Synthesis and Verification of Discrete Controllers for Robotics and Manufacturing Devices with Temporal Logic and the **Control-D** System.

by

Marco Antoniotti

September 1995

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University

Approved: ______________________

Professor Bhubaneswar Mishra

Research Advisor

© Marco Antoniotti

All Rights Reserved 1995

To my sister Franca,

who made this a better world

To my nephew Tommaso and my niece Agnese,

who will make this a better world

To Mary,

who makes this a better world

Acknowledgments

I am really indebted to my advisor Bud Mishra for his continuous support and friendship during my years as a graduate student. He has been a been a guide through fields that were unknown to me and has been a guide during troubled times. There are still many fields I will need to wander through and I am sure I will find his traces there.

I want to thank Professor Mohsen Jafari of Rutgers University for his help with the CRAMTD project and for kindness and enthusiasm since our first meeting at a conference.

A person who deserves much credit for my achievements is Stefania Bandini of the Universitaá degli Studi di Milano, who has always been able to keep me on my toes and to come to my aid whenever necessary. She has always been the source of precious advice.

Dr. R. Kurshan of AT&T Bell Laboratories was very kind and helpful during the final phase of my thesis by providing valuable information on many technical details and developments in the theory of related topics.

I want to thank Professor J. T. Schwartz for his valuable comments on my work.

Professor R. Wallace is the person who introduced me to the Walking Machine problem. He designed and worked on the hardware and was very helpful in giving me feedback in order to keep my research firmly set on a pragmatic foundation.

Friends like Giovanni Gallo and Alberto Policriti are a blessing for anybody who is doing research and who needs a companion to sneak out to the movies every once in a while.

I want to thank Professor K. Perlin, Professor S. Mallat, Professor D. Shasha and Professor R. Boppana for their kindness and for their suggestions over the past years.

Special thanks go to many people in Milan who have in various ways contributed to my success as a graduate student: Professor G. Mauri of the Dipartimento di Scienze dell'Informazione of the Universitaá degli Studi di Milano and all the friends at Quinary S.p.A., where the machines are named after Lisp functions.

Thanks are also due to Anina Karmen-Meade of the Department of Computer Science of NYU for her patience with my overly complicated administrative problems and to Fred Hansen for all his work in the Robotics Research Laboratory.

A group of people who deserves thanks and other things is constituted by my roommate over the years: Sunder Sethuraman, a great mathematician and a better soccer player than I'll ever be; David Bacon, Ron Even and Marek Teichmann, my three office mates who endured me and Toto Paxia, who *I* had to endure.

Finally, I thank Mary, whom I love, who loves me and, luckily, thinks the factorial of any number is 7.

Contents

1 Introduction **1**

2 Manufacturing, Robotics, Control Theory and Verification **5**

2.1 Models and Control 6

2.2 Classical Control Theory 8

2.3 Control Theory Applied to DES 12

2.3.1 Building Supervisory Controllers for CDES 14

2.4 Verification and Temporal Logic 20

2.4.1 Temporal Properties of Systems 21

2.4.2 Temporal Logics: Linear and Branching Time 23

2.4.3 Theorem Proving and Model Checking 28

2.4.4 Verification with Automata Theory 32

2.4.5 Reductions and Compositional Verification 33

2.4.6 Real Time Verification 33

2.4.7 Role of Verification Tools 34

2.5 Hybrid Systems Verification and Control 35

3 Temporal Logic Supervisor Synthesis with Verification **37**

3.1 Notational Preliminaries 39

3.1.1 CDES Notions and Notations 43

3.2 CTL Direct Synthesis of Supervisors 44

3.2.1 *Controlled* Semantics as *Model Restriction* 46

3.2.2 Main Objective and Notable Problems 47

3.2.3 Inductive Construction of the Controlled Semantics . . 54

3.2.4 Circumventing the Problems 77

3.2.5 CTL Soundness of Controlled Semantics 79

3.3 Simple CTL-Synthesis Algorithm 87

3.3.1 Existence of a Supervisor Synthesis Algorithm 88

4 The Control-D System **90**

4.1 Revised Algorithm and The Implementation 91

4.1.1 State Space Traversing Algorithm 92

4.1.2 Tradeoffs in the Treatment of Disjunctive Forms 99

4.1.3 Comparison with Standard Supervisor Synthesis Algorithm 102

4.1.4 Open Problem: *Symbolic* Representation 107

4.2 Verification and Synthesis 108

4.3 The Environment 108

4.3.1 Control-D Components and User Interface 109

5 Building a Discrete/Hybrid Controller for a Walking Machine 116

5.1 A Brief History of Walking Machines 117

5.2 Problems for Walking Machines 118

5.2.1 Leg Behavior Models and Gaits 119

5.2.2 Stability of Walking Machines 123

5.2.3 Synchronization and Real-time Control 123

5.3 Building Walking Machines Controllers with Control-D 124

5.3.1 Discrete Controller for the Walking Machine 127

5.3.2 Continuous Control Constraints 129

6 Ensuring Failure Behavior for the CRAMTD Manufacturing Line 134

6.1 CRAMTD Project Tray Packing Model 135

6.2 Failure Behavior Control 136

6.2.1 Modeling the Tray Packing Line 138

6.2.2 CTL Specification of Behavior 138

6.3 Concluding Remarks 143

7 Conclusion and Future Work 144

List of Figures

1.1 The walking machine example 3

1.2 Schematic drawing of the tray pack line of the CRAMTD project 3

2.1 The standard arrangement of plant $\mathcal{G}$ and supervisor $\langle \mathcal{R}, \varphi \rangle$. . 15

2.2 Schematic flowchart of the Supervisor building steps. 17

3.1 A case illustrating the requirements for **AX** supervisors 49

3.2 A counterexample for the intuitive supervisor map construction for disjunctions 51

3.3 Restrictions on the language satisfying $\mathbf{A}[b\,\mathbf{U}\,a]$. 55

3.4 Controlled Semantics for $\neg f$ without uncontrollable events . . 58

3.5 Controlled Semantics for $\neg f$ with uncontrollable events 60

3.6 An illustration of the problems with the controlled semantics for negation 61

3.7 A problematic case involving the supervisor map assignments needed to satisfy $\mathbf{AX}(p)$ and $f \equiv \mathbf{A}[\mathbf{AX}(p)\,\mathbf{U}\,q]$. 69

3.8 A problematic case for the semantic of the **EU** and **AU** operators. 83

4.1 Schematic of the Model Restriction Supervisor Synthesis (MRSS) Algorithm 93
4.2 The **label_state_graph** procedure 95
4.3 The function **label_AU**. 96
4.4 The procedure **label_AU_next**. 98
4.5 The procedure **model_synth** 98
4.6 A snapshot of the Control-D graphical environment 110
4.7 Control-D edit window 113

5.1 State diagram representing the movement of a single leg. 119
5.2 Hexapod tripod gait 121
5.3 Hybrid Controller Architecture for Walking Machine 125
5.4 Mini Actuator Leg Prototype 1 126
5.5 The FSM model of a Leg with uncontrollable event slip 127
5.6 Schematic representation of the walking machine 128
5.7 Simplified Geometric Model of the Walking Machine 130

6.1 Schematic of the tray packing line of the CRAMTD project . . 135
6.2 FSM model for the CRAMTD Check Weight Station 139
6.3 FSM model for the CRAMTD Reject-diverter 140
6.4 Model of the measurement communication 140
6.5 Control-D window with CRAMTD constraints 142

List of Tables

2.1 Syntax and informal Semantics for PLTL 26

2.2 Syntax and informal Semantics for CTL 27

Chapter 1

Introduction

A person's "productivity" can be measured in many ways. However, independent of how this productivity is measured, it appears that the advent of modern computing devices and software architectures has improved this ability by several factors of magnitude[1].

One of the aims of computer science has been to provide better software tools that would increase our productivity (and hopefully, leave us more time to engage in leisurely activities such as philosophizing or fishing). Two desirable characteristics of such tools are their "usefulness" in solving a given problem and the "soundness" of the theory on which they are founded.

[1] In spite of the fact that computers have been around for decades, there seems to be a *folk consensus* that this result has been actually achieved only recently outside research institutions and very high-tech industries (cfr. *Business Week*, July 17th, 1995: cover story on *Wages in America*).

This dissertation will describe how problems in robotics and manufacturing prompted the development of a theory of *Discrete Event Systems* (DES), how this theory has been applied and how a change in perspective leads to the construction of algorithms and tools which improve on the original formulation.

Two motivating examples form robotics and manufacturing will be used to substantiate the claims to be made. The first one is an application to the problem of synthesizing a controller for the synchronization of leg movements of a walking machine. The second is an application to the construction of control software for a food processing manufacturing line. (The latter work has been carried out in collaboration with the Department of Industrial Engineering of Rutgers University).

In the walking machine example the challenge is to build a discrete controller capable of producing reasonable gaits which follow simple principles of coordination. Moreover, the system presents many problems which have been recently addressed as "hybrid" between the continuous and the discrete viewpoints. Figure 1.1 shows a sample graphic output of the system. Chapter 5 contains a more thorough description of the example.

Manufacturing systems pose very interesting problems from the viewpoint of coordination and failure detection. Figure 1.2 shows a schematic of a *tray packing* line of the Combat Ration Automated Manufacturing Technology Demonstration (CRAMTD) of Rutgers University. Chapter 6 contains a more

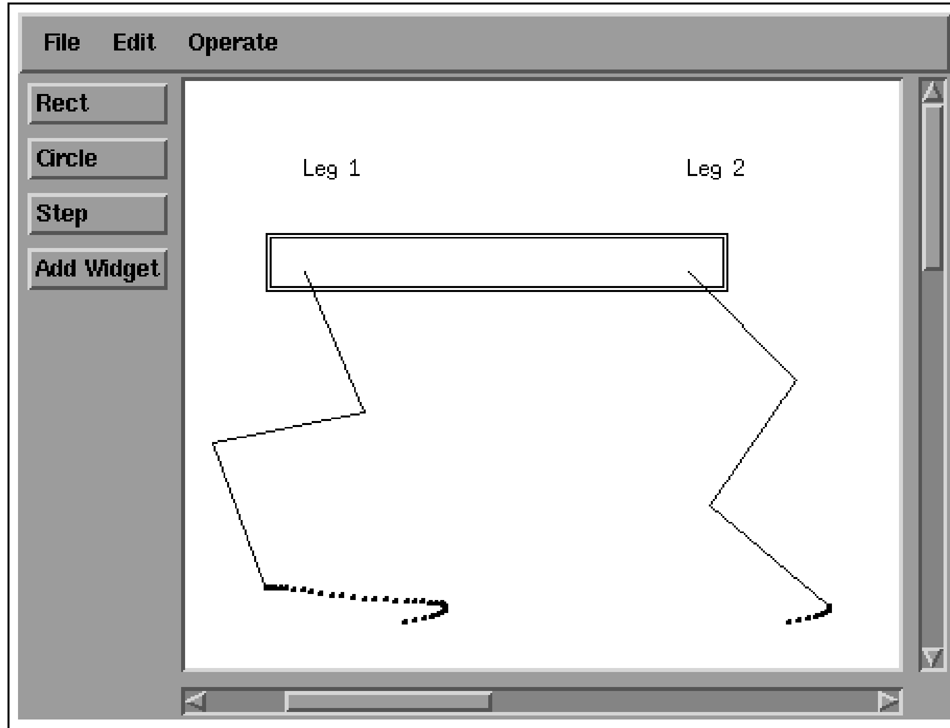

Figure 1.1: *The walking machine example.*

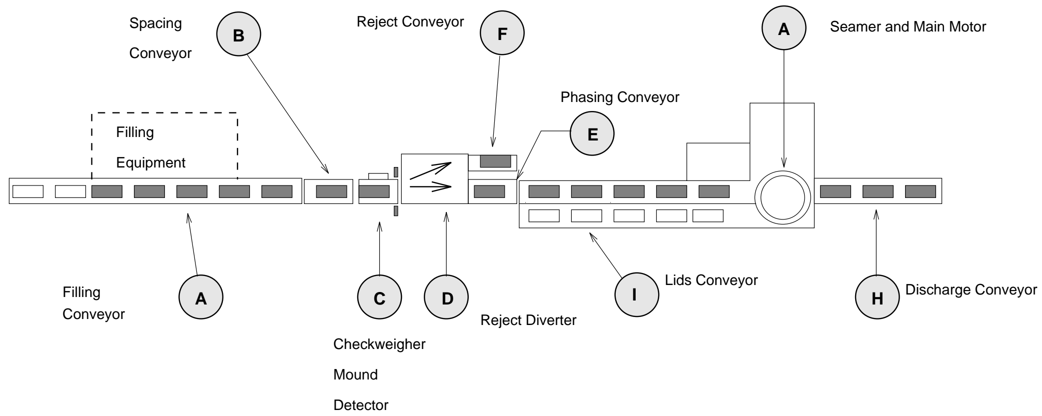

Figure 1.2: *Schematic drawing of the tray pack line of the CRAMTD project.*

detailed description of the application.

The CRAMTD system is composed of many subsystems, each of which can be represented by a finite state model. The result of the controller synthesis process for such a plant is packaged in intermediate format which can be translated into a variety of "programmable logic controllers" (PLC) and other software tools. In this case, the problems arise from the combinatorial explosion due to the nature of the model.

The dissertation is organized as follows. Chapter 2 contains an introduction that provides contexts for the developments described later in the dissertation. It contains a brief literature review and a description of standard tools: e.g. *control theory*, Ramadge and Wonham's theory of *controlled discrete event systems*, *temporal logic*, and *verification*. Chapter 3 contains a novel interpretation of propositional temporal logic as a tool for the synthesis of controllers for discrete event systems. Chapter 4 describes an algorithm for controller synthesis and its implementation using the Control-D tool-set, and then compares it with the "standard" algorithm for the same problem. Chapters 5 and 6 contain the descriptions of the walking machine and food manufacturing examples, respectively.

Chapter 2

Manufacturing, Robotics, Control Theory and Verification

This thesis studies the semi-automated synthesis and verification of control systems for robotics and manufacturing devices using formal methods in a discrete framework, and bears some resemblance to the theory of *controlled discrete event systems* (CDES) of Ramadge and Wonham's [67]. The aim is to study the integration of CDES theory with the techniques developed for the *specification* and *verification* of discrete event systems (DES). Many of these techniques rely on the application of some flavor of *temporal logic* [35].

This chapter begins with a description of the main concepts underlying the dissertation, followed by a review of the literature on various related subjects. It is worthwhile to note in advance that such review will necessarily be partial and incomplete, because of its the scope and the sheer amount of

material available in the literature.

2.1 Models and Control

The notions of *robotics* and *manufacturing* devices or systems encompass many artifacts which are currently employed in the production and servicing activities of the economy of many nations. Giving a precise definition of these terms is therefore a self defeating task. Nevertheless, science and engineering have provided many abstractions and tools to *model* such a wide variety of systems.

For the purpose of this dissertation, the robotics and manufacturing systems that will be considered, can be formally described in terms of *finite state machines* (FSM) [41]. Such discrete description will represent a certain "view" of the comprehensive behavior of the robotics or manufacturing system. I.e., it represents a *model* of certain behavioral aspects of the device. This model is *discrete* because by its nature the underlying formal tool (FSM) is. *Continuous* characteristics will be best described by different formal tools (e.g. *ordinary differential equations*, *linear* and *non-linear control theory*).

Models of robotics and manufacturing systems serve different purposes. They may simply be a description of the relationships among the components of a system, to be used as a documentation. More often, these models serve the purpose of analyzing the system in order to either *explain*, *predict*, or *control* its behavior. The nature of these models depends on which of these

tasks (or their combination) is to be performed.

Nowadays, explanation, prediction and control are supported by combinations of hardware and software embodied by computer systems. There is therefore a focus on those modeling tools that present desirable *computational* properties such as *low algorithmic complexity*.

This dissertation will focus on an approach to the control task based on a computational model whose characteristics make it relatively practical to implement while maintaining expressiveness. The basis of this computational model is the theory of CDES developed by Ramadge and Wonham in the mid 80's. This theory is deeply rooted in the classical *control theory* from which it borrows terminology and key concepts. It will be shown that by dropping the traditional control theory heritage, the resulting computational model gains in expressiveness and efficiency.

To substantiate these findings and demonstrate the feasibility of the approach, an implementation of the algorithms has been incorporated in the *Control-D* tool. The tool has been used to construct the controllers of two systems: a walking machine [12] and a manufacturing line [10].

Before discussing these results, an introduction to the key concepts as well as a (incomplete) review of the related (and vast) literature touching these subjects is in order.

