they have violated the requirement that they not enter the same work area simultaneously.

The completed plant model is the synchronous product automaton of the two machines, $M_1 \parallel M_2$, shown in Fig. 6-3.

The two separate FSM graphs have been "flattened" into a single graph structure representing the product behaviour of the two machines. Since there are no common events $\Sigma_1 \cap \Sigma_2 = \emptyset$, this plant is the shuffle of M_1 and M_2. Predictably, there are 9 states, since the shuffle produces a Cartesian product of the state labels $Q_1 \times Q_2$.

6.1.1.2 Example: Keen Control Synthesis

To examine how a controller is synthesized from the "flattened" plant model, we will use a specification that prevents the two machines from occupying their respective q_3 states at the time, that is, state $[q_3, q_3]$. An outline of an algorithm that uses the flattened plant model for controller synthesis is:

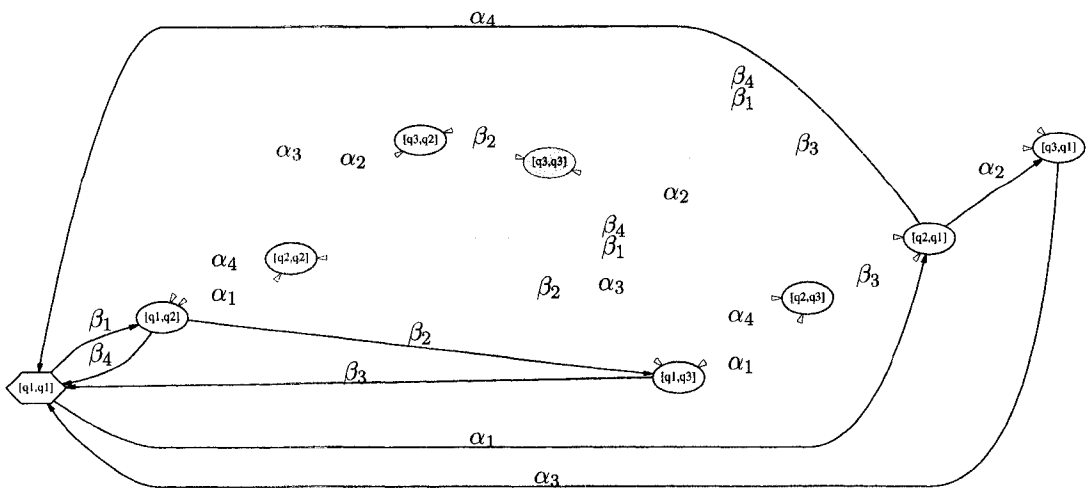


Figure 6-4: Controller graph for plant model of Figure 6-3.

1. Initially, $\Delta \leftarrow \emptyset$,

2. Form "flattened" product model $M_1 \parallel M_2$ by exhaustive state space search, once complete, the transition set Δ holds all of the graph transitions for the plant,

3. Do a depth-first reach on the flattened plant model, starting from the initial state $[q_1, q_1]$, until an illegal state is identified (e.g. blocking due to spec and marking criterion),

4. Step backwards through the graph, deleting transitions from Δ, until a controllable transition is encountered, deleting it from Δ,

5. Continue the reach (steps 2−3) until all remaining transitions in Δ are reachable and controllably safe.

To illustrate the keen algorithm, we will examine a single depth-first trace (Table 6.1. Assuming that step 1 of the algorithm has been completed already, the transition set Δ of the flattened product plant model contains all of the transitions pictured in the model of Fig. 6-3. Starting the reach with the initial product state of $q_{ps} = [q_1, q_1]$

Table 6.1: Example Runtime of Keen Algorithm

Event	$M_{1\|\|2}$ state	Transition Set, Δ	$\|\Delta\|$
	$[q_1, q_1]$	noop	24
β_1	$[q_1, q_2]$	noop	24
α_1	$[q_2, q_2]$	noop	24
β_2	$[q_2, q_3]$	noop	24
α_2	$[q_3, q_3]$ (illegal)	noop	24

Table 6.2: Example Runtime of Keen Algorithm, Removing Transitions

Event	$M_{1\|\|2}$ state	Transition Set, Δ	$\|\Delta\|$
	$[q_3, q_3]$	noop	24
α_2	$[q_2, q_3]$	$\Delta \leftarrow \Delta \setminus ([q_3, q_3], \alpha_2, [q_2, q_3])$	23
β_2	$[q_2, q_2]$	$\Delta \leftarrow \Delta \setminus ([q_2, q_3], \beta_2, [q_2, q_2])$	22
α_1	$[q_1, q_2]$	$\Delta \leftarrow \Delta \setminus ([q_2, q_2], \alpha_2, [q_1, q_2])$	21

and stepping forward through the graph:

Reaching the illegal state $[q_3, q_3]$, we now start to remove transitions from Δ, moving backwards through the graph (retracing the previous steps)

The process of removing transitions continues until the legal and reachable portions of the graph remain; i.e. the controller graph. The result of this controller synthesis is the controller structure of Fig. 6-4, with the number of transition in the transition set being pruned from $|\Delta| = 24$ down to $|\Delta| = 8$ transitions. In the figure, the controller is laid on top of the plant to show which transitions have been trimmed from the transition set of the full plant model.

A serious problem with the keen computation is that the number of states and transitions in the product model grows exponentially in the number of machines (sub models) that form the plant. For example, a simple factory model having 20 machines, each with a 3 state model, will have a product model of 3^{20}, or more than 3 billion states. An algorithm that utilizes the keen method for a realistically complex plant model, will likely run out of memory at step 1 of the above algorithm. This is the well-known *state explosion* problem of full state verification problems.

6.1.1.3 Lazy Computation: Product Object

There is an alternative to forming the "flattened" product graph. The product representing the plant model can be formed from a collection of its constituent models (Fig. 6-5). This hierarchical product object $M_{1\|2}$, is essentially a pointer to its constituent models, and instantiates all of the functions of the finite state product operation.

Continuing the example of the two machines, let the product state $x_{ps} = [x_1, x_2]$, then the transition function for the product object $\delta_{1\|2}$, is evaluated as follows

$$\delta_{1\|2}([x_1, x_2], \sigma) = [\delta_1(x_1, \sigma), \delta(x_2, \sigma)]$$

In Fig. 6-5, the product transition function can be seen as the parallel operation of the two sub-models, M_1 and M_2 (refer to Fig. 6-1 and Fig. 6-2). In this particular model, no synchronization between models M_1 and M_2 is necessary since there are no common events. Let $q_{ps} = [q_1, q_1]$ and $\sigma = \beta_1$, then

$$\begin{aligned} \delta_1(x_1, \sigma) &= \delta_1(q_1, \beta_1) = q_1 \\ \delta_2(x_2, \sigma) &= \delta_2(q_1, \beta_1) = q_2 \\ \delta_{1\|2}(x_{ps}, \sigma) &= \delta_{1\|2}([q_1, q_1], \beta_1) = [q_1, q_2] \end{aligned}$$

The product transition function is equivalent to the parallel combination of the transition functions of each of the constituent models.

How does the object-oriented plant model reduce synthesis complexity? Primarily, there is a savings in the use of space. An outline of the algorithm to synthesize a controller is as follows:

1. Initially, the transition set $\Delta \leftarrow \emptyset$,
2. Create a product object from M_1 and M_2,
3. Do a depth-first reach starting from the initial product state $[q_1, q_1]$, adding

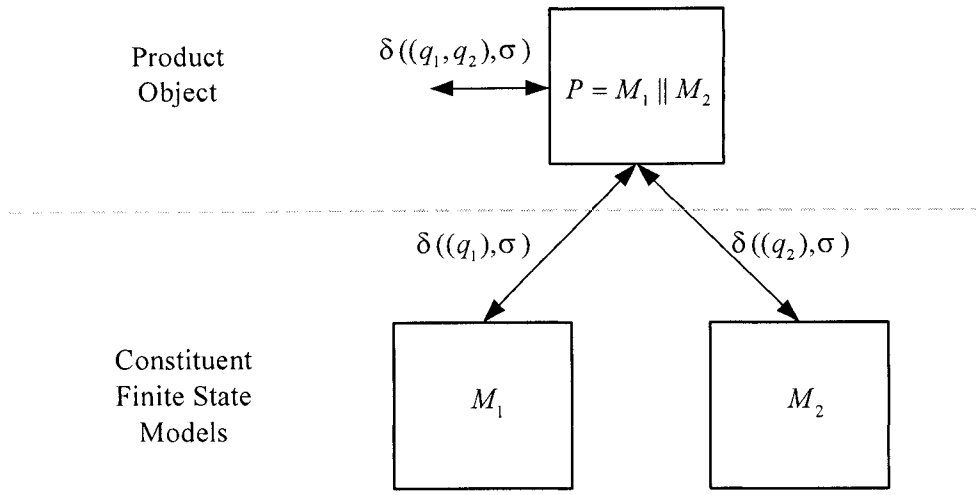


Figure 6-5: Plant model as a product object, illustrating overloaded transition function δ.

transitions to set Δ, until an illegal state is identified (e.g. blocking due to spec and marking criterion),

4. Step backwards through the graph, deleting transitions from set Δ, until a controllable transition is encountered, deleting it,

5. Continue the reach (steps $2-3$) until all remaining transitions in Δ are reachable and controllably safe.

To illustrate the lazy algorithm, we will examine one depth-first trace. Starting from the initial states of each of the constituent models, M_1 and M_2 and stepping forward, adding transitions: Now, stepping backward, and deleting uncontrollable transitions: Notes: (1) The transition $([q_2, q_3], \alpha_2, [q_3, q_3])$ is not added because it is illegal, (2) transition $([q_2, q_2], \beta_2, [q_2, q_3])$ is deleted because it is uncontrollable, and (3) transition $([q_1, q_2], \alpha_1, [q_2, q_2])$ is deleted to prevent the subsequent uncontrollable transitions of β_2 and α_2 (i.e. inhibiting α_1 enforces controllability).

Each depth-first reach into the model grows the transition set, followed by a reduction as the controllability condition is enforced. Continuing this process for the

Table 6.3: Example Runtime of Lazy Algorithm, adding Transitions

Event	M_1 state	M_2 state	Transition Set, Δ	$\lvert\Delta\rvert$
	$[q_1]$	$[q_1]$		0
β_1	$[q_1]$	$[q_2]$	$\Delta \leftarrow \Delta \cup ([q_1, q_1], \beta_1, [q_1, q_2])$	1
α_1	$[q_2]$	$[q_2]$	$\Delta \leftarrow \Delta \cup ([q_1, q_2], \alpha_1, [q_2, q_2])$	2
β_2	$[q_2]$	$[q_2]$	$\Delta \leftarrow \Delta \cup ([q_2, q_2], \beta_2, [q_2, 2])$	3
α_2	$[q_3]$	$[q_3]$		3

Table 6.4: Example Runtime of Lazy Algorithm, Removing Transitions

event	M_1 state	M_2 state	Transition Set, Δ	$\lvert\Delta\rvert$
α_2	$[q_2]$	$[q_3]$		3
β_2	$[q_2]$	$[q_2]$	$\Delta \leftarrow \Delta \setminus ([q_2, q_2], \beta_2, [q_2, q_3])$	2
α_1	$[q_1]$	$[q_2]$	$\Delta \leftarrow \Delta \setminus ([q_1, q_2], \alpha_1, [q_2, q_2])$	1

entire reachable model, the transition set of the completed controller will become $|\Delta| = 8$ transitions in size.

In this technique,

- the "flattened" plant product structure is never constructed,
- the product states (and product transitions) are only constructed *as needed* from the constituent models
- the specification is part of the product operation, thereby eliminating illegal transitions from the transition set *during construction*, of the controller.

During the computation, the transition set Δ will grow only slightly larger than the transition set of the final controller (due to controllability). The notion of constructing the product states "just in time" or only as needed is a powerful one, saving dramatically on memory requirements, and points to the strength of this approach for an implementation.

6.1.1.4 Example: Keen, Lazy Techniques

To contrast the two techniques, the keen method is a pruning process, while the lazy method is an additive process. In the lazy technique, the plant and specification are represented by a product object model, that allows the product to be computed in a "just in time" fashion. The product requires event synchronization between the models making up the product object. Providing there are some common events between specification and the plant models, the product will always have fewer transitions than the flattened plant model alone. This leads to a reduction in space complexity over the keen technique.

Example 6.1.1 *This example uses the analogy of navigating across a portion of a city (Fig. 6-6,* (1)*). The task of constructing a route that leads from the starting point (the green triangle) to a desired destination (the red octagon) is akin to computing a discrete event controller, with events corresponding to actions of left turn, right turn, or go straight; the states (or control points) correspond to the intersections of roads. Suppose the specification (a rule) is that directions that take us further from the destination (for example, in a euclidean sense) are illegal.Referring to Fig. 6-6, the keen algorithm begins at* (1) *with no knowledge of the map, but exhaustively builds a complete map* (2)*, by traversing all of the routes that reach the destination, building an exhaustive set of possible routes. Next, it checks each of these depth-first routes against the specification (the dotted contour lines indicate equal distance from the destination), removing transitions that are illegal, until only legal routes remain, as in* (3)*. With the lazy technique, we start with an empty transition set (map) at* (1)*. Routes are gradually added to the map by testing each route against the specification, until the entire legal and reachable map has been constructed in* (3)*. The result is that the lazy algorithm achieves an identical result to the keen technique, with a single step and with reduced space complexity. This is possible because the specification is used at design time, allowing illegal traces in many cases, to be eliminated before they are*

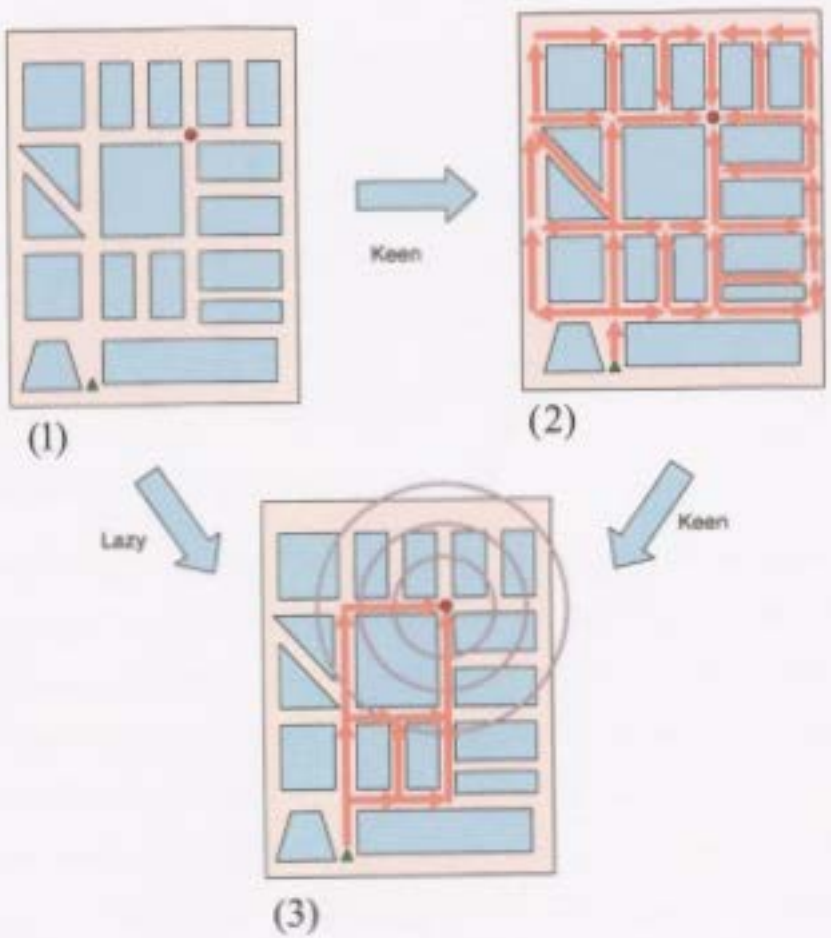


Figure 6-6: Map navigation analogy for controller synthesis algorithms.

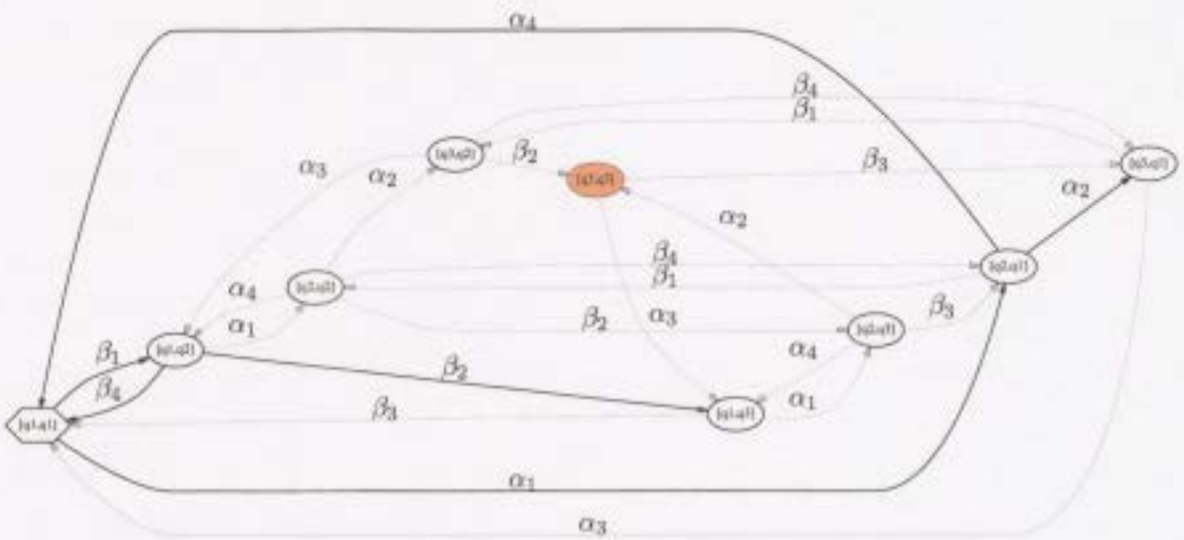


Figure 6-7: Limited lookahead 2 event controller graph for plant model of Figure 6-3.

fully constructed.

6.1.2 Limited Lookahead

Given a sufficiently large plant and specification, the controller synthesis process may be problematically large to compute. Limited lookahead is a technique that makes the complexity of controller synthesis more manageable. Limited lookahead refers to the fact that the reach depth of the controller reachability calculation is limited (truncated) during synthesis. This technique has been extensively examined in the context of discrete event supervisory control(Chung et al. 1992),(Chung, Lafortune and Lin 1994),(Hadj-Alouane, Lafortune and Lin 1994),(Kumar, Chung and Marcus 1998).

A lazy computation, combined with limited lookahead, is implemented as follows:

1. Initially, transition set $\Delta \leftarrow \emptyset$,

2. Instantiate a (non-flattened) product object from M_1 and M_2,

3. Do a depth-first reach starting from the initial state $[q_1, q_1]$, adding transitions

to set Δ, until either: (a) an illegal state is identified (e.g. blocking due to spec and marking criterion), or (b) the limited lookahead reach depth is reached,

4. Step backwards through the graph, deleting transitions from set Δ, until a controllable transition is encountered, deleting it,

5. Continue the reach (steps $2-3$) until all remaining transitions in Δ are reachable, less than the reach depth and controllably safe.

While limited lookahead is effective as a technique for complexity reduction, it is not possible to guarantee safety and non-blocking behaviour since the controller is designed for a partial model. Fig. 6-7 is the limited lookahead controller for the plant of Fig. 6-3 with a 2-event lookahead. This technique can be applied to either the keen or lazy techniques to provide a further reduction in space complexity. An additional benefit of the limited lookahead computation is that the space complexity is bounded. Since the size of the computation is known ahead of time, the computational resources can be planned making it more amenable to an online (real time) computation.

6.1.3 Online Computation

The terms *online* and *offline* refer to the synthesis of the controller: offline means the controller is completely synthesized before the execution of the control, while online control implies that the controller is synthesized during controller execution. Why include the controller synthesis as part of the runtime system? Online controller synthesis is a necessity to a limited lookahead control scheme. The controller map must be extended to match the moving system state by advancing the lookahead horizon. Moreover, the controller propagation process described in §5.5, allows us to deal with a time-varying model or specification. Online controller synthesis potentially allows the controller to react to modeled, but unexpected disturbances, that a controller constructed offline cannot. A requirement of online controller synthesis is that the

controller must be synthesized repeatedly in a real-time environment. Thus, the complexity of the controller synthesis computation must be of a predictable size, which, in turn, is why a limited lookahead scheme is amenable to online implementation.

6.2 Software

A software package has been designed that implements the modeling and controller synthesis framework described in the preceding sections. The package is called HySynth, for *Hy*brid *Synth*esis and is designed for the Matlab® environment. The software can be used to develop system models with a mixture of switched continuous and finite state models that represent the plant and specification of the target system. These models may be synchronized in a hierarchical product structure and then algorithmically manipulated to synthesize discrete event supervisors. Functions are also available to propagate the controllers and to simulate the controlled system. Additional functions are supplied to produce finite state graph outputs of the models in a variety of formats, including Postscript, Portable Document Format (PDF) and Virtual Reality Modeling Language (VRML). These outputs give the designer a means of visualizing the individual constituent models and the larger product graphs for troubleshooting purposes.

6.2.1 Architecture

HySynth exploits object-oriented programming (OOP) strategies. Models are stored as instantiations of appropriate classes:

`fsm` Class for deterministic finite state models, $G = (Q, \Sigma, \delta, \Gamma, q_0)$

`scm` Class for switched continuous models, $G = (\mathcal{F}, \Gamma, s_0)$

`product` Class for products of discrete event process objects, P

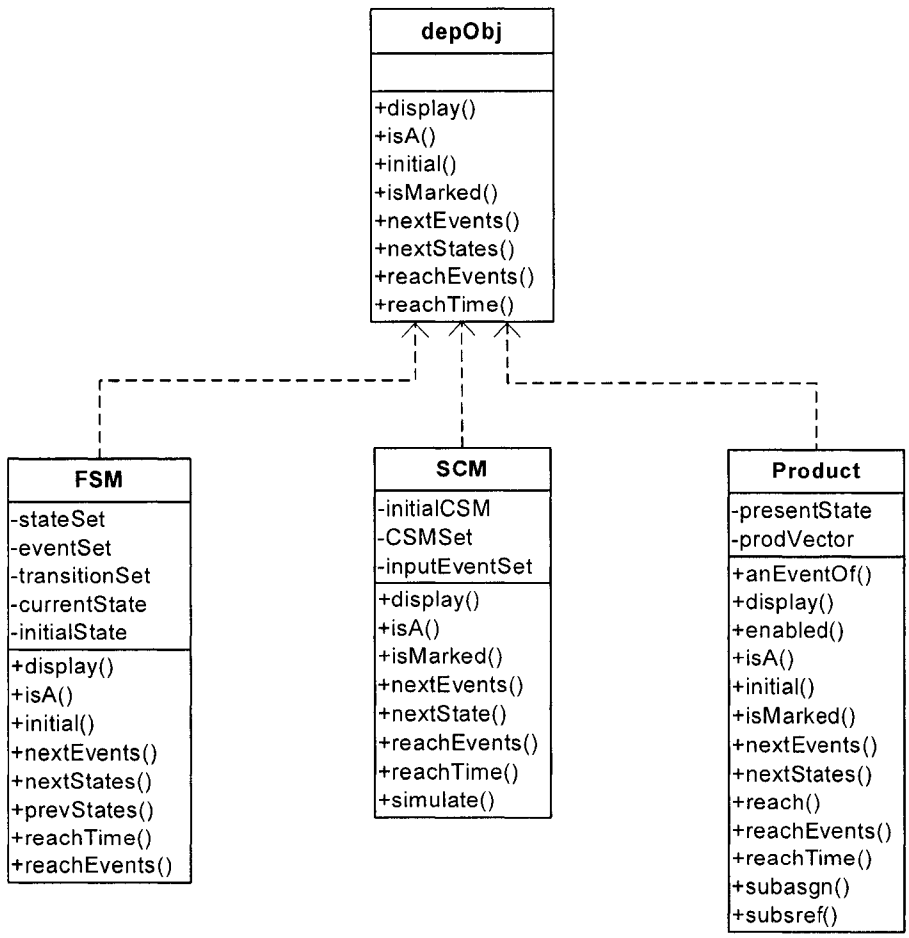


Figure 6-8: The discrete event process object class hierarchy.

Each of these classes is derived from an abstract base class, the discrete event process object class, or `depobj` class (Fig. 6-8). The `product` class acts as a heterogeneous container that stores a vector of `depobj` objects; i.e. types `fsm`, `scm` and also `product`. This recursive containment ability (a `product` may contain another `product`) permits hierarchical models to be constructed. The three classes present a common interface, implementing a variety of polymorphic methods. These methods permit the hierarchical models to be manipulated for analysis, simulation and control synthesis.

An example of a polymorphic function is the `nextEvents(state)` method that returns the set of next events for an object, given an argument of `state`. For a deterministic finite state machine $G = (Q, \Sigma, \delta, \Gamma, q_0)$, the `state` argument is a finite state, and the function evaluates the enabled events function Γ, returning a set of discrete events. For a switched continuous model $G = (\mathcal{F}, \Gamma, s_0)$, the function conducts a simulation for each $s \in \mathcal{F}$, with the `state` argument, a time-stamped continuous state. Simulation trajectories that cross the state boundaries (as defined by the respective set of partitioning functionals) generate events that become elements of the set of next events. This function equates to the enabled events function Γ_h of the HTG.

This discussion points out the need for a variety of types of states, since the domain of each function varies:

`finiteState` **Class** for finite state models, $q \in Q$

`ctsState` **Class** for continuous state variables, $x \in \mathbb{R}^n$

`pState` **Class** for product states, ps.

The domain of the product state has been deliberately omitted, to allow for a containment. These classes are derived from an abstract base class, the `stateObject` class (Fig. 6-9). As with the `product` class, the `pState` class is a heterogenous container, that stores a vector of `stateObject` objects. And like the `product` class, it can

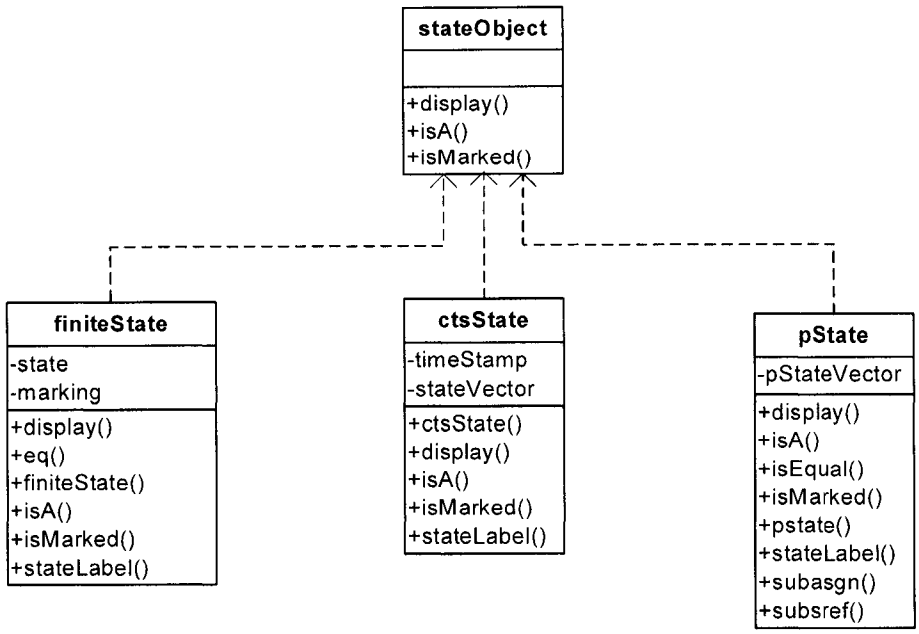


Figure 6-9: State object class hierarchy.

recursively store other `pstate` objects, allowing for states to match the hierarchical structure of the models.

The state space of a product object is not generated until run time, since the hierarchical object structure is maintained. By generating the state space only as needed using the lazy computation model, the costly computation of the "flattened" product state space is avoided. Fig. 6-10 illustrates a model constructed using the `product` class.

All of the algorithms that implement HySynth functions have been written to exploit this hierarchical storage of models. The core classes of HySynth in Fig. 6-8 and Fig. 6-9 have been implemented in Matlab scripting language, using the OOP programming features of Matlab (*MATLAB Programming* 2006). The resulting modeling framework essentially extends Matlab's interpretive command set to allow for modeling, analysis, controller synthesis and visualization.

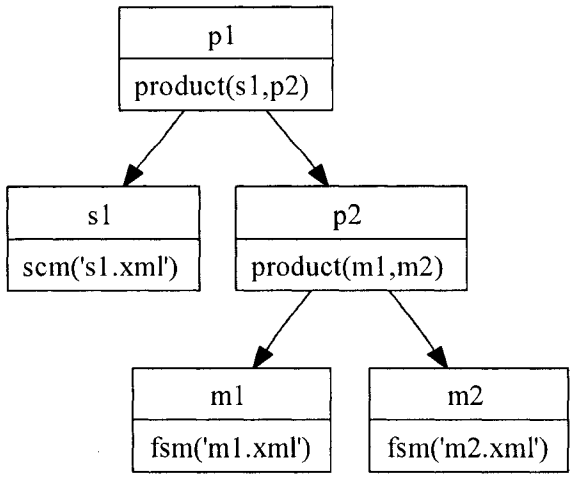


Figure 6-10: A model constructed as a set of hierarchical product objects.

The powerful continuous modeling and simulation capabilities of Matlab can also be embedded within our modeling framework through the `scm` class. While HySynth is currently implemented in Matlab, it could also be translated to any programming language that supports object oriented programming techniques[1].

6.2.1.1 Modeling Example

This example shows how the model of Fig. 6-10 can be constructed using a few commands.

```
m1 = fsm('m1.xml');        % read fsm models from source file
m2 = fsm('m2.xml');
p2 = product(m1,m2);       % product of m1 and m2
s1 = scm('s1.xml');        % get scm model
p1 = product(s1,p2);       % final model
```

[1] HynSynth is patterned on the architecture of the DES software OTCT, which was implemented in C++(O'Young 1992).

The `fsm()`, `scm()` and `product()` methods are the class constructors, allowing empty "placeholder" objects to be created or, as in this case, the models have been created from XML (extensible markup language) source files.

It should be emphasized that instantiating a product model object does not compute anything; it is merely a data structure with the models stored in a hierarchical fashion. If the product model `p2` is a specification and the SCM model `s1` is a plant, then a controller can be computed by finding the nonblocking reachability of `p1`. We begin by forming an initial state of the system that mirrors the hierarchy of the system model:

```
x0 = [20.1 32.7]                          % continuous state variable
t0 = 0.0                                  % initial time stamp
c0 = pstate(t0,x0);                       % initial time stamped cts state
q0 = pstate(initial(m1),initial(m2));     % initial state of specification
ps0 = pstate(c0,q0);                      % initial product state
```

Now the controller is formed by finding the nonblocking reachability of the model for some lookahead horizon, which in this case is 10 events:

```
[controller,exists] = reachEvents(p1,ps0,10);  %
```

The `reachEvents` function computes the controller transition structure (returned as `controller`) if it exists; indicated by the boolean value of return variable `exists`.