2.2 Classical Control Theory

The main ideas used in controlling physical devices can be traced back to Maxwell's governor example which also introduced the concept of *feedback control* for *dynamical systems*. This traditional control theory approach has been extensively formalized (originally by Shannon, Kalman and others – see [44, 77, 71]) and has resulted in rich linear and nonlinear theories treating *continuous* laws.

These theories introduce many key concepts that will be briefly reviewed in this section, in order to clarify their use in other parts of this dissertation. The following brief introduction is closely patterned after [32, 57].

A very abstract view of control theory considers a system which evolves over *time* (T) while producing *outputs* (Y) in response to *inputs* (U). The outputs are also dependent on the system *state*. Usually, inputs, outputs and system state are denoted with $u(t)$, $y(t)$, and $x(t)$, i.e. as functions of time. The exact nature of these functions and the underlying representation of time are the basis for taxonomy of control models.

The evolution of the values produced by these functions are called *trajectories*, or *histories*, or *traces*. The focus is usually on *state-space* trajectories

$$H_X \triangleq \{h_X : T \rightarrow X\},$$

and of output histories

$$H_Y \triangleq \{h_Y : T \rightarrow Y\}.$$

The dependency of state $x(t)$ on the inputs is usually denoted by a *state transition equation*

$$x(t) = \phi(t, t_0, x(t_0), u(t)),$$

or by a *state evaluation equation*

$$\begin{aligned} x(t_0) &= x_0, \\ x'(t) &= f(t, x(t), u(t)). \end{aligned}$$

These functions are coupled with a *output equation*

$$y(t) = g(t, x(t), u(t)).$$

ϕ, f, and g are application dependent functions.

With these definitions it is now possible to formulate the *control problem* for a dynamical system by distinguishing a *goal* subset of the trajectories in H_X. Controlling a dynamical system is equivalent to constraining it within a trajectory in the goal subset. According to the nature of the goal subset, several flavors of the control problem can be formulated. Two classical ones are the *servo problem* and the *state-avoidance problem.*

Servo Problem. Given a reference trajectory $g \in H_x$, define the goal subset as

$$G = \{h_X \text{ such that } |h_X - g| \leq \epsilon, \epsilon > 0\}.$$

I.e. the trajectory g must be "tracked" as closely as possible.

State-avoidance Problem. Given a set of states $Q \subset X$, define the goal subset as

$$G = \{h_X \text{ such that } \forall t.h_X(t) \notin Q\}.$$

I.e. the trajectory never reaches a state in Q (provided that $x_0 \notin Q$).

In order to constrain the system within the bounds specified by the trajectory goal set G, actions must be taken. These actions are represented as inputs to the system and are referred to as *control laws* or *control policies*. Control laws are chosen given the "current" state of the system, or the the "current" output or both. I.e. *feedback* is used in order to decide the "next" input to the system.

Often, goal trajectories may be unattainable. Control theory defines notions and methods to describe this situation. The key notion is that of *controllability* of a system. A point in the time-space $\langle \tau, x \rangle \in T \times X$ is said to be *controllable* with respect to a set $C \subseteq X$ of target states if and only if there exists a control action γ such that at a "time" $t > \tau$ the resulting point $\langle t, x' \rangle$ in the time-space is within C.

$$\langle \tau, x \rangle \overset{\gamma}{\rightsquigarrow} \langle t, x' \rangle, \text{ and } \langle t, x' \rangle \in C.$$

A dynamical system is *completely controllable* if and only if it is possible to transfer any state $x(t_0) \in X$ to any other state in X in a *finite* amount of "time".

The definition of controllability relies on precise information about the state of the system. Such information may not be available. The problem of recovering such information is the *observability problem* of a system. Controllability and observability may be related by the so-called *duality principle*.

A wide range of physical systems can be described and controlled in terms of *linear, time-invariant* models. For this class of models, which are characterized by a linear matrix form of the defining state and output equations, a very rich set of sophisticated mathematical tools has been developed over the years. *Non-linear dynamical systems* present many more difficulties.

Classical Control Theory Defining Traits

Control theory has been historically interested in *continuous* systems (over time T), i.e. systems whose models are represented with the tools of differential equations and linear algebra.

Another characteristic of control theory is the *uniformity of representation*. This principle applies both to basic research in control theory and in the practice of *designing* controllers for physical systems (robotics and manufacturing). It is common practice to represent $u(t), g(t), f(t)$, and the control task using the same underlying formalism.

This practice has been carried over to the modeling and control of systems which cannot be promptly represented with the tools of *continuous mathematics*. Section 2.3, contains some introductory remarks about the reformulation

of control theory for discrete event systems.

2.3 Control Theory Applied to DES

While continuous control theory has been very successful, many modern complex physical devices have proven to be not amenable to its techniques.

There are essentially two sources of problems. Many of these devices can be properly described only in a discrete or hybrid (i.e., mixture of discrete and continuous dynamics) setting. The plant has to be modeled in terms of a discrete set of states and transitions[1], which poses many problems. Secondly, the behavior desired of the ultimate system tends to be fairly complex.

In their original work, Ramadge and Wonham [66] describe a reinterpretation of the key concepts of control theory for systems whose underlying dynamics is represented in terms of *formal languages*. The system to be controlled is considered a *generator* ($\mathcal{G}$) of a language L (see [41]). This choice is well suited to represent the discrete nature of a wide variety of systems. Ramadge and Wonham introduced the term CDES to indicate this class of systems.

As a standard example, consider a machine on a shop floor. A high level

[1]The word *discrete* assumes two different meanings in control theoryİt is used to indicate a *discretization* of time T, or as a defining characteristic of the underlying state space. In this dissertation, unless explicitly noted, the term "discrete" always refers to system modeled with an underlying discrete state space.

model of its operation may be given in terms of three states idle, running, faulty. The normal operation of the machine is an alternation of the first two states, controlled by start and stop signals. Every once in while, the machine will break down, causing the model to move to the faulty state.

In its simplest form, a CDES is defined in terms of its generator

$$\mathcal{G} = (\Sigma, S, \delta, s_0).$$

Σ is the *alphabet* of the generator (similar to the *outputs* Y of a control theory specification), whose elements are called *events*. S is a finite set of states (the analogous of X). $\delta : \Sigma \times S \rightarrow S$ is the *unregulated* transition function, defined in the standard way. The state s_0 is the initial state. The generator is also called, with an obvious analogy, the *plant* to be controlled.

The main assumption on CDES is that the alphabet set is partitioned into two subsets. $\Sigma_c \subseteq \Sigma$ is a subset of *controllable* events, $\Sigma_u = \Sigma - \Sigma_c$ is a subset of *uncontrollable* events.

This assumption is the basis of the whole CDES theory. The control law to be applied to the plant can *disable* events in Σ_c preventing them from being generated. This disabling action constitutes the input of the system (in analogy to U).

An *admissible control* for a CDES is a set of disabling actions and can simply be represented as a subset of Σ_c or as a function

$$\gamma : \Sigma_c \rightarrow \{0, 1\}.$$

A $\sigma \in \Sigma$ is *enabled* if $\gamma(\sigma) = 1$, *disabled* otherwise. The set of all γ's is denoted by , .

The analogue of the state equation f is the *controlled* state transition function δ_c, which uses the definition of γ with the extension that $\gamma : \Sigma_u \to \{1\}$.

$$\delta_c : , \times S \times \Sigma \to S.$$

i.e.

$$\delta_c(\gamma, q, \sigma) = \begin{cases} p & \text{if } \delta(q, \sigma) = p \text{ and } \gamma(\sigma) = 1, \\ \perp & \text{otherwise.} \end{cases}$$

By substituting δ with δ_c in the definition of $\mathcal{G}$, the *controlled generator* $\mathcal{G}_c$ is obtained.

With these definition it is now possible to formulate the control problem for CDES's. I.e. design a *supervisor device* that selects the control inputs in such a way that the given CDES behaves in obedience to various constraints.

2.3.1 Building Supervisory Controllers for CDES

The analogy with continuous control theory is carried over in the specification of a supervisor for a CDES plant $\mathcal{G}_c$. Since the plant is a generator of a language ($\Lambda[\mathcal{G}]$), it is natural to assume a device *observing*, or *recognizing*, the events generated and producing the control inputs as needed.

Such a device is represented as a recognizer

$$\mathcal{R} = (\Sigma, R, \rho, r_0).$$

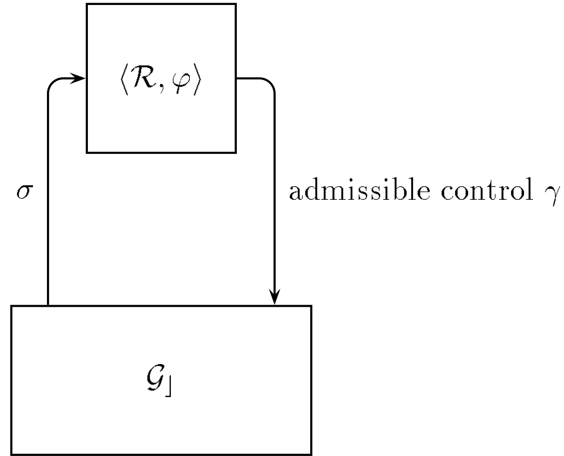

Figure 2.1: *The standard arrangement of plant $\mathcal{G}$ and supervisor $\langle \mathcal{R}, \varphi \rangle$.*

R is the recognizer set of states, ρ is its transition function and r_0 is its initial state. The recognizer has an associated map

$$\varphi : R \rightarrow ,\ ,$$

which represents the control law for the CDES. The pair $\langle \mathcal{R}, \varphi$ is the supervisor for $\mathcal{G}_\rfloor$. φ is called the *supervisor map* and represents the *state feedback law* for the plant. The supervisor is coupled with the plant in the standard arrangement shown in figure 2.1.

With these definitions, Ramadge and Wonham develop a theory which gives guarantees about the existence and the "constructibility" of supervisors.

CDES Theory: Main Results

The main result of the theory of CDES is the existence theorem for supervisors (cfr. [66]). The result is based on a notion of *controllability* of languages. Assume the standard language union, intersection, concatenation and prefix closure operations described in [41]. If K and L are languages over an alphabet Σ partitioned between controllable and uncontrollable events, then the language K is said to be *controllable* if

$$\bar{K}\Sigma_u \cap L \subseteq \bar{K}.$$

This condition ensures that given any sequence of events in the prefix of K ($\bar{K}$), any subsequent uncontrollable event will not produce a behavior that a recognizer for K would fail to detect.

The other main result of the theory concerns the structure of the family of controllable sublanguages of a given language K. This family is closed under union and intersection and contains a *supremal element*. The *supremal controllable sublanguage* of K, denoted by $K^{\uparrow}$, is therefore an approximation to K. The existence of this supremal element is the key to the construction of the actual supervisor for a wide variety of systems.

A Pragmatic Methodology

The existence of the supremal controllable sublanguage of a given language K, implies a procedure for the construction of a supervisor for a given system.

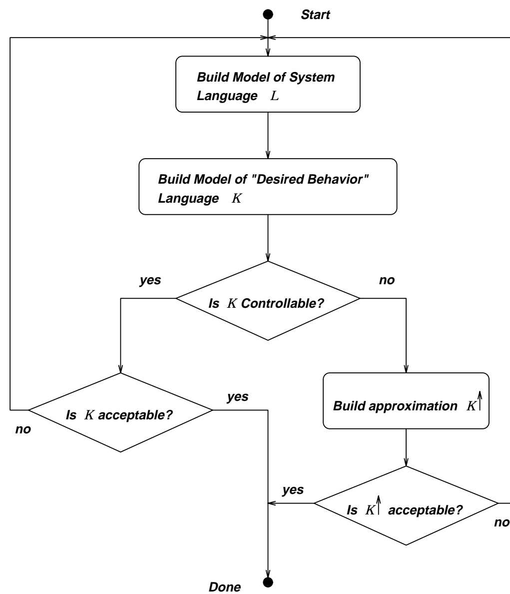

Figure 2.2: *Schematic flowchart of the Supervisor building steps.*

The supervisor $\langle \mathcal{R}, \varphi \rangle$ realizes the control law for a given set of constraints. These constraints are expressed in terms of a language K. If this language is not controllable, then its approximation $K^{\uparrow}$ can be built. This leads to a pragmatic procedure for the construction of supervisors (see Figure 2.2). Such a procedure is not an algorithm directly implementable. It is a guideline for a practitioner employing these notions to actually build a controller. In particular, the step checking for the "goodness" of the controlled behavior is completely up to the human. Ramadge and Wonham take this fact into account in their theory by introducing the notion of "acceptable" language (cfr. [66]).

The procedure is the basis for the design of a software environment for

the synthesis of discrete controllers for CDES's. In [13] a system is described that performs this "aiding" task.

The process of building a supervisor proceeds in two steps: first a supremal sublanguage for the specified K is built, then the actual supervisor map φ is synthesized.

There are restrictions to the kind of languages that can be effectively used with this procedure. The theory works well for *regular* languages. For larger classes of languages (e.g. *context-free*) the uniqueness of $K^{\uparrow}$ cannot be guaranteed. These results are contained in [64]. In the same paper, one can find an algorithm for the construction of the supremal controllable sublanguage of a $K^{\uparrow}$ with respect to a prefix closed language $\Lambda[\mathcal{G}]$. This algorithm assumes an FSM representation of $\mathcal{G}$ and $\mathcal{R}$ and produces the desired automata in $O(|S|, |R|)$, where $|S|$ is the cardinality of the set of states of the generator $\mathcal{G}$ and $|R|$ is the cardinality of the set of states of the recognizer $\mathcal{R}$. The synthesis algorithm proposed in this dissertation improves slightly on this result. Though the overall asymptotic complexity of the proposed synthesis algorithm does not change, it will be argued that the different representation used can significantly improve the practicality of the resulting tool.

Other Developments in Ramadge and Wonham's Theory

The part of the theory of CDES's described so far is only the basis which constitutes the background for the development of this dissertation. An overview

of other developments in the theory can be found in [67]. The other topics which have been investigated in the CDES framework regard observability and modular synthesis of supervisors [63]. The issue of *decentralized supervision* has been also investigated [68]. In this case the question asked is whether the action of several supervisors acting locally can achieve the same effect as a centralized one.

Another research direction concerns what kind of extensions can be introduced in the CDES framework preserving its basic characteristics. A crucial question concerns what classes of languages besides the regular ones admit the unique supremal sublanguage construction or what are the characteristics of other extensions to the basic model. For instance, ω-languages (and Büchi automata) as a basis for CDES are considered in [65], and *Vector Addition Systems* are considered in [47, 48].

The basic CDES model does not include *time* as a component. The only notion of the passage of time is derived from the generation of events by the plant. In this direction, a standard extension to the basic CDES model with discrete *time ticks* is presented in [18].

On the foundation of CDES, other extensions and reinterpretation of the theory have been proposed, mainly with the aim to reuse the wealth of knowledge developed within the temporal logic community. This dissertation falls in this last category. Additional references will be given in Section 2.4.

2.4 Verification and Temporal Logic

The CDES Theory described in Section 2.3 is geared toward synthesis of a controller for a given model of a device (either physical - like a robotics arm or a manufacturing line - or software). Yet, the question of producing such a controller is intertwined with that of determining the *correctness* of the results. Answering this question amounts to *verifying* that a *model* of the system has properties that can be proved in a "mathematical" sense. The underlying proof process ranges from fully automated algorithmic tools to "paper and pencil" techniques. An excellent historical survey of the topic of verification can be found in Chapter 1 of [46], along with a disquisition of the many meanings assumed by the term *"verification"*.

The argument in favor of automated (or semi-automated) proof procedures was made by Barwise (cfr. [46]) and it is based on the the complexity and lack of peer review of many correctness proofs for models of real-world systems. Therefore, since the early sixties, many researchers started working on applications of *theorem proving* for the proof of correctness of models specified in a given logic.

However, theorem proving in its most general sense is undecidable. Therefore, there have been an effort toward restricting the scope of the logics used for the task. Moreover, since most of the "interesting" properties of complex systems deal with time-dependent properties, the class of logics of choice is the one based on temporal modal operators, i.e. Temporal Logics.

Brief Classification of Temporal Logics

There are several flavors of temporal logic. Following the classification of Emerson's [35], different temporal logics can be distinguished according to the following criteria.

- *Propositional* vs. *First Order*.
- *Implicit* vs. *Explicit* Time.
- *Point* vs. *Interval* Time (especially for explicit time logics).
- *Discrete* vs. *Continuous* Time.
- *Branching* vs. *Linear* Time.

These distinctions (apart from the propositional vs. first-order one) refer to the underlying structure of time. For the purpose of this dissertation the focus will be directed at propositional, implicit, branching, discrete time logics. The exact meaning of this classification will become clearer in the next sections.

2.4.1 Temporal Properties of Systems

Several properties of a system can be classified as "temporal". Among them, there are *explicit time related* properties and *implicit time related* properties.

Implicit and Explicit Time Properties

Explicit time properties are of the kind "the lathe operation will last 3 minutes", or "the temperature of the water in the tank must drop 20 degrees within 10 minutes", and so on.

Implicit time properties do not mention explicit time constraints. Instead they express how a system will evolve base on the *history* of *observable* variables. E.g "The failure will be propagated", "In the long run we are all dead"[2].

Temporal logic has historically concentrated on this second kind of properties. In [50] summarize what classes of *properties* are the target of the "practicing verifier", like *safety*, *liveness* of a system.

Safety properties refer to conditions whose truth is ensured by the system. They are also referred to as "invariance" properties. Liveness properties express the fact that "something will happen", i.e. they express "eventuality".

Another set of properties that can be expressed in various temporal logics are *fairness* properties, which are particularly useful for systems composed of several *concurrently executing* devices. A further classification of fairness properties (namely, "impartiality", "justice", and "fairness") can be found in [24].

[2]This quote is ascribed to Lord J. M. Keynes. From "A random walk down Wall Street" by B. G. Malkiel, pg. 315.

Other Extensions

Of course, explicit time logics and extensions to the implicit time framework have been proposed in literature. Researchers in Artificial Intelligence have tackled this problem many years ago, though with an emphasis on "reasoning" and "knowledge representation". Typical references in this field are [4], [69] and also [32]. A more recent reference is [30]. Also, Section 2.4.3 contains some references that pertain to the field of Artificial Intelligence.

2.4.2 Temporal Logics: Linear and Branching Time

Temporal logic was introduced a long time ago as a modal logic [42] with a "temporal interpretation" of the modal operators $\Box$ (necessity) and $\Diamond$ (possibility) [61]. Therefore, the semantic of a temporal logic will rely on an interpretation of an underlying *Kripke structure*.

In order to achieve a high degree of automation (i.e. with reasonable computational complexity), the temporal logics used have been usually restricted to the *propositional* fragment (vs. *first order* temporal logic which is always undecidable – the Halting Problem is easily encoded – [35]). Furthermore they are split between *linear* and *branching time* logics.

The distinction between linear time temporal logic (in this context restricted to *propositional lineal temporal logic* – PLTL) and branching time temporal logic, lies in the assumption of the "form" of the underlying *time structure*.

The following table summarizes the distinctions between linear and branching time.

Linear	Branching
Course of time is linear.	Course of time has a branching *tree-like* structure.
Each *moment* has only one possible future moment.	Each moment has many possible future moments.
Temporal operators are about *states* along a single *path*.	Temporal operators are about states along the set of paths starting from the current state.

The semantics of PLTL is given in terms of *time-lines* over a set of *states* S. Time-lines are totally ordered sets $\langle S, < \rangle$ isomorphic to the natural numbers. There is an *initial moment* s_0 (no predecessor) and the time-line is *infinite* "in the future."

The semantics of branching time logic is given in terms of an infinite branching tree-like structure over a set of states S. The most popular branching time logic is the *Computational Tree Logic* (CTL) introduced by Clarke and Emerson in [36], which comes in many flavors distinguished by their expressive power and computational complexity.

Both linear and branching time logics assume the presence of a set AP of *atomic propositions* and an assignment of propositions to states.

$\Pi : S \rightarrow 2^{AP}$.

The semantics of PLTL is given in terms of a *linear time structure*

$$M = \langle S, x, \Pi \rangle,$$

where S is a set of states, $x = (s_0, s_1, s_2, \ldots)$ is a *full computation path*, or, more briefly, a *full-path*. *Partial computation paths* are defined as suffixes of full computation paths and denoted as $x^i = (s_i, s_{i+1}, s_{i+2}, \ldots)$ (therefore $x \triangleq x^0$).

The semantics of CTL is given in terms of a *branching time structure*

$$M = \langle S, R, \Pi \rangle.$$

Where S is – once again – a set of states and $R \subseteq S \times S$ is a total binary relation. Because of R, CTL is capable of expressing properties about *sets of paths* over M. To achieve this power it augments the syntax of PLTL with the two *path quantifiers* **A** ("for all paths") and **E** ("for some paths"). Technically, how PLTL is augmented with the path quantifiers determines different flavors of CTL. See [35] for a thorough account of the CTL variations.

The syntax and semantics of PLTL are summarized in Table 2.1, while those of CTL are summarized in Table 2.2.

Linear and branching time logics have different expressive powers. In particular, the version of CTL used in this dissertation is not the most powerful one. The more expressive CTL* subsumes CTL and PLTL by allowing unconstrained linear time formulæ within the scope of a path quantifier.

SYNTAX	SEMANTICS ($M \models f$)	DESCRIPTION
	BASE FORMULÆ	
p	$p \in \Pi(s)$	A proposition
$f_1 \vee f_2$	$f_1 \in \Pi(s)$ or $f_2 \in \Pi(s)$	A disjunction
$f_1 \wedge f_2$	$f_1 \in \Pi(s)$ and $f_2 \in \Pi(s)$	A conjunction
$\neg f$	$f \notin \Pi(s)$	A negation
$f_1 \Rightarrow f_2$	$f_1 \notin \Pi(s)$ or $f_2 \in \Pi(s)$	An implication
	TEMPORAL FORMULÆ	
$\mathbf{X}$(f)	$x^1 \models f$	f is true in the successor state of s_0
$f\ \mathbf{U}\ g$	$\exists j[x^j \models g \wedge \forall k < j(x^k \models f)]$	there exists a state in the future where g holds and such that f holds up to (i.e. *until*) there.
$\mathbf{F}(g)$	this is equivalent to true $\mathbf{U}\ g$	This is read as *eventually* g will hold.
$\mathbf{G}(f)$	this is equivalent to $\neg\mathbf{F}(\neg f)$	This is read as *henceforth* f holds.

Table 2.1: *Syntax and informal Semantics for PLTL under the assignment* Π. *The semantics is given with respect to a linear time structure* $M = \langle S, x, \Pi \rangle$.

Syntax	Semantics ($M \models f$)	Description
	Base Formulæ	
p	$p \in \Pi(s)$	A proposition
$f_1 \vee f_2$	$f_1 \in \Pi(s)$ or $f_2 \in \Pi(s)$	A disjunction
$f_1 \wedge f_2$	$f_1 \in \Pi(s)$ and $f_2 \in \Pi(s)$	A conjunction
$\neg f$	$f \notin \Pi(s)$	A negation
$f_1 \Rightarrow f_2$	$f_1 \notin \Pi(s)$ or $f_2 \in \Pi(s)$	An implication
	Temporal Formulæ	
$\mathbf{EX}(f)$	$f \in \Pi(s')$ and s' is a successor state of s	f will be true in some next state
$\mathbf{AX}(f)$	$f \in \Pi(s')$ for every s' successor of s	f will be true in all the next states
$\mathbf{E}[f_1\, \mathbf{U}\, f_2]$	If $s_0, s_1, \ldots, s_n$ is a sequence of states and at each of them $f_1 \in \Pi(s_i)$ for $i < n$ and $f_2 \in \Pi(s_n)$	There is a sequence of states where f_1 holds *until* f_2 will.
$\mathbf{A}[f_1\, \mathbf{U}\, f_2]$	For any sequence of states $s_0, s_1, \ldots, s_n$ at each of them $f_1 \in \Pi(s_i)$ for $i < n$ and $f_2 \in \Pi(s_n)$	There is a sequence of states where f_1 holds *until* f_2 will.
$\mathbf{EF}(f)$	There is *a sequence* of states where f will *eventually* hold (this is actually an abbreviation for $\mathbf{E}[\text{True}\, \mathbf{U}\, f]$)	This formula represents a *potential* event.
$\mathbf{AF}(f)$	For *any sequence* of states f will *eventually* hold (this is actually an abbreviation for $\mathbf{A}[\text{True}\, \mathbf{U}\, f]$)	This formula represents a *necessary* event.
$\mathbf{EG}(f)$	There is *a sequence* of states, f will *always* hold (this is actually an abbreviation for $\neg\, \mathbf{AF}(\neg f)$)	The formula f will always hold on some path.
$\mathbf{AG}(f)$	For *all sequences* of states, f will *always* hold (this is actually an abbreviation for $\neg\, \mathbf{EF}(\neg f)$)	This formula states a *global* and *invariant* property of the system.

Table 2.2: *Syntax and informal Semantics for* CTL *under the assignment* Π. *The semantics is given with respect to a branching time structure* $M = \langle S, R, \Pi \rangle$.

When restricted to the propositional case both linear and branching time logics are decidable, but with different algorithmic complexity. Theorem proving and *model checking* must take these differences into account in order to provide a viable verification environment.

2.4.3 Theorem Proving and Model Checking

Verifying a system with a temporal logic amounts to proving properties of the form $\mathcal{F} \equiv f \Rightarrow g$, where f and g are, usually temporal logic formulæ. Moreover, it is usually intended that f represents the "model" (in the "descriptive" sense) of the system to be verified and g is a formula representing the properties to be verified.

There are a variety of approaches that can be used to prove $\mathcal{F}$. Theorem proving approaches focus on the definition of a logical deduction calculus that can be used to derive theorems in an appropriate logic. The literature on automated theorem proving is vast by itself. Since Davis and Putnam put forth their procedure (see [31] for an account), numerous treatises on the topic have been published. Two rather arbitrary references are [23] for an account of "resolution based" theorem proving and [37] for an exposition of various modeling techniques employed in the Artificial Intelligence field.

For the class of systems which are the target of this dissertation and for the use of a temporal logic formalism within the theorem proving paradigm, an excellent example is the work by Manna and Pnueli. (A good reference is

[51].)

Automated theorem proving in its most general form requires a significant amount of human expertise to be useful. Moreover most of the decision procedures available – i.e. when the problem at hand is decidable – suffer from high algorithmic complexity. Using theorem proving for the temporal logic verification problem usually involves some for of PLTL, because of its decidability properties. However, the decision procedures for PLTL suffer from a very high complexity. The unsatisfiability of $f \wedge \neg g$ is PSPACE-complete [70].

A reformulation of the verification problem led to the development of a much more efficient procedure for branching time logic. The verification problem can be seen as a 2-inputs problem: the *model* of the system and a *property* to be checked. In [36, 24], the authors take this stand and produce a *model checking* algorithm which turns out to be of *linear* complexity for a model represented as an automata and a property represented in CTL. In this context, the formula $f \Rightarrow g$ is usually recast as $M \models g$ (M is a model for f).

The model checking algorithm is linear in the size of the set of states – the *state-space* – of the model of the system. However, the models are usually obtained through the *parallel composition* of distinct FSM's, an operation that yields a state-space of size exponential in the number of components[3]

[3] This is not to belittle the value of the model checking algorithm for CTL.

a phenomenon usually referred to as the *state-space explosion*. This fact spun off efforts into two directions: *compositional specifications* and *symbolic model checking*.

Some notes on compositional specifications appear later in this section. Symbolic model checking [22] takes advantage of a special representation of the model based on *Binary Decision Diagrams* (BDD's) [3, 21] to encode vast portions of the state space in a small amount of memory, while validating the specification g via a *fixed-point computation*. While the worst case analysis of this encoding does not improve on the original model checking algorithm, empirical results show a dramatic improvement in the memory requirements for many verification tasks.

"Synthesis" and the Tableau Method In [36] the authors lay the foundation for the use of temporal logic (in particular CTL) both for verification (by stating the model checking algorithm) and *synthesis* of computational structures.

In the field of temporal logic verification the term "synthesis" usually refers to the decision procedure that tries to prove the *satisfiability* of a well formed formula in a given temporal logic by constructing a model for it. A typical technique is the *tableau method* in its many different variants (cfr. [52, 76]). In this case, "synthesis" is a *1-input problem*.

In order to avoid confusion, it must be emphasized that the term "synthesis" used throughout this dissertation is to be understood in the sense

suggested by Ramadge and Wonham, and not in the accepted sense within the temporal logic framework.

Advantages of Branching Time Logics: Branching time may express more properties than simple linear time. CTL is strictly more expressive that PLTL. E.g. the distinction between the modalities **AF** and **EF** provide the ability to distinguish between "potentiality" and "eventuality".

Because of the better algorithmic complexity of the model checking procedure, CTL has been extensively used in systems like SMV [55] to prove properties of systems like the Futurebus+ hardware bus [26]. The theory and the tools presented in this dissertation use CTL as a specification language.

Advantages of Linear Time Logics: Linear time logics enjoy certain characteristics (not excluding the simpler time structure) that are not available to branching time logics, which makes them easier to handle. Their main attractiveness is the power to directly express fairness constraints that CTL cannot encode – fairness properties can be expressed using CTL*, at the price of exponential time algorithms for satisfiability *and* model checking.

As a final remark, it must also be noted that some recent empirical results indicate that PLTL model checking may be just as practical (see [25]) as the provably more efficient CTL.

Applications of Model Checking

Model checking has been successfully applied to the verification of VLSI designs, communication protocols, cache coherency protocols, and bus arbitration algorithms [20, 46]. The extensions to manufacturing and robotics presented here are rather novel, and raise many new issues.

2.4.4 Verification with Automata Theory

Temporal logic approaches to verification are not the only possible ones. There is a relation between temporal logic and *automata theory* that is rooted in the models given for these logics, which eventually are expressed in terms of the underlying Kripke structures (since, after all, temporal logics are modal logics). In particular, ω-automata on infinite strings are equally suitable for representing the properties listed in Section 2.4.1. Kurshan develops the theory of ω-automata for verification in [46]. Automata lend themselves to model checking and are more expressive than PLTL, though incommensurable in power with CTL.

The major advantage of this and related approaches (cfr. Chapter 1 of Kurshan's book) lies in the existence of *reduction* methods that make it possible to contain the state-space explosion inherent to the construction of many practical models and to proceed subsequently in the verification of a system in a step-wise fashion.

2.4.5 Reductions and Compositional Verification

In order to reduce the space requirements of model checking, several compositional procedures and reduction methods have been proposed. In [58] the logic CTL^- is proposed to this end. CTL^- is a syntactically restricted version of CTL that disallows existential path quantifiers and allows for a hierarchical check based on a "hiding" operation not dissimilar to the one proposed by Milner's for his Calculus of Communicating Systems (CCS) [56]. Another "compositional" version of CTL can be found in [27]. As already mentioned, automata based verification provides reduction methods for curbing the state-space explosion.

This dissertation will not discuss questions related to composition and reduction. The choice of CTL as the basic specification language limits by itself this development. However, all the methods and algorithms that will be developed should apply to CTL^-, thus, providing some form of hierarchical specification.

2.4.6 Real Time Verification

The distinction between "implicit" and "explicit" time properties is more acute when the systems being analyzed and verified involve *real time* properties. Recently, there have been proposed several fundamental models as well as extensions to temporal logic for this purpose. These models which introduce explicit time mostly in the form of "attached" constraints to the

temporal operators. A survey of some of these extended temporal logic and of their decidability properties can be found in [9]. A fundamental model which has been studied in recent year is the *timed automata*. References to this model can be found in [8]. The interest in this model lies in the existence of an encoding of real valued time information in a *finite* form suitable for symbolic model checking. Symbolic model checking for real time systems has been studied, for example, in [6] and [39].

Other approaches to these problems come from different communities. E.g. the Petri Nets (PN) community has developed many tools for the verification of time properties of PN models (e.g. see [14]).

2.4.7 Role of Verification Tools

The study of the algorithmic aspects of verification raises the question about the role of tools built upon this paradigm. The list of tools is now long. Two representative are SMV [55] and COSPAN [46].

The role of verifier systems is, by general agreement of the verification community, that of *sophisticated debuggers* which can catch "design errors" in the early phase of the development of a system. It will be argued that a tool capable of synthesizing (and verifying) controllers would be as beneficial to the early developing phases of a system, notably in such areas as robotics and manufacturing.

2.5 Hybrid Systems Verification and Control

Discrete and Continuous modeling tools are extremely useful in many situations. Yet, many devices and problems are better described as a combination of discrete and continuous models. Engineers refer to this interplay between discrete and continuous models as "mode switching". It is no surprise to see the emergence of efforts to integrate the discrete and continuous computational models. As an example, such a model is presented in [53] for the solution of the classic *peg–in–a–hole* problem.