6.2.1.2 Product Class Method Dependency

From the previous example, it is clear that the `product` class is central to the modeling and synthesis framework. In Fig. 6-11, the method dependency diagram for the product class is presented. In this figure, a variety of high-level functions are listed, such as `printAsPDFwithEvents()`, a function that prints the flattened product to be stored to a PDF file for some lookahead horizon specified in events. In the dependency

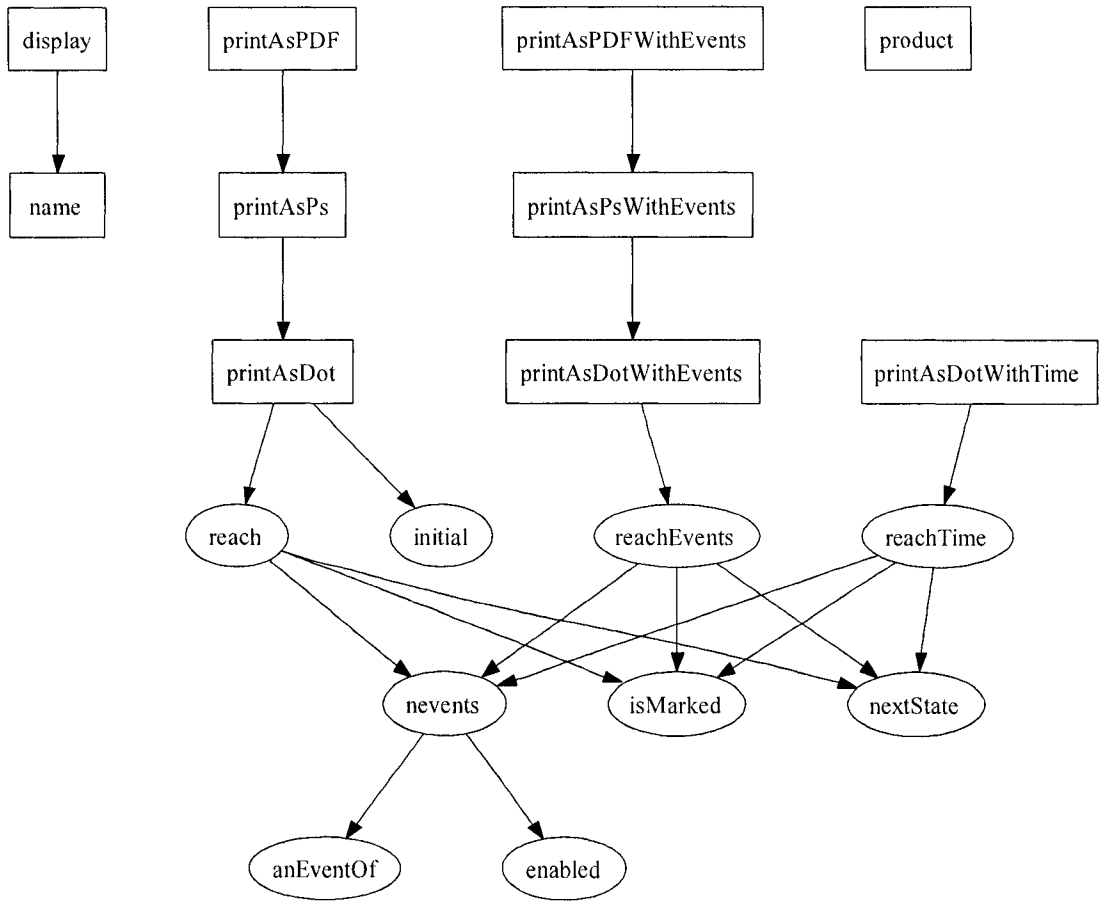


Figure 6-11: Method dependency for the product class. The product module is the constructor for this class.

Figure 6-12: Main menu of JFLAP automata and formal languages package.

diagram, recursive functions are indicated as ellipses. The recursion is necessary for the hierarchical storage of the `product` class.

6.2.2 User Interface

The user interface for HySynth was developed for prototyping purposes, but is reasonably easy to use. A simple graphical user interface is provided by a third-party software package, call JFLAP(Rodger and Finley 2006). This package is intended to be used as a tool for teaching students automata and formal languages, but for HySynth, it serves as a front-end for finite state machine capture. From the main menu (Fig. 6-12), the user selects the Finite Automaton option, which brings up an empty finite automaton capture window. In Fig. 6-13, the capture window has an automaton entered already. Once the designer has completed the design, the FSA may be stored to disk in an XML format file with default extension of .jff. The finite state model object in HySynth has a method that enables it to parse this file from disk, creating an `fsm` object.

The graph visualization capability of HySynth is based on the AT&T Graphviz graph layout engine (Gansner et al. 2002). Smaller graphs (<100 states) are useful to

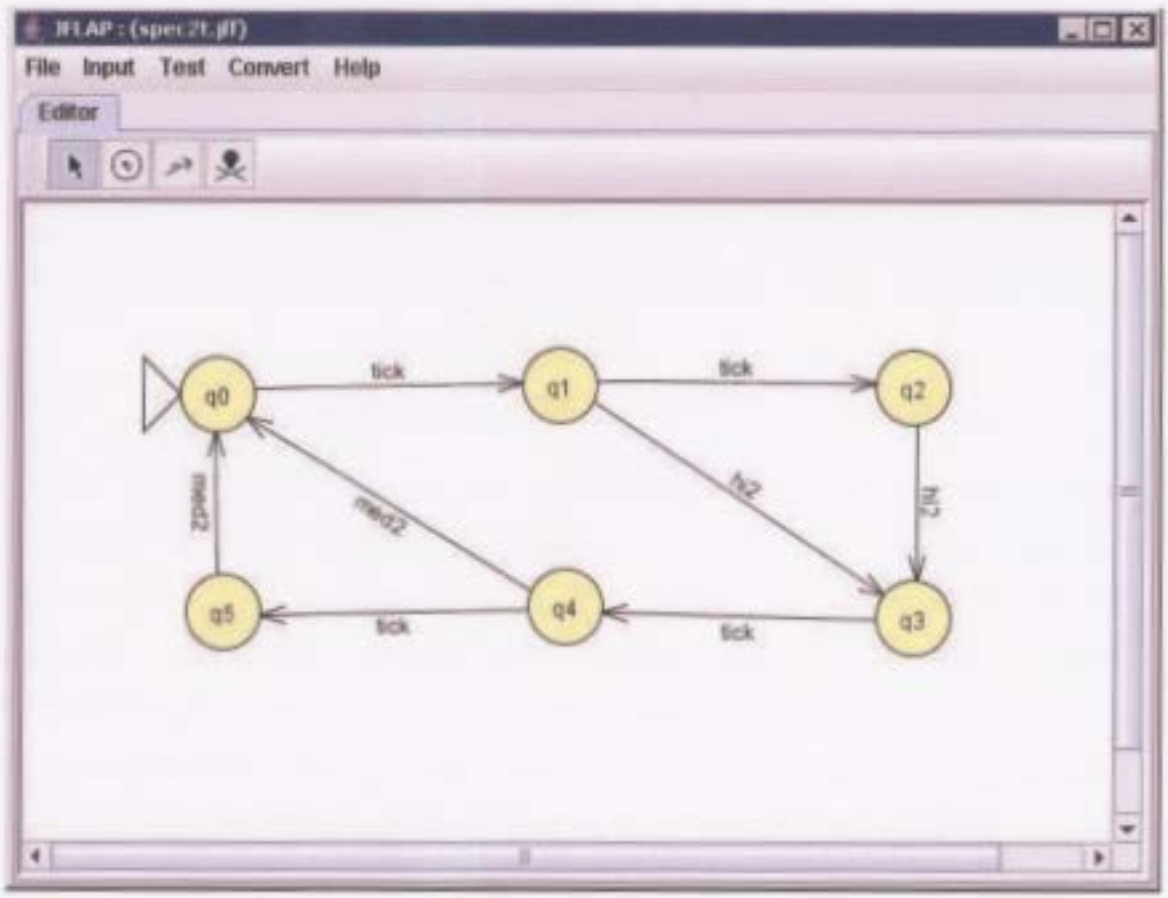


Figure 6-13: Finite state machine capture window.

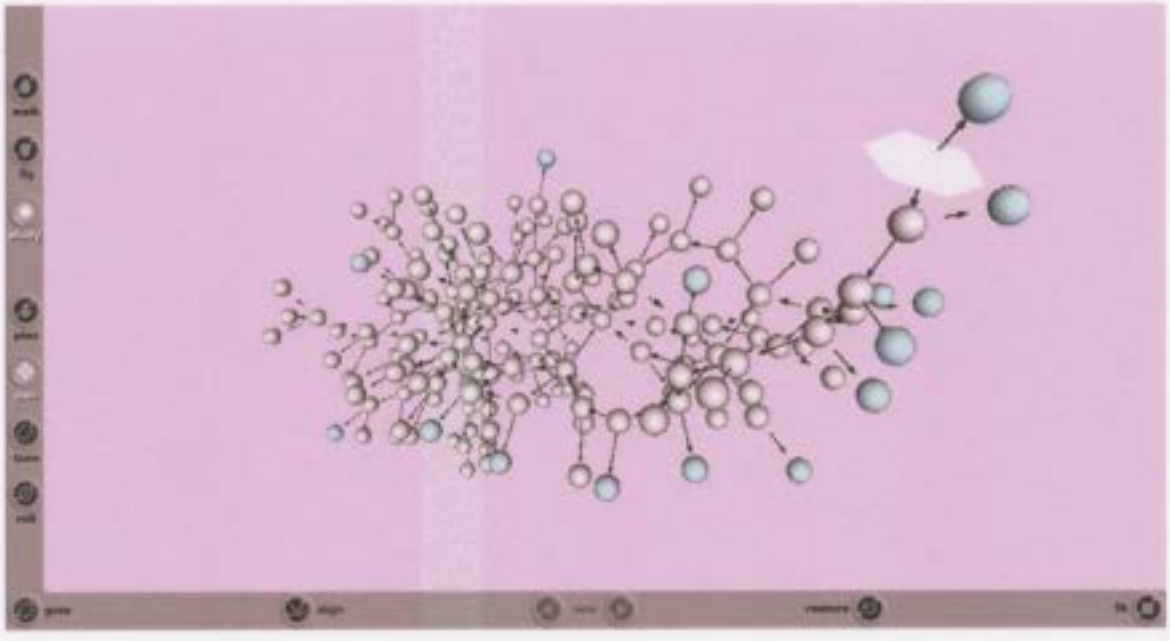


Figure 6-14: A three-dimensional view of a DES controller.

look at for debugging purposes. Even larger graphs (<5000 states) can be reasonably examined via a PDF file. Individual states can be found in PDF graphs with the standard search engine in the Adobe Acrobat reader. Colour and shape of nodes can be used to encode useful state information. Other options, such as 3 dimensional VRML visualizations may prove helpful for examining controller designs statically and also how they evolve through time and space using animations. Fig. 6-14 depicts a small controller in 3D, with initial state indicated by the hexagonal polygon, other controller time stamped states are spheres. The blue-coloured nodes indicate ESD states of the controller.

6.3 Algorithms

This section will provide some detail on the algorithms that have been designed to implement the SCM/FSM modeling, synchronization and controller synthesis.

6.3.1 SCM Functions

In order to implement the controller synthesis, the HTG state transition function δ_h and enabled events function Γ_h are required. At the heart of both algorithms is the evaluation of the solution of a continuous system model. Up to this point, an ordinary differential equation has been the "placeholder" for a broader class of simulations. In these examples, we assume (as before) Case II operation. so the solver must possess some sort of event detection. Event detection and location in ODE solvers is a well-studied problem and robust algorithms that add little computational overhead are available (Shampine and Gladwell 1991), (Shampine and Thompson 2000). Event detection is recognized as being an integral part of hybrid system simulation modeling and analysis (Alur et al. 2003), (Esposito, Kumar and Pappas 2003).

6.3.1.1 SCM Event Lookahead

Algorithm 6.1: An event-based reachability for the SCM $G = (\mathcal{F}, \Gamma, s_0, t)$

input : $\mathcal{R} \leftarrow \emptyset, s_0 \in \mathcal{F}, rd \geq 1, t \leftarrow t_0$
output: The set of continuous trajectories $x \in \mathcal{R}$ reachable in rd events

```
1 Function reachEvents(R, s, rd, t);
2 foreach s_i ∈ Γ(s) do
3 |   [x_i, t_i] ← simulate(s_i, t_i);
4 |   R ← R ∪ x_i;
5 |   if rd > 1 then
6 |   |   R ← R ∪ reachEvents(R, s_i, rd − 1, t);
7 |   end
8 end
9 return R
```

We shall revisit how the HTG is generated algorithmically from a SCM, based on the reachable continuous state space. One strategy for generating a HTG from an SCM is to predict its behaviour a fixed number of events into the future. We will use the abstract model of the SCM to demonstrate this with a depth-first reachability sweep in Algorithm 6.1. In line 3, the `simulate()` function is a generic continuous

dynamical simulation function that takes as its arguments the continuous system model s and an initial simulation time t. Starting from t, it returns the continuous solution (trajectory) x_i, to the first detected event, either the controller sample time Δt (associated with the *tick* event), or a partition crossing, whichever occurs first. The time at which the event occurred is also returned as t_i. Typically, but without loss of generality, $s \in \mathcal{F}$ are ordinary differential equations, and `simulate()` is an ODE solver that produces a solution x_i to the IVP posed by each continuous system model s. Each of these solutions is added to the reachable set of continuous trajectories $\mathcal{R}$. In line 5, the reach depth rd is tested to determine if the lookahead horizon has been reached. If not, the function calls `reachEvents()` recursively. The algorithm terminates with the continuous trajectories reachable in rd events returned in set $\mathcal{R}$.

6.3.1.2 SCM Time Lookahead

Algorithm 6.2: A time-based reachability for the SCM $G = (\mathcal{F}, \Gamma, s_0)$

input : $\mathcal{R} \leftarrow \emptyset, s_0 \in \mathcal{F}, t \leftarrow t_0, T \leftarrow t_0 + p\Delta t, p \in \mathbb{I}$
output: The set of continuous trajectories $x \in \mathcal{R}$ reachable in T time

```
1 Function reachTime(R, s, t, T);
2 foreach s_i ∈ Γ(s) do
3 |   [x_i, t_i] ← simulate(s_i, t);
4 |   R ← R ∪ x_i;
5 |   if (T − t_i) > 0) then
6 |   |   R ← R ∪ reachTime(R, s_i, t_i, T);
7 |   end
8 end
9 return R
```

The time lookahead strategy of Algorithm 6.2 predicts the SCM behaviour out to some fixed time horizon, T, relative to the initial simulation time t_0. The assumption is that t_0+T will be some integer multiple of the *tick* event time Δt, which guarantees that the simulation of line 3 will terminate with a *tick* event. In line 5, the simulation time is tested to see if it has reached the time horizon T; if not, the function calls

`reachTime()` recursively, with the new advanced simulation time t_i.

The sets generated by these algorithms represent the uncontrolled continuous behaviour of a system modeled as a SCM. For both time and event lookahead schemes, a HTG that corresponds to the continuous reachable trajectories can be constructed, since for each $x_i \in \mathcal{R}$ there exists a corresponding discrete event transition τ.

The HTG can be generated easily using modified versions of Algorithms 6.1 and 6.2, by modifying the `simulate()` function to return the detected output event $\sigma \in \Sigma_{out}$. The set of transitions $\mathcal{T}$ for the HTG can then be assembled.

6.3.1.3 Functions for HTG Traversal

Both δ_h and Γ_h algorithms assume that a solver with event detection exists. We first consider the Γ_h , or `enabledEvents` function in Algorithm 6.3 The algorithm assumes

Algorithm 6.3: *nextEvents* method of the *SCM* class

input : $G \leftarrow (\mathcal{F}, \Gamma, s_0), c_0 \leftarrow (t_0, x_0) \in (\mathbb{R} \times \mathbb{R}^n)$, tick time Δt
output: The set of enabled output events.

```
1 Function nextEvents(G, c_0, Δt) ;
2 nextEventSet ← ∅ ;
3 foreach s_i ∈ Γ do
4 |   solution of ODE posed by s_i on time interval T = t_0 + Δt;
5 |   T ← t_0 + Δt;
6 |   σ ← solveODE(f_i, x_0, T);
7 |   nextEventSet ← nextEventSet ∪ σ;
8 end
9 return (nextEventSet)
```

that all candidate continuous system models are eligible for execution. A solution for each CSM is executed on the Δt control cycle interval. The solver returns the output event, which will either be the detected event, or *tick* if no events are detected before the simulation terminates.

The transition function δ_h, is not implemented exactly as defined (Def. 4.8.4).

Rather, the set of next states that matches the output event $\sigma \in \Sigma_{out}$ is computed

$$\delta_{scm}(c, \sigma) = \{c' : c' = \delta_h(c, \sigma') \text{ if } \omega(c, \sigma') = \sigma\}$$

Roughly, this equates to finding the set of all simulations that generate the discrete output event σ. Algorithm 6.4 implements δ_{scm}. Note the similarity between each of the algorithms. Indeed, these functions actually solve the same ODEs, because the discrete transition information is derived form the continuous system models. For better computational efficiency, a transition table is usually computed, and the $\delta_h(\delta_{scm})$ and Γ_h are evaluated by performing a table lookup. The reachability code is designed to update the transition table only once at each choice point; i.e. the continuous simulations are executed once for each eligible CSM. Any subsequent calls to δ_h and Γ_h are then simply table lookups.

Algorithm 6.4: *nextStates* method of the *SCM* class

input : $G \leftarrow (\mathcal{F}, \Gamma, s_0), c_0 \leftarrow (t_0, x_0) \in (\mathbb{R} \times \mathbb{R}^n), \sigma \in \Sigma_{out}$, tick time Δt
output: The set of next states.

```
1  Function nextStates(G, c_0, Δt) ;
2  nextStateSet ← ∅ ;
3  foreach s_i ∈ Γ do
4  |   solution of ODE posed by s_i on time interval T = t_0 + Δt;
5  |   T ← t_0 + Δt;
6  |   [t_f, x_f, σ_e] ← solveODE(f_i, x_0, T);
7  |   if (σ_e == σ) then
8  |   |   c ← (t_f, x_f);
9  |   |   nextStateSet ← nextStateSet ∪ c;
10 |   end
11 end
12 return (nextStateSet)
```

6.3.2 Product Synchronization Functions

In the previous section, algorithms for traversing hybrid transition graphs were given. In this sections, algorithms for the product or synchronized versions of these functions is given.

6.3.2.1 Product Next States Function

Algorithm 6.5: Algorithm *nextStates* method of the *product* class

input : Product structure P, a product state $ps \in (\mathcal{C} \times Q)^N$, where N is the number of models in the product object P, $\sigma \in \Sigma$.
output: The set of next states Q_n for the product P.

```
1 Function nextStates(P, ps, σ) ;
2 stateSet[N] ← ∅;
3 foreach G ∈ P, state ∈ ps do
4 |  stateSet[i] ← nextStates(state, σ);
5 end
6 nextStateSet ← crossProduct(stateSet);
7 return (nextStateSet)
```

The implementation of the product transition function $\delta_{||}$ for an N-ary product object P, is illustrated in Algorithm 6.5, the `product::nextStates()` function. Since δ_{scm} (Algorithm 6.4), `scm::nextStates()` returns a set of next states Q_n instead of a single state, the algorithm for $\delta_{||}$ must also return a set of product states if there is a SCM as one of the objects in the product. From the modeling class diagram of Fig. 6-8, the `product`, `scm` and `fsm` classes all implement `nextStates()`. Therefore, a product object may contain other product objects (hierarchically nested product objects), in which case the `nextStates()` function call in line 4 is a recursive function call.

Essentially, the algorithm evaluates $\delta()$ for each object in the product (lines $3-5$), then compiles the next state set in line 6, by forming a cross product of each state set. In the case where each object in product P is a FSM, the function returns a set

with a single element.

6.3.2.2 Product Next Events Function

Algorithm 6.6: Algorithm *nextEvents* method of the *product* class

input : Product structure P, a product state $ps \in (\mathcal{C} \times Q)^N$, where N is the number of models in the product object P.
output: The set of enabled events Σ_n for the product P.

```
1  Function nextEvents(P, ps) ;
2  enabledEventSet ← ∅;
3  foreach G ∈ P, state ∈ ps do
4  |  enabledEventSet ← enabledEventSet ⋃ nextEvents(G,state) ;
5  end
6  foreach G ∈ P, state ∈ ps do
7  |  if (σ ∉ Γ_g(state)) ∧ (σ ∈ Σ_g) then
8  |  |  enabledEventSet ← enabledEventSet \ σ;
9  |  end
10 end
11 return (enabledEventSet)
```

The enabled events function $\Gamma_{||}()$ for an N-ary product object P, is illustrated in Algorithm 6.6, the `product::nextEvents()` function.

6.3.3 Nonblocking Reachability

Synthesis of a nonblocking safe controller is based on the discrete event reachability. Each of the algorithms that construct this reachability tree are, without loss of generality, based on a depth-first recursive reach. Thus, the reachability tree in each case is formed a trajectory at a time, with each branch being computed temporarily and pruned backwards as necessary before being added to the transition set. Essentially, for each lookahead type, the corresponding algorithm seeks to remove incomplete trajectories, i.e. those that do not reach the horizon.

Algorithm 6.7: A recursive reachability algorithm that returns the non-blocking HTG transition set of an SCM/FSM product structure for an integer event lookahead horizon.

input : Product structure $P, \mathcal{T} \leftarrow \emptyset$, initial product state $ps \in \mathcal{C} \times Q, rd \geq 1$

output: The set of transitions reachable in rd events for the product P

```
1  Function reachEvents(P,ps,rd);
2  if rd ≤ 0 then
3  |  return (true)
4  end
5  nonBlocking ← false;
6  enabledEventSet ← nextEvents(P,ps) ;
7  foreach σ ∈ enabledEventSet do
8  |  nextStatesSet ← nextStates(P,ps,σ) ;
9  |  foreach ns ∈ nextstateSet do
10 |  |  flag ← reachEvents(P,ns,rd-1);
11 |  |  if flag then
12 |  |  |  tranSet ← tranSet ∪ [ps, σ, ns];
13 |  |  end
14 |  |  nonBlocking ← nonBlocking ∨ flag;
15 |  end
16 end
17 return (nonBlocking)
```

6.3.3.1 Event Reachability

The controller is constructed from the product of plant and specification models using a modified reachability sweep (Algorithm 6.7). This recursive function takes arguments of a product model object, a product state object, and the lookahead horizon. This algorithm demonstrates a limited lookahead depth-first reachability. In this case, the lookahead horizon is specified by the number of events, the *rd* parameter in the function call. The function terminates when every branch of the reachability has been explored, either reaching the event lookahead or not. Providing that there exists at least one complete trajectory, the function will terminate, returning a status of boolean *true*, indicating that the reachability is non-empty. The function constructs a transition set $\mathcal{T}_R$ for the graph of the reachable controlled state space. Transitions are only placed in the transition set as they are verified to be non-blocking. This requires that the algorithm traverse all the way to the lookahead horizon to verify that a trajectory is complete. Thus, the transition set is built from the lookahead horizon backwards in this technique.

6.3.3.2 Time Reachability

As an alternative, the lookahead horizon of the reachability can be specified as a time horizon in terms of simulation time intervals, Δt, represented by the event label *tick*. In Algorithm 6.8, the *reachDepth* parameter of the `productReach` function now specifies the number of *tick* events for the lookahead horizon. The recursive function call decrements *rd* only if the enabled event is a *tick*. If the event is not a *tick*, the recursive call is made with *rd* unchanged. This function will only terminate if all trajectories are nonzeno, since if time is not able to advance, then the function will call itself *ad infinitum*. Assuming that the function does terminate, the resulting transition set will consist of only those trajectories having *rd tick* events in each generated string.

The time horizon can also be specified in dense time $T \geq t$. The algorithm will not

Algorithm 6.8: A recursive reachability algorithm that returns the nonblocking HTG transition set of an SCM/FSM product structure for an integer tick time lookahead horizon.

input : Product structure P, transition set $\mathcal{T} \leftarrow \emptyset$, initial product state $ps \in \mathcal{C} \times Q$, time horizon integer $td \geq 1$ *ticks*.
output: The set of transitions $\mathcal{T}$ reachable in td *tick* events for the nonblocking product P.

```
1  Function reachTime(P,ps,td);
2  if td ≤ 0 then
3  |  return true;
4  end
5  nonBlocking ← false;
6  enabledEventSet ← nextEvents(P,ps);
7  foreach σ ∈ enabledEventSet do
8  |  nextStateSet ← nextStates(P, ps, σ);
9  |  foreach ns ∈ nextStateSet do
10 |  |  if σ = tick then
11 |  |  |  flag ← reachTime(P,ns,td-1);
12 |  |  else
13 |  |  |  flag ← reachTime(P,ns,td);
14 |  |  end
15 |  |  if flag then
16 |  |  |  τ ← [ps, σ, ns];
17 |  |  |  transitionSet ← transitionSet ⋃ τ;
18 |  |  end
19 |  |  nonBlocking ← nonBlocking ∨ flag;
20 |  end
21 end
22 return nonBlocking;
```

be presented here due to the similarity with Algorithm 6.8. Recall that the discrete plant state is a continuous product state object consisting of the $[t_e, x_e]$ continuous state x_e and the simulation time t_e at the occurrence of the event. Essentially, the dense time can be extracted from the plant product state. This time t_e can be tested against the lookahead time T to determine if the horizon has been reached.

6.3.3.3 Combination Reachability

A combination of time and event lookahead horizons can also be utilized. Such a combined strategy has the benefit of assuring an upper bound that balances the objectives of both types of lookahead schemes. It reduces complexity in the following situations:

1. **Event Lookahead**: Continuous system model dynamics that are slow changing and thus generate few events (except for tick events) will terminate on a time horizon instead of continuing until the event limit.

2. **Time lookahead:** continuous system model dynamics and a partitioning structure that leads to dense switching behaviour (limit-cycle behaviour) will terminate on the event limit instead of continuing until the time limit.

The algorithm is simply a blend of the event and time horizon reachability algorithms. Essentially, the horizons are specified as integers $td \leq rd$. Whichever horizon is encountered first sets the returned nonblocking flag *true*, indicating the complete trajectory for this depth first reach to be nonblocking.

6.3.4 Fail-safe Controller Synthesis

For fail-safe control, an ESD state must be reachable (Proposition 5.4.1) from the current system state. Algorithm 6.10 builds a transition set that is pruned according to nonblocking and ESD state reachability rules. It also returns the nonblocking and

Algorithm 6.9: A recursive reachability algorithm that returns the non-blocking HTG transition set of an SCM/FSM product structure for a combination lookahead horizon based on either an integer number of events or an integer number of *ticks*.

input : Product structure $P, \mathcal{T} \leftarrow \emptyset$, initial product state $ps \in \mathcal{C} \times Q$, $1 \leq td \leq rd$

output: The set of transitions reachable in rd events or td ticks for the product P

```
1  Function reachCombo(P,ps,rd,td);
2  if rd ≤ 0 then
3  |  return (true)
4  end
5  if td ≤ 0 then
6  |  return (true)
7  end
8  nonBlocking ← false;
9  enabledEventSet ← nextEvents(P,ps) ;
10 foreach σ ∈ enabledEventSet do
11 |  nextStateSet ← nextStates(P, ps, σ);
12 |  foreach ns ∈ nextStateSet do
13 |  |  if σ = tick then
14 |  |  |  flag ← reachCombo(P,ns,rd,td-1);
15 |  |  else
16 |  |  |  flag ← reachCombo(P,ns,rd-1,td);
17 |  |  end
18 |  |  if flag then
19 |  |  |  τ ← [ps, σ, ns];
20 |  |  |  transitionSet ← transitionSet ⋃ τ;
21 |  |  end
22 |  |  nonBlocking ← nonBlocking ∨ flag;
23 |  end
24 end
25 return nonBlocking;
```

Algorithm 6.10: A recursive reachability algorithm that computes the nonblocking and ESD reachable HTG transition set of an SCM/FSM product structure for an integer event lookahead horizon.

input : Product structure $P, \mathcal{T} \leftarrow \emptyset$, initial product state $ps \in \mathcal{C} \times Q, rd \geq 1$

output: The set of transitions in rd event lookahead for the nonblocking and ESD reachable product P

```
1  Function reachEvents(P,ps,rd);
2  if (isMarked(p) ∨ isMarked(ps)) then
3  |  return ([true, false])
4  end
5  if rd ≤ 0 then
6  |  return ([false, true])
7  end
8  nbFlag ← false;
9  esdFlag ← false;
10 enabledEventSet ← nextEvents(P,ps) ;
11 foreach σ ∈ enabledEventSet do
12 |  nextStatesSet ← nextStates(P,ps,σ) ;
13 |  statusSet ← ∅;
14 |  foreach ns ∈ nextstateSet do
15 |  |  [esd, nonblocking] ← reachEvents(P,ns,rd-1);
16 |  |  if (esd ∧ nonblocking) then
17 |  |  |  τ ← [ps, σ, ns];
18 |  |  |  transitionSet ← transitionSet ⋃ τ;
19 |  |  |  esdFlag ← esdFlag ∨ esd;
20 |  |  |  nbFlag ← nbFlag ∨ nonBlocking;
21 |  |  else
22 |  |  |  nextStateSet ← nextStateset \ ns;
23 |  |  end
24 |  |  if esd then
25 |  |  |  statusSet ← statusSet ⋃ [esd, nonblocking];
26 |  |  end
27 |  end
28 end
29 return ([esdFlag, nbFlag], nextStateSet, statusSet)
```

ESD reachable status, and a matching set of next states (or transitions). This set of transitions, along with the nonblocking and ESD flags, are used to choose the next controller action (Table 5.1).The algorithm presented here is based on event horizon reach, but can be modified to be based on integer tick time, dense time, or combination lookahead.

6.3.4.1 ESD and State Marking

ESD states must be indicated specially; in this implementation, they are called *marked* states. The method `isMarked()` is implemented by subclasses of `stateObject` and `depObj`. Thus, marking may be jointly specified for individual states as well as for models. If a model (a `depObj`) is marked, then all of its states are considered to be marked. ESD marking has to retain the hierarchical structure of the `product` and `pstate` classes; a hierarchical truth object is returned for `product` or `pstate` classes that instantiate the `isMarked()` method. For example, let us look at the hierarchical system model of Fig. 6-10. The following commands generate the logical data structures that represent the model and state marking illustrated in Fig. 6-15:

```
modelMarking = isMarked(p1);     %p1 is the system model
stateMarking = isMarked(ps1);    %ps1 is the current product state
```

In line 2 of Algorithm 6.10, the logical OR ($\vee$) symbol is an overloaded operation. Without presenting the algorithm, the resulting marking is a element-wise OR between the "branch ends" of the model and the state structures; but with the requirement that the individual product object and product state object truth results must all be true for a product model or product state to be true (AND). This special test for marking is conducted to determine the ESD marking, and as is evident in Fig. 6-15, for this example the result is true.