New research directions in this field can be found in [38] and in [16]. These efforts constitutes the coming together of different communities. The research directions can be roughly categorized into two groups. The first one studies the mathematical foundations of the combination of discrete and continuous systems, while the second concentrates on the fitting of existing formalisms to the task. Many of the formalisms are rooted in the temporal logic (e.g. [7] and [60]) and control theory communities [59]. The main objective of these studies is to find a proper "combined" model for both discrete and continuous behavior of a given system and to produce a "syntactic" specification of the control law to be obeyed. The thesis by Desphande [33] follows this direction.

In this dissertation, the topic of hybrid systems is only touched upon when discussing one of the testbeds for the proposed system. No comprehensive theory is developed: information of "continuous" nature is only used as a supplement in the synthesis of a viable discrete controller for a walking ma-

chine. Nonetheless, this approach seems to be quite powerful, in the kind of practical situations arising in robotics and manufacturing.

Chapter 3

Temporal Logic Supervisor Synthesis with Verification

This chapter contains the theoretical framework for the application of the tools of temporal logic to the *Supervisor Synthesis Problem* for CDES.

The aim of this approach is to use a more viable specification formalism for the constraints on the trajectory of a plant, than having to express them directly in the form of an auxiliary language (the language K in the CDES formalism [67]). This is a departure from the control theoretical metaphor that characterizes the CDES modeling and synthesis procedure. It can be argued that "representational" simplicity can be gained by such a departure. The specification language that will be used is the standard CTL described in [36].

First of all, it must be noted that the idea of using automata or temporal

logic for the synthesis of "controller programs" is not new and dates back to the early work of Arbib [44] and others[1]. Synthesis of programs based on temporal logic goes back to [1]. More specifically, the combination of temporal logic with the supervisor synthesis problem within the Ramadge and Wonham framework has been studied in [49], and with a slight variation (and extensions to the hybrid case) by [33].

The choice of CTL as a specification language is novel. To the best of the author's knowledge, this is the first attempt to formulate the synthesis of controllers for CDES based on a branching time logic. All the works mentioned use some form of PLTL as their base model. Choosing a branching time instead of a linear time logic was mainly a matter of relative usefulness of the paradigm vs. the linear time framework. One of the main reasons for such a choice was the existence of the the linear time model checking algorithms and their efficient implementations (which form the basis for the supervisor synthesis algorithm described in Chapter 4) Another reason was the CTL stronger expressive power.

This Chapter is organized as follows. Section 3.1 contains an introduction to the notation used throughout the chapter. Section 3.2 contains the definition of the semantic characterization for a specification given in CTL with respect to a system to be controlled. Section 3.3 ensures the existence of an algorithm capable of synthesizing the correct supervisor that will ensure the

[1] Who must forgive the forgetfulness of the writer.

satisfiability of the CTL specification. The algorithm is exponential in time. Chapter 4 will present an algorithm which, under a set of mild restrictions with respect to the full CTL, achieves a linear time complexity.

3.1 Notational Preliminaries

The terminology used in this and subsequent chapters is that of the Theory of Formal Languages, more precisely, the *regular* languages [41]. The definition of CDES given in Section 2.3 serves as a basis of the following discussion. All the languages mentioned subsequently are intended to be regular. Also, it is always assumed that the underlying Kripke structure has the form of a FSM (denoted either with $\mathcal{P}$ or $\mathcal{G}$) with components (Σ, S, δ, s_0). Finally $M[L]$ denotes the FSM associated to a language and $\Lambda[\mathcal{M}]$ denotes the language recognized by a machine $\mathcal{M}$.

Language, Active Set, Previous and Next Set at a state s

In order to denote the language and the *active alphabet* relative to a given state of the FSM, the following notations are used: $L[s]$ and $\Sigma[s]$. $L[s]$ is the language that can be generated using s as start state. $\Sigma[s]$ is the subset of Σ that can be "followed", starting from a given state s.

The notations

$$\mathbf{N}[s] \triangleq \{s' \mid \exists \sigma \in \Sigma . \delta(s, \sigma) = s'\},$$

$$\mathbf{P}[s] \quad \triangleq \quad \{s' \mid \exists \sigma \in \Sigma . \delta(s', \sigma) = s\}.$$

indicate the set of "next" and "previous" states relative to a "current" state s.

Language Operations

The following *language operations* on arbitrary regular languages L, L_1, L_2 are assumed.

$L_1 \cup L_2$	The set-theoretic *union* operation.
$L_1 \cap L_2$	The set-theoretic *intersection* operation.
L^c	The set-theoretic *complementation* operation.
$L_1 L_2$	The *concatenation* operation.
$\bar{L}$	The *prefix closure* operation.
$L_1 \setminus L_2$	The *language quotient* operation defined as

$$L_1 \setminus L_2 \triangleq \{x \mid \exists y \in L_2 . (xy \in L_1)\}.$$

The following shorthand will also be used:

$$L[s] \setminus_{q_i} L[q_i], \;\text{ for } 1 \leq i \leq n,$$

where $\{q_i\} \subseteq S$ and $s \overset{\delta}{\rightsquigarrow} q_i$ (i.e. there is a path in $\mathcal{P}$ from s to q_i), is to intended to denote

$$(((L[s] \setminus L[q_1]) \setminus L[q_2]) \ldots \setminus L[q_n]).$$

Languages concatenation operations

The *language concatenation* operation is usually defined in terms of the *string concatenation* operation. This "low level" operation is usually defined to have the empty symbol/string ϵ to be the *unit* element. Therefore

$$\begin{aligned} \epsilon\epsilon^* &= \epsilon, \\ \epsilon w &= w\epsilon = w. \end{aligned}$$

The usual definition of the *language* concatenation for two languages L_1 and L_2 is

$$L = L_1 L_2 = \{uw \mid u \in L_1 \wedge w \in L_2\}.$$

This definition is augmented by the two following cases in order to accommodate the *null language* $\mathcal{E} = \{\epsilon\}$ and the *empty language* $\Phi = \emptyset$.

With these definition the concatenation operation behaves as follows

$$\mathcal{E}L = L\mathcal{E} = L. \tag{3.1}$$

and

$$\Phi L = L\Phi = \Phi. \tag{3.2}$$

Therefore, in the present framework, language concatenation behaves like multiplication with $\mathcal{E}$ and Φ as $\mathbf{1}$ and $\mathbf{0}$ respectively. Definition 3.2 will be especially useful when the semantics of a CTL formulæ under supervision will be defined.

Logical *Entailment* Operators

The *logical entailment* operator $\models$ usually denotes truth in a structure M which is defined according to the needs of the task. In CTL literature, the structure M is defined in terms of the underlying Kripke structure and the notation

$$M, s \models f,$$

where f is a CTL formula and s is a state and the entailment, denotes the truth of f.

However, the constructions that will follow in this chapter, "parameterize" the notion of truth, making it possible to generate undue confusion. The following notation is therefore introduced, in order to keep the discussion simple

$$\begin{aligned} L, \varphi, s &\;|\!\models\; f, \text{ or} \\ L, s &\;|\!\models\; f. \end{aligned}$$

L is a language and φ is the standard "supervisor map" of CDES (see Section 2.3). The second form is used as a shorthand unless there is a need to specify explicitly the supervisor map.

The notation

$$\begin{aligned} M, \varphi, s &\models f, \text{ or} \\ M_{\varphi}, s &\models f, \end{aligned} \tag{3.3}$$

with M the underlying Kripke structure (i.e. a FSM like $\mathcal{P}$), could have been used instead of $L, \varphi, s \; ||\!\!= f$. Yet, it was found to make the treatment of some of the construction more complicated. It was found that using the "language" as a characterization of the the semantics for a given CTL formula, yields a more immediate characterization of the behaviors allowed for the model (e.g. see the **AU** case).

There is one more important rationale behind the choice of the notation introduced in (3.3). In the CDES context, the supervisor map acts on the basis of the "tracking" action performed by the recognizer $\mathcal{R}$. Therefore it can be used to perform a control action which is not *locally limited* to the current state. CTL path formulæ serve the purpose of expressing such non local properties, and the characterization of their semantics by means of languages is an useful "shorthand."

3.1.1 CDES Notions and Notations

There are two key notions within CDES Theory: the distinction between *controllable* and *uncontrollable* events and the definition of the supervisor map.

The definition of controllable and uncontrollable events leads immediately to the definition of supervisor map.

$$\varphi_f \; : \; Q \times \Sigma_c \to \{0, 1\},$$
$$\varphi_f \; : \; Q \times \Sigma_u \to \{1\}.$$

That is, at each state, a decision must be made whether to enable (with the supervisor map yielding a 1) or disable (yielding a 0) the next transitions[2], in dependence of the formula f and of its sub-formulæ. When no confusion arise, the f subscript will be dropped. Note that uncontrollable events are always "enabled".

3.2 CTL Direct Synthesis of Supervisors

One of the aims of the recasting of the CDES paradigm into a temporal logic framework is to give a reasonable reinterpretation of its basic concepts. The most important concept is that of supervisor with its components, the recognizer $\mathcal{R}$ and the supervisor map φ.

In the temporal logic setting proposed in the following sections and chapters, the recognizer $\mathcal{R}$ is equal to the model of the plant $\mathcal{P}$, and the supervisor map φ is synthesized from the CTL specification $\mathcal{S}$. This is the main difference from the CDES approach, where $\mathcal{R}$ plays the role of the specification[3].

This synthesis problem can be denoted as the *Temporal Logic Supervisor Synthesis Problem* or *CTL Supervisor Synthesis Problem.* The formulation

[2]Ramadge and Wonham define the supervisor map in a slightly more general way. Refer to their original paper for a comparison [66].

[3]Technically, this is not quite correct: $\mathcal{R}$ is the result of the manipulation – possibly identical – of an "initial" FSM which is reduced in order to satisfy the "controllability" condition (cfr. Chapter 4 for a more detailed explanation). However, the imprecision does not bear on the following argument.

of such a problem addresses the question whether there exists a language L such that

$$L \subseteq \Lambda[\mathcal{P}], \quad (3.4)$$

$$L, \varphi_f, s_0 \;\; |\!\models \;\; f, \quad (3.5)$$

where $\Lambda[\mathcal{P}]$ is the plant language, s_0 is its *initial state* and f is a CTL formula indicating the *desirable properties* of the controlled plant. The language L is obtained by *restricting* $\Lambda[\mathcal{P}]$ by means of φ_f[4].

In other words, the task is to find which supervisor map restricts the model $\mathcal{P}$ in such a way to satisfy $\mathcal{S}$. In the following sections, the supervisor map is shown associated with each state and with each sub-formula of $\mathcal{S}$.

Remark: Once the language L has been found, the following questions can be asked.

1. Is $L \neq \Phi$?

2. If $L = \Phi$, is there a subset of the specification[5] that is satisfiable by some other language L'?

Question (2) leads immediately to the notion of "approximation" of a sub-language, which Ramadge and Wonham treat formally with the notion of

[4]The dependence from φ_f may be omitted for simplicity. The notation becomes $L, s_0 \;|\!\models f$.

[5]We will take the specification to be a conjunction of CTL formulæ. This is standard practice.

supremal controllable sublanguage (see Section 2.3.1).

3.2.1 *Controlled* Semantics as *Model Restriction*

The supervisor map φ is constructed by an algorithm that can be interpreted as a modification of the standard model checking labeling algorithm [24]. The main idea is to "cut off" the (controllable) branches that would make a subformula unsatisfiable. During this process, the language that makes a certain formula satisfiable in a state s is computed and kept in memory, to be used subsequently.

The modified semantics also defines how the supervisor map can be built in order to ensure the satisfiability of the formula under consideration. The resulting semantics is called the *controlled semantics* of CTL, and it is constructed in an inductive way.

Semantics "Collapse" Operation

The inductive construction of the controlled semantics specifies for each formula the *supervisor map assignment*. Since specifying the map assignment must always take into account controllable and uncontrollable events, the following convention for specifying the controlled semantics of a formula f is adopted.

> Whenever the map assignment tries to satisfy a formula f in a state s by disabling an uncontrollable event from $\Sigma_u[s]$, then the

semantic clause *collapses* to

$$\Phi, s \ \| \!\!\models f. \tag{3.6}$$

In other words, when the only way to satisfy a formula f in a state s would be through a "controllability" breach, then the state is simply marked as *no good* by "assigning" to it the empty language Φ, thus preventing recursive building of a nonempty language satisfying a set of formulæ for the complete system.

3.2.2 Main Objective and Notable Problems

The characterization of the semantics for a set of CTL formulæ is not trivial. The branching time nature of the logic introduces many problems which require a careful balancing act in the definition of the interplay between the supervisor map and the language satisfying a given formula.

A very simple construction for the supervisor would either allow every transition or block all of them (thus yielding $\Phi, s_0 \ \| \!\!\models \mathcal{S}$) based on the result of checking for satisfiability of $\mathcal{S}$ via the model checking algorithm (i.e. checking whether $\mathcal{P}, s_0 \models \mathcal{S}$). Of course these are two extremes and it is desirable to find some "middle ground".

The examination of the trade-offs leads the definition of the controlled semantics. Its characterization is eventually given in terms of the *soundness theorem* of Section 3.2.5. I.e. the *restricted* plant $\mathcal{P}$ must be a model (in the

model-theoretic sense) of the specification $\mathcal{S}$.

Section 3.2.3 contains an inductive definition of the controlled semantics for a CTL specification. Many of the clauses comprise elaborate structures that circumvent the problems that might creep in the supervisor if care is not taken. Three major problems are described below.

- *Consistency*: The presence of uncontrollable events makes it difficult to find assignments for the supervisor maps that satisfy the specification and that do not impose constraints too tight to the languages.
- *Choice*: Disjunction of path formulæ may lead to different supervisors, which yield different languages satisfying $\mathcal{S}$, which therefore break down the notion of soundness of the construction.
- *Look-ahead computations*: Path formulæ (mostly pure "until" formulæ like $\mathbf{A}[f_1 \, \mathbf{U} \, f_2]$) eventually lead to states where the "terminal" f_2 must be satisfied. The supervisor assignment that satisfies this terminal formula may interfere with the satisfaction of the over all "until" formula, and is governed by the configuration of the underlying graph of the plant FSM.

Some examples of these problems are listed in the next section.

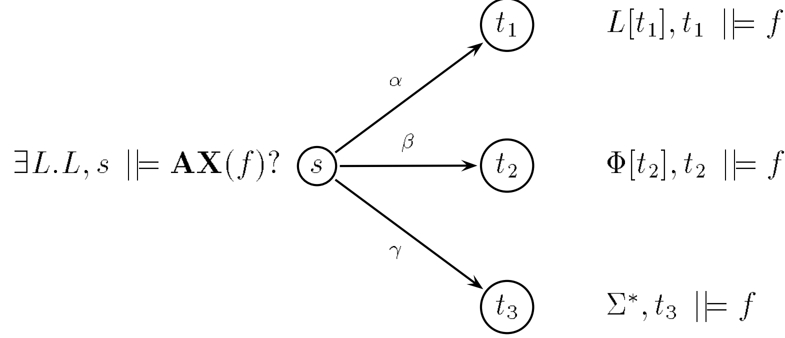

Figure 3.1: *Suppose that states t_1, t_2, and t_3 allow f to be satisfied by the languages shown in the picture. In this situation the supervisor must be able to set $\varphi(s)(\beta)$ to 0. Otherwise we could get into a state (t_2) where it might be the case that $\Sigma[t_2] \cdot L', t_2 \;||\!\!= \neg f$ for some $L' \subseteq \Sigma^*$. Assuming that $\{\alpha, \beta, \gamma\} \subseteq \Sigma_c$, the language satisfying $\mathbf{AX}(f)$ at state s is $\alpha L[t_1] \cup \gamma\Sigma^*$.*

Supervisor Map "Consistency" in the Presence of Uncontrollable Events

In anticipation of the problems that the controlled semantics must take care of, the reader may consider the case depicted in Figure 3.1.

If the event β were uncontrollable then it would be impossible to prevent the transition to state t_2, where the only possible evolution of the system would be to *block*[6].

[6] This is a case where the supervisor map would yield a *blocking* behavior - cfr. Ramadge and Wonham [ibid.].

The Choice Problem Due to Disjunctions

Figure 3.2 contains a problematic example for the temporal logic supervisor synthesis problem with disjunction. A supervisor is needed that satisfies the formula

$$\mathbf{AX}(\mathbf{AG}(p_1)) \vee \mathbf{AX}(\mathbf{AG}(p_2)). \tag{3.7}$$

for the plant language L of Figure 3.2.

The language L generated by the automata in fig. 3.2 is simply

$$L = \alpha_1\beta_1^* + \alpha_2\beta_2^* + \alpha_3(\beta_1^* + \beta_2^*). \tag{3.8}$$

The controllable events are α_1, α_2 and α_3.