For the `scm` class, no states are marked internally, but discrete states of an SCM can be designated as marked by synchronizing a FSM with the SCM to provide the

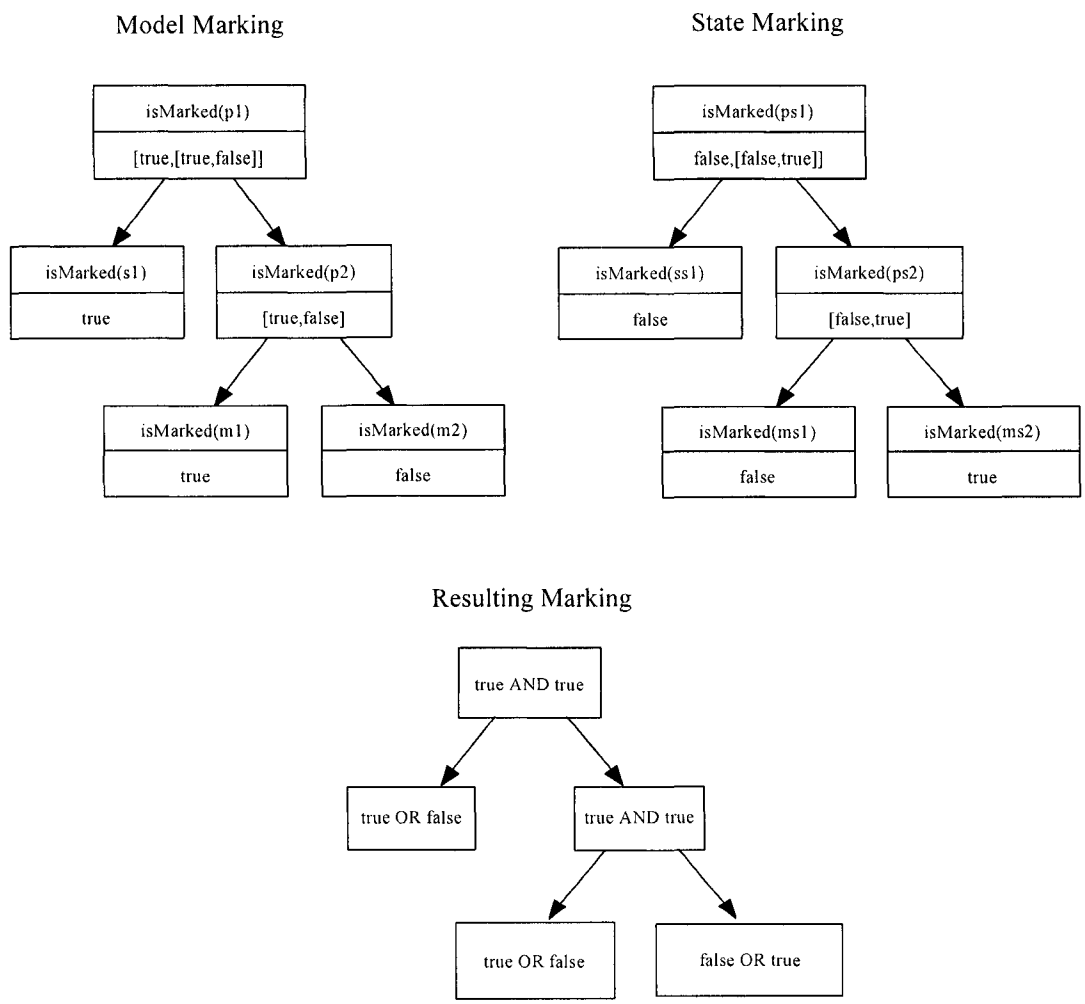


Figure 6-15: Hierarchical marking structures for object and state marking and the resulting marking decision.

appropriate marking. Alternatively, as in this example, all states may be marked in the SCM by using the object marking to always override the state marking.

6.4 Complexity

The complexity of the controller synthesis computation is of utmost importance to the practical implementation of an online controller. In previous sections, considerable care was taken to outline the various approaches to reducing complexity, including lazy computation, and limited lookahead. This section will outline the expected space and time complexity of implementing the hybrid controller synthesis.

Essentially, the process of online synthesis consists of generating a locally safe controller repeatedly at different instances in space, and more importantly, time. The controller is assembled in a just-in-time fashion, so it is essential that the computation can be computed reliably, in a finite amount of time. The controller is represented by a directed acyclic graph that is the result of the product of a switched continuous models and one or more finite state models. The plant, represented by the switched continuous model, can be "flattened", that is, its reachable state space can be calculated on a reduced time horizon combined with the limited switching framework (choice points) to produce a finite state model. In §4.6, the cardinality of the reachable state space $\mathcal{R}$ of a SCM was derived for Case I and Case II switching (Eq. 4.7 and Eq. 4.4). Since there exists a transition in the HTG for every $x_i \in \mathcal{R}$, the number of transitions is as follows:

$$pq + 1 \leq |\mathcal{T}_R| \leq \frac{r^{pq+1} - 1}{r - 1}, \; r > 1$$

where q is the length of the longest SC trajectory (the maximum reach depth) and r is the number of possible continuous system models (control actions) available at any choice point. Since both q and r are a function of the specification model, the plant model and the particular product state at the moment the controller is to

be computed, it is not possible to find the exact controller size, but it is possible to classify the order of the calculation in space and time. the following sections examine the complexity for a system modeled by a single SCM.

6.4.1 Constant Event Reach (Plant)

The constant event reach has a fixed number of states which is determined by integer q, the reach depth, and integer r, the branching factor (the number of enabled continuous system models). Thus, the number of transitions in the graph that implements the finite state model is

$$|\mathcal{T}_R| = \frac{r^{q+1} - 1}{r - 1}, \; r > 1$$

providing that there are exactly r choices at each choice point.

Lemma 6.4.1 (Space Complexity Reach (Events)) *Let $G = (F, \Gamma, s_0)$ be a switched continuous model. The space complexity of the reachability calculation with constant event horizon (reach depth) is polynomial in r.*

Proof. With the number of events in any complete switched continuous trajectory (string) $|\xi| = q$ fixed, the number of transitions in the HTG is a function of r

$$|\mathcal{T}_R| = f(r) = \frac{r^{q+1} - 1}{r - 1}, \; r > 1$$

with r large, this converges to r^q. Therefore for $q > 1$, the space complexity is $O(r^q)$ which is polynomial in r. ■

6.4.2 Constant Time Reach (Plant)

With constant time reach, the maximum reach depth in events pq, is not consistent for each SC trajectory $\xi \in \mathcal{R}$, and is unpredictable depending upon the dynamics and partitioning structure of the SC model $G = (\mathcal{F}, \Gamma, s_0)$. Recall that the reachable

state space $\mathcal{R}$ of an SC model of a fixed-time horizon of $p\ \Delta t$, with finite number of partition switches per time interval of $q = |\xi|$ has an upper bound of

$$|\mathcal{R}| \leq \frac{r^{pq+1} - 1}{r - 1},\ r > 1$$

which corresponds directly to the number transitions$|\mathcal{T}_R|$ in the HTG of G.

Lemma 6.4.2 (Space Complexity Reach (Breadth)) *Let $G = (F, \Gamma, s_0)$ be a switched continuous model. The space complexity of the reachability calculation with a fixed number of model choices (reach breadth) is exponential in pq.*

Proof. Let $pq = q'$. Clearly with r constant, the number of states in the reachability graph, N, is a function of q'

$$|\mathcal{T}_R| = f(q') = \frac{r^{q'+1} - 1}{r - 1},\ r > 1$$

Therefore, the worst-case space complexity is $O(r^{q'})$, that is exponential in pq. ■

6.4.3 Complexity With Control

This section examines the computational cost of computing a controller. The number of possible branches that can be made at any choice point is dictated by the number of available dynamics $r = |\Gamma|$, and by the number of these choices that are disabled by the specification. This disablement is unpredictable and is determined by how "tight" the specification is. Any other synchronous model connected to the plant may also constrain the plant branching behaviour, even though it may not necessarily be considered as part of the specification.

- State complexity of the controller has an upper bound which is polynomial in r, and exponential in q. In practice, however, due to the discrete-event interaction of the SCM with other plant and specification models, the complexity will always be less.

- Time complexity for reachability of a directed graph is $O(N)$, in the number of nodes or states of the controller, N. Therefore, time complexity for forming the controller, in terms of q and r, is essentially the same complexity as the state complexity. Therefore time complexity is at worst, exponential, but will always be better depending on the interaction of the SCM with other models.

- A 'tight' specification helps to reduce complexity. The implication of a "tight" specification however, is that a larger set of plant behaviours will be disabled. This effectively reduces the available control choices, increasing the likelihood of blocking, and an unwarranted emergency shutdown.

6.4.4 Empirical Complexity

It is not possible to analytically predict the exact complexity of controller synthesis for a particular system model, but it is possible to evaluate it empirically.

To demonstrate complexity reduction, we will use the tank control example of §4.9 again. The plot of Fig. 6-16 shows the controller size for this example (at the same initial state) for a range of event lookahead horizons. Plotted on a logarithmic scale, the number of transitions in the uncontrolled plant increases by approximately one order of magnitude for an increase in lookahead horizon by 2 events. For this example, there is a considerable decrease in controller size due to the inclusion of the specification; an approximate curve fit has been applied (in blue) to the controller data.

The size of the controller also varies depending on the initial condition; a simulation of the tank control example repeated for 50 controller updates, shows a considerable variation of the controller size as the controller is propagated (Fig. 6-17).

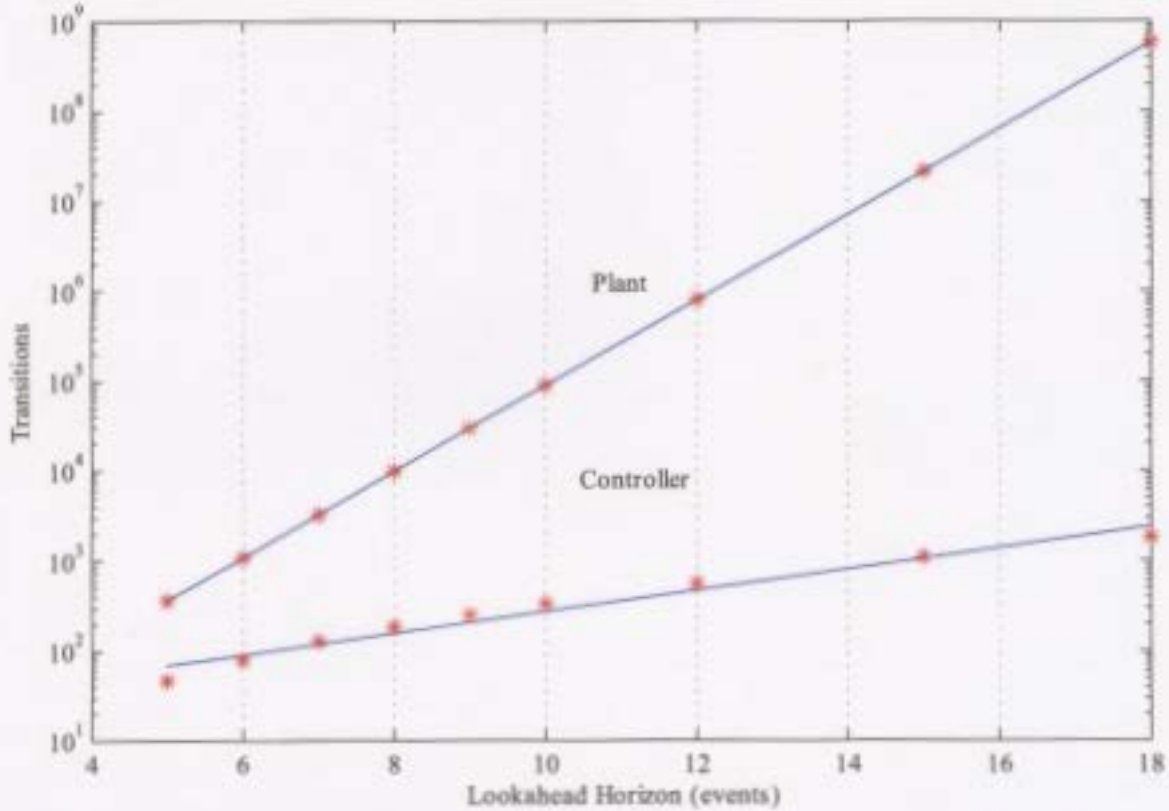


Figure 6-16: Emiprical controller complexity results for tank filling example.

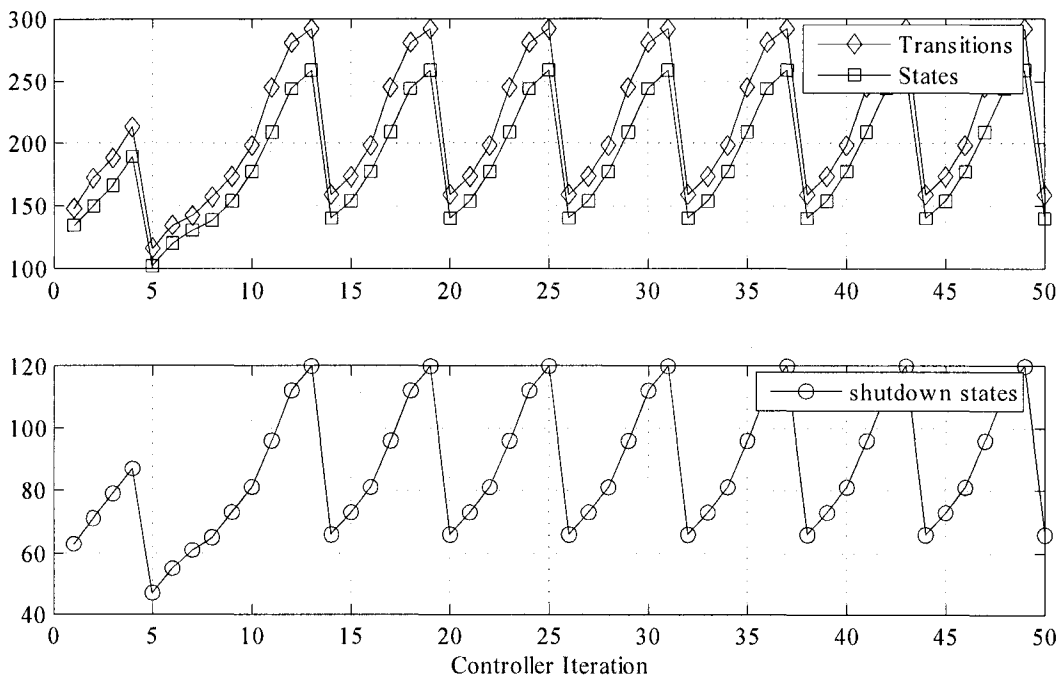


Figure 6-17: Controller size measured in number of states, number of transitions, and number of emergency shutdown states.

6.5 Summary

This chapter has taken the theoretical concepts of the preceding chapters and described one possible implementation. The controllers that we wish to synthesize are implemented online (i.e. a real time environment); thus the entire computational framework has been designed to efficiently achieve this goal. The primary challenge is to manage the computational complexity of the controller synthesis, since the controller is to be implemented online. We achieve a significant reduction in computational complexity by combining the specification with the controller at design time. The hierarchical modeling framework allows for a "lazy" or just in time assembly of the controller from its constituent models. The reduction in complexity comes from the fact that illegal traces can be eliminated without expanding the entire flattened state space of the model. In fact, since we are dealing with a hybrid model that theoretically has an infinite state space, the entire model is impossible to compute. Therefore, a limited lookahead horizon technique is used to compute the reachable state space. The lookahead horizon may be specified in events, integer (tick) time, dense time, or a combination of these.

A software package called HySynth has been developed to help evaluate the effectiveness and the practicality of the theoretical approach. The HySynth computational engine exploits object oriented programming techniques to implement the lazy computational strategy. Written for the Matlab environment, the modeling framework can leverage the wide range of general and special purpose numerical simulation toolboxes available for Matlab.

In the next chapter, we will examine specific application examples, in which HySynth is used to formulate hybrid system models, compute discrete event supervisory controllers, and to simulate the closed-loop system behaviour.

Chapter 7

Applications

This chapter presents two application examples that demonstrate in more detail, the system modeling process and controller synthesis for hybrid systems using the switched continuous model. The HySynth software package has been used to model, synthesize and simulate the controlled systems that are presented here. The first example, the control of liquid in two tanks, is a common benchmark hybrid control design problem. The second example is a detailed industrial example based on the control of a vessel and its associated systems. This example in particular demonstrates the utility of the control techniques developed in this document, and represents the first time a hybrid control design has been attempted for this application.

7.1 Tank Level Control

Recall the example of §4.9 (p.82) in which a SCM was developed for a system consisting of a tank of some fluid. In this example, we will develop a specification, synthesize an online controller and simulate its operation. This example points out the basics of control synthesis and illustrates the use of the HySynth software package.

With the plant modeled as a SCM, the desired closed loop behavior can be specified in the form of finite state machines. The desired behavior is modeled by the finite state

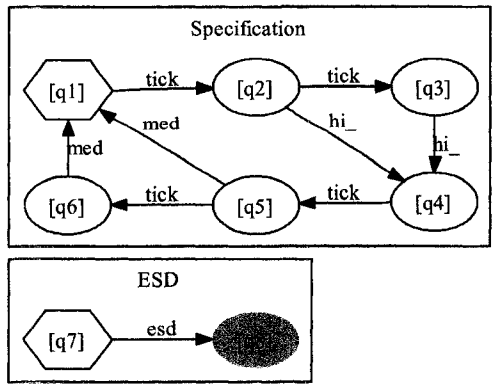


Figure 7-1: Specifications for the legal behavior, and the emergency shutdown.

models of Fig. 7-1, and is composed of two parts; the performance specification and the emergency shutdown behavior. The specification programs the controller to fill the tank to the high level then drain it to the medium level and repeat the cycle. The timing for this operation is specified coarsely, indicating that the cycle must complete in a minimum of two *tick* events, and a maximum of four *tick* events. The event set for the specification is $\Sigma_{spec} = \{ovf, hi, med, unf, tick\}$. The inclusion of the *ovf* and *unf* events in the specification event set forces these events to be illegal because of the event synchronization rule between the plant and the specification. Here is an example where safety is represented by blocking (of entering unsafe states).

The ESD model specifies the ESD state marking, thus the dynamics of the ESD procedure are handled in the SCM, but the designation of the state as a safe shutdown state is handled in the finite state model (provided that all other models are also explicitly marked, or neutral with respect to marking).

Using the HySynth software as described in §6.2.2, a model is constructed by loading individual model objects form source files (stored in XML format) that have been designed using the graphical user interface. For this example, the following command sequence produces the basic model:

```
tank = scm('tank.xml');      % create scm from each source file
esd = fsm('esd.xml');
spec = fsm('spec.xml');
p = product(tank,spec,esd); % create product object
```

The product object p is the basis for the controller synthesis. To synthesize a controller, the current state and time of the system are initialized by creating the initial continuous product state for the SCM

```
x0 = ctsState(26);    % the initial cts variable
t0 = ctsState(0)      % initial simulation time
c0 = pstate(x0,t0);   %
```

Now, the initial product state for p, taking the default initial conditions (as defined by the FSM) is:

```
ps  = pstate(c0,initial(spec),initial(esd));
```

The hybrid transition graph of a controller for an eight event lookahead is generated by running a reachability command on the product system object:

```
[struct] = reachEvents(p,ps,8);
```

The `reachEvents()` command executes an eight-event reachability on the product system, starting from the argument initial product state. The controller update time (corresponding to the tick event) has been set to 90 seconds for this example. If successful (a controller exists with respect to the specification), a data structure `struct` is returned containing the set of legal next events and the ESD reachability and nonblocking status of each subgraph. The size of this graph is too large to present here, but a representative example constructed using a three event lookahead is given in Fig. 7-2. This graph has been laid out using the *neato* engine, part of the AT&T Graphviz graph layout suite (Gansner et al. 2002). The result is that the initial state of the tree is in the center of the graph, with the immediate legal control actions surrounding it like spokes from a hub. By examination, there are three possible control choices. the transition to the upper right and the transition

to the lower left (both generate *tick* output events) are priority 1 subgraphs because they each have at least one nonblocking (complete) trajectory, in addition to being able to reach an emergency shutdown state (in blue). There is a priority 2 trajectory available, directly to the shutdown state (the *esd* output event).

7.1.1 Example: ESD Controller Operation

To illustrate how a controller will enforce a safe shutdown, we modify the single tank model to include a deliberate shutdown program. This programmed failure is modeled by the FSM of Fig. 7-4, which after three *hi* events blocks the system from proceeding.

In the simulation (Fig. 7-5), the programmed failure forces the controller to issue a shutdown command *sd*, which opens the purge valve, bringing the system to an ESD state (tank drained, $h \leq 0.5$). The size of the controller during this simulation is the subject of Fig. 7-6. Once the controller has forced a shutdown, the controller size shrinks as fewer available trajectories are available, and the system closes in on the ESD state.

7.1.2 Controlling Two Tanks

More complex dynamics are generated by this system if a second tank is added to the plant (Fig. 7-7). The two tank system is widely used as a benchmark for control techniques due to the richness of dynamics that it presents. A survey of the literature shows a wide range of control approaches that have been taken, including: robust control (μ synthesis) (Smith and Doyle 1988), Lyapunov-stable switched systems approach (Ecker and Malmborg 1999), timed and hybrid automata (Stursberg, Kowalewski, Hoffmann and Preusig 1997) and a discrete abstraction and supervisory controller in (Su et al. 2003).

Some modifications to the SCM are required, since an additional control valve has been added. A second purge valve, also labeled P has been added as well, to permit

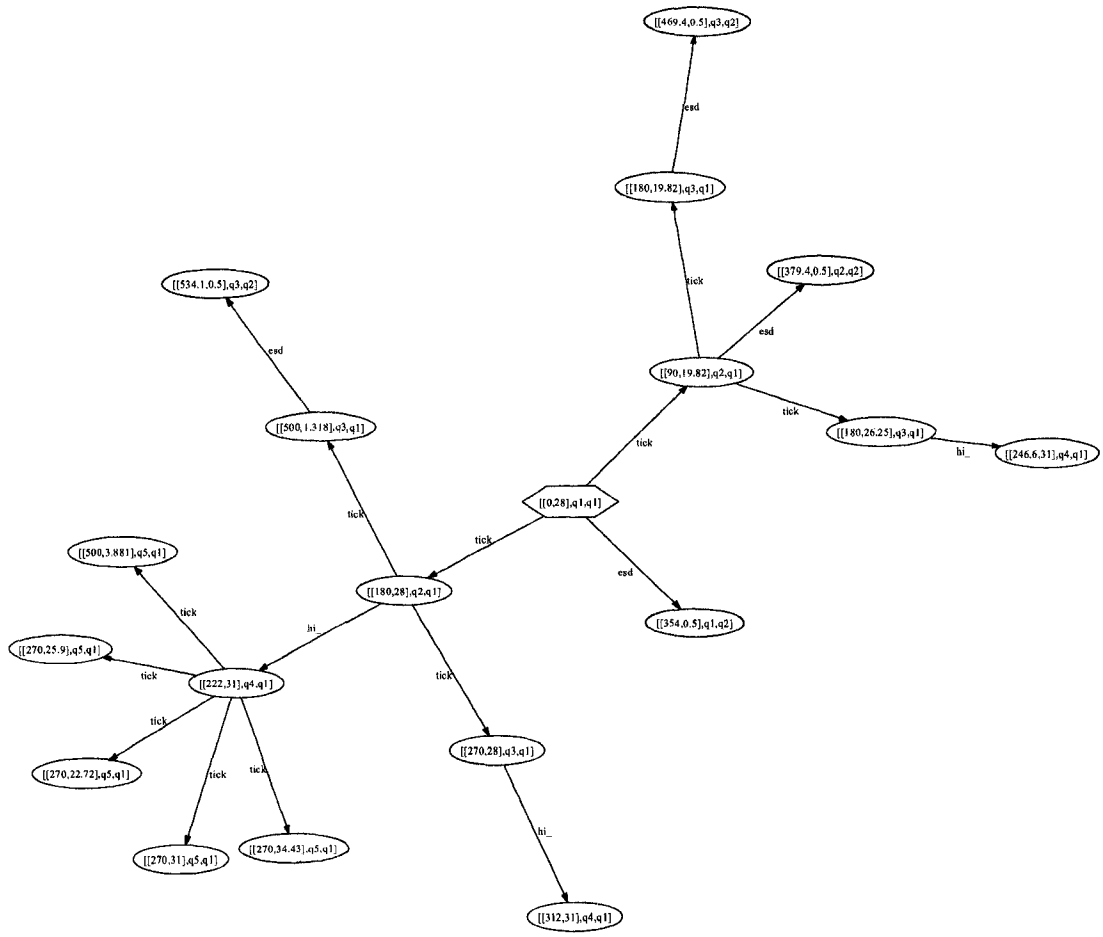


Figure 7-2: A three-event controller sturcture for the tank, with input (control) events omitted.

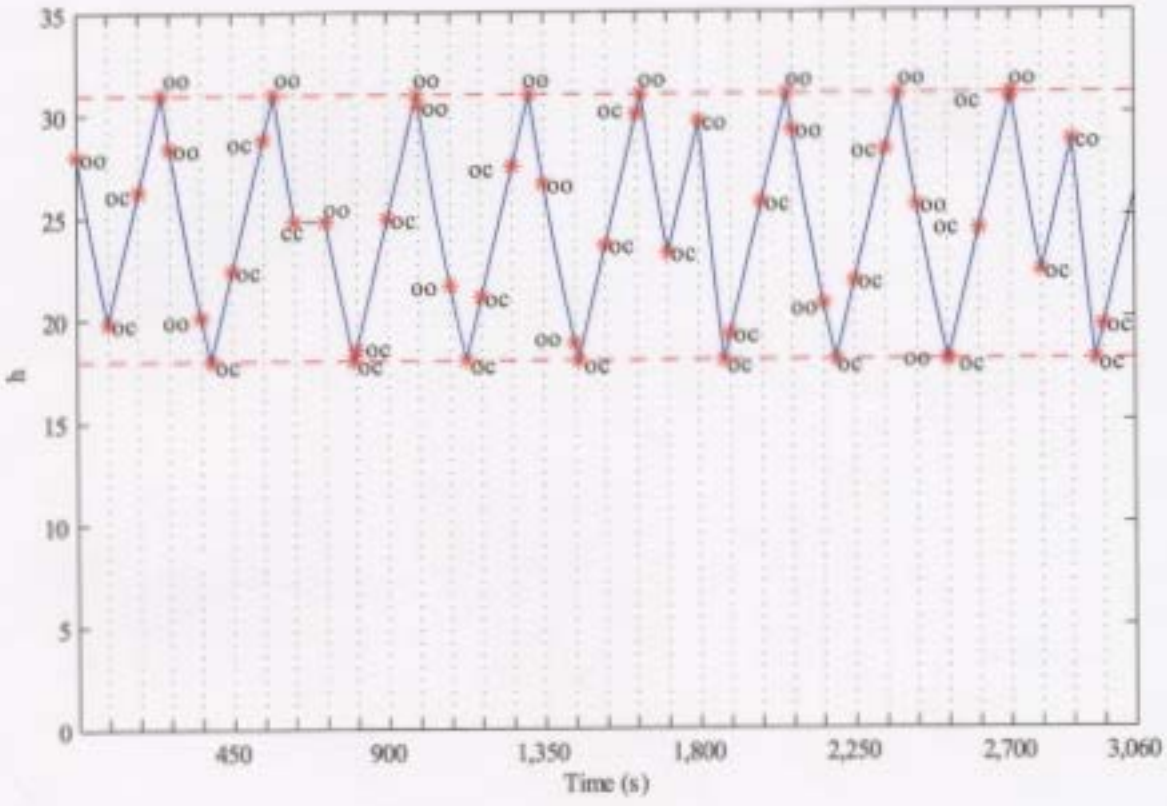


Figure 7-3: Plot of fluid level versus time, with control actions superimposed.

Figure 7-4: Programmed shutdown specification used to test online controller shutdown

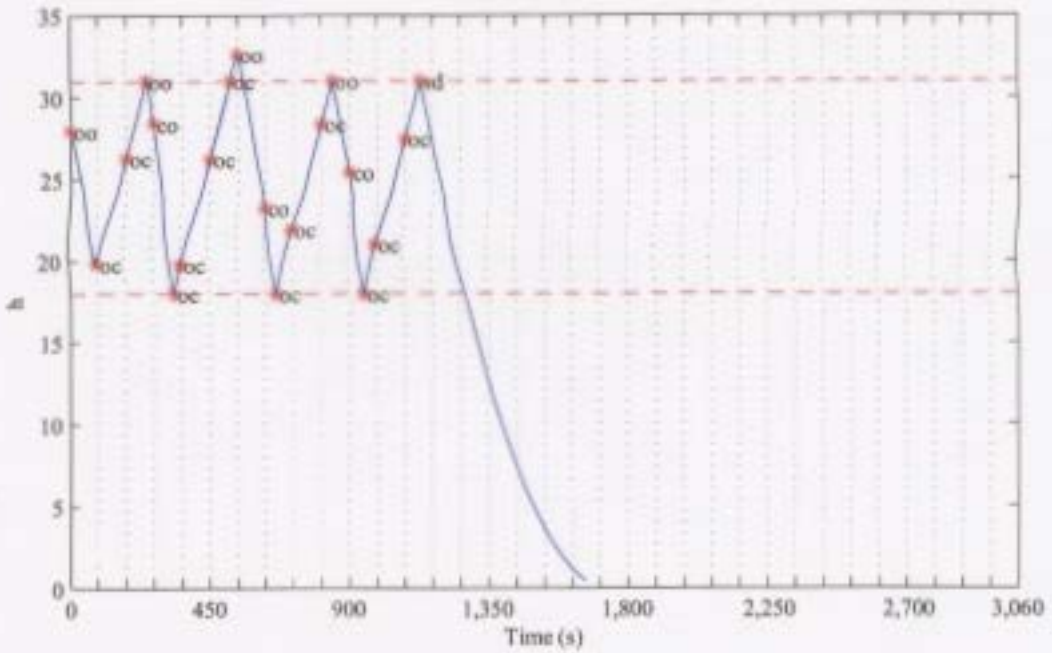


Figure 7-5: A trace showing an emergency shutdown, forced by synchronization with the shutdown program of Figure 7-4.