Suppose now that the following assignment of propositions to states are given.

$$\begin{aligned}
\Pi(s_1) &= \{p_1\}, \\
\Pi(s_2) &= \{p_1, p_2\}, \\
\Pi(s_3) &= \{p_2\}.
\end{aligned}$$

With this assignment it is possible to start labeling the states with CTL formulæ in the following way

$$\begin{aligned}
\beta_2^*, s_3 &\;||\!\!= \{\mathbf{AG}(p_2), \mathbf{AX}(\mathbf{AG}(p_2))\}, \\
\beta_1^*, s_1 &\;||\!\!= \{\mathbf{AG}(p_1), \mathbf{AX}(\mathbf{AG}(p_1))\}, \\
\beta_1^*, s_2 &\;||\!\!= \{\mathbf{AG}(p_1)\}, \text{ and} \\
\beta_2^*, s_2 &\;||\!\!= \{\mathbf{AG}(p_2)\}.
\end{aligned}$$

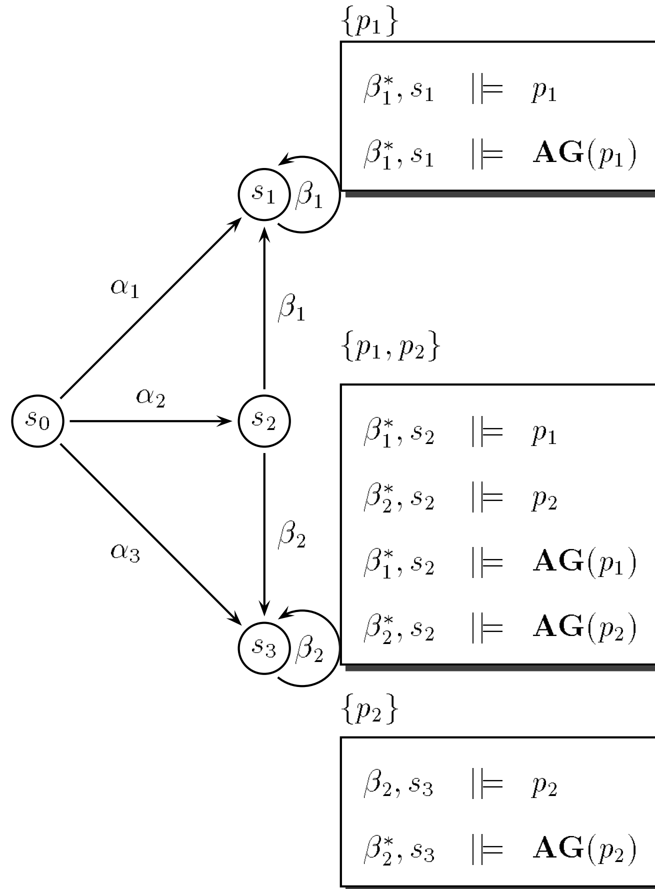

Figure 3.2: *A counterexample which shows the difficulties for the synthesis of a supervisor for a* CTL *disjunction. We cannot find a reasonable supervisor* φ *for* $\mathbf{AX}(\mathbf{AG}(p_1)) \vee \mathbf{AX}(\mathbf{AG}(p_2))$ *even if the supervisors for the subformulæ are well defined.*

Note that state s_0 cannot be labeled with a meaningful language. I.e., no meaningful language L' (under a control action implemented by φ) can be found such that

$$\begin{aligned} L', s_0 &\parallel\!\models \mathbf{AX}(\mathbf{AG}(p_1)) \vee \mathbf{AX}(\mathbf{AG}(p_2)), \quad \textit{i.e.} \\ M[L', \varphi], s_0 &\models \mathbf{AX}(\mathbf{AG}(p_1)) \vee \mathbf{AX}(\mathbf{AG}(p_2)). \end{aligned}$$

In fact, when formula (3.7) is eventually considered in state s_1, neither of the two **AX** disjuncts can be satisfied unless some control action is enforced, either by disabling the α_1 or the α_2 transitions (as it was implicitly assumed when labeling s_3 with $\beta_1^* \parallel\!\models \mathbf{AG}(p_1)$ – here it was assumed that β_2 was disabled).

For the two disjuncts $\mathbf{AX}(\mathbf{AG}(p_1))$ and $\mathbf{AX}(\mathbf{AG}(p_2))$ in (3.7), there are the following cases.

1. *disabling* α_1:

$$(\alpha_2 + \alpha_3)\beta_2^*, s_1 \parallel\!\models \mathbf{AX}(\mathbf{AG}(p_2)), \tag{3.9}$$

2. *disabling* α_2:

$$(\alpha_1 + \alpha_3)\beta_1^*, s_1 \parallel\!\models \mathbf{AX}(\mathbf{AG}(p_1)). \tag{3.10}$$

The following abbreviations will be useful

$$\begin{aligned} L_1 &\triangleq (\alpha_1 + \alpha_3)\beta_1^*, \\ L_2 &\triangleq (\alpha_2 + \alpha_3)\beta_2^*. \end{aligned}$$

It can now be noted that it is impossible to label s_1 with the union of L_1 and L_2, as intuition may suggest,

$$L_1 \cup L_2, s_1 \ \| \!\models \mathbf{AX}(\mathbf{AG}(p_1)) \vee \mathbf{AX}(\mathbf{AG}(p_2)),$$

because of the control action that must be performed in order to ensure that either of the disjuncts is satisfied[7]. That is, by labeling state s_1 and formula (3.7) with $L_1 \cup L_2$ there is some *lost information* about what control actions we are enforcing. This problem arises because of the *choice* inherent to the disjunction.

Lookahead Computations

The choice of defining the controlled semantics in terms of languages recognized by a given machine (the plant $\mathcal{P}$) is also dictated by the necessity to represent evolutions of the system beyond the state where a given formula is satisfied. CTL path formulæ characterize these evolutions, however, their controlled semantics must also characterize the interactions between their sub-formulæ. These interactions may hinder the desired characteristic of the supervisor map to eventually restrict the behavior of the plant to become a model for $\mathcal{S}$.

As an example of this problem, consider the very simple example of Figure 3.3. The supervisor map must ensure that eventually $M[L, \varphi] \models \mathbf{A}[b\,\mathbf{U}\,a]$.

[7]The same problem happens in state s_2.

However, since state s_1 is labeled with a, then a naïve supervisor map assignment would allow any suffix language $L[s_1]$ of $\Lambda[\mathcal{P}]$ to be generated undisturbed.

However, the naïve supervisor map is "blind" to the other possible evolutions of the plant. The evolution of the plant by the event τ puts the plant in state s_2, where the only possible acceptable event satisfying $\mathbf{A}[b\,\mathbf{U}\,a]$ is γ. Therefore, there are at least two computation paths which "interfere" during the construction of a supervisor.

This can be explained in terms of the interference due to the conflicting goals of "model checking satisfiability" and of "greatest possible size" of the supervised behavior of the plant[8].

3.2.3 Inductive Construction of the Controlled Semantics

This section contains the description of the controlled semantics for each CTL formula. The construction for each formula is divided into three parts, accompanied by short discussions about why certain choices were made.

1. A *semantics* clause.

2. A *supervisor map assignment* clause.

[8]Note that it would be possible to selectively disable the event σ at state s_2 by introducing a notion of "history" in the system. This choice was discarded because of the extra complications that it introduces in the construction of the supervisor map.

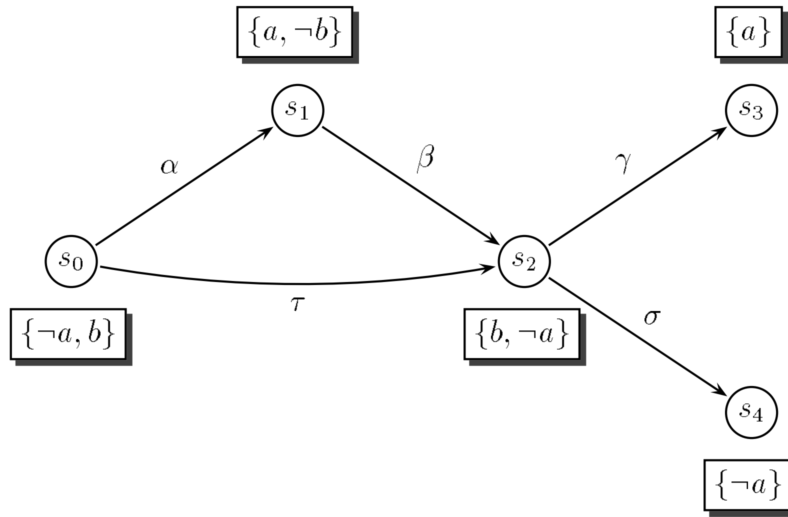

Figure 3.3: *This example shows why some restrictions are necessary on the controlled semantics for* $f = \mathbf{A}[b\,\mathbf{U}\,a]$. *The language satisfying* f *at* s_0 *is* $L = (\alpha\beta + \tau)\gamma$. *However, the language satisfying* a *at state* s_1 *is* $L_a[s_1] = \beta(\gamma + \sigma)$.

3. When appropriate, a language *non emptiness* condition[9] (which will also ensure that the supervisor map assignment is meaningful).

In the following we will always assume that for events $v \in \Sigma_u$ the supervisor map φ at state s will be defined as

$$\varphi(s)(v) \leftarrow 1,$$

as the standard theory requires. As it will become clear, the "language non emptiness" condition will state when the supervisor map assignment would make the specification non satisfiable.

$\boxed{p \in AP}$

Here are the definitions for a formula comprising a simple proposition p

SEMANTICS

$$L[s], s \; ||\!\!= p \text{ iff } p \in \Pi(s).$$

The semantic rule for the atomic propositions is standard and it is meant to capture the intuition that if a proposition is true in a certain state, then any action can be taken without fear of compromising the "local" truth value[10].

[9]Abbreviated LNE condition, henceforth.

[10]As an aside, the case of a negated proposition $\neg p$ could be considered here. The appropriated semantic clause would be

$$\Phi, s \; ||\!\!= p \text{ iff } \neg p \in \Pi(s).$$

The treatment of $\neg p$ would then be symmetric.

Supervisor Map Assignment

$$\phi(s)(\sigma) \leftarrow 1 \text{ for all } \sigma \in \Sigma[s].$$

The assignment for the supervisor map must in this case be the "most liberal" possible.

Case: $\Sigma[s] = \emptyset$

The special case when $\Sigma[s] = \emptyset$ must be considered[11]. The semantic rule is thus specified as

$$\mathcal{E}, s \;\; ||\!\!= p.$$

In this way, once in state s the only thing that the system could do and still maintain p true is to "block".

Remark: The LNE condition is meaningless in this case.

□

$\boxed{\neg f}$

In the standard CTL semantics, negation is easily handled. The controlled semantics of a negation must instead be specified considering the different outcomes which depend on the presence of uncontrollable events.

[11]This case happens when the Kripke structure (i.e. the underlying automaton) has a "sink" state. This is what Ramadge and Wonham appropriately call a *blocking* system.

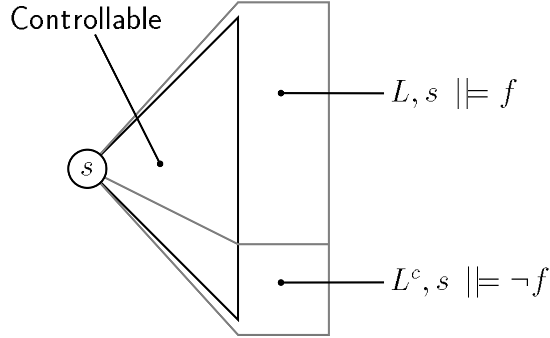

Figure 3.4: *The characterization of the controlled semantics of f and $\neg f$ at state s, without uncontrollable events ($\Sigma_u = \emptyset$).*

SEMANTICS

$$L, s \;||\!\!= \neg f \quad \text{iff} \quad L, s \;||\!\!\not= f. \tag{3.11}$$

or

$$L, s \;||\!\!= \neg f \quad \text{iff} \quad L^c, s \;||\!\!= f. \tag{3.12}$$

Note that the semantics of the negation depends on the ability to actually "complement" the language L_f (i.e. the language such that $L_f, s \;||\!\!= f$), by means of the supervisory action.

Unfortunately, this works with no problems only when there are no uncontrollable events in $\Sigma[s]$ (see fig. 3.4 for an illustration of this fact). In this

case, the supervisor map assignment is simply the one described by equations. (3.13) and (3.14). However, the presence of uncontrollable events can prevent the system from achieving the language L^c by means of supervision.

Figure 3.5 illustrates this situation in the "simple" case where L does contain all the sequences of events starting with an uncontrollable one[12].

With uncontrollable events, the LNE condition tells us when $\neg f$ can only be satisfied by Φ at state s.

SUPERVISOR MAP ASSIGNMENT

$$\phi(s)(\sigma) \leftarrow 1, \quad \text{if} \quad \sigma \in \Sigma_c[s] \wedge \sigma w \in L, \text{ for some } w \in \Sigma^*, \tag{3.13}$$

$$\phi(s)(\sigma) \leftarrow 0, \quad \text{if} \quad \sigma \in \Sigma_c[s] \wedge \sigma w \notin L, \text{ for all } w \in \Sigma^*. \tag{3.14}$$

This assignment ensures that the semantics of negation is the one expressed in (3.11) and (3.12).

LANGUAGE NON-EMPTINESS CONDITION

The condition that ensures the satisfiability of $\neg f$ at state s by a non empty language $L_{\neg f}$ is the following.

$$\forall_{\sigma \in \Sigma_u[s]} . \left(\neg \exists_{L'} . \left((L' \subseteq \Sigma^*) \wedge (\sigma L', s \;|\!\!\models f)\right)\right). \tag{3.15}$$

When this condition holds, the semantics collapses

$$\Phi, s \;|\!\!\models \neg f. \tag{3.16}$$

[12]A more precise illustration would consider the case where each of L_f and $L_{\neg f} = L_f^c$ both contain controllable *and* uncontrollable prefix events.

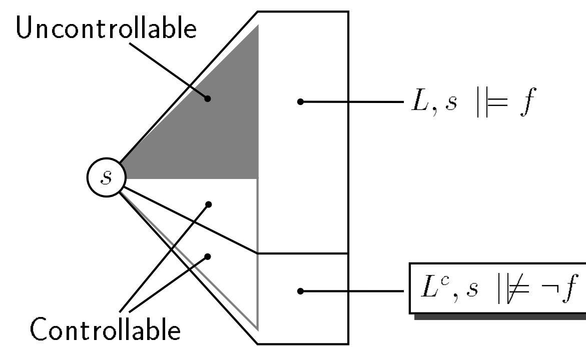

Figure 3.5: *When $\Sigma_u[s] \neq \emptyset$, while it could be possible to achieve satisfiability for a formula f by a language L, it might not be possible to use L^c to satisfy $\neg f$.*

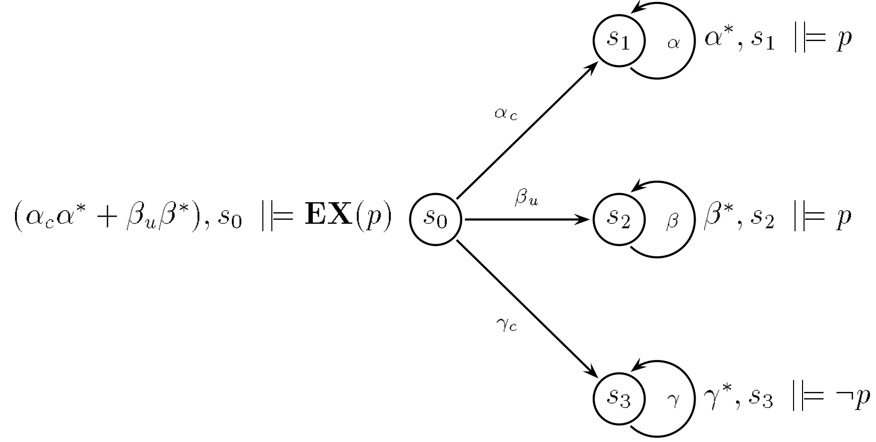

Figure 3.6: *An illustration of the difficulties that we encounter with negation and synthesis of supervisors. In this case we assume that* $\Pi(s_1) = \Pi(s_2) = \{p\}$ *and* $\Pi(s_3) = \{\neg p\}$ *with* $p \in AP$. *While it is easy to come up with a supervisor that satisfies* $\mathbf{EX}(p)$, *it is impossible to find one that satisfies* $\neg\mathbf{EX}(p) \equiv \mathbf{AX}(\neg p)$.

Remark: The fact that sometime a meaningful supervisory action at a state s cannot be achieved because of negation should not be surprising. In [66], the existence of a (proper) supervisor is subject to the "controllability" of the language K (the "desired behavior").

Since in the controlled semantic characterization for the negation of f the supervisor is actually switching between a "desired behavior" – as specified by a formula f – and its "opposite", it is also switching between a "potentially controllable" (in the Ramadge and Wonham sense) language satisfying f and an uncontrollable one satisfying $\neg f$.

As a further example of this phenomenon consider the case depicted in fig. 3.6.

□

AX(f)

The universal *next time* operator has the following characterizations for its semantics and supervisor map.

SEMANTICS

$$L, s \;|\!\models \mathbf{AX}(f) \text{ iff}$$

$$\text{for } \underline{\text{all}} \text{ states } t \text{ such that } \delta(s,\sigma) = t \text{ and } \varphi(s)(\sigma) = 1,$$

$$L[t], t \;|\!\models f \text{ and } L = \bigcup_t \sigma L[t]. \tag{3.17}$$

The semantic clause is subject to the restriction of the associated LNE.

SUPERVISOR MAP ASSIGNMENT

The semantics of the **AX** operator is ensured by the following assignment.

$$\phi(s)(\sigma) \leftarrow 1, \text{ if}$$

$$(\delta(s,\sigma) = s') \wedge (L'[s'], s' \;|\!\models f) \wedge (L'[s'] \neq \Phi), \tag{3.18}$$

$$\phi(s)(\sigma) \leftarrow 0, \text{ if}$$

$$(\delta(s,\sigma) = s'') \wedge [(L''[s''], s'' \;|\!\models f) \Rightarrow (L''[s''] = \Phi)]. \tag{3.19}$$

Remark: The clause specifying when the supervisor map can disable the transition reflects the fact that two different supervisors may at the same time satisfy both f and $\neg f$ at a state s. Therefore, it must be ensured that the supervisor will lead to a state where the formula f (from $\mathbf{AX}(f)$) will be satisfied by a non empty language.

Remark: Of course, all the uncontrollable events must lead to states q_i such that $L[q_i], q_i \;\; ||\!\!= f$. The supervisor assignment is obviously assumed to be equal to 1 for all $\upsilon \in \Sigma_u[s]$.

Language Non–emptiness Condition

The LNE condition condition is similar to the conditions appearing in the supervisor map assignment, plus a general condition on uncontrollable events.

$$\forall_{\kappa \in \Sigma_c[s]}.((\varphi(s)(\kappa) = 1) \wedge (\delta(s,\kappa) = s') \wedge (L'[s'], s' \;\; ||\!\!= f) \\ \Rightarrow \;\; L'[s'] \neq \Phi), \tag{3.20}$$

$$\exists_{\upsilon \in \Sigma_u[s]}.((\delta(s,\upsilon) = s') \wedge (L'[s'], s' \;\; ||\!\!= f) \\ \Rightarrow \;\; L'[s'] \neq \Phi). \tag{3.21}$$

Again, when (3.20) and (3.21) do not hold it must be concluded that

$$\Phi, s \;\; ||\!\!= \mathbf{AX}(f). \tag{3.22}$$

Remark: Note the use of universal quantification over all $\kappa \in \Sigma_c[s]$ in (3.20). Its use intends to stress the "universality" character of the **AX** operator. As

it will be seen soon, an existential quantifier will be used for the **EX** operator.

□

EX(f)

The existential next-time operator has the characterizations shown below. Note that there are some technical points that *do* raise some difficulties that must eventually be resolved. This is not surprising, since existential nature of **EX** implies a *choice* that must eventually be made in the evolution of the system.

Semantics

$L, s \; \| \!\!\models \mathbf{EX}(f)$ iff

for <u>some</u> states t, such that $\delta(s, \sigma) = t$,

$$L[t], t \; \| \!\!\models f \text{ and } L = \bigcup_{\substack{\sigma \in \Sigma[s] \\ s' = \delta(\sigma, s)}} \sigma L[s'].$$

The semantic clause is subject to the LNE condition for the **EX** operator.

Supervisor Map Assignment

$\phi(s)(\sigma) \leftarrow 1$, for all $\sigma \in \Sigma[s]$.

As usual, we assume that $\varphi(s)(v) \leftarrow 1$, for all $v \in \Sigma_u[s]$.

Remark: The supervisor map assignment must be as liberal as possible. It is not relevant whether the system takes some step that will lead to a state s' (from $\delta(s, \sigma) = s'$), where $\Phi, s' \;|\!\models f$. As long as at least one of the "next" states satisfies f with a non empty language, the semantic clause is ensured. The LNE condition states exactly this intuition.

LANGUAGE NON–EMPTINESS CONDITION

We can ensure that the language L satisfying $\mathbf{EX}(f)$ at state s is non empty, as long as at least one of the "next" states satisfies f with a non empty language.

$$\exists_{\sigma \in \Sigma[s]}.\,((\delta(s, \sigma) = s') \wedge (L'[s'], s' \;|\!\models f) \wedge (L'[s'] \neq \Phi))\,.$$

This is the only requirement to ensure that $\mathbf{EX}(f)$ is satisfied at a state s by a non empty language.

□

$\boxed{\mathbf{A}[f_1 \,\mathbf{U}\, f_2]}$ ———

Recall that standard CTL semantics for the **AU** operator is as follows:

$M, s \models \mathbf{A}[f_1 \,\mathbf{U}\, f_2]$ iff

for <u>all</u> paths $(s_0 = s, s_1, s_2, \ldots)$,

$$\exists_{i \geq 0}.\,((M, s_i \models f_2) \wedge \forall 0 \leq j < i.(M, s_j \models f_1))\,.$$

Where M is the underlying Kripke structure (or FSM plus the assignment Π).

The **AU** (and **EU**) operator involves several "paths" in the underlying FSM. Hence its controlled semantics (along with the supervisor map assignment) cannot be determined "locally", i.e. just by looking at the state s or the set $\mathbf{N}[s]$; instead, the controlled semantics must be defined by looking at the paths leading away from s, as in the standard CTL case[13].

SEMANTICS

In order to define the language L such that

$$L, s \;||\!\!= \mathbf{A}[f_1 \,\mathbf{U}\, f_2],$$

the following definitions are introduced.

Given the underlying plant $\mathcal{P}$ (or $\mathcal{G}$) a *full-path* of it is an infinite sequence of states $(s_0, s_1, s_2, \ldots)$. Note that a full-path starts at the initial state s_0. Otherwise, the path is a *suffix path* $x^i = (s_i, s_{i+1}, s_{1+2}, \ldots)$.

Next there are two auxiliary definitions about sets of states and suffix languages.

1. The set of states $\mathbf{S}_{f_1}$ (resp.$\mathbf{S}_{f_2}$) where f_1 (resp. f_2) is satisfied.

$$\mathbf{S}_{f_1} \quad \stackrel{\triangle}{=} \quad \{s \mid \exists_{L \neq \Phi}.\,(L, s \;||\!\!= f_1)\},$$

[13]The semantics given for the "until" formulæ in standard CTL literature can be given in terms of a *fixpoint* operation (see for example [36]). For the $\mathbf{A}[f_1 \,\mathbf{U}\, f_2]$ formulæ we have that

$$\mathbf{A}[f_1 \,\mathbf{U}\, f_2] \equiv \mu\mathcal{F}.f_2 \vee (f_1 \wedge \mathbf{AX}(\mathcal{F})).$$

Where $\mathcal{F}$ is a set of states of the underlying FSM.

$$\mathbf{S}_{f_2} \triangleq \{s \mid \exists_{L \neq \Phi}.\,(L, s \;|\!\models f_2)\}.$$

2. The set of suffix languages satisfying f_1 (resp. f_2) over the elements of $\mathbf{S}_{f_1}$ (resp. $\mathbf{S}_{f_2}$).

$$\mathbf{L}_{f_1} \triangleq \{L \mid (s \in \mathbf{S}_{f_1}) \wedge (L, s \;|\!\models f_1)\},$$
$$\mathbf{L}_{f_2} \triangleq \{L \mid (s \in \mathbf{S}_{f_2}) \wedge (L, s \;|\!\models f_2)\}.$$

Define also the following set.

$$\mathbf{U}[s] \triangleq \{q \mid (q \in \mathbf{N}[s]) \wedge (L[q], q \;|\!\models \mathbf{A}[f_1 \,\mathbf{U}\, f_2])\}\,. \tag{3.23}$$

That is to say, the set of successor states where the formula $\mathbf{A}[f_1 \,\mathbf{U}\, f_2]$ is satisfied by a language $L[q]$. Finally, indicate with

$\varphi_1(p)$ the supervisor map that ensures the satisfiability of f_1 at a state p,

$\varphi_2(p)$ the supervisor map that ensures the satisfiability of f_2 at a state p.

The controlled semantics for **AU** needs to resort to the standard CTL semantics in order to avoid the the problem depicted in Figure 3.3.
Consider the set of full-paths

$$X_{\mathbf{AU}} \triangleq \{x \mid \text{ subject to the conditions shown below}\}. \tag{3.24}$$

There exist $i \leq 0$, $L_1 \in \mathbf{L}_{f_1}$, and $L_2 \in \mathbf{L}_{f_2}$ such that

1. $L_2, \varphi_2, x^i \models f_2$ and

2. $\forall_j.\,((j < i) \Rightarrow (L_1, \varphi_1, x^j \models f_1))$.

The definition of $X_{\mathbf{AU}}$ is the basis for the standard CTL semantics. Now consider a restricted subset of $X_{\mathbf{AU}}$ containing those full-paths for which there exists a prefix that does not contain a given state s (the notation $x_p x^k$ denotes a full-path with a prefix x_p of length $k-1$).

$$(X_{\mathbf{AU}} \mid s) \triangleq \{x \mid \exists_k. \Big((k > 1) \wedge (x_p x^k \in X_{\mathbf{AU}}) \wedge (s \notin x_p)\Big)\}, \qquad (3.25)$$

where $s \in S$.

The definition of the language L is given in two clauses.

1. $s \in \mathbf{S}_{f_2}$: Consider the full-paths $x_p s_i y \in (X_{\mathbf{AU}} \mid s)$ for some s_i, such that $usws_i z \in X_{\mathbf{AU}}$. I.e. s_i is the first state *after* s where the suffix path starting at s and another full-path over which $\mathbf{A}[f_1 \,\mathbf{U}\, f_2]$ is satisfied cross[14]. Now, the language L is defined as

$$L \triangleq (L[s] \setminus_{s_i} (L_{f_1}[s_i])^c), \qquad (3.26)$$

In other words, the language satisfying f_2 at state s is "chopped" off the languages not satisfying f_1 at the states s_i.

2. $s \in \mathbf{S}_{f_1}$ (with some $L_1[s] \in \mathbf{L}_{f_1}$):

$$L \triangleq \bigcup_{q \in \mathbf{U}[s]} \sigma \cdot L[q], \quad \text{if } \delta(\sigma, s) = q, \text{ where} \qquad (3.27)$$

$$\varphi(s)(\sigma) = 1 \ \text{ and}$$

$$\overline{\sigma \cdot L[q]} \subseteq \overline{L_1[s]}. \qquad (3.28)$$

[14]Technically there are an infinite numbers of such paths. However it is always possible to restrict the number to a finite one, if cycles in the structure $\mathcal{P}$ are "collapsed".

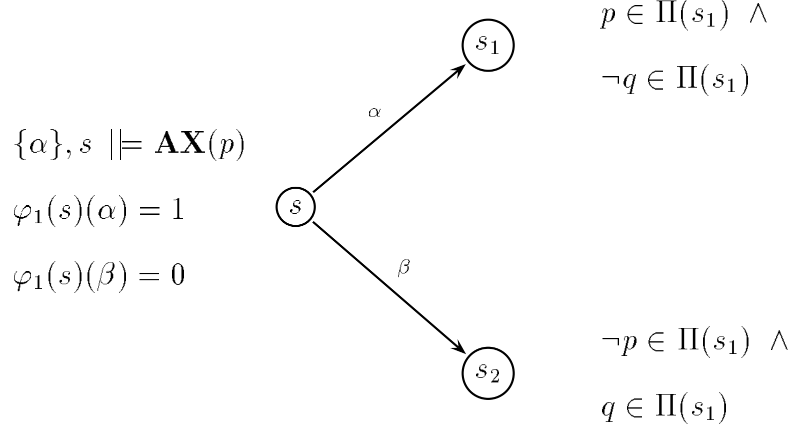

Figure 3.7: *A problematic case involving the supervisor map assignments needed to satisfy* $\mathbf{AX}(p)$ *and* $f \equiv \mathbf{A}[\mathbf{AX}(p)\,\mathbf{U}\,q]$ *at state* s*. Formula* $\mathbf{AX}(p)$ *is satisfied with the* φ_1 *assignment shown, but, by reversing it,* $\mathbf{A}[\mathbf{AX}(p)\,\mathbf{U}\,q]$ *would be satisfied by the language* $L = \{\beta\}$*, if we did not impose any constraints on the supervisors* φ_1 *and* φ_f*.*

Remark: Condition (3.26) ensures that the language that satisfies f_2 at a state s is properly "trimmed" in order to make sure that its concatenation with the prefix where f_1 is satisfied does not interfere with the sub-language corresponding to another full-path as depicted in Figure 3.3.

Remark: Condition (3.28) may not seem necessary, but it was introduced because it guarantees an "extensionality" property of the supervisor map assignments. Figure 3.7, shows a problematic case where the formula $\mathbf{AX}(p)$, coupled with q. If condition (3.28) were not in place, it would be possible

to satisfy both $\mathbf{AX}(p)$ and $\mathbf{A}[\mathbf{AX}(p)\,\mathbf{U}\,q]$, but with two different supervisor maps. This seems quite unnatural and the restriction imposed by (3.28), automatically rules out this possibility. (Of course, the LNE condition for **AU** will reflect this state of affairs).

Supervisor Map Assignment

The supervisor map assignment that ensures the semantics for the **AU** operator is similar to the one for the **AX** one.

$$\varphi(s)(\sigma) \leftarrow 1, \quad \text{if}$$

$$(s \in \mathbf{S}_{f_2}) \wedge \varphi_2(s)(\sigma) = 1 \tag{3.29}$$

$$\varphi(s)(\sigma) \leftarrow 1, \quad \text{if}$$

$$(\delta(s,\sigma) = q) \wedge (L[q] \;|\!\models \mathbf{A}[f_1\,\mathbf{U}\,f_2]) \wedge (L[q] \neq \Phi) \wedge \left(\overline{\sigma \cdot L[q]} \subseteq \overline{L_1[s]}\right) \tag{3.30}$$

$$\varphi(s)(\sigma) \leftarrow 0, \quad \text{otherwise} \tag{3.31}$$

The last conjunct in the second clause is condition (3.28) again. As usual, it must be true that $\forall v \in \Sigma_u[s].\varphi(s)(v) \leftarrow 1$. If not the LNE condition makes the language empty.

Remark: The assignment (3.29) may seem to contradict the characterization of the language satisfying $\mathbf{A}[f_1\,\mathbf{U}\,f_2]$ given in (3.26). However, it must be remembered that the supervisor map assignment is *local*. The characteriza-

tion given in (3.26) is ensured by the inductive construction of the supervisor assignments.

Language Non-emptiness condition

The LNE for the **AU** operator states – as in the previous cases – that any attempts to disable uncontrollable events will result in an empty language $L[s]$. The resulting $L[s]$ will also be empty when the union operation in (3.27) would yield an empty language.

$$\forall_{\kappa \in \Sigma_c[s]} . \left((\varphi(s)(\kappa) = 1) \Rightarrow L[\delta(s,\kappa)] \neq \Phi \right).$$

$$\forall_{\upsilon \in \Sigma_u[s]} . \left(\left(\begin{array}{ll} & \delta(s,\sigma) = q \\ \wedge & L[q] \;\; ||\!\models \mathbf{A}[f_1 \, \mathbf{U} \, f_2] \\ \wedge & \overline{\sigma \cdot L[q]} \subseteq \overline{L_1[s]} \end{array} \right) \Rightarrow L[\delta(s,\upsilon)] \neq \Phi \right).$$

Otherwise, it must be concluded that

$$\Phi, s \;\; ||\!\models \mathbf{A}[f_1 \, \mathbf{U} \, f_2].$$

□

$\boxed{\mathbf{E}[f_1 \, \mathbf{U} \, f_2]}$ ————————————————

The **EU** operator can be treated in essentially the same way as the **AU** operator. However, the treatment reserved for the **EX** operator must also be taken into account.