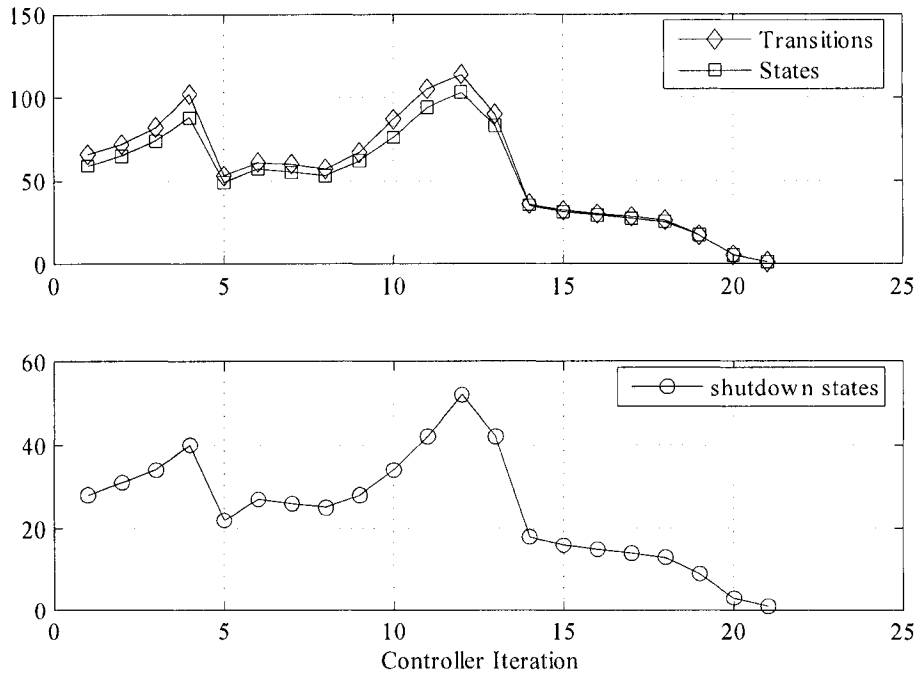


Figure 7-6: The variation in controller size for the programmed failure simulation pictured in Figure 7-5

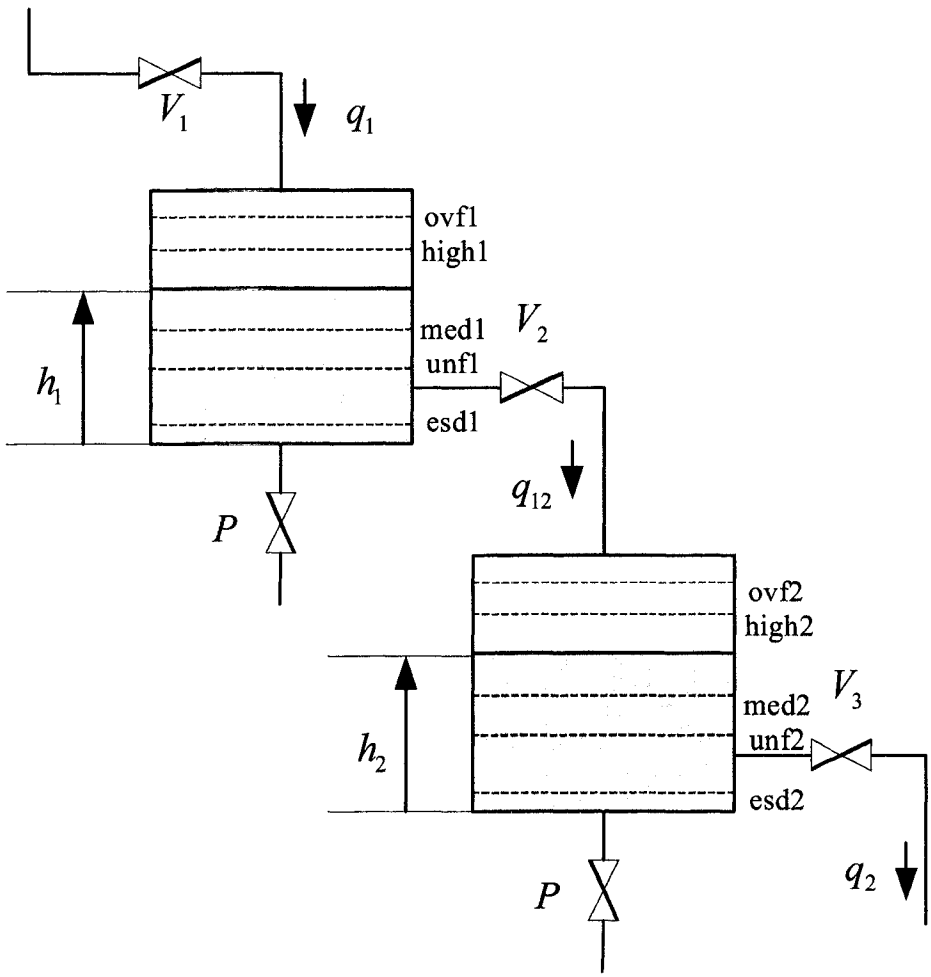


Figure 7-7: Two tank system schematic.

Table 7.1: Valve control structure

	Controls			
σ_{in}	V_1	V_2	$V3$	P
i_1	0[a]	0	0	0
i_2	0	0	1	0
i_3	0	1	0	0
i_4	0	1	1	0
i_5	1	0	0	0
i_6	1	0	1	0
i_7	1	1	0	0
i_8	1	1	1	0
sd	0	0	0	1

[a] Valve open = 1, closed = 0

the emergency shutdown of the two tanks by draining them (Table 7.1). We will assume the operation of these valves is slaved to the sd input event. The actuator control vector is adjusted accordingly, to allow for the added control valve, V_3.

$$\begin{aligned} V_1, V_2, V_3, P &\in \{0,1\} \\ u_c &= [V_1, V_2, V_3, P]^T \end{aligned}$$

The continuous dynamics of the tanks are modeled by the following differential equation:

$$\dot{h} = \begin{bmatrix} \frac{dh_1}{dt} \\ \frac{dh_2}{dt} \end{bmatrix} = \begin{bmatrix} q_{mi} & -\frac{\sqrt{\rho g h}}{\rho A R_{t2}} & 0 & -\frac{\sqrt{\rho g h}}{\rho A R_{tP}} \\ 0 & \frac{\sqrt{\rho g h}}{\rho A R_{t2}} & -\frac{\sqrt{\rho g h}}{\rho A R_{t3}} & -\frac{\sqrt{\rho g h}}{\rho A R_{tP}} \end{bmatrix} u_c \tag{7.1}$$

where R_{t2}, R_{t3}, R_{tP} are the turbulent resistances of valves V_2, V_3 and P respectively. Additional continuous dynamics require additional functionals and output events; these are summarized in Table 7.1.

There are nine sets of possible continuous dynamics once the possible control vectors u_c specified by Table 7.1 and the dynamics of Eq. 7.1 have been combined. As before, each actuator setting, along with a set of state partitioning functionals, forms a separate CSM which will be embedded in the switched continuous model.

Table 7.2: Output events, with associated functionals and hypersurface crossing directions for the two tank control synthesis problem.

σ_{out}	Functional	Zero-crossing	Alarm
ovf1	$F_1(h) = h_1 - 33$	$\uparrow$	over filled (tank 1)
hi1	$F_2(h) = h_1 - 31$	$\uparrow$	high
med1	$F_3(h) = h_1 - 18$	$\downarrow$	medium
unf1	$F_4(h) = h_1 - 15$	$\downarrow$	under filled
ovf2	$F_5(h) = h_2 - 33$	$\uparrow$	over filled (tank 2)
hi2	$F_6(h) = h_2 - 31$	$\uparrow$	high
med2	$F_7(h) = h_2 - 27$	$\downarrow$	medium
unf2	$F_8(h) = h_2 - 20$	$\downarrow$	under filled
esd	$F_9(h) = (h_1 - 0.5) \wedge (h_2 - 0.5)$	$\downarrow$	emergency shutdown

The CSMs corresponding to input events i_1 to i_8 share the same set of functionals $\Psi_1 = \{F_1, F_2, F_3, F_4, F_5, F_6, F_7, F_8\}$ and the CSM for emergency shutdown operation (both purge valves P open) has $\Psi_2 = \{F_9\}$. The functionals are defined in Table 7.2 along with their associated output events. In this case, the *esd* output event is signaled when both tanks are drained ($h_1, h_2 \leq 0.5$).

The specification for this system will be similar to the single-tank system; we wish both tanks to cycle between an upper (*hi* event) and lower limit (*med* event). The fill/drain cycle timing is specified in coarse (tick) time $2\Delta t \leq t \leq 4\Delta t$, where $\Delta t = 90$ seconds.

In Fig. 7-8 the finite state models are given for this specification. The HySynth commands to build the model are as follows:

```
s1 = fsm('spec1.xml');          %load the tank 1 specification
s2 = fsm('spec2.xml');          %load the tank 2 specification
spec = product(s1,s2);          %create the spec product object
plant = scmodel('tank.xml');    %two tanks SC model
esd = fsm('esd.xml');           % esd specification
p1 = product(plant,spec,esd); % Create the controller model
```

The result of these commands is the product model structure of Fig. 7-9. A

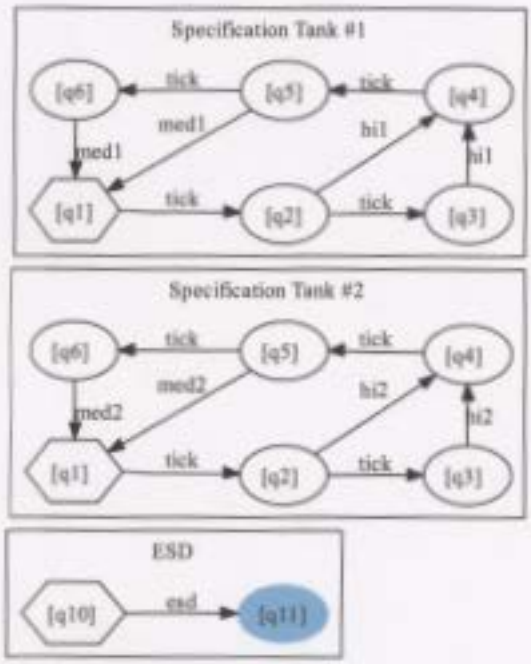


Figure 7-8: The specification models for the two tank level controller.

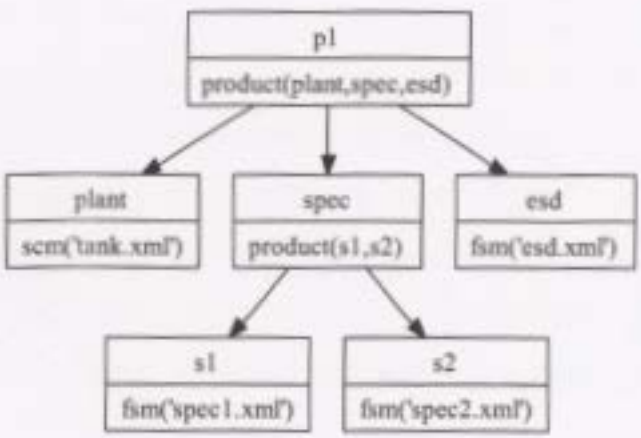


Figure 7-9: The hierarchical model structure for the two tank controller synthesis.

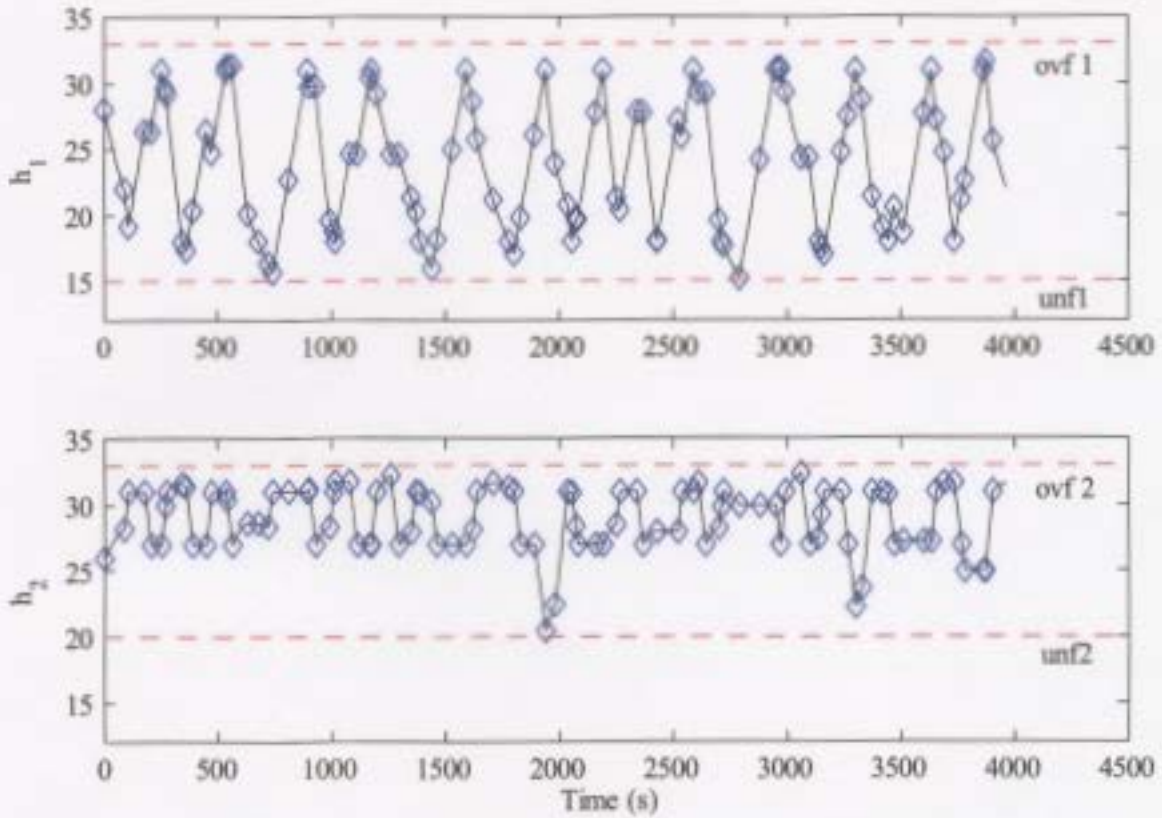


Figure 7-10: Simulation of the online controller using a 5-event lookahead horizon and random control choice.

controller synthesis based on an event horizon reach of 5 events was used for the simulation. The controller advances using a random selection of control actions from the available legal subset of input events (as determined by the online controller) at each of 100 choice points. The resulting control performance is pictured in Fig. 7-10, in which the state variables, h_1 and h_2 have been plotted versus time. The top trace is h_1, the level of tank 1 and the lower trace is h_2, the fluid level in tank 2. The choice points, where the controller has acted, are indicated by diamond markers on the traces.

In Fig. 7-11, a section of this simulation has been enlarged with the two traces overlapped to show the time history in more detail, with h_1 the blue trace, and h_2

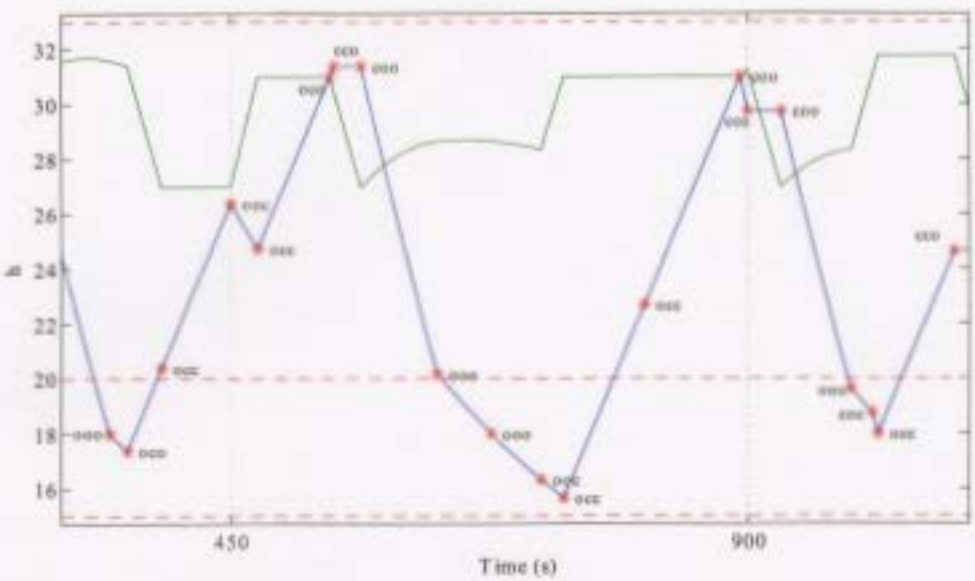


Figure 7-11: Detail of figure 7-10, showing the sequence of actuator (input) events selected by the controller.

the green trace. The controller actions (input events) have been plotted to show the valve settings that were selected at each choice point. By design, this is a fail-safe controller, therefore there is always an emergency shutdown state within five events of the system state, or the controller would not have existed. In addition, the controller does not block because the shutdown procedure is never initiated.

The code used to generate this control sequence is given in Fig. 7-12.

7.1.3 Reducing Controller Size: Two Tanks

In §6.4, it was claimed that a more restrictive specification would reduce controller size, thereby improving the speed with which the controller can be computed. Model size may be determined empirically by offline simulation. In an iterative fashion, a designer may adjust the specification to "tune" the controller size and, of course, the legal or controlled behaviour of the system. To illustrate this idea, we will compare the

```
%
% Program to synthesize a DES controller
% and simulate it for 100 iterations
%
spec1 = fsm('spec1.xml');                          % tank 1 spec
spec2 = fsm('spec2.xml');                          % tank 2 spec
spec1 = addEvents(spec1,['ovf1';'unf1']);          % augment event set
spec1 = markAll(spec1);                            % all states marked
spec2 = addEvents(spec2,['ovf2';'unf2']);
spec2 = markAll(spec2);
spec = product(spec1,spec2);
esd = fsm('esd.xml');
plant = scmodel('tanks.xml');
p = product(plant,spec,esd);
x0 = ctsState([28;26])                             % initial state for sim
t0 = ctsState(0)
c0 = pstate(t0,x0)                                 % time stamp the state
ps = pstate(c0,initial(spec),initial(esd))         % initialize controller
% do 100 controller synthesis/simulations
for i = 1:100
    % synthesize the controller
    [flag,ns,gSize] = printAsDotWithEvents(p,ps,7,filename);
    if ~flag
        disp('Controller Block')
        return(false)                          % controller empty, quit
    else
       % Choose controller action here
        ps = selectNextState(ns);               % pick random next state
        ev = lookupInputEvent(plant,ps(i),ps)   % get the control event
    end
end
return (flag)
```

Figure 7-12: MATLAB code used to synthesize 100 controllers for the 2 tank system.

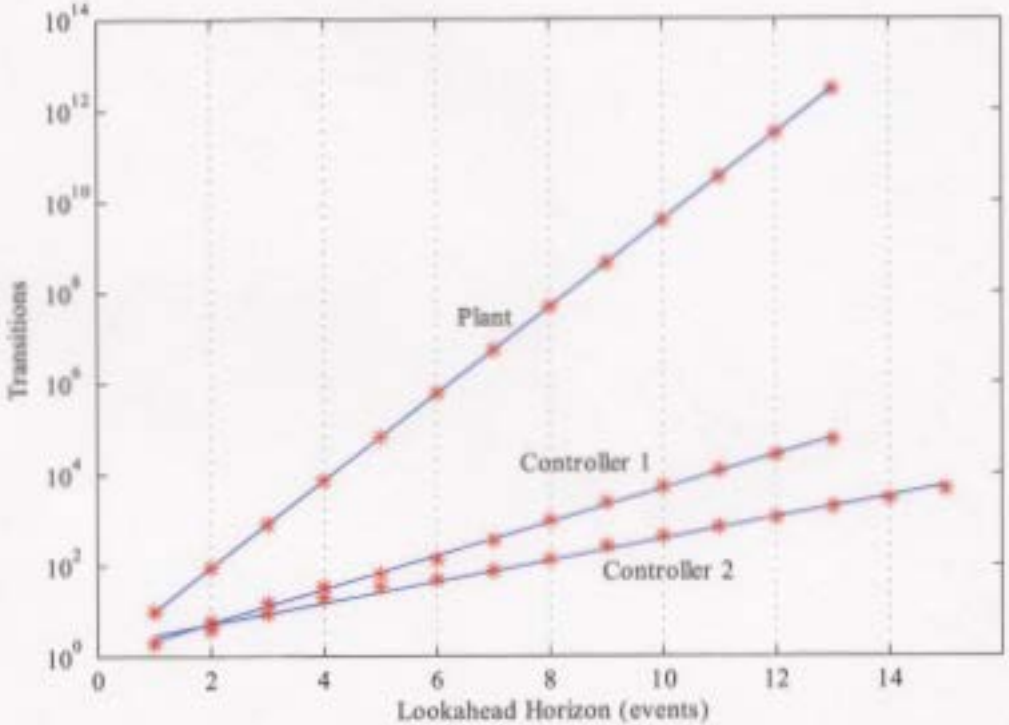


Figure 7-14: Comparative size as a function of events for the plant, Controller 1 (see specification of Figure 7-8), and of Controller 2 (see specification of Figure 7-13).

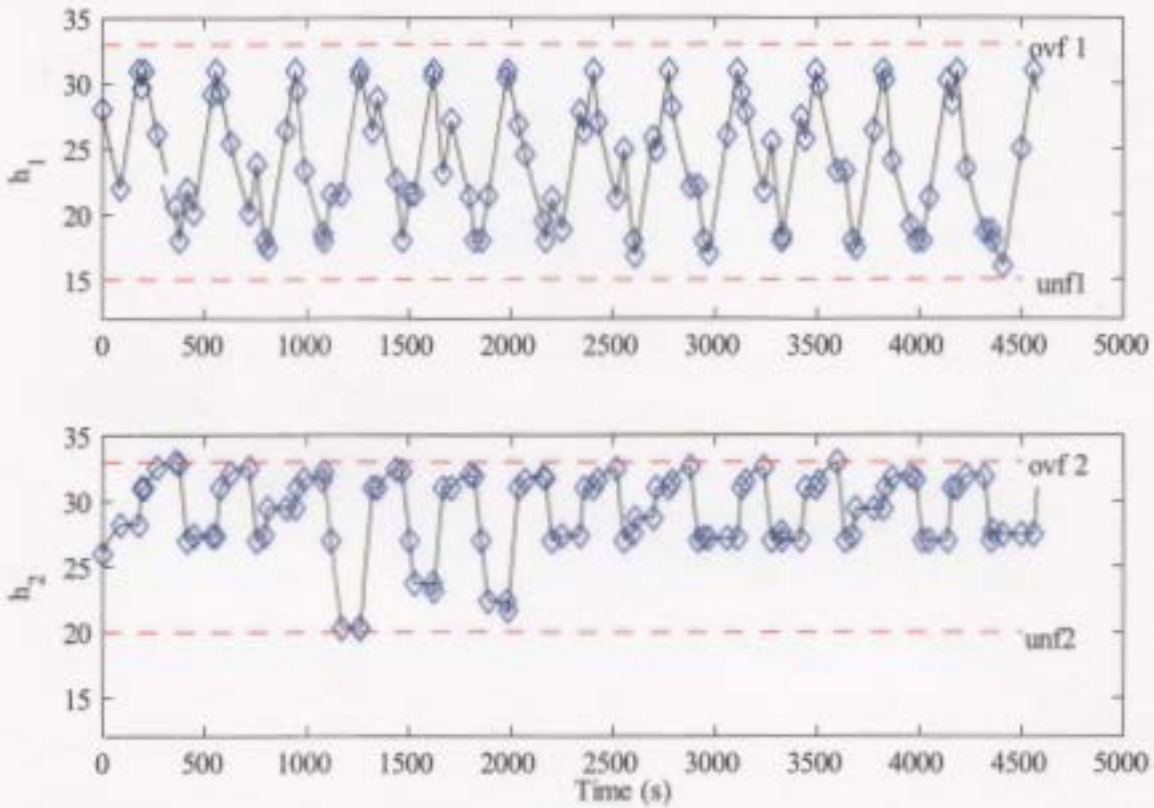


Figure 7-15: Simulation of the online controller using a 7-event lookahead horizon, random control choice, and the restrictive specification of Figure 7-13.

the fact that the controller choice mechanism is random.

7.2 Manoeuvring of a DP Vessel

In this section, we will introduce a control application that requires more complex embedded continuous dynamics than the previous examples. This application also is used to demonstrate a controller with a human choice mechanism: an example of human-in-the-loop control.

Dynamic positioning (DP) is defined in the marine engineering community as the automatic control of a vessel's position and heading using it's thrusters. The DP control system may also be used in combination with the vessel's rudders, and passive restraints such as a mooring system. Typically, DP systems are installed on vessels that need to automatically maintain station for long periods of time in a variety of weather conditions. For a general, non-mathematical treatment of the subject, the reader is referred to (Morgan 1978) and (Hancox 2001). For a mathematically rigorous version, the reader is referred to (Fossen 1994). Almost all theoretical study of DP control has been devoted to various types of continuous control strategies, particularly optimal control. These techniques have been refined and in practical use for many years. As in many other industrial applications, the challenge for the "next generation" of ship control systems is the integration of DP control with other shipboard control functions, such as power management, which require logic and appropriate sequencing (Weingarth 2002), (Millan, Smith and O'Young 2002). Currently, such functions are served by highly skilled operators.

A challenging control problem is the FPSO and shuttle tanker offloading application. In many areas of the world, the oil from subsea oil fields is pumped and stored by a specialized vessel known as a floating production storage and offloading vessel (FPSO). The FPSO is usually moored over a manifold on the sea floor from which the oil is pumped. Risers (large flexible transfer hoses) carry the oil from the subsea

Figure 7-16: Terra Nova FPSO offloading oil to a shuttle tanker. The FPSO is also flaring excess gas.(Photo courtesy of Petro-Canada)

manifold to the FPSO, often entering through a swiveling manifold system on the underside of the vessel. Finally, the oil is transferred at sea from the FPSO to a shuttle tanker, which takes up station in tandem, at the FPSO stern. The task of oil transfer at sea is complex and dangerous due to the close proximity of the two vessels. There is a risk of collision if they get too close to each other, or of transfer hose breakage and an oil spill if they drift too far apart. The shuttle tanker may use some sort of passive restraint system (a rope called a hawser line) as a backup, but no tension is applied to it. Thus, the shuttle tanker must maintain station behind the FPSO using only its propulsion system which is controlled by the DP controller. Fig. 7-16 is an example of a FPSO and shuttle tanker offloading operation.

Occasionally, the FPSO may have to turn in order to realign itself if the prevailing environmental conditions change direction. Since the FPSO rotates about the swiveling manifold on the hull, the shuttle tanker must also swing, but through a greater arc. This is known as a weathervaning manoeuvre, and requires coordination and care by the operators of both vessels. The most important factor influencing the ability of these vessels to carry out their operations is the power system.

We will now examine in detail a control example in which we model the shuttle

tanker operation and synthesize a control system for a weathervaning manoeuvre.

7.2.1 Vessel Power System

On most modern vessels, the propulsion system is powered from an electrical generation system. As a result, the performance of the power system directly affects the propulsion, and thus the ability of the DP system to maintain station. For this reason, the DP control system is often integrated with the power generation system so that these systems may be coordinated. For the sake of this example, we will assume a power generation system having 2 main generators ($MG1$ and $MG2$) and a propulsion system with 4 steerable propulsion units, $T1 - T4$ (called azimuthing thrusters). The azimuthing thruster units are designed so that they can be turned to direct thrust in the appropriate direction relative to the vessel. In Fig. 7-17 the electrical schematic for the power distribution and propulsion systems is depicted for a hypothetical DP vessel. Normally, the two main generators with rated capacity of 15 Megawatt (MW) supply the main propulsion bus via the transformers TR1 and TR2. Switchgear at S1 and S2 enable the generators to be taken off line. A backup generator, designed for so-called "hotel" load (i.e. lighting, domestic loads) can be placed on the propulsion bus in event of emergency via switch S3. The azimuthing thrusters, are supplied by thyristor drives SCR $1 - 4$, which control the propellor speed. Having azimuthing capability, the thrusters can be rotated continuously through 360 degrees to direct their thrust in the most appropriate direction.

For this example, we will assume that the main generators are running and that switching of generators onto the bus is instantaneous. While it is possible to model the power system dynamics, they will be neglected for this example.

7.2.2 Vessel Manoeuvring Model

For purposes of manoeuvring control, a vessel model can be limited to 3 degrees of freedom (DOF), since we are only interested in controlling yaw angle, surge displace-

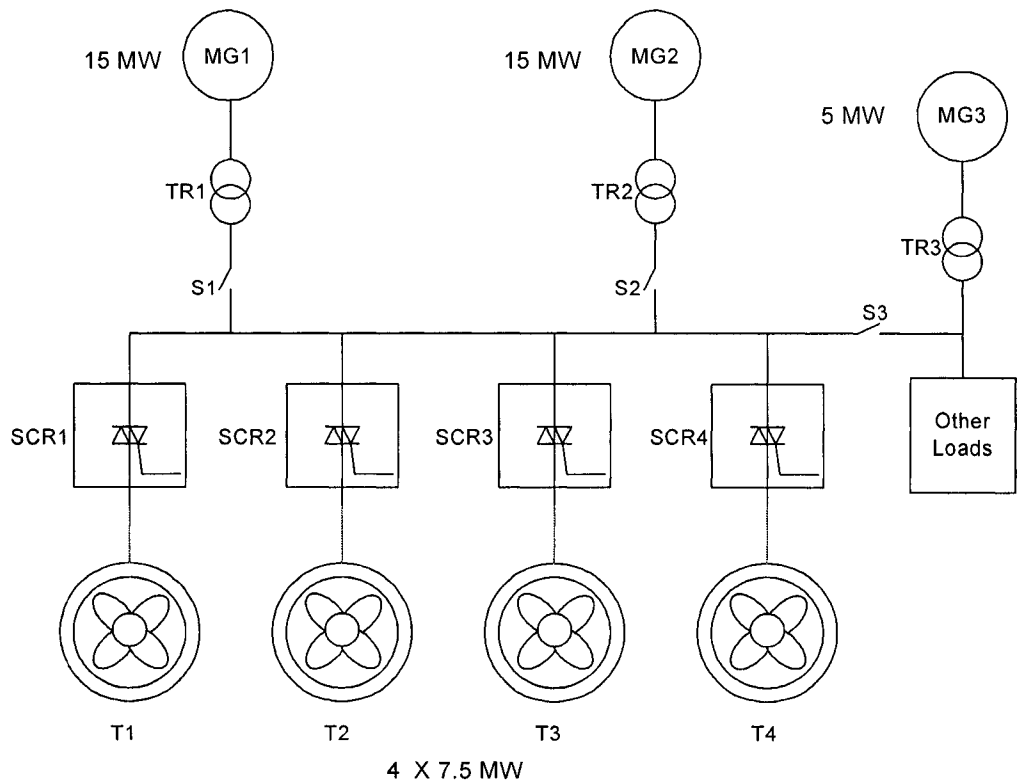


Figure 7-17: The power distribution and propulsion load schematic for a hypothetical vessel.