As in the case of the **AU** operator, the standard CTL semantics for **EU**is as follows:

$$M, s \models \mathbf{E}[f_1 \,\mathbf{U}\, f_2] \text{ iff}$$

$$\text{for } \underline{\text{some}} \text{ paths } (s_0 = s, s_1, s_2, \ldots),$$

$$\exists_{i \geq 0}. \left((M, s_i \models f_2) \wedge \forall_{0 \leq j < i}. \left(M, s_j \models f_1 \right) \right).$$

The construction for the semantics clause of the **EU** operator will be a recursive one[15], as in the case of **AU**.

SEMANTICS

The definitions for $\mathbf{S}_{f_1}$, $\mathbf{S}_{f_2}$, $\mathbf{L}_{f_1}$, $\mathbf{L}_{f_2}$, φ_1 and φ_2, introduced for the **AU** operator are assumed to be the same. The set $\mathbf{U}[s]$ is redefined as

$$\mathbf{U}[s] \triangleq \{q \mid (q \in \mathbf{N}[s]) \wedge (L[q], q \;|\!\models \mathbf{E}[f_1 \,\mathbf{U}\, f_2])\}.$$

I.e., the set of successor states where the formula $\mathbf{E}[f_1 \,\mathbf{U}\, f_2]$ is satisfied by a language $L[q]$.

The sets of full-paths $X_{\mathbf{EU}}$ and $(X_{\mathbf{EU}} \mid s)$ are defined analogously to (3.24) and (3.25) for the **AU** case.

The definition of the language L becomes

---

[15] The corresponding fixpoint characterization is the following.

$$\mathbf{E}[f_1 \,\mathbf{U}\, f_2] \equiv \mu\mathcal{F}.f_2 \vee (f_1 \wedge \mathbf{EX}(\mathcal{F})).$$

The terminology is the same for the **AU** case.

1. $s \in \mathbf{S}_{f_2}$:

$$L \triangleq L[s]. \tag{3.32}$$

2. $s \in \mathbf{S}_{f_1}$ (with some $L_1[s] \in \mathbf{L}_{f_1}$): consider the set of states

$$F_2 = \{s' \mid s' \in \mathbf{S}_{f_2} \wedge \exists_{s'' \in \mathbf{U}[s]}. \left(s'' \overset{\delta, \varphi_{\mathbf{EU}}}{\rightsquigarrow} s'\right)\}.$$

   I.e. F_2 is the set of states in $\mathbf{S}_{f_2}$ which is reachable from s under supervision. The language L is therefore defined as

$$L \triangleq (L_1[s] \setminus_{(s' \in F_2)} (L_2[s'])^c), \quad \text{for } L_2[s'] \in \mathbf{L}_{f_2}. \tag{3.33}$$

Remark: The characterization of the language L for **EU** is therefore almost symmetrical to that of **AU**. The rationale behind this choice is that, whenever the system reaches a state where f_1 is satisfied by a non empty language, then it makes sense to limit its evolution only to those behaviors that actually make the full **EU** form satisfied.

Supervisor Map Assignment

The supervisor map assignment for the **EU** operator is very similar to the **AU** case.

$$\varphi(s)(\sigma) \leftarrow 1 \quad \text{if} \quad (s \in \mathbf{S}_{f_2}) \wedge (\varphi_2(s)(\sigma) = 1),$$

$$\varphi(s)(\sigma) \leftarrow 0 \quad \text{if} \quad (s \in \mathbf{S}_{f_2}) \wedge (\varphi_2(s)(\sigma) = 0),$$

$$\varphi(s)(\sigma) \leftarrow 1 \qquad \text{otherwise}.$$

Remark: As in the case of the **EX** operator the supervisor map assignment should be as liberal as possible. The only obligation of supervisor map assignment is to respect the constraints imposed by the satisfiability condition of f_2 (via φ_2).

Language Non–emptiness Condition

The non-emptiness of the language L such that $L, s \;|\!\models \mathbf{E}[f_1 \, \mathbf{U} \, f_2]$ is imposed by conditions similar to those for the **EX** case.

$$\begin{aligned} \exists_{\sigma \in \Sigma[s]}.\ ((s \in \mathbf{S}_{f_1}) & \\ \wedge \quad (\delta(s,\sigma) = s') & \\ \wedge \quad (L'[s'], s' \;|\!\models \mathbf{E}[f_1 \, \mathbf{U} \, f_2]) & \\ \wedge \quad (L'[s'] \neq \Phi)). & \end{aligned} \tag{3.34}$$

Note that L in the case $s \in \mathbf{S}_{f_2}$, is non empty by definition, and we can therefore assume that φ_2 does not try to disable any uncontrollable event.

Moreover, since no restrictions are imposed on the next states, it is not necessary to distinguish between controllable and uncontrollable events in order to define the LNE, when $s \in \mathbf{S}_{f_1}$.

If the above condition does not hold then the semantic clause becomes $\Phi, s \;|\!\models \mathbf{E}[f_1 \, \mathbf{U} \, f_2]$.

□

$\boxed{f_1 \wedge f_2}$

The standard CTL semantics for conjunctions is shown below.

$$M, s \models f_1 \wedge f_2 \quad \text{iff} \quad M, s \models f_1 \ \text{ and } \ M, s \models f_2$$

Where M is the underlying FSM (or Kripke structure).

Semantics

The semantics clause for conjunctions is straightforward.

$$L, s \ ||\!\!= f_1 \wedge f_2 \qquad \text{iff} \qquad L_{f_1}, s \ ||\!\!= f_1 \ \text{ and } \ L_{f_2}, s \ ||\!\!= f_2 \qquad (3.35)$$
$$\text{where} \quad L = L_1 \cap L_2.$$

Supervisor Map Assignment

The definition of the local supervisor map assignment is simply

$$\varphi_{\wedge}(s)(\sigma) \leftarrow 1 \qquad \text{if} \qquad \varphi_{L_{f_1}}(s)(\sigma) = 1 \wedge \varphi_{L_{f_2}}(s)(\sigma) = 1,$$
$$\varphi_{\wedge}(s)(\sigma) \leftarrow 0 \quad \text{otherwise.}$$

Language Non–emptiness Condition

Since the supervisor map assignment is derived from those of the two subformulæ, the LNE must simply ensure that the intersection of the two languages is not empty. Locally, this amounts to check that the two map assignments are not disjoint.

□

$\boxed{f_1 \vee f_2}$

Disjunctions of well formed formulæ pose problems. It turns out that it is not possible to define their semantics using the CTL guidelines, without paying a price.

The standard CTL semantics for disjunctions is the following.

$$M, s \models f_1 \vee f_2 \quad \text{iff} \quad M, s \models f_1 \ \text{ or } \ M, s \models f_2,$$

Where M is the underlying FSM (or Kripke structure).

Semantics

While somewhat unintuitive, the controlled semantics for disjunctions must be defined in the following terms.

$$L, s \ \|\!\models f_1 \vee f_2 \quad \text{iff} \quad L_{f_1}, s \ \|\!\models f_1 \ \text{ or } \ L_{f_2}, s \ \|\!\models f_2 \tag{3.36}$$

$$\text{where } \ L = \begin{cases} L_{f_1} & \text{if } \ L_{f_2} \subseteq L_{f_1}, \\ L_{f_2} & \text{if } \ L_{f_1} \subseteq L_{f_2}, \\ \Phi & \text{otherwise.} \end{cases}$$

The above definition is so restrictive because the languages L_{f_1} and L_{f_2} must be guaranteed to be "compatible". As seen in Section 3.2.2, there are serious problems for the intuitive semantics.

Remark: The notion creeping in the characterization of disjunctions is that of *non conflicting* languages that Ramadge and Wonham use to treat "modular" supervisor synthesis [67].

Supervisor Map Assignment

Given the above definition of controlled semantics the following characterizations of the supervisor map assignment for disjunctions is satisfactory.

$$\varphi_{\vee}(s)(\sigma) \leftarrow 1 \quad \text{if} \quad \varphi_{L_{f_1}}(s)(\sigma) = 1 \vee \varphi_{L_{f_2}}(s)(\sigma) = 1,$$

$$\varphi_{\vee}(s)(\sigma) \leftarrow 0 \quad \text{if} \quad \text{the containment conditions do not hold.}$$

Note that in absence of the containment conditions, in both cases it could be concluded that $L \subset L_{f_1} \cap L_{f_2}$ and $L \subset L_{f_1} \cup L_{f_2}$.

Language Non–emptiness Condition

The LNE condition for disjunctions is also a constraint on the applicability of the supervisor map construction for both disjunctions and conjunctions. The containment conditions in (3.36) serve as the LNE condition for disjunctions.

$$L_{f_1} \subseteq L_{f_2}, \tag{3.37}$$

$$L_{f_1} \supseteq L_{f_2}. \tag{3.38}$$

Since it is assumed that either $L_{f_1} \;|\!\models f_1$ or $L_{f_2} \;|\!\models f_2$, the resulting supervisor assignment will still respect the constraint on the events $\upsilon \in \Sigma_u[s]$.

□

3.2.4 Circumventing the Problems

As noted in Section 3.2.2 there are some problems with the specifications of the controlled semantics for some classes of formulæ– e.g. the problem of

"choice" with disjunctions – which may hinder the automatics construction of a supervisor map.

The steps that can be taken in order to resolve these problems depend on the nature of application of the CTL synthesis scheme, which, it should be remembered, is aimed at the aiding of a practitioner whose job is to build a controller program. The following alternatives are available.

- The scheme can simply be declared erroneous.
- The logic used can be further restricted.
- The user of the scheme (as it will be embodied by a computer program) can be forced to make a choice as to which disjunct will be considered acceptable.

It would be desirable to avoid the first alternative. The other two are briefly discussed next.

Restricting or Changing the Logic. The Logic CTL^- [58] could solve the problems since it disallows disjunctions over *path* formulæ[16].

Leaving the choice to the user. This is a viable alternative, but it could imply an overwhelming work for the user in the case of very intricate specifications. As a counterargument, it could be noted that (from literature,

[16] Technically, a superset of CTL^- would be used since the **EX** operator is not disallowed.

especially on CTL-based analysis) the specifications usually come in the form of rather short formulæ which are then conjoined.

The choices made for the implementation of the Control-D system are described in Chapter 4.

3.2.5 CTL Soundness of Controlled Semantics

Section 3.2.3 contains the definitions for the "controlled semantics" of a set of CTL formulæ for a given plant specification $\mathcal{P}$ and discusses the limitations on the approach due to the implicit choices of the formalism. This section introduces a notion of *soundness* for the construction and forms a basis for the development of a supervisor map synthesis algorithm.

Notation: If φ is a supervisor map, then we use $M[L, \varphi]$ to indicate the FSM of language L operating under the supervision of φ.

Soundness Theorem

The notion of soundness that will be introduced is carefully tailored in order to allow for reuse of standard results coming from the temporal logic verification field. More precisely, it will be ensured that *the specification $\mathcal{S}$ is satisfied (i.e. can be verified) by the constrained behavior of the plant $\mathcal{P}$.*

Theorem: *If there exists a supervisor map φ and a language $L \neq \Phi$, such that for a plant FSM $\mathcal{P}$ with s_0 as initial state we have*

1. $L \subseteq \Lambda[\mathcal{P}]$ *and*

2. $L, \varphi, s_0 \ \Vdash \mathcal{S}$,

with $\mathcal{S}$ *being a set of* CTL *formulæ. Then*

$$M[L, \varphi], s_0 \models \mathcal{S}.$$

(Note that this last statement is to be intended in the standard CTL *sense).*

Remark: The theorem states that if a supervisor map restricts the plant behavior to L, then the specification can be verified with respect to the "restricted" machine $M[L, \varphi]$.

Proof. The proof is by induction on the state space and on the structure of the formulæ in $\mathcal{S}$.

Without loss of generality, suppose that $\mathcal{S}$ contains a single formula[17] f. The proof proceeds by cases.

Case: $f \equiv p$ **(with** $p \in \Pi(s_0)$**).**

The definition of the supervisor assignment is such that the result follows immediately

$$L, s_0 \ \Vdash f \quad \Rightarrow \quad M[L, \varphi], s \models f.$$

[17] In practice, this seems to be the most likely case. The specification will be given as a single conjunction.

Case: $f \equiv \mathbf{AX}f'$.

In this case too, the definitions of the supervisor assignment and of the LNE condition condition are such that the result follows immediately.

$$L, s_0 \;||\!\!= \mathbf{AX}f' \;\;\Rightarrow\;\; M[L,\varphi], s \models \mathbf{AX}f'.$$

Case: $f \equiv \mathbf{EX}f'$.

As above, for the case $f \equiv \mathbf{AX}f'$.

Case: $f \equiv \mathbf{E}[f_1\,\mathbf{U}\,f_2]$.

By assumption, there exists a language L and a supervisor map φ such that

$$L, s_0 \quad ||\!\!= \quad \mathbf{E}[f_1\,\mathbf{U}\,f_2].$$

There are two sub-cases to be considered.

Sub-case 1: $L, s_0 \;||\!\!= f_2$ (with $L \neq \Phi$).

$L, s_0 \;||\!\!= f_2$ implies that exist $\sigma_1, \sigma_2, \ldots, \sigma_k \in \Sigma$ and $L_1, L_2, \ldots, L_k \subseteq \Sigma^*$ such that $L = \bigcup_{i=1}^{k}(\sigma_i L_i)$ and for all $i \in 1,\ldots,k$, $\varphi(s_0)(\sigma_i) = 1$. From this, the definition of controlled semantics, the construction of the supervisor map, and by the inductive hypothesis, it follows that $M[L,\varphi], s_0 \models f_2$.

Next, note that

$$M[L,\varphi], s_0 \models f_2$$

implies

$$M[L,\varphi], s_0 \models \mathbf{E}[f_1\,\mathbf{U}\,f_2].$$

To see this fact, suppose that the contrary holds: since $L \neq \Phi$, it must be the case that $M[L,\varphi], s_0 \not\models \mathbf{E}[f_1\,\mathbf{U}\,f_2]$, but this would contradict the fact that $L, s_0 \;|\!\models f_2$ and thus $M[L,\varphi], s_0 \models f_2$.

It remains to be shown that $M[L,\varphi] \models \mathbf{E}[f_1\,\mathbf{U}\,f_2]$ implies $M[L,\varphi] \models f_2$. Which follows immediately from the definition of controlled semantics for the **EU** operator.

Sub-case 2: $L, s_0 \;|\!\not\models f_2$ (and $L \neq \Phi$).

Since $L, s_0 \;|\!\models \mathbf{E}[f_1\,\mathbf{U}\,f_2]$, it must be the case that, because of the definition of the controlled semantics, $L, s_0 \;|\!\models f_1$. Hence, as in the sub-case 1, $M[L,\varphi], s_0 \models f_1$.

It remains to show that, as the language L is "unwound" (under the supervision φ) the result will still be $M[L,\varphi], s_0 \models \mathbf{E}[f_1\,\mathbf{U}\,f_2]$. The argument is again by induction. However the case depicted in Figure 3.8, will require special attention.

Base Case:

The base case will be for a s' such that $\delta(s_0,\sigma) = s'$ and $\varphi(s_0)(\sigma) = 1$, i.e. paths in M of length 1 will be considered.

There are two sub-cases to consider.

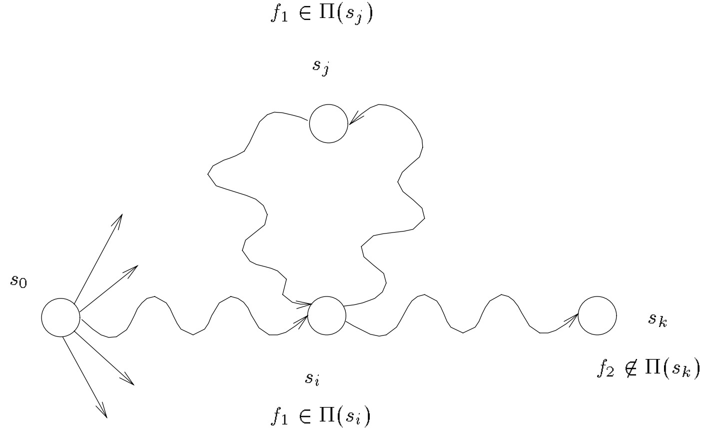

Figure 3.8: A problematic case for the semantic of the **EU** and **AU** operators.