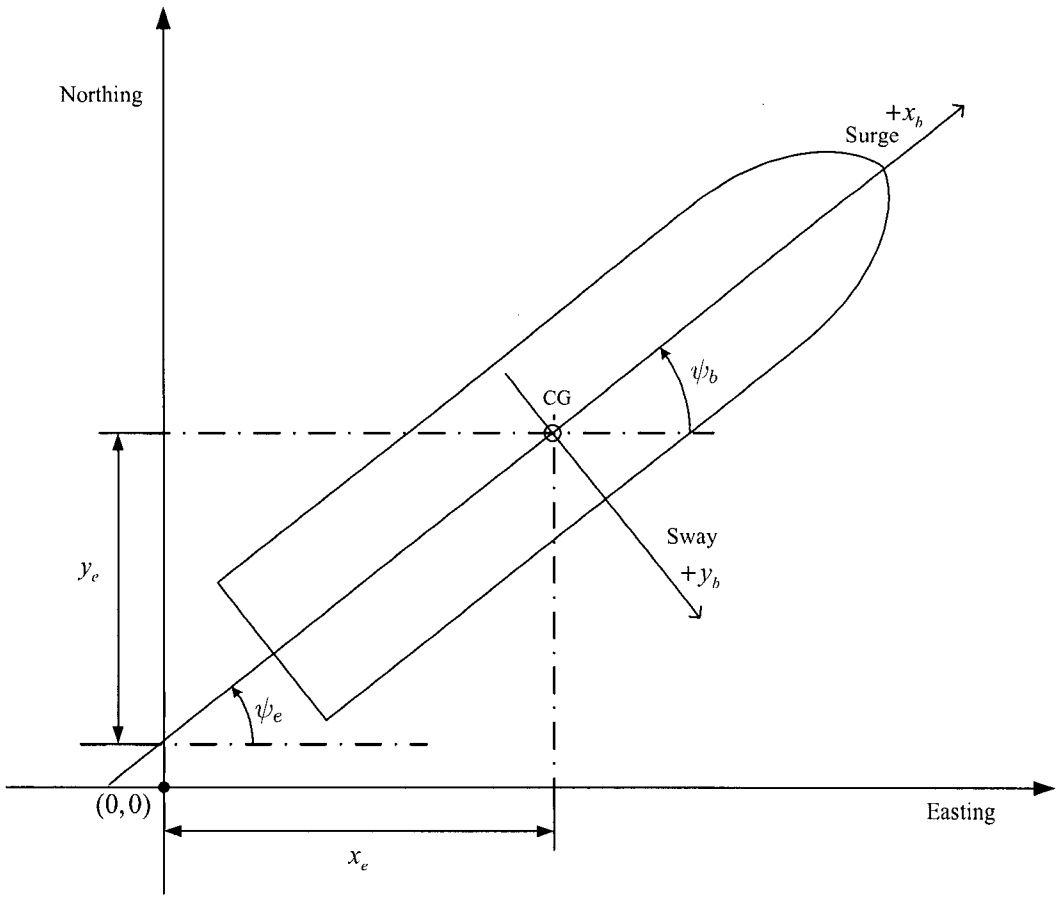


Figure 7-18: Vessel cartesian coordinate systems. Body coordinates are denoted by subscript b and earth coordinates by subscript e.

ment and sway displacement. Roll and pitch angles and heave (vertical) displacement cannot be controlled and so are unnecessary to model. The rotation of the vessel (about its centre of gravity) within the plane is North-referenced and is called heading, denoted ψ_e. The rotational component in the body reference frame ψ_b, is the same as heading, but to distinguish it from the absolute coordinate, it is called yaw. In general, the subscript e is used to denote earth-referenced (inertial) coordinates and the body referenced coordinate frame is denoted by subscript b. Let the vector $\mathbf{x}_e$ represent the earth-referenced 3 DOF position vector of the vessel and $\mathbf{x}_b$ denotes the position 3 DOF vector in the body frame:

$$\begin{aligned} \mathbf{x} &= \begin{bmatrix} x_b & y_b & \psi_b \end{bmatrix} \\ \mathbf{x}_e &= \begin{bmatrix} x_e & y_e & \psi_e \end{bmatrix} \end{aligned}$$

Since heading angle and yaw are equivalent, we will use ψ as the default rotation about the center of gravity (CG) of the vessel. The coordinate transformation $J(\psi)$ takes the earth referenced measurements into the body frame of reference:

$$\mathbf{x} = J(\psi)\mathbf{x}_e$$

$$\begin{bmatrix} x_b \\ y_b \\ \psi \end{bmatrix} = \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ \sin(\psi) & -\cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ \psi \end{bmatrix}$$

We define the velocity and acceleration vectors accordingly:

$$\begin{aligned} \mathbf{v} &= \frac{d}{dt}\mathbf{x} = \begin{bmatrix} u & v & \phi \end{bmatrix} \\ \dot{\mathbf{v}} &= \frac{d}{dt}\mathbf{v} = \begin{bmatrix} \dot{u} & \dot{v} & \dot{\phi} \end{bmatrix} \end{aligned}$$

The simplified (linear) dynamics of a freely-floating (i.e. unmoored) surface vessel can then be characterized by the following vector differential equation:

$$M\dot{\mathbf{v}}+D\mathbf{v}=\boldsymbol{\tau} \tag{7.2}$$

$\boldsymbol{\tau}$ is the force and moment vector acting upon the vessel that arises from the sum of the control forces, $\boldsymbol{\tau}_c$ and the environmental forces (current, waves and wind), $\boldsymbol{\tau}_e$, each defined in the inertial frame.

$$\boldsymbol{\tau} = \boldsymbol{\tau}_c + \boldsymbol{\tau}_e$$

M is a positive definite matrix ($M = M^T$) containing the inertial and hydrodynamic added mass terms for the vessel as follows:

$$M = \begin{bmatrix} m + X_{\dot{u}} & 0 & 0 \\ 0 & m + Y_{\dot{v}} & mx_G + Y_{\dot{r}} \\ 0 & mx_G + Y_{\dot{r}} & I_{zz} + N_{\dot{r}} \end{bmatrix}$$

where m is the vessel's mass, I_{zz} is the yaw moment of inertia, $X_{\dot{u}}$, $Y_{\dot{v}}$, are the hydrodynamic added mass in the surge axis, sway axes respectively and $N_{\dot{r}}$ is the added moment of inertia in the yaw axis. The off-diagonal terms are symmetrical (this follows, since the vessel is symmetrical about both the surge and sway axes), and feature a hydrodynamic added mass term $Y_{\dot{r}}$ due to the cross-coupling between the sway and yaw axes. mx_G is present when the vessel's control point (CP)[1] is not the same as the center of mass (CG) of the vessel. The longitudinal distance between CP and CG is x_G is (Fig. 7-18).

[1]The control point is the point in the body coordinate frame which is positioned by the DP controller.

Table 7.3: The FPSO vessel particulars.

Vessel Particular	Full Scale
Length Overall (LOA)	290 m
Displacement, ∇	193,000 m^3
Mass	197,632 tonnes
Yaw Radius of Gyration	57 m
Beam	45 m
Longitudinal CG	145 m

D is a matrix containing linear hydrodynamic damping terms:

$$D = \begin{bmatrix} X_u & 0 & 0 \\ 0 & Y_v & Y_r \\ 0 & N_v & N_r \end{bmatrix}$$

the diagonal terms X_u, Y_v and N_r, are the surge sway and yaw damping. The off-diagonal terms Y_r and N_v are respectively, the sway-yaw and yaw-sway damping terms.

Assumptions that have been made to simplify this model are that centripetal and Coriolis forces are negligible because yaw rates are relatively small, and hydrodynamic added mass and damping are constant. Since most DP vessels are designed for stationkeeping, and vessel velocities are low, this assumption is also reasonable. The hydrodynamic added mass and damping can be determined by specialized software, estimated from existing ships for which these parameters are already known, or determined empirically by model testing or full-scale ship trials. The detailed vessel particulars for the vessel that will be simulated are given in Table 7.3. For the simulation, we will use non-dimensional quantities for convenience. The so called *bis* system ((Fossen 1994), p. 94) is a convenient system for low-speed manoeuvring models, since it is not based on vessel forward speed as other systems are. The bis system non-dimensional scaling factors for a surface vessel are given in Table 7.4. Typically, the vessel LOA is used for the length factor L, g is the acceleration due to gravity, ρ

Table 7.4: Nondimensional scaling factors.

Quantity	Scale Factor
Mass	$\rho\nabla$
Length	L
Linear Velocity	$\sqrt{gL}$
Angular velocity	$\sqrt{\frac{g}{L}}$
Force	$\rho g\nabla$
Moment	$\rho g\nabla L$
Time	$\sqrt{\frac{L}{g}}$

is the density of sea water (1025 kg/m^3). As an example, the non-dimensional mass of the vessel at full displacement is $m = 1$.

7.2.3 Closed Loop Control

For this example, the discrete event controller will supervise a closed loop continuous controller (i.e. the DP control system). Therefore, the switched continuous model will be developed around CSMs that model the closed-loop dynamics of the vessel. Figure 7-19 is a block diagram of a typical DP control system. The system is commanded with a 3 DOF setpoint command in earth referenced coordinates. The vessel's 3 DOF position $\mathbf{x}_e$ is measured with a variety of sensors and passed through a state estimator[2]. The error signal is converted to body coordinates, and control gains are applied to determine a controller demand. Measurements of the wind speed and direction are used to calculate a feedforward wind load, which is summed to the controller demand. A thruster allocation block determines how this controller $\boldsymbol{\tau}_c$ demand will be divided amongst the available thrusters, taking the geometry of their hull arrangement into account. Not pictured in the figure is the optimal state estimation current and wave generated forces and moment; these are summed into the controller demand.

[2]Typically, a Kalman filter is used to remove sensor noise.

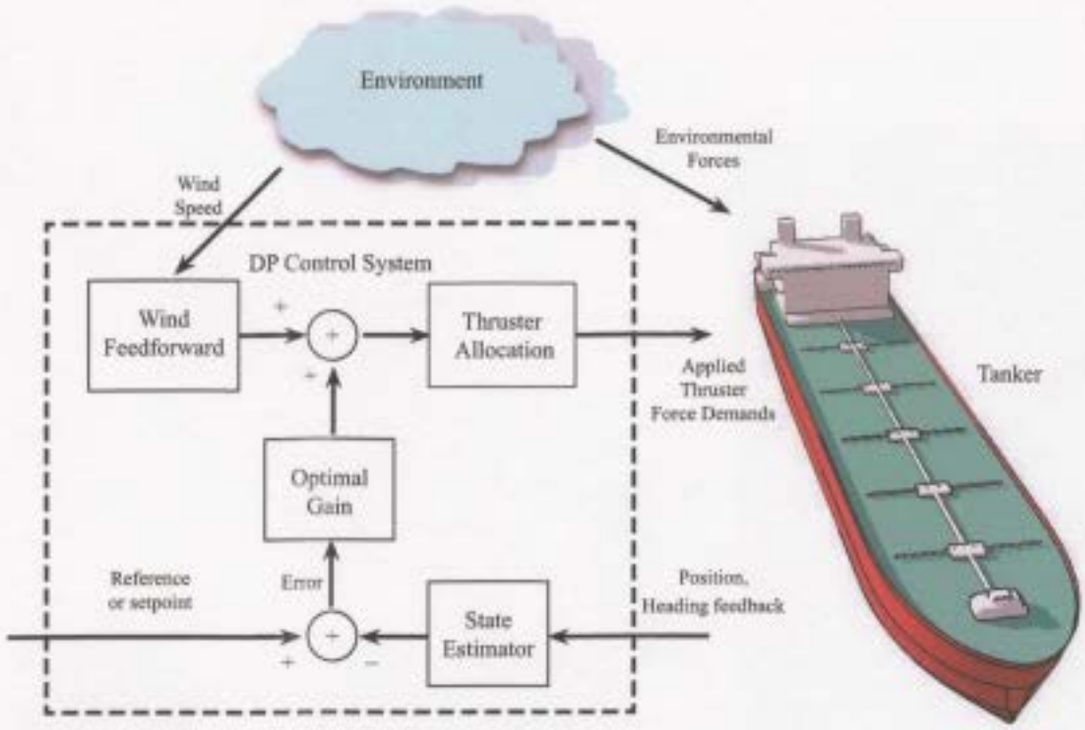


Figure 7-19: Block diagram of DP control system.

7.2.4 Thruster Allocation

Azimuthing thrusters are ideal actuators since they can be turned to direct the necessary force in any desired direction. In Fig. 7-20 the vessel thruster arrangement is pictured. The relation between the control demand and the individual actuator demands is as follows

$$\tau_c = T_a T_{th}$$

where T_{th} is a vector of thruster demands in Cartesian coordinates, and T_a is the thruster allocation matrix, defined as follows

$$T_{th} = \begin{bmatrix} T_{1x} & T_{1y} & \dots & T_{4x} & T_{4y} \end{bmatrix}^T$$

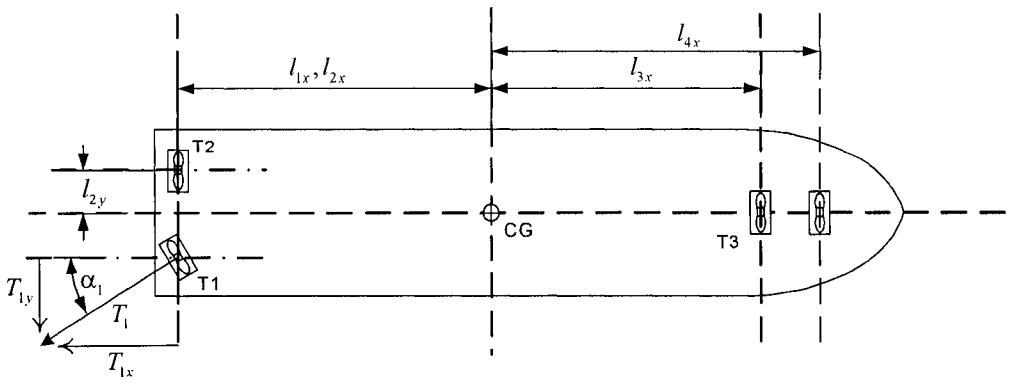


Figure 7-20: Vessel thruster arrangement and coordinate reference frame.

and

$$T_a = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ l_{1y} & l_{1x} & l_{2y} & l_{2x} & l_{3y} & l_{3x} & l_{4y} & l_{4x} \end{bmatrix} \tag{7.3}$$

In 7.3 matrix entries of 1 indicate that 100% is available from the thruster if it is rotated to the appropriate direction. The bottom row are the lever arm distances that generate moment about the CG. From Fig. 7-20, the lever arms are $l_{1y} = l_{2y}; l_{1x} = l_{2x}; l_{3y} = l_{4y} = 0$, and filling in the values with non-dimensional units

$$T_a = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0.1 & 0.45 & -0.1 & 0.45 & 0 & -0.4 & 0 & -0.45 \end{bmatrix}$$

Solving for the unknown T_{th} requires finding the Moore Penrose generalized inverse of T_a

$$T_{th} = T_a^{\dagger} \boldsymbol{\tau}_c$$

where $T_a^{\dagger}$ is the generalized inverse of T_a. Thrust vector T_{th} can be converted from Cartesian coordinates to an azimuth angle command and thrust demand pair $\begin{bmatrix} \alpha & T \end{bmatrix}^T$

Table 7.5: Example vessel thrust limits as a function of power system configuration.

Electrical Configuration	Total Thrust	Thruster Saturation
One Main Generator, 15 MW	3 MN	750 kN
Both Main Generators, 30 MW	6 MN	1.5 MN
Standby Generator 5 MW	1 MN	250 kN

for each thruster as follows

$$T_{th} = \begin{bmatrix} \alpha_1 & T_1 & \dots & \alpha_4 & T_4 \end{bmatrix}^T$$

The thrusts are minimal in a least-squares sense, but may exceed the thrust limit for the actuator. In a real thruster, the maximum thrust is dependent on many factors, including the speed of the thruster through the water and the proximity and wake direction of other thrusters. In this simulation, the thrusts T_{1-4} will simply be clamped at a saturation limit which will be determined by the electrical bus power available. For this simulation, the thrust/electrical power relation is summarized in Table 7.5.

The various modeling details given in these sections provide for a reasonable ship simulation model. In the next section a DES supervisor for the tanker will be developed.

7.2.5 Supervisory Controller Design

We will design a supervisory controller for an FPSO tanker offloading system. The coordinate frame and general arrangement of the vessels and safe operating areas is detailed in Fig. 7-22. Since the FPSO is attached to a mooring and rotates about this point, it is convenient for the supervisory controller to command the tanker and FPSO in a rotating coordinate frame We now redefine the earth-referenced Cartesian coordinate frame to a polar coordinate frame centered at the FPSO mooring point where r is the radial distance of a vessel (CG) from the origin, and $\Pi \leq \theta \leq \Pi$

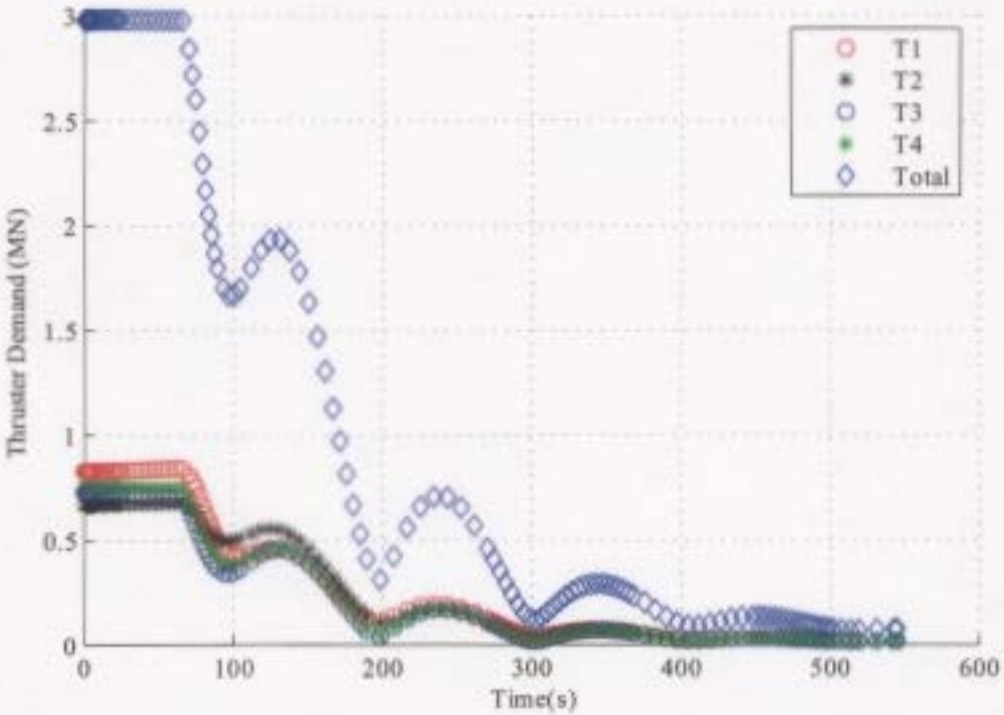


Figure 7-21: Individual thrusts for a step maneouver in yaw, using one generator.

is the angle of rotation, while ψ is the vessel heading, unchanged from the other coordinate frames. All ranges r in this diagram are nondimensional. The scenario we are designing a controller for is a weathervaning manoeuvre that only the tanker carries out. When the flare is lit on the FPSO, the forward deck temperature of the tanker can rise to dangerous levels. the written operating procedures (Allan 1999) require that the operator of the FPSO contact the tanker and request that it move in order to minimize the deck heating. Since the flare is on the starboard side of the FPSO, movement of the tanker slightly to the port side of the FPSO's stern has the desired effect. during this move, the appropriate separation between the vessels must still be maintained to prevent collision or hose breakage. In Fig. 7-22 the green zone is the normal safe area in absence of a flare. The red-coloured zone is a "keep-out" area due to subsea risers that may be damaged by the tanker's thrusters. The blue area to the port side of the FPSO is the safe area while the flare is operating. We will design a controller to enforce safe operation during a flare event.

7.2.5.1 Partitions and Output Events

We begin the modeling process by defining the partitioning functionals for the system SCM in Table 7.6. In this case, we use a partial state vector

$$x = \begin{bmatrix} r & \theta & \psi & t \end{bmatrix}$$

Note that we are using the polar coordinate frame with origin at the mooring point of the FPSO. Events tfp and tfs signal when the vessel longitudinal axis is out of alignment with the coordinate frame; the goal is to keep the bow of the tanker aimed towards the FPSO at all times[3]. The esd emergency shutdown is assumed to be achieved once the vessel has safely reached a radial distance $r_{sd} \geq 1.85$.

[3] Alternatively, the origin of the polar coordinate frame could be placed at the stern of the FPSO.

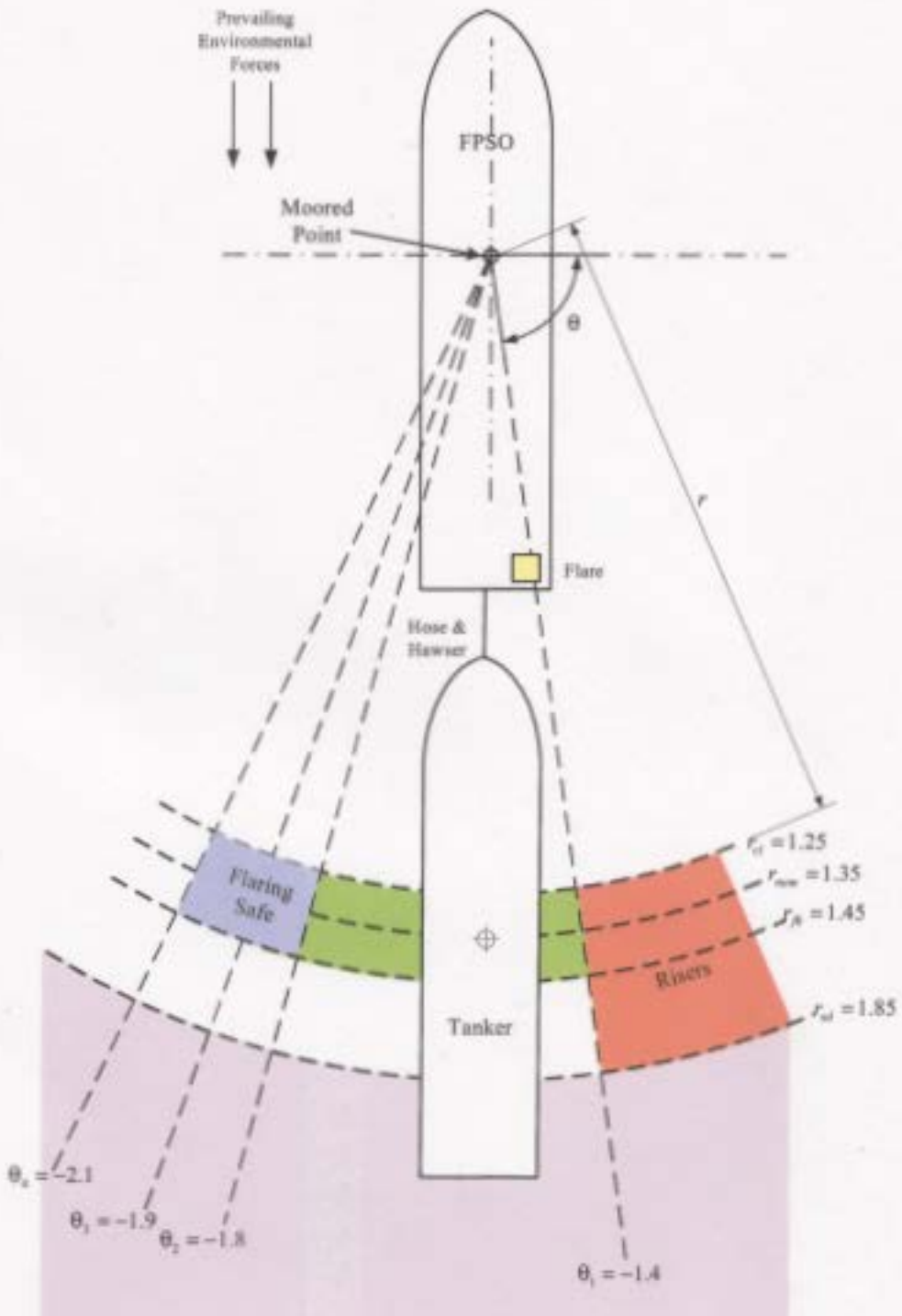


Figure 7-22: The FPSO and tanker offloading system.

Table 7.6: Output events, with associated functionals and hypersurface crossing directions for the DP vessel control synthesis problem.

σ_{out}	Functional	Zero-crossing	Alarm
tcl	$F_1(x) = r - 1.25$	$\downarrow$	too close to FPSO
tfb	$F_2(x) = r - 1.45$	$\uparrow$	too far from FPSO
o3	$F_3(x) = \theta + 1.4$	$\uparrow$	riser area guard
o4	$F_4(x) = \theta + 1.9$	$\downarrow$	enter flare safe area from green
o5	$F_4(x) = \theta + 2.1$	$\downarrow$	cw exit flare safety area
o6	$F_4(x) = \theta + 1.8$	$\uparrow$	ccw exit flare safety area
tfp	$F_7(x) = \pi - (\psi - \theta) - 0.2$	$\uparrow$	misalignment to port
tfs	$F_8(x) = \pi - (\psi - \theta) + 0.2$	$\downarrow$	misalignment to starboard
esd	$F_9(x) = r - 1.85$	$\uparrow$	emergency shutdown, fall back
tick	$F_{10}(x) = \sin(2\pi t/\Delta t)$	$\downarrow$	controller update

Table 7.7: Control actions available to the DES supervisor. Controls are specified as setpoint jog commands to the DP controller, and are in non-dimensional units and the FPSO polar coordinate reference system.

	Controls						
σ_{in}	r_{jog}	θ_{jog}	ψ_{jog}	g			Description
α_1+	0	0.1	0	1	0	0	jog cw with one generator
α_1-	0	−0.1	1	1	0	0	jog ccw with one generator
α_2+	0	0.15	0	1	1	0	jog cw with two generators
α_2-	0	−0.15	1	1	1	0	jog ccw with two generators
fwd	1	−0.1	0	1	0	0	move ahead with one generator
back	1	0.1	1	1	0	0	move astern with one generator
hold	0	0	0	1	0	0	hold station with one generator
sd	1.85†	‡	‡	1	0	1	shutdown on emergency power

† in absolute coordinates; ‡ indicates a don't care input

7.2.5.2 Controller Actions

The control actions available to the controller for this model are listed in Table 7.7 and are associated with the corresponding input event labels $\sigma_{in} \in \Sigma_{in}$. The controls are the commands that will be sent to the DP control system. The controls r_j, θ_j and ψ_j are "jog" commands which are summed with the current state of the system

to develop an absolute setpoint for the DP controller.

$$\begin{aligned} r_{sp} &= r + r_{jog} \\ \theta_{sp} &= \theta + \theta_{jog} \\ \psi_{sp} &= \psi + \psi_{jog} \end{aligned}$$

The control indicated by $g = \begin{bmatrix} S1 & S2 & S3 \end{bmatrix} \in \{0, 1\}$ is a vector corresponding to the generator switchgear of Fig. 7-17. Within the controller simulation, the effect of this switchgear control input is that it sets the saturation limit of the thrusters as per Table 7.5.

7.2.5.3 Modeled Environmental Load

In heavy environmental loading, it is necessary to align a vessel with the prevailing direction of this load in order to minimize the thruster effort required to stay on station, and in the case of the moored vessel, it reduces the vessel motion. For our scenario, we are assuming that the FPSO is aligned with this environmental load. When the tanker moves around to avoid the flare, it encounters a load that progressively increases in proportion to the misalignment of the vessel with the load vector. As the tanker moves from out of the "shadow" of the FPSO and its beam is exposed to the waves and wind, this will tend to drag the vessel off station. We will assume that this load is a modeled effect. Thus, this force is predictable and it can be embedded in the continuous system models.

7.2.5.4 Specifications

For the first example, we will direct the vessel to rotate from its initial position, with a heading of $\psi = \pi/2$ and positioned directly behind the FPSO in the green zone (Fig. 7-22). The assumption is that the FPSO has communicated to the tanker that it will commence flaring gas so the tanker must move to the flare safe area.

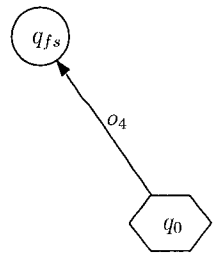


Figure 7-23: Inadequate specification with no timing.

Typically an offshore marine operation like this has a set of written procedures for the vessel operators to follow; an example of this is the Terra Nova FPSO/Tanker Joint Operations Manual (JOM) (Allan 1999). These procedures contain detailed written descriptions of various activities that involve both vessels, and contained within this manual is a specification of the maximum flare dwell time and the safe area for the tanker. Essentially we wish to take the descriptive procedure and encode it as a specification.

There are a variety of approaches to designing an effective specification, and thus a corresponding online controller. With no knowledge of the system, the least restrictive specification may be appropriate as a starting point; starting with a very restrictive specification may lead to a non-existent controller. The least restrictive specification for this system is to request an $o4$ event occur (Fig. 7-23). Adding the following events to event set as follows,

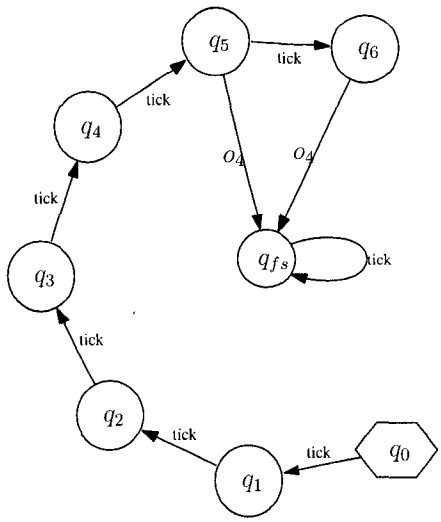
$$\Sigma = \{o4\} \cup \{tcl, tfb, o3, o5, o6, tfs, tfp\}$$

effectively prohibits the included events from occurring. This specification enforces safety, but there is no guarantee that the controller will find its way to the flare safe area because there is no time explicitly mentioned (the *tick* event is not included).

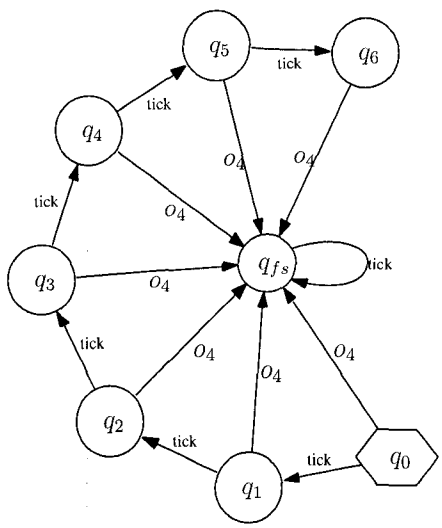
Effectively this means that the system has been commanded to reach the safe area but nothing has been said about the timing of this activity. This specification is unsuitable since there is no upper bound on the time that the vessel can remain in the green zone while the flare is operating. An additional problem with this specification is that the controller synthesized with this approach will have to use a low time or event lookahead to avoid the exponential growth in controller size.

A different approach is to choose specification that includes some specific timing information. The designer must specify to a greater or lesser degree the time at which the flare safe zone will be entered, since if the vessel remains too long in the vicinity of the flare, it will violate the system safety. However, too "tight" a specification can lead to blocking and an unnecessary emergency shutdown. Too "open" a specification, and controller complexity may become a problem. One possible approach is to "calibrate" the trajectory by manually running simulations offline to test the time required to reach safety from a variety of initial conditions. The specification can then be tailored to admit the trajectories with the "calibrated" timing. Such "over-specification" defeats the purpose of this type of online controller synthesis, since the set of control solutions is decided offline. This approach is shown in Fig. 7-24(b). In this specification, an upper limit of 6 events has been specified for the vessel to move to the flare safe area. Some flexibility has been built in by allowing for it to take place sooner in 5 events. Without calibration, it is possible that this specification may eliminate control trajectories that will take the vessel to the flare safe area before this time.

The specification of Fig. 7-24(b) does not specify a lower time limit for the manoeuvre, but effectively is the same as that of Fig. 7-24(a) in terms of complexity. By not specifying the lower limit, we have made no presumption of the vessel initial condition or dynamics. The upper bound on the specification time (6 events) is derived not from the vessel dynamics, but is based on the maximum time the vessel can linger in the green zone while the flare is lit, which is derived directly from the



(a)



(b)

Figure 7-24: (a) An example of overspecification, (b) a better specification.

operational procedures manual (the JOM). This will be the specification used for the controller synthesis and simulation.

7.2.6 Results

The modeling information of the preceding sections was used to develop a switched continuous model. Using HySynth to simulate the closed-loop system, the following test results were obtained for supervisory control of the weathervaning manoeuvre. The tick time used for these simulations was $\Delta t = 100$ (unitless).

7.2.6.1 Control With Random Choice

Running a simulation of the weathervaning manoeuvre using the specifications of Fig. 7-24, generally will result in a shutdown if the event or time lookahead horizon is not large enough to include state q_6 of the specification. The control synthesis has to have large enough lookahead to perceive that state q_6 is blocking if the vessel cannot subsequently reach the flare safe area (state q_{fs}). With an event or time lookahead that is too short, the control system is not able to distinguish between an output string of 5 *ticks* that is moving to safety from a an output string that is simply staying within the green zone. This is clearly illustrated in Table 7.8, in which the controller has a 5 event lookahead combined with a random control choice mechanism. The table is read from left to right: e.g. at step 1, there are a total of $|A| = |A_e + A_{en}| = 6$ control actions available for the operator to choose amongst. The second and third columns are the legal input event subsets, that are eligible as control actions: recall that A_e is the set of control actions that lead only to ESD and A_{en} is the set of events that lead to ESD reachable and nonblocking states.

Starting at Step 1, the controller has 5 priority 1 choices (ESD reachable and non-blocking), so using a random choice mechanism, it actually selects a_2^+ which is driving with both generators in the wrong direction. It continues to do this action in the next step, when suddenly the specification has reached a block in the lookahead; i.e. $o4$

Table 7.8: A summary of the vessel controller simulation, see Figure 7-25.

Step	A_e	A_{en}	Size†	σ_{in}	σ_{out}
1	$\{sdd\}$	$\{\alpha_1^-, \alpha_1^+, \alpha_2^-, \alpha_2^+, hold\}$	4503	α_2^+	*tick*
2	$\{sdd\}$	$\{\alpha_1^-, \alpha_1^+, \alpha_2^-, \alpha_2^+, hold\}$	4378	α_2^+	*tick*
3	$\{sdd, \alpha_1^-, \alpha_1^+, \alpha_2^-, \alpha_2^+, hold\}$	$\emptyset$	1319	α_2^+	*tick*
4	$\{sdd, \alpha_1^-, \alpha_1^+, \alpha_2^-, hold\}$	$\emptyset$	193	α_2^-	*tick*
5	$\{sdd, \alpha_1^-, \alpha_1^+, \alpha_2^-, \alpha_2^+, hold\}$	$\emptyset$	59	α_2^+	*tick*
6	$\{sdd, \alpha_1^-, \alpha_1^+, \alpha_2^-, hold\}$	$\emptyset$	9	α_2^-	*tick*
7	$\{sdd\}$	$\emptyset$	1	*sdd*	*esd*

† in graph transitions

can no longer be synchronized. The available control actions all lead to shutdown, $A_{en} = \emptyset$. Fortunately, by constructing the controller as emergency shutdown reachable, at least the system will be able to shutdown gracefully. The remaining control actions continue to select at random from set A_e and the system shuts down at Step 7. This simulation is pictured in Fig. 7-25, in which the top trace is a plot of the earth-referenced position vector $\mathbf{x}_e$ versus time (all quantities are non-dimensional). The controller actions and the resulting output events have also been placed on the plot. The upper line of text are the output events, while the lower line is the string of controller actions or input events. In Fig. 7-25 the lower trace is a plot of the vessel velocity as a function of time. A plan view of the tanker's movements of Fig. 7-26 aids interpretation of the vessel actions.

7.2.6.2 Human In the Loop Control

The fundamental premise of human in the loop control is that the human, the operator, has some sort of system knowledge which enables him/her to make an "informed" decision. The strength of our control approach is that the controller can assist an operator by removing from the entire set of available control choices the ones that lead unambiguously to unsafe states. Presumably the controller can also consider many more system trajectories than the operator can in the same amount of time. Yet, without a useful choice mechanism as we saw in the previous section, the controller

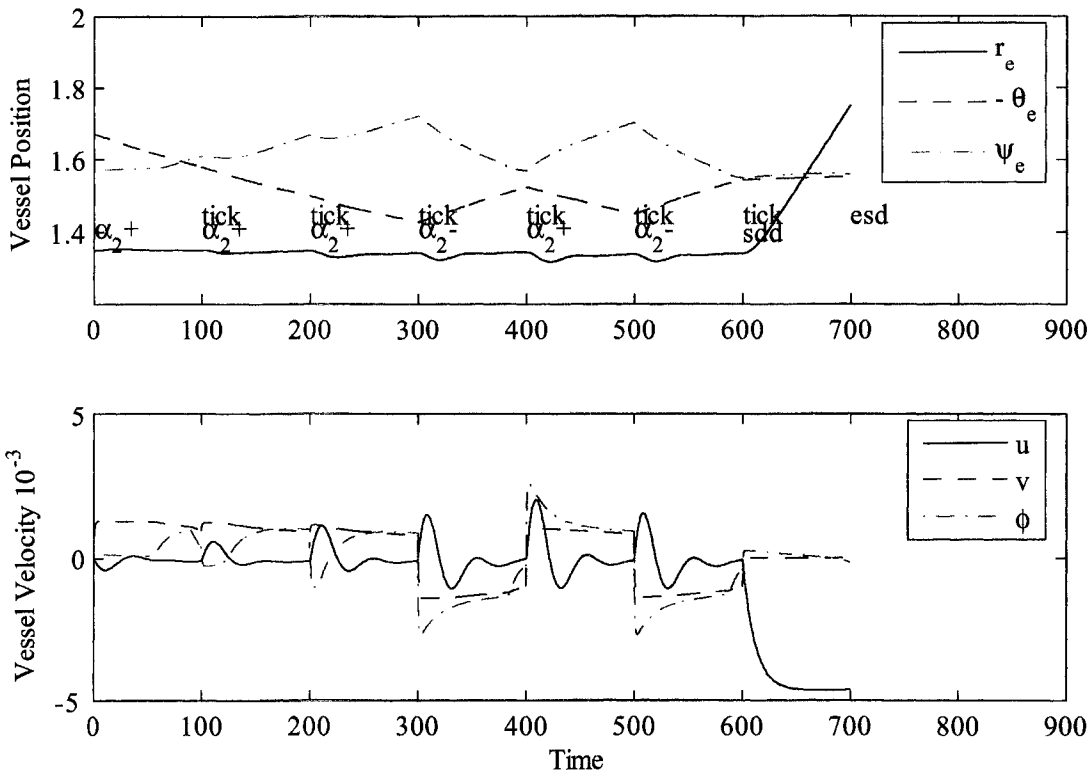


Figure 7-25: Simulation with inadequate event horizon. Using a random choice mechanism, the system goes to ESD.

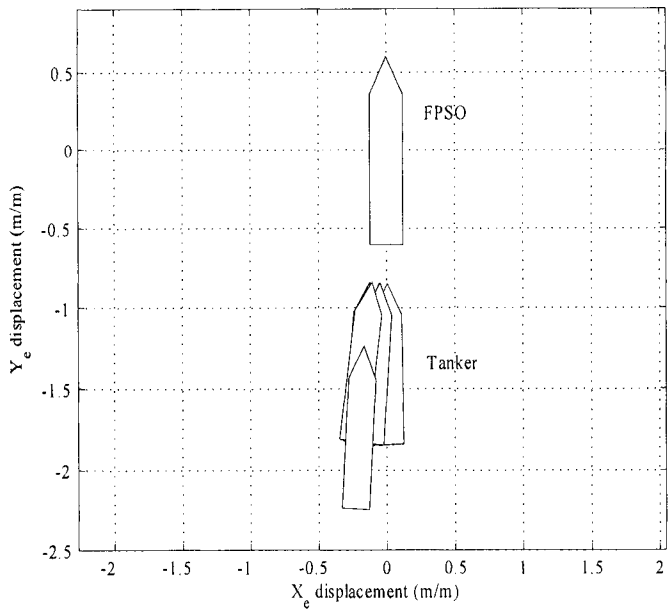


Figure 7-26: An overhead view of the shutdown.

is unable to drive the system to the desired objective unless it is very carefully specified, and/or the event or time lookahead is large enough to encompass enough of the specification to decide unambiguously between blocking or safe outcomes. Fig. 7-27 presents a block diagram of the HIL supervisory control hierarchy that is implemented in this section.

In this example, the specification of Fig. 7-24(b) was used again for control synthesis, but this time with a human operator as the selection mechanism. With an event lookahead of 4 events, this simulation demonstrates that some operator knowledge combined with the DES controller easily outperforms the automated random choice mechanism. The simulation was produced by constructing the ESD reachable controller and giving the operator an opportunity to select the control action at each controller update. The system is advanced through a simulation to the next time step and then the process repeats itself by computing the controller for the new time step, and so on.

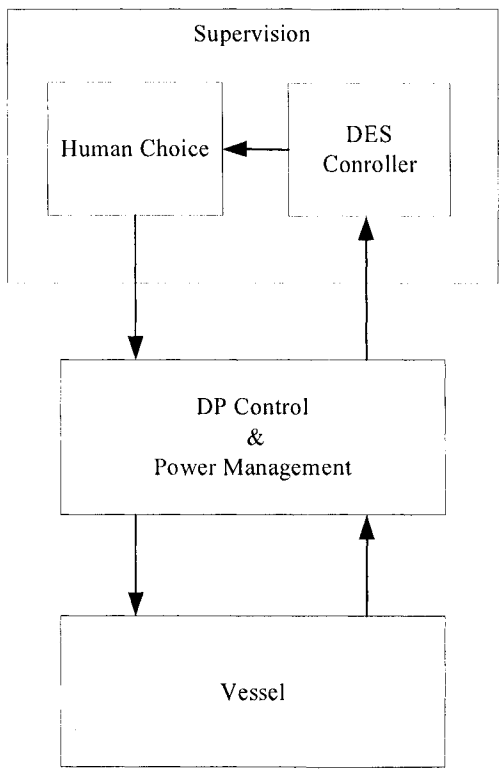


Figure 7-27: A block diagram of the HIL control arrangement for this example.

Table 7.9: A summary of the HIL controller simulation, see Figure 7-28.

Step	A_e	A_{en}	Size†	σ_{in}	σ_{out}
1	$\{sdd\}$	$\{\alpha_1^-, \alpha_1^+, \alpha_2^-, \alpha_2^+, hold\}$	4539	α_1^-	$tick$
2	$\{sdd\}$	$\{\alpha_1^-, \alpha_1^+, \alpha_2^-, \alpha_2^+, hold\}$	4486	α_1^-	$tick$
3	$\{sdd, \alpha_1^+, \alpha_2^+\}$	$\{\alpha_1^-, \alpha_2^-, hold\}$	1507	α_1^-	$tick$
4	$\{sdd, \alpha_1^-, \alpha_1^+, \alpha_2^+, hold\}$	$\{\alpha_2^-\}$	304	α_2^-	$tick$
5	$\{sdd, \alpha_1^-, \alpha_1^+, \alpha_2^+, hold\}$	$\{\alpha_2^-\}$	66	α_2^-	$tick$
6	$\{sdd, \alpha_1^-, hold\}$	$\{\alpha_2^-\}$	42	α_2^-	$tick$
7	$\{sdd\}$	$\{\alpha_2^-\}$	130	α_2^-	o_4
8	$\{sdd\}$	$\{\alpha_2^-\}$	444	α_2^-	$tick$
9	$\{sdd\}$	$\{\alpha_1^-, \alpha_1^+, \alpha_2^-, \alpha_2^+, hold\}$	1448	$hold$	$tick$
10	⋮	⋮	⋮	⋮	⋮

† graph size in transitions

The result of a HIL control simulation is summarized in Table 7.9. A knowledgeable operator knows that the vessel must move to port to reach its destination (the flare safe area), this rules out the actions that will carry the vessel away from the target, α_1^+, α_2^+, and the action that does nothing, *hold*. The event α_1^- is selected by the operator because it saves fuel by using only one generator. This same decision approach continues to be exercised for the next two steps. At step 4 though, the operator is suddenly presented with the fact that if the system is to continue, the second generator must be switched onto the propulsion bus, and so α_2^- must be selected. For the next 5 steps, the α_2^- event is the only choice that permits the vessel to continue to operate. After the control action of Step 7, the vessel crosses into the flare safe area, signified by the o_4 event. Once within the area, the choice of control actions returns, and the size of the controller grows as more legal moves are available again. For the remainder of this run (15 controller updates), the operator continues to select control actions that are in the A_{en} column and that are reasonable for stationkeeping, i.e. moves requiring only one generator.

A picture of what happened during this simulation is given by the time series of the position and velocity pictured in Fig. 7-28. A plan view of the two vessels during the manoeuvre is given by Fig. 7-29.

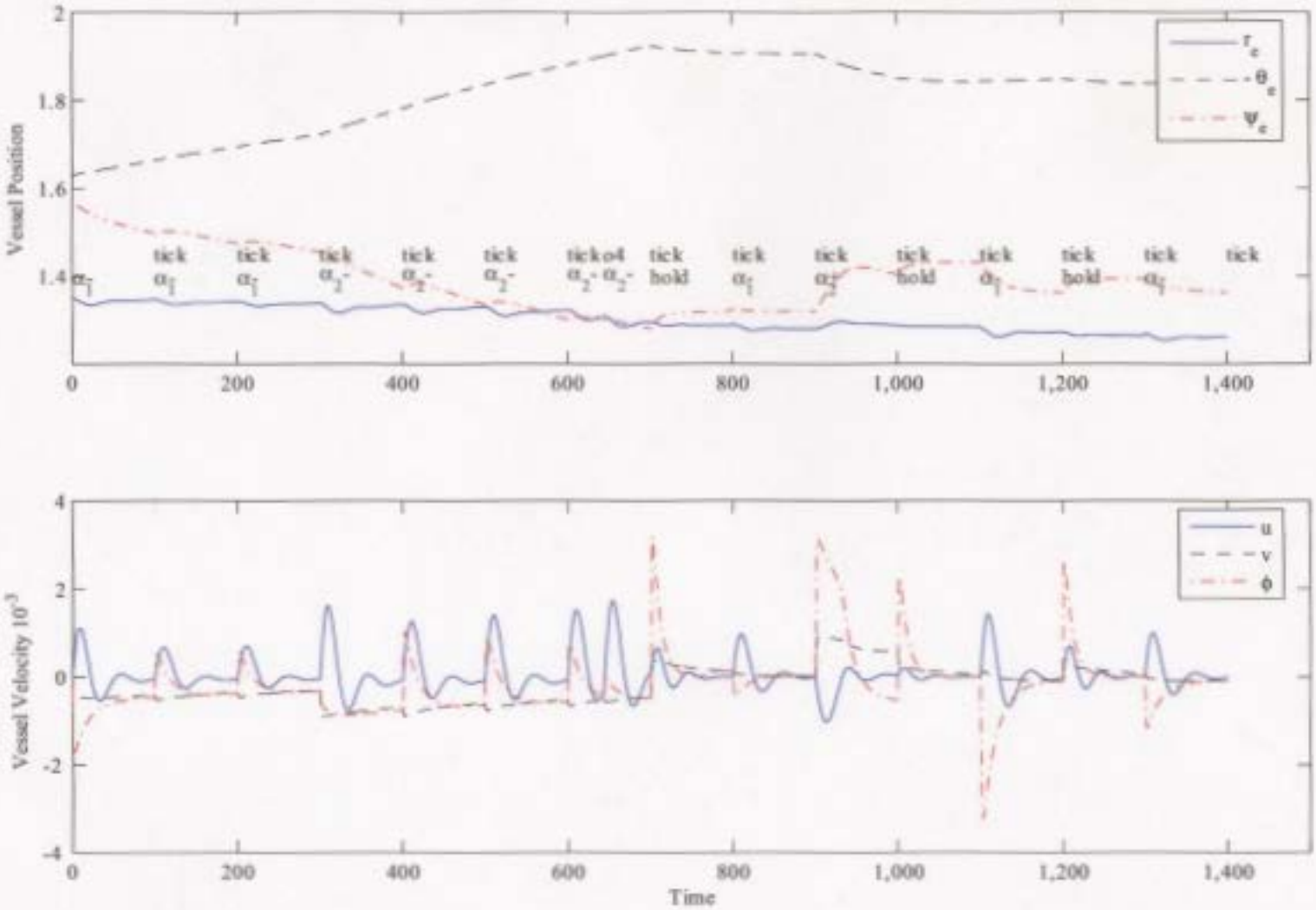


Figure 7-28: The HIL control simulation results showing a time series plot of the vessel position vector (top trace) and the vessel velocities (lower trace).

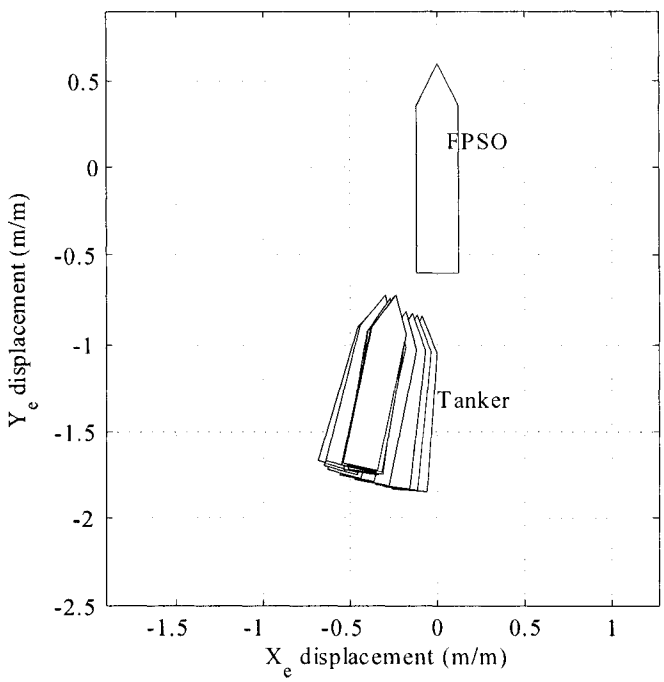


Figure 7-29: An overhead view of the tanker movement during the successful weathervaning maneouver of the HIL controller.

7.3 Remarks

The continuous system model of Chapter 3 was proposed as the basis for modeling the continuous dynamics of a hybrid system. Formally, the CSM accommodates systems whose dynamics can be modeled by nonlinear ODEs. The only constraint placed on these functions is that they be Lipschitz-continuous, in order to ensure finite discrete abstractions, but this is a theoretical consideration. The numerical solution of ODEs is well understood, but as any numerical solution is simply an approximation of the theoretical system model, there has to be a "leap of faith"; i.e. since there is theoretically a solution to an ODE, then the numerical solver must find an approximation of the solution that is sufficient for our purposes. Some of the issues of course are:

- Will the solver terminate?,
- What type of solver should be used?; i.e. one must choose appropriate stiffness, solver tolerance, etc.
- Will the solver locate the events?

So clearly, even ODE solvers have limitations that make the fundamental jump from theory to implementation uncertain.

Given that there will be implementation-specific issues with our control framework implementation, this implies that the theory should be used as a guide only. While nonlinear ODEs may be appropriate for modeling many types of continuous systems, is it possible that our computing framework may accommodate a broader range of continuous simulation tools?

In many industries, application-specific simulation tools already exist; if control system designers are to be convinced to use a hybrid control design package, they will not be willing to alter their simulation tools to match the problem. Within our

framework, it may be possible to encapsulate existing simulation tools, provided they meet some qualitative, but not theoretically rigorous conditions.

Starting with the assumption that there exists a continuous simulation tool that, given an initial condition and some parameters, produces a numerical solution for a particular system model. Then, we wish to specify the conditions that this simulation is comparable to a system of ODEs an numerical ODE solver. A simulation tool, when given a set of parameters:

1. must always produce an output (solution existence),
2. must be repeatable for the same parameters (solution uniqueness)
3. the solution can be computed in less time than it takes the actual system to execute (real-time implementation).

Whether the latter requirement (3) is met, hinges primarily on the complexity of the simulation and the extent to which the specification limits the legal trajectories of the plant. The computational hardware including the installed memory and CPU power may also influence (3) to a lesser extent.

If a simulation tool meets each of the above requirements, then with suitable wrapper functions (object methods), an SCM can be built around it, and an online hybrid controller is feasible.

We have demonstrated in the vessel control example of §7.2, that a complex continuous simulation may be embedded seamlessly into the SCM framework and the online controller synthesis works as expected. This vessel simulation appears to conform to practical requirements 1 and 2 above, and may even meet requirement 3 given a choice of appropriate lookahead horizon, control choice mechanism and CPU power. For this size of vessel, the actual controller update time Δt is

$$\Delta t = \sqrt{\frac{L}{g}} * 100 = 538 \text{ seconds}$$

Given the current state of computing hardware, it is conceivable that such a controller update time is more than adequate to implement this controller in real time. Clearly this would be the subject of future work.

7.4 Summary

This chapter has demonstrated the utility of the switched continuous modeling framework as a basis for simulation and control synthesis for hybrid systems. The theoretical results of the preceding chapters have been confirmed, particularly those regarding controller complexity and guaranteed controller safety. Some of the issues that have been dealt with in this chapter are: the process of developing a SCM, the encapsulation of the continuous variable models, the selection of appropriate specifications, and the effect of control choice mechanism on the system safety. The concept of the controller being able to maintain safety in the limited lookahead environment has been demonstrated through emergency shutdown examples.

The vessel control application involving human in the loop control points to the efficacy of combining a very simple heuristic in the control choice mechanism with the "brute force" of the exhaustive state-space search to yield better nonblocking controller behaviour despite the limited lookahead horizon. Furthermore, the heuristic might also be exploited to guide the search of the state space, thereby reducing the computational burden.

Chapter 8

Conclusions and Future Work

8.1 Contributions

This thesis develops four main themes with respect to hybrid system control: modeling, controller synthesis, computation and application. We will now highlight the contributions in each of these areas.

8.1.1 Model

The SCM model (Chapter 4) is designed for the automated synthesis of discrete event supervisory control systems. This model admits switching of continuous system dynamics in both a time and state-dependent fashion and retains the full generality of nonlinear continuous model dynamics. This is a significant advantage over most other hybrid synthesis and verification techniques that require the hybrid system models to use simplified continuous dynamics. The SCM also admits the inclusion of generalized simulations, enabling system designers to leverage existing numerical simulation tools, by embedding them in the SCM.

8.1.2 Control Synthesis

With the online controller, prior to each control decision, a hybrid transition graph is constructed based on a prediction of the discretized continuous dynamics for each eligible continuous system model in synchronism with other DES plant and specification processes. This graph is safe by design, because the safety specification was used to construct the graph. This graph is further pruned to eliminate blocking traces; i.e. those that do not carry the system to the horizon. By inclusion of emergency shutdown states, we guarantee safe operation by ensuring that these ESD states are always reachable within the controller graph's limited horizon; this is the fail-safe controller.

8.1.3 Computation

The computation model is central to the controller implementation. Due to the infinite state space of the hybrid model, limited lookahead horizon and finite control set are built-in limitations of the model. In order to extend the model through time (into an infinite time horizon) the controller is recomputed by incrementally extending the lookahead horizon in a moving horizon scheme. Since this task must be implemented online, an efficient means of model storage and computation are necessary. We describe an object-oriented modeling scheme that allows for compact storage of the plant and specification models; the hierarchical model storage avoids the state explosion that normally results from "flattening" the system model. Furthermore, the controller graph is easily constructed from the hierarchical model using a lazy technique that helps to reduce the intermediate size of the graph reachable set. The controller size complexity is bounded above by an expression that is exponential in lookahead horizon, which implies that the synthesis algorithm is, for the worst-case, exponential. However, the lower bound expression is linear in lookahead horizon, implying that the computation may not always be intractably large. The controller size is a function of the model state and time and the "tightness" of the specification.

A heavily specified system helps to reduce the controller size. The trade-off is that overspecification can lead to controller blocking.

8.1.4 Application

Application examples (Chapter 7) demonstrate the flexibility of the modeling scheme and provide empirical results for a reasonably complex industrial control problem. The ship control example demonstrates failsafe embedded simulations, controller synthesis, online control and HIL control, all novel concepts that have not been used in this application before.

8.2 Future Work

Future work will focus on the areas of modeling , control synthesis, applications, and improvements to the computational engine. Naturally, the impact of work in each of these areas is interconnected, so they will be carried out in parallel.

In the area of modeling, the objective will be to add the ability of our SCM/CSM modeling scheme to explicitly incorporate unmodeled continuous disturbance. Currently, the model accommodates modeled disturbance by simply including the disturbance model with each of the switched continuous system models. Various possible approaches exist, including exploiting the results of robust control for performance and stability guarantees or Monte Carlo simulation techniques.

It will be useful to develop other results around control choice mechanisms of various kinds. For example, HIL control appears to offer an improvement in controller performance or optimality, something that is not addressed well by the DES supervisory controller theory. HIL may not be suitable for every application, so automated control choice mechanisms must also be studied.

From the applications standpoint, it is desirable to apply the control technique to some of the more complex benchmark hybrid control problems, in order to pro-

vide further comparison of this technique to others. The ship control problem will continue to be actively studied, by looking at power system and thruster configuration robustness and multi-vessel coordination. A future model test may be planned to prove the controller in a real-time control environment. Future applications will also involve supervision and coordination of unmanned aerial vehicles (UAV) and autonomous underwater vehicles (AUV), as these are both funded research projects currently being carried out by the National Research Council. In both cases, there is access to physical modeling and numerical simulation capabilities, in addition to the underwater vehicles and aircraft. Both of these applications focus on the coordination of multiple agents. Studying these problems from an application perspective will help to focus future improvements to the modeling and control techniques.

Going hand in hand with improvements to modeling and applications, is the need for further development of the computational tool HySynth. Essentially, the software must be taken in two directions: 1) design time tool and 2) a run time tool. At present HySynth is a proof-of-concept code. It should be further expanded as the design-time tool; existing hybrid and DES libraries may prove to be useful extensions to HySynth. The goal will be to create an easy-to-use design environment. The second aspect of the software development effort should be focussed towards a run-time implementation that is optimized for the real-time online control environment. This involves such considerations as real-time operating system (OS) selection, hardware platform and the software development environment.

References

Abdelwahed, S., Su, R. and Neema, S.: 2005, A feasible lookahead control for systems with finite control set, *Proceedings of the 2005 IEEE Conference on Control Applications*, IEEE, pp. 663–668.

Allan, H.: 1999, Joint operations manual - FPSO and shuttle tanker, *Technical Report TN-PE-OP04-X00-001*, Terra Nova Petro-Canada.

Alur, R., Courcoubetis, C. and Dill, D. L.: 1993, Model-checking in dense real-time, *Information and Computation* **104**(1), 2–34.

Alur, R., Dang, T., Esposito, J., Hur, Y., Ivancic, F., Kumar, V., Lee, I., Mishra, P., Pappas, G. and Sokolsky, O.: 2003, Hierarchical modeling and analysis of embedded systems, *Proceedings of the IEEE* **91**(1), 11–28.

Alur, R. and Dill, D. L.: 1994, A theory of timed automata, *Theoretical Computer Science* **126**(2), 183–235.

Balas, G. J. and Packard, A. K.: 1996, *The structured singular value μ-framework*, CRC Press, pp. 671–688.

Balluchi, A., Benvenuti, L. and Sangiovanni-Vincentelli, A.: 2005, Hybrid systems in automotive electronics design, *Proceedings of the 44th IEEE Conference on Decision and Control*, pp. 5618–5623.

Balluchi, A., Natale, F. D., Sangiovanni-Vincentelli, A. L. and van Schuppen, J. H.: 2004, Synthesis for idle speed control of an automotive engine, *in* R. Alur and G. J. Pappas (eds), *Proceedings of Hybrid Systems: Computation and Control(HSCC04), 7th International Workshop*, Vol. 2993 of *Lecture Notes in Computer Science*, Springer, pp. 80–94.

Bayen, A. and Tomlin, C.: 2003, Real time discrete control law synthesis for hybrid systems using MILP: application to congested airspace, *Proceedings of the 2003 American Control Conference*, pp. 4620–4626.

Bemporad, A., Heemels, W. and de Schutter, B.: 2002, On hybrid systems and closed-loop MPC systems, *IEEE Transactions on Automatic Control* **47**(5), 863–869.

Bengtsson, J., Larsen, K. G., Larsson, F., Pettersson, P. and Yi, W.: 1995, UPPAAL - a tool suite for automatic verification of real-time systems, *Hybrid Systems*, pp. 232–243.

Bernstein, D. S.: 2002, Feedback control: an invisible thread in the history of technology, *IEEE Control Systems Magazine* **22**(2), 53–68.

Blondel, V. D. and Tsitsiklis, J. N.: 2000, A survey of computational complexity results in systems and control, *Automatica* **36**(9), 1249–1274.

Blondel, V. and Tsitsiklis, J.: 1999, Complexity of stability and controllability of elementary hybrid systems, *Automatica* **35**(3), 479–489.

Boel, R., Cao, X.-R., Cohen, G., Giua, A., Wonham, W. M. and van Schuppen, J. H.: 2002, Unity in diversity, diversity in unity: Retorspective and prospective views on control of discrete event systems, *Discrete Event Dynamic Systems: Theory and Applications* **12**, 253–264.

Bourdon, S. E., Lawford, M. and Wonham, W. M.: 2005, Robust nonblocking supervisory control of discrete-event systems, *IEEE Transactions on Automatic Control* **50**(12), 2015–2021.

Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S. and Yovine, S.: 1998, KRONOS: A model-checking tool for real-time systems, *in* A. J. Hu and M. Y. Vardi (eds), *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, Vol. 1427, Springer-Verlag, pp. 546–550.

Braatz, R. D., Young, P. M., Doyle, J. C. and Morari, M.: 1994, Computational complexity of μ calculation, *IEEE Transactions on Automatic Control* **39**(5), 1000–1002.

Brandin, B. A. and Wonham, W.: 1992, The supervisory control of timed discrete event systems, *Proceedings of the 31st IEEE Conference on Decision and Control*, pp. 3357–3362.

Branicky, M.: 1998, Multiple lyapunov functions and other analysis tools for switched and hybrid systems, *IEEE Transactions on Automatic Control* **43**, 475–482.

Brave, Y. and Heymann, M.: 1991, Control of discrete event systems modeled as hierarchical state machines, *Proceedings of the 30th IEEE Conference on Decision and Control*, pp. 1499–1504.

Brooks, C., Cataldo, A., Lee, E. A., Liu, J., Liu, X., Neuendorffer, S. and Zheng, H.: 2005, Hyvisual: A hybrid system visual modeler, *Technical Memorandum UCB/ERL M05/ 24*, University of California, Berkeley, CA 94720.

Carloni, L., DiBenedetto, M., Pinto, A. and Sangiovanni-Vincentelli, A.: 2004, Modeling techniques, programming languages, and design toolsets for hybrid systems, *Technical Report IST-2001-38314*, Columbus Project.

Cassandras, C. G. and Lafortune, S.: 1999, *Introduction to Discrete Event Systems*, Kluwer Academic Publishers.

Chung, S., Lafortune, S. and Lin, F.: 1992, Limited lookahead policies in supervisory control of discrete event systems, *IEEE Transactions on Automatic Control* **37**(12), 1921–1935.

Chung, S., Lafortune, S. and Lin, F.: 1994, Supervisory control using variable lookahead policies, *Discrete Event Dynamic System: Theory and Applications* **4**(3), 237–268.

Chutinan, A. and Krogh, B. H.: 2003, Computational techniques for hybrid system verification, *IEEE Transactions on Automatic Control* **48**(1), 64–75.

Coleri, S., Ergen, M. and Koo, T.: 2002, Lifetime analysis of a sensor network with hybrid automata modeling.

David, A. and Yi, W.: 2000, Modelling and analysis of a commercial field bus protocol, *Proceedings of the 12th Euromicro Conference on Real Time Systems*, IEEE Computer Society, pp. 165–172.

Daws, C., Kwiatkowska, M. Z. and Norman, G.: 2004, Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM, *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 5, pp. 221–236.

Ecker, J. and Malmborg, J.: 1999, Design and implementation of a hybrid control strategy, *IEEE Control Systems Magazine* **19**(4), 12–21.

Esposito, J., Kumar, V. and Pappas, G.: 2003, Multi-agent hybrid system simulation, *Proceedings of the 40th IEEE Conference on Decision and Control*, IEEE, pp. 780–785.

Fehnker, A., Vaandrager, F. and Zhang, M.: 2003, Modeling and verifying a Lego car using hybrid I/O automata.

Fossen, T.: 1994, *Guidance and Control of Ocean Vehicles*, John Wiley & Sons.

Francis, B., Henton, J. W. and Zames, G.: 1984, $\mathcal{H}_\infty$-optimal feedback controllers for linear multivariable systems, *IEEE Transactions on Automatic Control* **AC-29**(10), 888–900.

Gansner, E., Koustofios, E. and North, S.: 2002, *Drawing Graphs with Dot*, AT&T.

Gaudin, B. and Marchand, H.: 2005, Safety control of hierarchical synchronous discrete event systems: A state-based approach, *Proceedings of the 13th Mediterranean Conference on Control and Automation*, pp. 889–895.

Giorgetti, N., Pappas, G. J. and Bemporad, A.: 2005, Bounded model checking of hybrid dynamical systems, *Proceedings of the 44th IEEE Conference on Decision and Control*, pp. 672–677.

Gohari, P. and Wonham, M.: 2000, On the complexity of supervisory control design in the RW framework, *IEEE Transactions on Systems, Man, and Cybernetics* **30**(5), 643–652.

Grewal, M. and Andrews, A.: 1993, *Kalman Filtering Theory and Practice*, Prentice Hall.

Hadj-Alouane, N., Lafortune, S. and Lin, F.: 1994, Variable lookahead supervisory control with state information, *IEEE Transactions on Automatic Control* **39**(12), 2398–2410.

Hancox, M.: 2001, *Towing, Positioning and Hook-Up for Offshore Production*, Vol. 8, Oilfield Publications Limited.

Harel, D.: 1987, Statecharts: A visual formalism for complex systems, *Science of Computer Programming* **8**(3), 231–274.

Heemels, W., de Schutter, B. and Bemporad, A.: 2001, On the equivalence of classes of hybrid dynamical models, *Proceedings of the 40th IEEE Conference on Decision and Control* **1**, 364–369.

Henzinger, T. A.: 2000, The theory of hybrid automata, *Verification of Digital and Hybrid Systems* **170**, 265–292.

Henzinger, T. A. and Ho, P.: 1995, Algorithmic analysis of nonlinear hybrid systems, *in* P. Wolper (ed.), *Proceedings of the 7th International Conference On Computer Aided Verification*, Vol. 939, Springer Verlag, Liege, Belgium, pp. 225–238.

Henzinger, T. A., Ho, P. and Wong-Toi, H.: 1996, *A User Guide to HYTECH.*

Henzinger, T. A., Ho, P. and Wong-Toi, H.: 1997, HyTech: A model checker for hybrid systems, *Software Tools for Technology Transfer* **1**, 110–122.

Henzinger, T. A., Kopke, P. W., Puri, A. and Varaiya, P.: 1998, What's decidable about hybrid automata?, *Journal of Computer and System Sciences* .

Henzinger, T. A. and Wong-Toi, H.: 1995a, Linear phase-portrait approximations for nonlinear hybrid systems, *Hybrid Systems*, pp. 377–388.

Henzinger, T. A. and Wong-Toi, H.: 1995b, Using hytech to synthesize control parameters for a steam boiler, *Formal Methods for Industrial Applications*, pp. 265–282.

Hespanha, J. and Morse, A.: 2002, Switching between stabilizing controllers, *Automatica* **38**(11), 1905–1917.

Hespanha, J. P.: 2004, *Control Systems, Robotics, and Automation*, Encyclopedia of life support systems, Eolss Publishers Co. Ltd., Oxford, chapter Stabilization through Hybrid Systems.

Heymann, M., Lin, F., Meyer, G. and Resmerita, S.: 2002, Analysis of zeno behaviours in hybrid systems, *Proceedings of the 41st IEEE Conference on Decision and Control*, pp. 2379–2384.

Johansson, K. H., Egerstedt, M., Lygeros, J. and Sastry, S.: 1999, On the regularization of zeno hybrid automata, *Systems & Control Letters*, Vol. 38, Elsevier, pp. 141–150.

Kalman, R. E.: 1960, A new approach to linear filtering and prediction problems, *Transactions of the ASME–Journal of Basic Engineering* **82**(Series D), 35–45.

Kaynak, O., Erbatur, K. and Ertugrul, M.: 2001, The fusion of computationally intelligent methdologies and sliding mode control - a survey, *IEEE Transactions on Industrial Electronics* **48**(1), 4–17.

Khalil, H. K.: 2002, *Nonlinear Systems*, third edn, Prentice Hall.

Knopp, K.: 1956, *Infinite Sequences and Series*, Dover.

Koutsoukos, X. and Antsaklis, P.: 2001, Hierarchical control of piecewise linear hybrid dynamical systems based on discrete abstractions, *Technical Report ISIS-2001-001*, ISIS Group, University of Notre Dame.

Koutsoukos, X., Antsaklis, P., Stiver, J. and Lemmon, M.: 2000, Supervisory control of hybrid systems, *Proceedings of the IEEE*, IEEE, pp. 1026–1048.

Kumar, R., Chung, H. M. and Marcus, S. I.: 1998, Extension based limited lookahead supervision of discrete event systems, *Automatica* **34**(11), 1327–1344.

Kumar, R. and Garg, V. K.: 1995, *Modeling and Control of Logical Discrete Event Systems*, Kluwer Academic Publishers.

Lee, E. A.: 2003, Overview of the Ptolemy project, *Technical Memorandum UCB/ERL M03/25*, University of California, Berkeley, CA, 94720, USA.

Leith, D. and Leithead, W.: 2000, Survey of gain scheduling analysis & design, *International Journal of Control* **73**(11), 1001–1025.

Lemmon, M. D., He, K. X. and Markovsky, I.: 1999, Supervisory hybrid systems, *IEEE Control Systems* pp. 42–55.

Liberzon, D.: 2003, *Switching in Systems and Control*, Systems and Control: Foundations and Applications, Birkhauser, Boston, MA.

Lin, F. and Wonham, M.: 1988, On observability of discrete-event systems, *Information Sciences* **44**, 173–198.

Lin, H. and Antsaklis, P.: 2005, Stability and stabilizability of switched linear systems: A short survey of recent results, *Proceedings of the 2005 IEEE International Symposium on Intelligent Control*, pp. 24–29.

Lynch, N., Segala, R. and Vaandrager, F.: 2003, Hybrid I/O automata, *Technical Report MIT-LCS-TR-827d*, MIT Laboratory for Computer Science, Cambridge, MA 02139.

MATLAB Programming: 2006, The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098.

Meder, C.: 1997, *TTCT User Manual V0.5.0*, Systems Control Group, University of Toronto.

Michel, A. N.: 1996, Stability: the common thread in the evolution of feedback control, *IEEE Control Systems Magazine* **16**(3), 50–60.

Millan, J. and O'Young, S.: 2000, Hybrid modeling of tandem dynamically positioned vessels, *Proceedings of the 39th IEEE Conference on Decision and Control.*

Millan, J. and O'Young, S.: 2006, Hybrid system control using an online discrete event supervisory strategy, *IFAC Conference on Analysis and Design of Hybrid Systems*, IFAC.

Millan, J. P.: 2006, On-line supervisory control of hybrid systems using embedded simulations, *Proceedings of the 8th International Workshop on Discrete Event Systems WODES06.*

Millan, J., Smith, L. and O'Young, S.: 2002, Coordination of FPSO and tanker offloading operations, *Proceedings of the MTS Dynamic Positioning Conference 2002.*

Mitra, S., Wang, Y., Lynch, N. and Feron, E.: 2003, Application of hybrid I/O automata in safety verification of pitch controller for model helicopter system, *Technical Report MIT-LCS-TR-880*, MIT Lab for Computer Science, Cambridge, MA 02139.

Moor, T., Raisch, J. and Davoren, J.: 2001, Computational advantages of a two-level hybrid control architecture, *Proceedings of the 40th IEEE Conference on Decision and Control*, pp. 358–363.

Morgan, M.: 1978, *Dynamic Positioning of Offshore Vessels*, PPC Books, Division of Petroleum Publishing Company.

Mosterman, P. J.: 2002, HYBRSIM - a modeling and simulation environment for hybrid bond graphs, *Journal of Systems and Control Engineering* **216**, 35–46.

Oden, J. T.: 1979, *Applied Funcitonal Analysis*, Prentice Hall.

O'Young, S.: 1991, On the synthesis of the supervisors for timed discrete event processes, *Technical Report 9107*, Department of Electrical and Computer Engineering, University of Toronto.

O'Young, S. D.: 1992, Systems control group report, Object TCT: Users guide, *Technical report*, Department of Electrical Engineering, University of Toronto.

Palm III, W. J.: 2000, *Modeling, Analysis, and Control of Dynamic Systems*, second edn, John Wiley & Sons.

Potocnik, B., Bemporad, A., Torrisi, F., Music, G. and Zupancic, B.: 2004, Hybrid modelling and optimal control of a multiproduct batch plant, *Control Engineering Practice* **12**(9), 1127–1137.

Raisch, J.: 2000, Discrete abstractions - an Input/Output point of view, *Mathematical and Computer Modeling of Dynamical Systems* **6**(1), 6–29. Special Issue on Discrete Event Models of Continuous Systems.

Raisch, J. and O'Young, S.: 1998, Discrete approximation and supervisory control of continuous systems, *IEEE Transactions on Automatic Control* **43**(4), 569–573.

Ramadge, P. and Wonham, W.: 1987, Supervisory control of a class of discrete event processes, *SIAM Journal on Control Optimization* **25**(1), 206–230.

Ramadge, P. and Wonham, W.: 1989, The conrol of discrete event systems, *Proceedings of the IEEE* **77**(1), 81–98.

Rodger, S. H. and Finley, T. W.: 2006, *JFLAP: An Interactive Formal Languages and Automata Package*, Jones & Bartlett Publishers.

Rudie, K. and Wonham, M.: 1992, Think globally, act locally: Decentralized supervisory control, *IEEE Transactions on Automatic Control* **37**(11), 1692–1708.

Shampine, L. and Gladwell, I.: 1991, Reliable solution of special event location problems for odess, *ACM Transactions on Mathematical Software* **17**(1), 11–25.

Shampine, L. and Thompson, S.: 2000, Event location for ordinary differential equations, *Computers and Mathematics with Applications* **39**, 43–54.

Skogestad, S. and Postelthwaite, I.: 1993, *Multivariable Feedback Control Analysis and Design*, Wiley.

Smith, R. and Doyle, J.: 1988, The two tank experiment: A benchmark control problem, *Proceedings of the IEEE American Control Conference* **3**, 403–415.

Stiver, J. A., Koutsoukos, X. D. and Antsaklis, P. J.: 2000, An invariant-based approach to the design of hybrid control systems, *Technical Report ISIS-2000-001*, ISIS Group, University of Notre Dame.

Stursberg, O.: 2004, A graph search algorithm for optimal control of hybrid systems, *Proceedings of the 43rd IEEE Conference on Decision and Control*, pp. 1412–1417.

Stursberg, O., Fehnker, A., Han, Z. and Krogh, B. H.: 2003, Specification-guided analysis of hybrid systems using a hierarchy of validation methods, *IFAC Conference on Analysis and Design of Hybrid Systems*, IFAC.

Stursberg, O., Kowalewski, S., Hoffmann, I. and Preusig, J.: 1997, Comparing timed and hybrid automata as approximations of continuous systems, *Hybrid Systems IV, LNCS 1273*, Springer-Verlag, pp. 361–377.

Su, R., Abdelwahed, S., Karsai, G. and Biswas, G.: 2003, Discrete abstraction and supervisory control for switching systems, *IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 1, IEEE, pp. 415–421.

T. Stauner, O. Mueller and M. Fuchs: 1997, Using HYTECH to verify an automotive control system, *in* O. Maler (ed.), *Hybrid and Real-Time Systems*, Springer Verlag, LNCS 1201, Grenoble, France, pp. 139–153.

Thevenon, L. and Flaus, J.-M.: 2000, Modular representation of complex hybrid systems: application to the simulation of batch processes, *Simul. Pr. Theory* **8**(5), 283–306.

Torrisi, D. and Bemporad, A.: 2001, Discrete-time hybrid modeling and verification, *Proceedings of the 40th IEEE Conference on Decision and Control*, pp. 2899–2904.

Torrisi, F. and Bemporad, A.: 2004, HYSDEL - a tool for generating computational hybrid models, *IEEE Transactions on Control Systems Technology* **12**(2), 235–249.

UMDES Software Library: 2006.
URL: *http://www.eecs.umich.edu/umdes/toolboxes.html*

Weingarth, L.: 2002, Avoiding catastrophes in dynamic positioning; integrating key parameters using a systems approach, *IADC SPE Drilling Conference.*

Williams, S. J.: 1990, A review of μ and its applications, *IEE Colloquium on Successful Industrial Applicaitons of Multivariable Analysis* pp. 7/1–7/14.

Witsenhausen, H. S.: 1966, A class of hybrid-state continuous-time dynamic systems, *IEEE Transacitons on Automatic Control* pp. 161–167.

Wonham, W. and Ramadge, P.: 1987, On the supremal controllable sublanguage of a given language, *SIAM Journal on Control Optimization* **25**(3), 637–659.

Zhang, J., Johansson, K. H., Lygeros, J. and Sastry, S.: 2000, Dynamical systems revisited: Hybrid systems with zeno executions, *in* N. A. Lynch and B. H. Krogh (eds), *Hybrid Systems: Computation and Control, Third International Workshop, HSCC 2000, Pittsburgh, PA, USA, March 23-25, 2000, Proceedings*, Vol. 1790 of *Lecture Notes in Computer Science*, Springer, pp. 451–464.

Appendices

Appendix A

Continuous System Modeling

A continuous dynamical system can be described by the first-order vector differential equation,

$$\frac{dx}{dt} = f(x, u, t) \tag{A.1}$$

where x is the state vector, $x \in \mathbb{R}^n$, and u is the system input vector, $u \in \mathbb{R}^m$. The state variables, $x_1, x_2, \ldots x_n$, and the system input variables, $u_1, u_2, \ldots u_m$ are functions of time, $t \in \mathbb{R}$. Since this is a vector system, it is composed of n scalar first-order nonlinear differential equations:

$$\begin{aligned}
\frac{dx_1}{dt} &= f_1(x_1, \ldots x_n, u_1 \ldots u_m, t) \\
\frac{dx_2}{dt} &= f_2(x_1, \ldots x_n, u_1 \ldots u_m, t) \\
&\vdots \\
\frac{dx_n}{dt} &= f_n(x_1, \ldots x_n, u_1 \ldots u_m, t)
\end{aligned}$$

where each state variable $x_i \in \mathbb{R}$, each input $u_j \in \mathbb{R}$, and each $f_i : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \rightarrow \mathbb{R}$.

The class of systems that we wish to consider can be modeled by an unforced, ordinary differential equation (ODE),

$$\dot{x}(t) = f(x, t) \tag{A.2}$$

where $\dot{x} = \frac{dx}{dt}$, and $f : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R}^n$. Suppose u is some function of time, $u = g(t)$ and u is also a function of the state x, (for example due to state feedback), then u can be described as a function of both time, t and state, $x = g(x,t)$. Substituting $u = g$ into Eq. A.1, allows the independent variable u to be eliminated, providing the dynamics associated with function g are absorbed into f. This means that, without loss of generality, results developed for the unforced state equation (Eq. A.2) are equally applicable to the forced system model (Eq. A.1).

A.0.1 Elementary Topology

Definition A.0.1 (Euclidian Metric) *The Euclidean metric or 2-norm is defined as*

$$\forall x, y \in \mathbb{R}^n,\ d(x,y) = \|x - y\|_2 = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Definition A.0.2 (Metric Space) *Let $X \subseteq \mathbb{R}^n$, then the set X with the euclidean metric is called a metric space and is denoted $(X, \|\cdot\|)$.*

Definition A.0.3 *Let $(X, \|\cdot\|)$ be a metric space, $x_0 \in X$ and $r > 0$, then an open ball is defined as*

$$B_o(x_0, r) = \{x \in X : \|x_0 - x\| < r\}$$

likewise, the corresponding closed ball is defined as

$$B_c(x_0, r) = \{x \in X : \|x_0 - x\| \leq r\}$$

An open ball can be thought of as the interior of a ball, excluding its boundary, while a closed ball includes the boundary.

Definition A.0.4 (Open Set) *A subset Q of the metric space $(X, \|\cdot\|)$ is an open set with respect to the metric $\|\cdot\|$ provided that Q is a union of open balls.*

Definition A.0.5 (Closed Set) *A subset $\bar{Q}$ of a metric space X, is a closed set with respect to the metric $\|\cdot\|$, provided that its complement $X\backslash\bar{Q}$ is an open set with respect to X. The overbar notation will be used to indicate the closure of a set (a closed set).*

Note that henceforth for convenience, the 2 subscript will be dropped from the Euclidian metric and it will be implied by the notation $\|\cdot\|$.

Definition A.0.6 (Initial Value Problem) *Given an open subset $D \subset \mathbb{R}^n \times \mathbb{R}$, a continuous function $f : D \to \mathbb{R}^n$, and a point $(x_0, t_0) \in \mathbb{R}^n \times \mathbb{R}$, we wish to find a solution $x(t)$ to the equations $\dot{x} = f(x,t), x(t_0) = x_0$. This is known as an initial value problem (IVP).*

A solution to the IVP is given by the continuously differentiable function:

$$x(t) = x_0 + \int_0^t f\left(x(\tau)\right) d\tau$$

which is a solution in the sense of Caratheodory.

Definition A.0.7 (Local Lipschitz Continuity) *A function f, is locally Lipschitz continuous if for each $x, y \in D \subset \mathbb{R}^n$ and $t \in [t_0, t_1] \subset \mathbb{R}$,*

$$\|f(x,t) - f(y,t)\| \leq L \|x - y\| \tag{A.3}$$

where $L > 0$ is the Lipschitz constant.

Remark A.0.1 (Existence and Uniqeuness of Solutions) *If a function f is locally Lipschitz continuous, then the solution, $x(t)$ to the IVP $\dot{x} = f(x(t), t)$, $x(t_0) = x_0$, exists and is unique over the interval $[t_0, t_1]$.*

Definition A.0.8 (Global Lipschitz Continuity) *The function f is said to be globally Lipschitz continuous if there exists a single Lipschitz constant, $L > 0$, such that for all $x, y \in \mathbb{R}^n$ and all $t \in \mathbb{R}$ Eq. A.3 is true.*

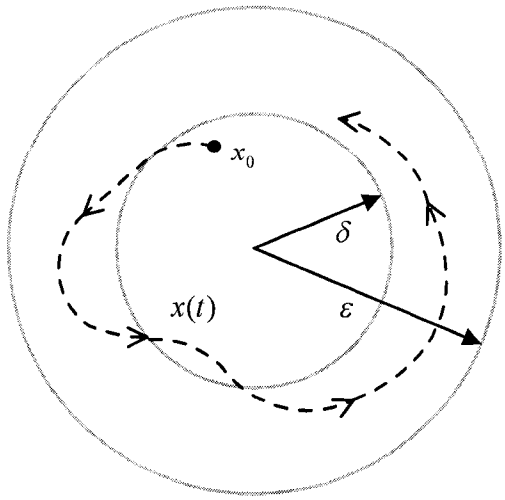


Figure A-1: Lyapunov stability

Lipschitz continuity is a stronger requirement than strict continuity, but is weaker than continuous differentiability (Khalil 2002).

A.0.2 Lyapunov Stability

Definition A.0.9 (Equilibrium Point) *Given a continuous system model* $\dot{x} = f(x,t)$, $x(t_0) = x_0$, $x \in \mathbb{R}^n$, *then* x^*, *is an equilibrium point if*

$$f(t, x^*) = 0, \ \forall t \geq 0$$

Definition A.0.10 (Lyapunov Stability) *An equilibrium point* $x^* = 0$ *is stable in the sense of Lyapunov at* $t = t_0$ *if for all* $t > t_0$ *and for all* $\varepsilon > 0$, *there exists* δ *such that*

$$\|x(t_0)\| < \varepsilon \implies \|x(t)\| < \delta, \ \textit{for all } t \geq t_0 \tag{A.4}$$

Definition A.0.11 (Uniform Lyapunov Stability) *An equilibrium point* $x^* = 0$ *is uniformly stable in the sense of Lyapunov if there exists* $\delta(\varepsilon)$, *independent of* t_0,

such that Eq. A.4 holds true.

Without loss of generality, the equilibrium point has been located at the origin for the preceding definitions. In fact, it is possible for an equilibrium point to be located anywhere in the state space of a system. Results for non-zero equilibrium points are identical since an equilibrium can be translated from anywhere in the state space to the origin.

Definition A.0.12 (Unstable Equilibrium) *An equilibrium point x^* is unstable if it is not stable.*

Appendix B

State Partitioning

In the case where additional partitions may need to be established on an already existing partitioned state space, the results of Lemma 3.2.4, Lemma 3.2.3 and Theorem 3.2.1 of Chapter 3 are extended here. The assumption is that we are starting with a family of subsets H such that $|H| = M$ and H is the result of a partitioning operation that has already been applied to the state space such that $\bigcup_{j=1}^{M} Q_j = \mathbb{R}^n$.

Lemma B.0.1 (Set Partition Upper Bound) *Let $H = \{Q_j \subseteq \mathbb{R}^n : 1 \leq j \leq M\}$ be a family of pairwise disjoint sets such that $\bigcup_{j=1}^{M} Q_j = \mathbb{R}^n$, and let $\Psi = \{F_i : \mathbb{R}^n \rightarrow \mathbb{R},\ 1 \leq i \leq N\}$ be a family of functionals. The set partition operation (II) of Def. 3.2.7,will produce a family of sets, H', such that $|H'| = M \times 2^N$.*

Proof. Show the upper bound by induction on the number of functionals, i, for any $|H| = M$ and assuming maximal intersection, which means that $Q_j \cap \mathcal{N}(F_i) \neq \emptyset$

for all $Q_j \in H$ and for all $F_i \in \Psi$. The base case is for $|\Psi| = 1$ or $N = 1$ functionals:

$$\begin{aligned}
|H'_1| &= P_f^1(H, \mathcal{N}(F)) = \left| \bigcup_{j=1}^{M} P_s(Q_j, \mathcal{N}(F)) \right| \\
&= |P_s(Q_1, \mathcal{N}(F))| + |P_s(Q_2, \mathcal{N}(F))| + \ldots \\
&\quad + |P_s(Q_M, \mathcal{N}(F))| \\
&= \underbrace{2 + 2 + \ldots + 2}_{M} \\
&= 2M \\
&= M \times 2^N
\end{aligned}$$

so the base case is consistent with the original hypothesis. The inductive hypothesis is that $|H'_N| = M \times 2^N$. It remains to show that $|H'_{N+1}| = M \times 2^{N+1}$. Due to maximal intersection, $Q_j \cap \mathcal{N}(F_{N+1}) \neq \emptyset$, for all $Q_j \in H'_N$.

$$H'_{N+1} = P_f^{N+1}(H'_N, \mathcal{N}(F_{N+1})) = \bigcup_{j=1}^{M \times 2^N} P_s(Q_j, \mathcal{N}(F_{N+1}))$$

and therefore,

$$\begin{aligned}
|H'_{N+1}| &= \left| \bigcup_{j=1}^{|H'_N|} P_s(Q_j, \mathcal{N}(F_{N+1})) \right| \\
&= |P_s(Q_1, \mathcal{N}(F_{N+1}))| + |P_s(Q_2, \mathcal{N}(F_2))| + \ldots \\
&\quad + \left| P_s(Q_{|H'_N|}, \mathcal{N}(F_{N+1})) \right| \\
&= \underbrace{2 + 2 + \ldots + 2}_{|H'_N|} \\
&= 2 \times |H'_N| \text{ and by the inductive hypothesis,} \\
&= 2 \times M \times 2^N \\
&= M \times 2^{N+1}
\end{aligned}$$

■

Lemma B.0.2 (Set Partition Lower Bound) *Let* $H = \{Q_j \subseteq \mathbb{R}^n : 1 \leq j \leq M\}$ *be a family of pairwise disjoint sets such that* $\bigcup_{j=1}^{M} Q_j = \mathbb{R}^n$ *and let* $\Psi = \{F_i : \mathbb{R}^n \rightarrow \mathbb{R},\ 1 \leq i \leq N\}$ *be a family of functionals such that there is minimal intersection. The set partition operation (II) of Def. 3.2.7,will produce a family of sets, H', such that* $|H'| = M + N$.

Proof. The lower bound is proven by induction on the number of functionals, i, for any $|H| = M$. The base case is for $|\Psi| = 1$ (i.e. $N = 1$). With minimal intersection for example $Q_M \in H$ such that $Q_M \cap \mathcal{N}(F) \neq \emptyset$, then

$$\begin{aligned}
H_1' &= P_f^1(H, \mathcal{N}(F_1)) \\
|H_1'| &= \left| \bigcup_{j=2}^{M} P_s(Q_j, \mathcal{N}(F_1)) \right| + |P_s(Q_M, \mathcal{N}(F_1)| \\
&= |P_s(Q_1, \mathcal{N}(F_1)| + |P_s(Q_2, \mathcal{N}(F_1)| + \ldots \\
&\quad + |P_s(Q_{M-1}, \mathcal{N}(F_1)| \\
&\quad + |P_s(Q_M, \mathcal{N}(F_1)| \\
&= \underbrace{1 + 1 + \ldots + 1}_{M-1} + 2 \\
&= M - 1 + 2 \\
&= M + 1
\end{aligned}$$

Thus the base case is consistent with the hypothesis. Now the inductive hypothesis is that $|H_N'| = M + N$. It remains to show $|H_{N+1}'| = M + N + 1$. Suppose there exists $Q_{M+N} \in H_N'$ and $|H_N'| = |H_{N+1}|$ then:

$$H_{N+1}' = P_f^{N+1}(H_N', \mathcal{N}(F_{N+1})) = \bigcup_{j=1}^{M+N} P_s(Q_j, \mathcal{N}(F_{N+1}))$$

therefore

$$\begin{aligned}
|H'_{N+1}| &= \left|\bigcup_{j=1}^{|H'_N|-1} P_s(Q_j, \mathcal{N}(F_{N+1}))\right| + \left|P_s(Q_{|H'_N|}, \mathcal{N}(F_{N+1}))\right| \\
&= |P_s(Q_1, \mathcal{N}(F_{N+1}))| + |P_s(Q_2, \mathcal{N}(F_{N+1}))| + \ldots \\
&\quad + \left|P_s(Q_{|H'_N|-1}, \mathcal{N}(F_{N+1}))\right| \\
&\quad + \left|P_s(Q_{|H'_N|}, \mathcal{N}(F_{N+1}))\right| \\
&= \underbrace{1+1+\ldots+1}_{|H'_N|-1} + 2 \\
&= |H'_N| - 1 + 2, \text{and by the inductive hypothesis,} \\
&= M + N + 1
\end{aligned}$$

■

Theorem B.0.1 (Set Partitioning Operation Boundedness) *Let* $H = \{Q_j \subseteq \mathbb{R}^n : 1 \leq j \leq M\}$ *be a family of pairwise disjoint sets such that* $\bigcup_{j=1}^{M} Q_j = \mathbb{R}^n$ *and let* $\Psi = \{F_i : \mathbb{R}^n \to \mathbb{R},\ 1 \leq i \leq N\}$ *be a family of functionals. The set partition operation (II) of Def. 3.2.7 produces a family of sets* H'*, such that* $M + N \leq |H'| \leq M \times 2^N$.

Proof. Let $|\Psi| = N$, and let $|H| = M$ it follows from Lemma B.0.2 that the cardinality of the set partitioning operation has a lower bound of $|H'| = M + N$. Lemma B.0.1 establishes an upper bound on the returned family of sets of $|H'| = M \times 2^N$, hence the result is proven. ■

Appendix C

Discrete Event System Modeling

C.1 Finite State Machines

The finites state machine (FSM) model is one of many possible representations of a discrete event system (DES), and has been used extensively since it lends itself readily to the analysis of such systems.

Definition C.1.1 *A deterministic FSM model is a five-tuple:*

$$G = (X, \Sigma, \Delta, x_0, X_m)$$

where:

X is the set of states,

Σ is the set of events that cause G to change states

Δ is the transition set, the set of all labeled transitions

x_0 is the initial state, $x_0 \in X$

X_m is the set of marked states, $X_m \subseteq X$

Definition C.1.2 *A transition function is defined in G denoted by* $\delta(x, \sigma)!$ *with* $x \in X$, $\sigma \in \Sigma^*$, *if* $\exists x \in X$ *and* $(x, \sigma, \acute{x}) \in \Delta$.

Definition C.1.3 *The language generated by G,* $L(G)$, *is the set of all symbols and concatenated symbols (strings) that would be generated by starting at* x_0 *and exercising all possible transitions in sequence across the entire state space of the automaton.*

Definition C.1.4 *The language marked by G,* $L_m(G)$, *is the set of all strings s, for which* $\delta(x_0, s)!$ *in G and* $\delta(x_0, s) \in X_m$. *The marked states represent the completion of some task and the strings s, are able to take the system from the initial condition to the completion of the task, a marked state.*

Definition C.1.5 *Two automata,* G_1 *and* G_2, *are said to be equivalent if* $L(G_1) = L(G_2)$ *and* $L_m(G_1) = L_m(G_2)$.

Definition C.1.6 *The language* $L(G_2)$ *is said to be a sublanguage of* $L(G_1)$ *if* $\forall s \in L(G_2), s \in L(G_1) \Rightarrow L(G_1) \subseteq L(G_2)$.

The event set of the generator G, is partitioned, $\Sigma = \Sigma_{uc} \cup \Sigma_c$ where Σ_{uc} is the set of uncontrollable events and Σ_c is the set of controllable events.

The graphical representation of a hypothetical machine modeled by a FSM is pictured in Fig. C-1. In the figure, states are labeled and are represented by the nodes of the graph. The arcs connecting the nodes are state transitions and are labeled with events. A single-ended arc pointing into a node denotes the initial state. Marked states are denoted by a double circle.The initial state of the system, x_0 is the Idle state (labeled I), indicated by the arrow pointing in; it is also a marked state. Other states in the system are Working (W, marked), Down (D), and Scrapped (S), $X = \{I, W, D, S\}$. The marker states are $X_m = \{I, W\}$. The machine's event set is $\Sigma = \{\alpha, \beta, \kappa, \lambda, \mu, \nu\}$, with the event labels representing the following actions:

α the machine starts, Idle to Working,

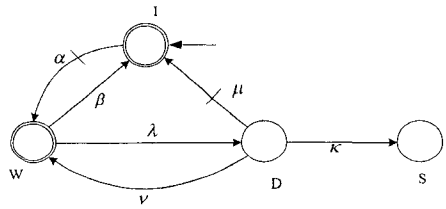


Figure C-1: FSM representation of a simple machine.

β completes its work and returns to the Idle state,

λ breaks down, going to the Down state,

μ gets repaired, returning to the Idle state,

ν gets repaired, returning to the Working state,

κ is scrapped, moving to the Scrap state.

The stroke through the α and μ arcs denotes that these transitions are controllable, in the sense that they can be disabled through the action of some controller. Hence, $\Sigma_C = \{\alpha, \mu\}$, $\Sigma_U = \{\beta, \kappa, \lambda, \nu\}$, and $\Sigma_C \cap \Sigma_U = \emptyset$.

C.1.1 Combining Multiple Automata

More complex system behaviors can be modeled by combining multiple machine models using synchronous composition. Two operations are discussed below.

C.1.1.1 Product

The product operation is denoted by the operator $\times$. Given two automata, $G_1 := (X_1, \Sigma_1, \Delta_1, x_{0,1}, X_{m,1})$ and $G_2 := (X_2, \Sigma_2, \Delta_2, x_{0,2}, X_{m,2})$, the product , $G_1 \times G_2 := (X, \Sigma, \Delta, x_0, X_m)$, they may execute a common event in $\Sigma_1 \cap \Sigma_2$ concurrently. For the product, it can be shown that $L(G_1 \times G_2) = L(G_1) \cap L(G_2)$ and $L_m(G_1 \times G_2) =$

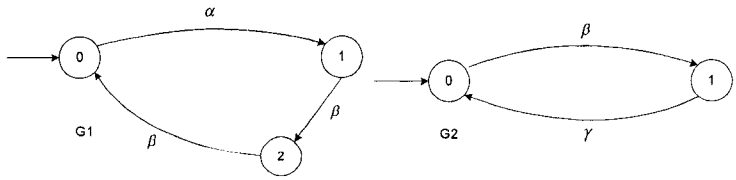


Figure C-2: Two simple FSM models with $\Sigma_1 = \{\alpha, \beta\}$ and $\Sigma_2 = \{\beta, \gamma\}$.

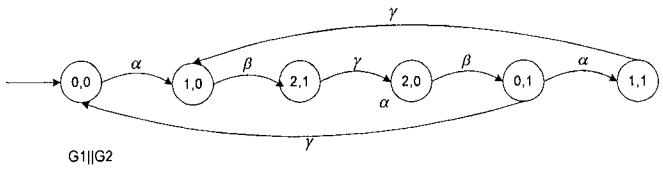


Figure C-3: The synchronous product $G_1||G_2$ of automata in figure C-2.

$L_m(G_1) \cap L_m(G_2)$. Referring to the two automata pictured in Fig. C-2, with $\Sigma_1 = \{\alpha, \beta\}$ and $\Sigma_2 = \{\beta, \gamma\}$ it is clear that the product will be $L(G_1 \times G_2) = \{\epsilon\}$ and the marked language is $L_m(G_1 \times G_2) = \{\epsilon\}$. If either of the initial states in G_1 or G_2 was not marked, the marked language would reduce to $L_m(G_1 \times G_2) = \emptyset$. The first event in G_1, α, is prevented from occurring because G_2 does not share that event; therefore they are blocking each other from further execution.

C.1.1.2 Synchronous Product

For control system synthesis, the synchronous (parallel) product is useful for making system interconnections. In the synchronous product, the automata must synchronize on common events but events private to each automaton, $(\Sigma_1 \setminus \Sigma_2) \cup (\Sigma_2 \setminus \Sigma_1)$, are allowed to occur at any time within their respective automata. Referring again to Fig. C-2, the synchronous product operation produces the resulting automaton in Fig. C-3. In the simple product, both automata were blocked from executing because the initial transition of G_1, α, was not a shared transition and so could not be executed. The synchronous product allows G_1 to execute α, which then allows both automata

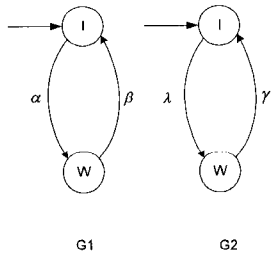


Figure C-4: Automata for shuffle product.

to synchronously execute the β transition and so on. It should be noted that the state names for product automata are derived from the state names of the respective composing automata as follows : (x_1, x_2) where x_1 is the current state in G_1 and x_2 is the current state in G_2. Controllable transitions in $\Sigma_1 \cap \Sigma_2$ will be disabled together even if the transition is prevented from occurring in one of the composing automata only due to the fact that they must be executed synchronously.

There are two special cases of the synchronous product. The first occurs when $\Sigma_1 \cap \Sigma_2 = \emptyset$. In this case, all events are private to their respective automata and the synchronous product, $G_1 \parallel G_2$ is called the shuffle product, since it consists of all possible shuffles of the respective alphabets, Σ_1 and Σ_2.The second special case is when $\Sigma_1 = \Sigma_2$, with the synchronous product ($\|$) reducing to the product ($\times$), since all transitions will be executed synchronously:$L(G_1 \parallel G_2) = L(G_1) \cap L(G_2)$ and $L_m(G_1 \parallel G_2) = L_m(G_1) \cap L_m(G_2)$.Fig. C-4 illustrates how the shuffle product is the result of the synchronous product operation, since $\Sigma_1 = \{\alpha, \beta\}$ and $\Sigma_2 = \{\gamma, \lambda\}$, $\Sigma_1 \cap \Sigma_2 = \emptyset$; two machines that perhaps work on their respective tasks side by side. The synchronous product of these two devices is graphically represented in Fig. C-5.

The synchronous product is a powerful tool, since it reduces the task of composing complex systems to that of designing the individual component automata and then synchronizing them together to produce the entire system behaviour. The same technique can be extended to more than two automata easily since the properties

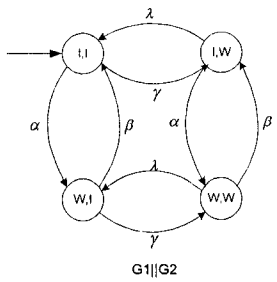


Figure C-5: Shuffle product of automata from figure C-4.

of commutativity and associativity hold for the synchronous product operation:

$$G_1 \parallel G_2 = G_2 \parallel G_1$$

$$G_1 \parallel (G_2 \parallel G_3) = (G_1 \parallel G_2) \parallel G_3$$

Very complex DES models can be constructed by combining multiple FSMs in this way.

Appendix D

DES Control Synthesis

Control systems engineers wish to have automated methods by which they can synthesize a controller that modifies the behavior of their target (the plant) within the limitations of the plant (controllability). Manual methods of synthesis can work for small DES systems, but an algorithmic approach to synthesis is needed. The DES control problem is analogous to the continuous control problem (See Fig. D-1), except that the system communicate via symbols (discrete values) instead of with continuous signals. The environment, an uncontrollable element, seeks to disturb the plant, thus presenting another obstacle for the controller.The environment, usually unmodeled, actively disturbs the plant and is given, which in this sense means we are aware of how it affects the plant. The three systems are closely coupled in an embrace where each affects the other. If each of these elements are viewed as agents in a DES, then

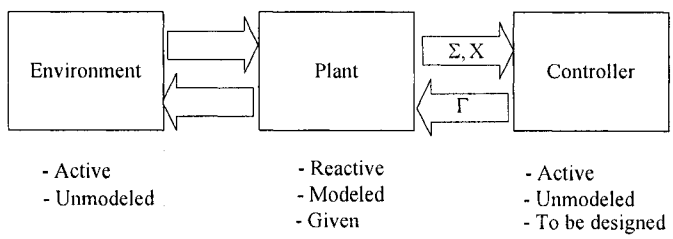


Figure D-1: Conceptual visualization of the control process.

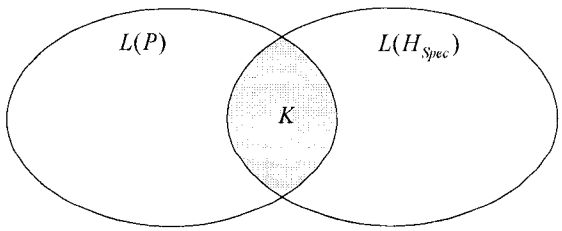


Figure D-2: Linguistic interpretation of legal language $K = L(P) \cap L(H_{spec})$ by synchronous product of plant and specification automata.

their interaction can be modeled using FSM models connected through some form of product connection.

A method of formally synthesizing a supervisory controller for discrete event systems (DES) was reported by Ramadge and Wonham (Ramadge and Wonham 1989), (Ramadge and Wonham 1987), and will be referred to as the RW synthesis from here on. In this framework, a finite state automaton model $P := (X_P, \Sigma_P, \Delta_P, x_{0,P}, X_{m,P})$, generating language $L(P)$, completely describes the behavior of the discrete event plant that is to be controlled. A second automaton model, H_{spec}, generating language $L(H_{spec})$, represents the specification or desired (controlled) behaviour. The synchronous product of these two automata produces an automaton that generates the legal language, $K = L(P) \cap L(H_{spec})$, of the closed-loop process (Fig. D-2).

The task of synthesizing a discrete event supervisor (controller) S, is to enforce the legal language on the plant P. The supervisor is also a FSM:

$$S := (X_S, \Sigma_S, \Delta_S, x_{0,S}, X_{m,S})$$

The supervisor accomplishes this by monitoring the plant, and disabling the controllable transitions in order to preempt actions by the plant that could uncontrollably cause the system to violate the legal language. The closed-loop connection of a DES supervisor and plant is shown in Fig. D-3. The supervisor is able to monitor $s \in L(P)$, the string (a trace) of all events executed so far by the plant. The closed-loop lan-

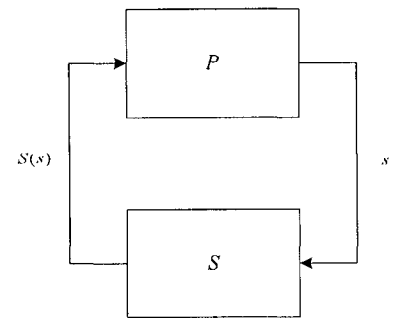


Figure D-3: Closed-loop interconnection of supervisor S controlling plant P by disablement.

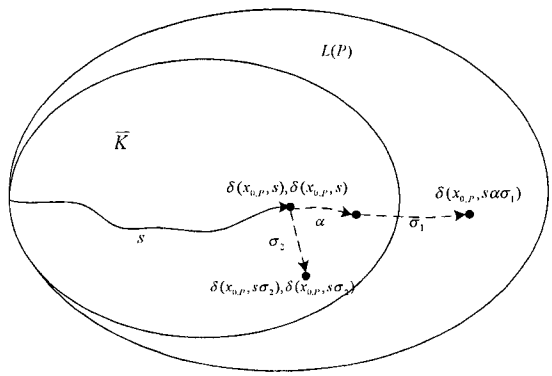


Figure D-4: Diagram illustrating controllability of K with respect to P.

guage, $L(S/P) \subseteq K$. $L(S/P)$ is read as "S controlling P". The string s has taken P to a state $\delta(x_{0,P}, s) \in X_P$ and supervisor S to state $\delta(x_{0,S}, s) \in X_S$ (Fig. D-4). At this state in $S \parallel G$, $(\delta(x_{0,S}, s), \delta(x_{0,P}, s))$, an uncontrollable event σ_1 and a controllable event, α are executable. The supervisor must disable the controllable event α that leads subsequently to the uncontrollable event $\sigma_1 \in \Sigma_P$ that causes the plant to violate the legal language. This is accomplished by excluding the α event from the supervisor and since the two systems are synchronized, the plant is prevented from executing. The supervisor cannot disable σ_1 or σ_2, because by definition, uncontrollable events cannot be disabled, in any case, σ_2 does not violate the legal language.

This implies the controllability condition which states that a supervisor S exists

such that $L(S/G) = \overline{K}$ if and only if:

$$\overline{K}\Sigma_{uc} \cap L(P) \subseteq \overline{K}$$

where $\overline{K}$ is the prefix closure of K. This is also stated that the language K is *controllable* with respect to plant P and Σ_{uc}.

D.0.2 Supremal Controllable Sublanguage - Safety Guarantee

Now suppose that the K we have formed is not controllable with respect to P, i.e. $\overline{K}\Sigma_U \cap L(P) \not\subseteq \overline{K}$, then what is the largest sublanguage of K that is controllable? This is known as the supremal controllable sublanguage, denoted as $K^{\uparrow C}$. Define the class of all controllable sublanguages of K:

$$C_{in}(K) = \{L \subseteq K : \overline{L}\Sigma_U \cap L(P) \subseteq \overline{L}\}$$

The union of all controllable sublanguages must be the largest, or supremal controllable sublanguage:

$$K^{\uparrow C} := \bigcup_{L \in C_{in}(K)} L$$

The goal of RW synthesis is to find an automaton that enforces $K^{\uparrow C}$ for a particular plant. This automaton is known as the supremal controllable sublanguage generator (SCSG). The SCSG is the optimal controller for a particular plant and specification since it disables the fewest number of events (most permissive) in the plant to ensure the legal language (safety) is not violated. It is analogous to the $\mathcal{H}_\infty$ controller of the previous section in this sense being designed to handle the worst case and hence guaranteeing safety.

D.0.3 Infimal Controllable Sublanguage - Performance Guarantee

Defining the class of prefix-closed, controllable superlanguages:

$$C_{out}(K) = \{L \subseteq \Sigma^* : (K \subseteq L \subseteq L(P)) \wedge (\overline{L} = L) \wedge (\overline{L}\Sigma_{uc} \cap L(P) \subseteq \overline{L})\}$$

The infimal controllable superlanguage is the intersection of all controllable superlanguages:

$$K^{\downarrow C} := \bigcap_{L \in C_{out}(K)} L$$

If K represents the minimal required language of a plant as opposed to the maximum legal behaviour, then the automaton that enforces $K^{\downarrow C}$ is the smallest (optimal) controller that can be guarantees that the closed-loop system will meet this specification. Optimal with respect to performance.

To summarize the relationship between each of the languages:

$$\emptyset \subseteq K^{\uparrow C} \subseteq K \subseteq \overline{K} \subseteq K^{\downarrow C} \subseteq L(P)$$

where

$K^{\uparrow C}$ is the supremal controllable sublanguage,

K is the maximum legal language, or minimum performance,

$\overline{K}$ is the prefix closure of K,

$K^{\downarrow C}$ is the infimal prefix-closed controllable superlanguage,

$L(P)$ is the plant language.

D.0.4 Nonblocking Controllability

A desirable feature of a supervisory controller is nonblocking behaviour, also known as liveness. So in addition to the controllability condition on the legal language, $\overline{K}\Sigma_{uc} \cap L(P) \subseteq \overline{K}$, the legal language must also be non-blocking:

$$K = \overline{K} \cap L_m(P)$$

That is, the controller should not able to "stop" execution at an unmarked state in P. So the following is true of a nonblocking supervisor:

$$\begin{aligned} L_m(S/P) &= (L_m(P) \cap L_m(H_{spec}))^{\uparrow C} \\ L(S/P) &= \overline{(L_m(P) \cap L_m(H_{spec}))^{\uparrow C}} \end{aligned}$$

D.1 DES Controller Synthesis Software

Several software packages are available that implement the RW synthesis, including TCT, OTCT and UMDES (University of Michigan DES library). The first package was TCT, developed at the University of Toronto. A similar and related software package is OTCT, which has a script-based interface as opposed to a menu-based GUI. OTCT lends itself better to automated operation, since jobs are submitted in as a batch using the scripting language. Automata are specified using a text file description. Once read into the OTCT workspace, the automata are represented as objects and can be manipulated with various functions. The three that shall be considered here are:

- **sync(x,y):** forms the synchronous product of two argument objects, x and y,
- **supfcBySync(x,y):** computes the maximally permissive controller given the plant,x, and the generator of the specification, y,

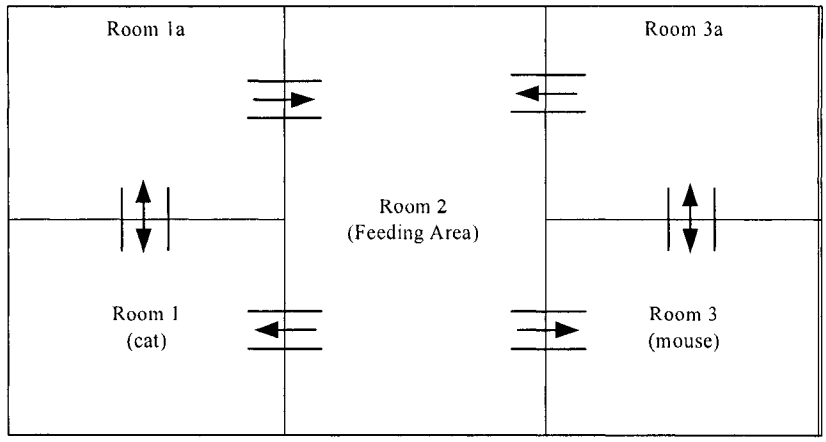


Figure D-5: Cat and mouse "toy" supervision problem, adapted from (Ramadge and Wonham 1989).

- **condat(c):** computes the controller state feedback map for the controller c.

To illustrate RW synthesis, consider the simple example illustrated by Fig. D-5, in which a cat and mouse share a house. Each animal starts in a separate room that is accessible only to itself, but they share a common feeding area. The entrances to the feeding area can be disabled by the controller (which is to be designed). If the two animals occupy the feeding room (area 2) together, the cat will eat the mouse. The specification is that the cat eating the mouse must be prevented in a least restrictive way, i.e. the doors to the feeding area are disabled only when they have to be. The cat executes events $\Sigma_c = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ and the mouse executes $\Sigma_m = \{\beta_1, \beta_2, \beta_3, \beta_4\}$. Motion detectors provide information regarding the movements of the animals between their anterooms (areas 1A and 3A), and their 'home' areas (areas 1 and 3). Events $\alpha_1, \alpha 4$, and β_1, β_4 signify these moves by the animals. Controllable transitions are α_2 and β_2, which are the events of the two respective animals entering the common feeding room. The assumption is that the gate can be disabled instantly (or, at the very least, before each animal acts). For the time being, this appears to be a reasonable assumption, since a controller is

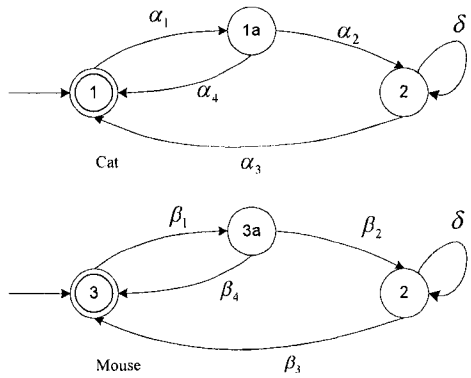


Figure D-6: Cat and Mouse automata.

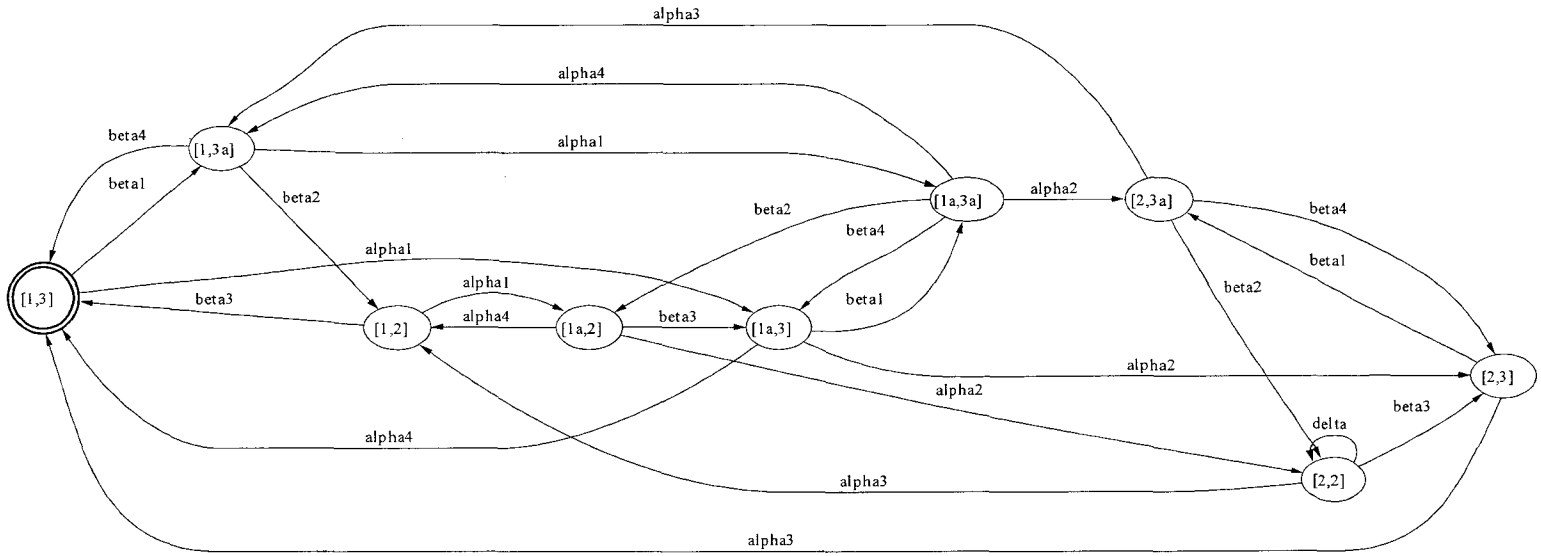
likely to be much faster than the physical surroundings. Events α_3 and β_3 are the events signifying the animals leaving the common feeding area. The diagram has been transferred into functional automata in Fig. D-6. An additional event has been added, δ, which can only be executed should both animals reach state 2 (the joint feeding area).

By inspection of the synchronous product automaton (Fig. D.1), it is clear that we want to avoid entering the product state (2,2). Linguistically speaking, this means that we which to avoid any strings in the product language L_P, that lead to this state or that end in a δ. The controller needs to trim this state, which can be done by disabling the controllable transitions that lead to it, α_2 and β_2 The controller for this plant can be generated by running the *supfcBySync* procedure ion the synchronous product of the plant and the specification (the specification is given in Fig. D-7).

The resulting controller is presented in Fig. D-8, indicating that the (2,2) state has been trimmed. Running the *condat* function on the supervisor, results in the following trace:

```
Control Data:


PLANT: [2,$q_{2}$]
```



Plant formed from the synchronous product of cat and mouse automata.

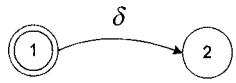


Figure D-7: The specification indicates that the δ event must be excluded since it takes the system away from a marked state.

```
SUPER: [[2,$q_{2}$],1]
DELAY: $\beta _{2}$
PLANT: [$q_{2}$,2]
SUPER: [[$q_{2}$,2],1]
DELAY: $\alpha _{2}$
```

A program could be constructed to implement this controller, assuming that the state of the plant can be observed at all times, based on the controller map:

```
begin control{
    if plant_state = [2,$q_{2}$] then disable $\beta _{2}$;
    if plant_state = [$q_{2}$,2] then disable $\alpha _{2}$;
}end control
```

D.1.1 Timed Discrete Event Models

RW supervisory synthesis theory was extended by O'Young and Brandin and Wonham (Brandin and Wonham 1992) by applying the same techniques to timed finite state automata or timed transition model (TTM). While the pure DES model has the system dynamics completely abstracted, this extension allows for some of the dynamics of the modeled system to be included, since time has been added. Fig. D-9 shows a TTM version of the cat automaton of the previous section (p. 256). The notation $\alpha_2[2, \infty]$ means that the α_2 event is only admitted only after 2 time periods ('ticks') have transpired. So in this example, the cat must stay at state q_2 at least 2 ticks before the α_2 transition can be taken to state 2. The FSM equivalent of this

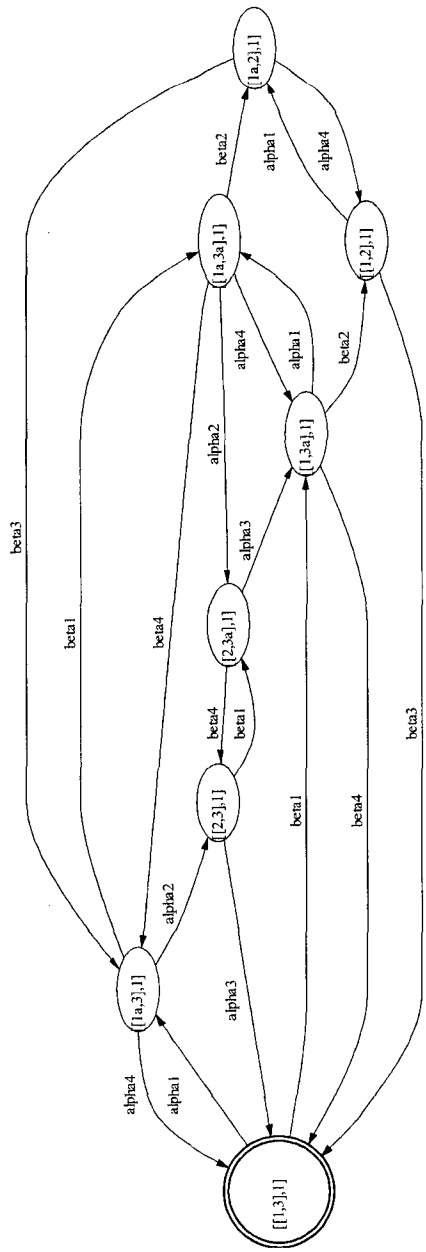


Figure D-8: Supervisor for the Cat/Mouse system that enforces mutual exclusion in a maximally permissive (optimal) sense.

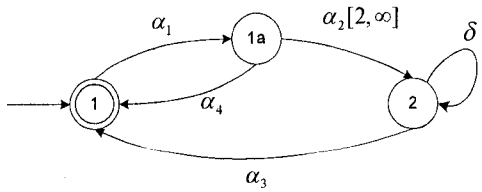


Figure D-9: A TTM model of the cat automaton.

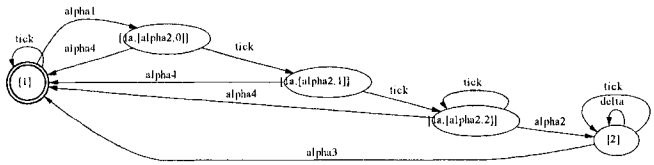


Figure D-10: DES equivalent model of cat of timed automaton.

timed discrete model is shown in Fig. D-10. The equivalent is produced by adding states to delay timed transitions by the appropriate number of ticks. State labeling for these added states is derived from the transition name and the number of ticks that have occurred. States that do not have timed transitions leading from them must be self-looped with tick transitions to prevent blocking with the passage of time.

This extension to FSMs is a simple way of incorporating timing in a DES model. DES system can be comprised of a combination of FSMs and TTMs easily by self-looping states in FSMs with tick events. Due to the added states for the timed transitions, such models tend to have many more states than simple FSM models. Care must be taken to only use timed transitions when necessary, to reduce model complexity.

Appendix E

Hybrid System Modeling

E.1 Hybrid Automata

Hybrid system modeling is a formal method of modeling that attempts to capture both the discrete and continuous properties of a system. The state of a hybrid system having n real variables and m boolean variables at any moment in time corresponds to a point in the state space of the system $\mathbb{R} \times \mathbb{B}$. The ideal hybrid model allows the dynamics of a model to change in a discrete fashion, so as to model the failure of a component or a sudden switch in operating points of a system. The hybrid automaton (Fig. E-1) was proposed as a model of such a hybrid system.

Definition E.1.1 *A hybrid automaton model is an eight-tuple:*

$$H = (X, L, T, F, inv, jump, \Sigma, init)$$

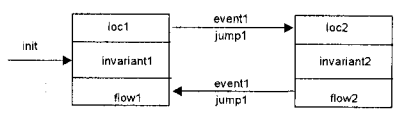


Figure E-1: A hybrid automaton.

where:

X is a finite set of n continuous variables in $\mathbb{R}$, $X = \{x_1, x_2, \ldots x_n\}$

L is a finite set of j locations (states), $L = \{loc_1, loc_2, \ldots loc_j\}$

T is a finite set of discrete jumps between locations

F is the set of flow conditions

inv is a labeling function that assigns an invariant at each location. An invariant is a flow constraint used to force a transition to another state.

jump is a labeling function that is attached to each jump

Σ is a finite set of synchronizing events

init is a labeling function assigning an initial condition at each location.

The flow conditions describe how the continuous variables evolve through time. Ideally, these could take on the usual differential equation form of $\dot{x} = f(x, t)$, however such a flow condition is not feasible for algorithmic verification, since it has been shown to be undecidable. If the flow conditions are changed to constant rate; i.e. $\dot{x} = a$, then the verification is semidecidable. A hybrid automaton having such constant-rate flow conditions is known as a linear hybrid automaton (LHA).

At least one software tool exists for conducting automated analysis and verification of hybrid systems, called HyTech (Henzinger et al. 1997). Systems described by LHA's can be verified for correctness against some specification that is a combination of discrete and continuous conditions. Traces of the path to the correct (or incorrect) system state can be generated by either reaching forward or backward. Additionally, one powerful tool is the ability to parameterize invariant conditions, a useful design tool.

Appendix F

HySynth: Hybrid Control Synthesis Software Package

F.1 Introduction

The HySynth software package was developed to implement the concepts that have been set forth in this thesis. HySynth is an object-oriented package that allows for hybrid system modeling and controller synthesis, hence the name Hy (for Hybrid) and Synth (for Controller Synthesis). The package allows for the modeling and synthesis of controllers using a combination of SC models and FS models. HySynth was developed within the Mathworks' Matlab environment, thus allowing it to take advantage of the numerous ODE solvers and the many other toolboxes that are available to Matlab users. This permits a designer the flexibility to embed existing Matlab simulations within the switched continuous model framework.

F.2 Brief Overview

The HySynth package has been developed using the object-oriented features of Matlab. A number of objects are available for constructing hybrid models, and various

commands are available to manipulate models and to synthesize and simulate controllers. The methods associated with each object extend the basic functionality of Matlab script. Application examples of HySynth are detailed in Chapter 7.

F.2.1 Objects

The modeling objects are:

`fsm` This is the basic DES building block, the finite state automaton model. An `fsm` object can be constructed graphically by the JFLAP GUI and stored to an XML file. The constructor method is then invoked to instantiate the object; e.g. `m1=fsm('m1.xml');`. Once created, the object is referred to by its variable name, `m1`. Alternatively, the finite state machine object can be created programmatically by defining the various sets: the state set, event set, transition function, and so on.

`scm` The basic switched continuous system model. Currently no GUI has been designed for the scm object, so it must be constructed using a Matlab script. The user defines a set of continuous dynamics, as Matlab function handles, which point to the desired continuous system models. The continuous systems are captured as Matlab functions in the form of a generalized nonlinear time-varying differential equation; i.e. $\dot{x} = f(x, t, params)$. A corresponding set of input event labels should be defined, one label for each continuous system model. Partitioning functionals are defined via an event function that implements the event detection in conjunction with the Matlab ODE solvers. For each partition, a set of output event labels must be defined.

`product` The synchronous product object, a hierarchical object made up of `fsm`, `scm` or other `product` objects. For example, given a pair of existing FSM objects, `m1` and `m2`, the product constructor function is used as follows to create their product: `p1 = product(m1,m2)`.

The state objects are:

`finiteState` The finite state object, a finite state label, a Matlab string.

`ctsState` The continuous state object, storage of continuous (state) variables $x \in \mathbb{R}^n$, a Matlab double vector.

`pstate` The product state object. A hierarchical object, made up of two or more `finiteState`, `ctsState`, or other `pstate` objects.

F.2.2 Commands

Once a system model has been constructed, various methods are available to modify it, analyze it, and to view its structure. This section contains a partial listing of commands that a user needs to create a controller structure.

Commands that apply to `fsm` objects only:

`addEvents` adds the argument list of events to the event set

`markAll` mark all of the states as ESD states.

`unMarkAll` unmark all ESD states.

`addEvents` adds the argument list of events to the event set

Commands common to all modeling objects:

`initial` returns the initial state of a system model

`printAsDot` print the argument object as a Graphviz format file in .dot format to the argument filename.

`printAsPs` print the argument object as an Adobe PostScript format file to the argument filename.

`printAsPDF` print the argument object as an Adobe PDF format file to the argument filename.

`printAs*WithEvents` print limited event lookahead (reachability) of the argument object to the argument filename. Prints to one of three file formats, replace wildcard * with one of `Dot`, `Ps` or `PDF`.

`printAs*WithTime` print limited time lookahead (reachability) of the argument object to the argument filename. Prints to one of three file formats, replace wildcard * with one of `Dot`, `Ps` or `PDF`.

`simulate` given an argument input event, performs a prediction until the next choice point.