1. $L[s'], s' \;|\!\models f_2$

   The machine having s' as a start state is such that

   $$M[L[s'], \varphi], s' \models f_2,$$

   hence it can be concluded that

   $$((L \setminus L[s'])L[s'])\,, s' \;|\!\models \mathbf{E}[f_1 \,\mathbf{U}\, f_2].$$

   But by the definition of the controlled semantics this mean that

   $$M[L, \varphi], s_0 \models \mathbf{E}[f_1 \,\mathbf{U}\, f_2].$$

2. $L[s'], s' \;|\!\not\models \mathbf{E}[f_1 \,\mathbf{U}\, f_2]$

   This case does not influence the proof, unless all the s' make the formula false. Yet, this cannot be the case because of the assumption that L is not empty and by the supervisor map construction for **EU**.

Inductive Case:

Now suppose that the statement is true for paths of length $i-1$, that is for suffixes of L of length $i-1$. I.e., we have that

$$L[s'], s' \ |\!\models f_1 \text{ and } L[s'], s' \ |\!\models \mathbf{E}[f_1 \, \mathbf{U} \, f_2]$$

where $s' = \delta(s_0, \sigma)$ for some $\sigma \in \Sigma$ such that $\varphi(s_0)(\sigma) = 1$. In particular $M[L[s'], \varphi], s' \models \mathbf{E}[f_1 \, \mathbf{U} \, f_2]$. Therefore, the machine M can be extended by an extra step in order to make it recognize L. Eventually $M[L, \varphi], s_0 \models \mathbf{E}[f_1 \, \mathbf{U} \, f_2]$ can be achieved by this extension process.

Case: $f \equiv \mathbf{A}[f_1 \, \mathbf{U} \, f_2]$**.**

As in the **EU** case, there exists a language L and a supervisor map φ such that

$$L, s_0 \ |\!\models \mathbf{A}[f_1 \, \mathbf{U} \, f_2].$$

Again, there are two cases to be considered.

Sub-case 1: $L, s_0 \ |\!\models f_2$ (and $L \neq \Phi$).

By the same argument used for the **EU** operator, from the definition of the controlled semantics for the **AU** operator it can be concluded that

$$(L, s_0 \ |\!\models f_2) \text{ iff } (M[L, \varphi], s_0 \models \mathbf{A}[f_1 \, \mathbf{U} \, f_2]).$$

Sub-case 2: $L, s_0 \;|\not\models f_2$ (and $L \neq \Phi$).

As in the case of the **EU** operator, since it has been assumed that $L, s_0 \;|\models \mathbf{A}[f_1\,\mathbf{U}\,f_2]$, it must be the case that $M[L,\varphi], s_0 \models f_1$, by the definition of the controlled semantics for **AU**[18].

It must now be shown that $M[L,\varphi], s_0 \models \mathbf{A}[f_1\,\mathbf{U}\,f_2]$ holds after "unwinding" the language L under the supervision of φ. Note that the induction argument used for the **EU** case, does not apply for **AU**, since *all* the next states must be considered.

Since $L \neq \Phi$ it must be the case that the LNE condition for **AU** does not hold. Consider the subset of the events "enabled" at state s_0 by the supervisor map φ. Each of these events leads to a state $s' \in \mathbf{U}[s_0]$ as defined by the controlled semantics of **AU** – see (3.23).

Now partition these states in $\mathbf{U}[s_0]$ into two disjoint subsets[19]

$$
\begin{aligned}
\mathbf{U}_{f_1} \cup \mathbf{U}_{f_2} &\subseteq \mathbf{U}[s_0] \text{ and} \\
\mathbf{U}_{f_1} \cap \mathbf{U}_{f_2} &\neq \emptyset.
\end{aligned}
$$

$\mathbf{U}_{f_1}$ is the set of states s' where $L[s'], s' \;|\models f_1$; $\mathbf{U}_{f_2}$ is the set of states s'' where $L[s''], s'' \;|\models f_2$.

Given this partition, the following statements can be verified.

[18] Moreover, by the definition of the controlled semantics of the **AU** operator, there must exist another language L_{f_1} such that $L_{f_1}, s_0 \;|\models f_1$ and $L \subseteq L_{f_1}$.

[19] In general, we cannot ensure that all the states in $\mathbf{U}[s]$ for any s, can be reached from it. In fact the LNE and the "prefix containment" (3.28) conditions may prevent this from happening.

1. For all the states $s'' \in \mathbf{U}_{f_2}$ the argument used in sub-case 1 can be applied in order to assert that $M[L[s''], \varphi], s'' \models \mathbf{A}[f_1 \, \mathbf{U} \, f_2]$.

2. For all the states $s' \in \mathbf{U}_{f_1}$, the present argument can be iterated in order to achieve the same conclusion. The only caveat is to consider carefully the situation depicted in fig. 3.8 (the situation applies also to the **AU** case), in order to avoid a circularity in the argument. Note however, that the unwinding of the argument can be coupled with the concomitant marking of the *strongly connected components* (SCC) of the underlying Kripke structure under the supervision of φ (after all it is a finite structure). This marking process will avoid the circularity of the argument.

   The iterative application of the two present arguments finally yields that $M[L[s'], \varphi], s' \models \mathbf{A}[f_1 \, \mathbf{U} \, f_2]$.

Hence, from the two previous arguments, the structure of the underlying Kripke structure (the "plant" $\mathcal{P}$) and the definition on the supervisor map assignment for **AU**, it follows that

$$M[L, \varphi], s_0 \models \mathbf{A}[f_1 \, \mathbf{U} \, f_2].$$

Case: $f \equiv f_1 \vee f_2$.

Given the conditions in (3.36) and the supervisor map construction defined for the disjunction case, the result follows immediately.

$$L, s_0 \; ||\!\!= (f_1 \vee f_2) \quad \Rightarrow \quad M[L, \varphi], s \models (f_1 \vee f_2).$$

Case: $f \equiv f_1 \wedge f_2$.

As in the case of disjunctions, given the conditions in (3.35) and the supervisor map construction defined for the conjunction case, the result follows immediately.

$$L, s_0 \; ||\!\!= (f_1 \wedge f_2) \quad \Rightarrow \quad M[L, \varphi], s \models (f_1 \wedge f_2).$$

□

Having established this relationship between the notion of controlled semantics and the standard definition of satisfiability in the model checking context (for CTL), it is now time to start answering some basic questions about the algorithms for the synthesis task.

3.3 Simple CTL-Synthesis Algorithm

The existence of an algorithm capable of synthesizing the supervisor map is established in this section. The algorithm is very simple, but has a very high

time complexity. Chapter 4 provides a description of a better algorithm and its implementation.

3.3.1 Existence of a Supervisor Synthesis Algorithm

It is a simple matter to come up with a supervisor synthesis algorithm that would produce a map which respects the semantics definitions just given (with the restriction that either (3.37) or (3.38) holds):

Algorithm:

INPUT: FSM model of a *plant*, $\mathcal{P}$ and a specification as a set $\mathcal{S}$ of CTL formulæ.

METHOD:

1. For each state $s \in Q_{\mathcal{P}}$ choose a subset of $\Sigma[s]$ (containing all of $\Sigma_u[s]$) – i.e. choose the supervisor map for state s.

2. Run the MODEL CHECKING ALGORITHM on the resulting supervised $\mathcal{P}$.

3. If $\mathcal{S}$ is satisfied return the supervisor map. We are done.

4. Repeat step 1 with a different choice.

Of course this algorithm terminates (there is a finite number of choices that can be made), its computational complexity is horrible and the a supervisor map produced may satisfy the specification only by Φ.

Complexity: There are at most $O\left(\mid S \mid \cdot \mid \mathcal{P} \mid \cdot 2^{\mid \mathcal{P} \mid}\right)$ choices for the supervisor in the outer loop[20] besides the linear cost of the CTL model checking algorithm. Hence the algorithm has an exponential time complexity.

The existence of the simple-minded exponential algorithm should not be surprising, given the finiteness of the entities considered. A better algorithm is possible by a straightforward re-elaboration of the standard model checking algorithm given in [24].

[20]The size of $\mathcal{P}$ FSM graph is an upper bound on the effective size of the set of choices, which depends on $\mid \Sigma \mid$ and $\mid \delta \mid$.

Chapter 4

The **Control-D** System

Building a system that embodies a theoretical result poses many interesting challenges in the present setting. There are many choices that have to be made.

- How does one represent the elementary building blocks of algorithms (by efficient data structures) that on paper are straightforward?
- How does one resolve the problems that the theory leaves open?
- What kind of audience is targeted by the system being designed?.
- What kind of *user interface* the system should have?

The Control-D system is an experimental tool that addresses these problems. This chapter contains a description of its core synthesis algorithm and implementation. Section 4.1 describes the synthesis algorithm, the trade-offs

needed to achieve a reasonable usability of the system and an analysis of its complexity. Section 4.2 discusses how the Control-D tool can be used for *both* verification of system properties and synthesis of discrete controllers. Section 4.3.1 describes the user environment.

4.1 Revised Algorithm and The Implementation

The controlled semantics for CTL given in Chapter 3 was built in such a way as to make it easy to construct a state-space search algorithm for the supervisor synthesis problem. The algorithm presented here is a modification of the *labeling* algorithm presented in [24], and it is called the *model restriction supervisor synthesis* algorithm (MRSS).

Limitations: The algorithm embodied in the most recent version of Control-D uses heuristics and imposes some limitations in the case of "until" formulæ, negations, and disjunctions. In the last two cases, the controlled semantics requires that for each state s the overall supervisor action restricts the language satisfying a (negation or disjunction) formula f by actually "looking into the future". This amounts to check, for each state, all the possible supervisor assignments, hence falling back to the exponential algorithm shown at the end of Chapter 3. The simple heuristics that will be introduced here allows

the algorithm to maintain a linear time complexity, while not compromising the controlled semantics.

Remark: One may ask, why bother with the *labeling* algorithm, while the tendency has been toward the use of BDD's and *symbolic encoding* (cfr. [21, 22, 13]) of the state space?

First of all the BDD based algorithms eventually encode the labeling scheme. Therefore, the labeling scheme turns out to be necessary to understand the BDD encoding. Secondly, there is some consensus among practitioners of verification, that BDD's, though extremely useful, are not so easy to manage (due to the requirements of the heuristics on variable ordering). Finally, the emergence of new algorithmic techniques based on *compositional* modeling [27], might eventually make the "state space sweeping" schemes more appealing.

4.1.1 State Space Traversing Algorithm

The MRSS algorithm builds the supervisor map φ in an incremental way by sweeping the state space of a plant $\mathcal{P} \triangleq \langle Q, \Sigma, s_0, \delta \rangle$ (it therefore assumes a complete reachability graph construction), while considering sub-formulæ of a specification $\mathcal{S}$. It is therefore a variation of the algorithm found in Clarke and Emerson's original paper [24].

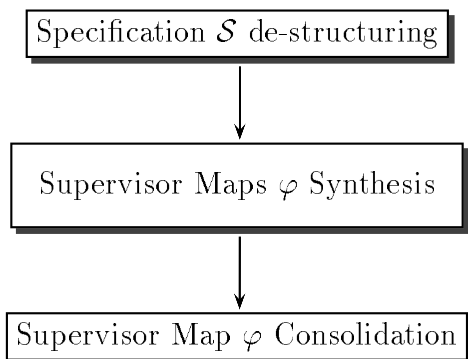

Figure 4.1: *Three stages schematic of the Model Restriction Supervisor Synthesis Algorithm.*

Algorithm Stages

The MRSS algorithm works in three stages. Figure 4.1 shows a schematic of the control flow. The first stage is a very simple de-structuring operation of the specification $\mathcal{S}$ (cfr. [24]). The specification formula is broken down into sub-formulæ arranged in a topological order. The second stage "labels" each state with an appropriate supervisor map. Finally in the third stage the supervisor map for the specification $\mathcal{S}$ is reconstructed from the maps for the component sub-formulæ.

The first stage is straightforward. The second stage is where the algorithm spends most of its time. The following notation is used to explain how the algorithm works.

- f is a sub-formula of the specification $\mathcal{S}$.
- $\mathcal{F}$ is the set of all the sub-formulæ of $\mathcal{S}$ ($f \in \mathcal{F}$, since $\mathcal{S}$ is usually a conjunction).

The MRSS algorithm uses the following internal map during the labeling phase

$$\textsf{wff-state-restriction} : \mathcal{F} \times S \times \Sigma \rightarrow \{1, 0\},$$

where S is the set of states of the underlying FSM representation of the plant $\mathcal{P}$. It is obvious that

$$\textsf{wff-state-restriction}(f) = \varphi_f.$$

I.e. the wff selects its appropriate supervisor maps in the overall map wff-state-restriction.

From the definition of wff-state-restriction it is immediate to infer an estimate of the space requirements for MRSS algorithm. The space requirement is $O(|S| \cdot |\mathcal{F}| \cdot |\Sigma|)$[1].

Labeling Phase

The core of the MRSS algorithm works by labeling each state with the appropriate supervisor depending on the wff under consideration. The procedure **label_state_graph** shown in Figure 4.2 is an outline of the actual algorithm. Each of the sub-procedures in **label_state_graph** traverses the state-space

[1] Where the size of $\mathcal{S}$ is the size of the state-space.

```
Procedure  label_state_graph(P, f)
        The arguments are a graph (actually a FSM) encoding the be-
        havior of the plant P, and a CTL formula f.
begin
        if  f ∈ AP          →  label_proposition(P, f);
            f = ¬f'         →  label_negation(P, f);
            f = AX(f')      →  label_all_next(P, f);
            ...
            f = A[f1 U f2]  →  label_all_until(P, f);
            f = EF(f')      →  label_possibly(P, f);
            ...
            f = AG(f')      →  label_invariant(P, f);
            f = EG(f')      →  label_weak_inv(P, f);
        fi
        return wff-state-restriction
end
```

Figure 4.2: *The* **label_state_graph** *procedure.*

and augments the map wff-state-restriction according to the semantic specification given in Chapter 3. Most of the sub-procedures perform a standard Depth First Search (DFS) [28] of the state-space. However, disjunctions and negations require special care and the tradeoffs among these cases will be dealt with separately.

As an example, consider the procedure **label_all_until** which traverses the state space while building the supervisor map for the formula $f = \mathbf{A}[f_1 \, \mathbf{U} \, f_2]$. The DFS traversal uses the {WHITE,GREY,BLACK} labeling scheme used in [28]. The procedure **label_all_until** performs, a simple initialization routine for the DFS and then calls the real traversal function (**label_AU**) with the start state of $\mathcal{P}$. The definition of **label_AU** is given in Figure 4.3.

```
Function  label_AU(P, f, s) : boolean
        The arguments are the plant P, a CTL f = A[f1 U f2] formula,
        and a state s.
begin
        color(s) ← GREY
        if (L ≠ Φ), s |⊨ f2  then
                There is a non empty language L satisfying f2.
                carry_over_supervisor(f2, f, s, wff-state-restriction);
                color(s) ← BLACK;

        else if Φ, s |⊨ f1  then
                f1 is satisfied only by an empty language at s.
                wff-state-restriction(f)(s) ← 0;
                color(s) ← BLACK;

        else
                f1 is satisfied by a non empty language at s.
                Thus, the supervisor at s for f depends on the paths originating at s.
                label_AU_next(P, f, s);
                color(s) ← BLACK;

                Finally check for the AU LNE condition.
                if check_AU_LNE(P, f, s)  then
                        wff-state-restriction(f)(s) ← 0;
end
```

Figure 4.3: The function **label_AU**.

The first two cases of the **label_AU** function are straightforward. The third case involves a recursive call that realizes the DFS traversal of the state space.

The procedure **label_AU_next** (cfr. Figure 4.4) traverses the state-space while checking for forward and backward edges. Both procedures **label_AU** and **label_AU_next** modify the supervisor map for $\mathbf{A}[f_1 \mathbf{U} f_2]$ formula, based on the inductive hypothesis that the supervisor maps for f_1 and f_2 are known. This fact is guaranteed by the topological sort of the specification $\mathcal{S}$ from the first phase of the algorithm.

With respect to the controlled semantics, **label_AU** and **label_AU_next** produce a smaller language satisfying f. This is a consequence of the treatment of "back edges" in **label_AU_next**. This simplification is similar to the one made in [24] for the CTL model checking algorithm.

The correctness of the algorithm for the **AU** operator derives from the inductive construction of the controlled semantics and from a straightforward argument on the behavior of the DFS coloring scheme.

The overall algorithm is driven by the procedure **model_synth** depicted in Figure 4.5, which ties together the first two phases of the algorithm. Since each of the wff-specific functions is a DFS on the state-space, the overall complexity of the MRSS algorithm is linear. However, in order to achieve these results, some heuristics and restrictions had to be imposed with respect to the controlled semantic construction. Section4.1.2 explains these restrictions.

```
Procedure label_AU_next(P, f, s)
begin
    for all {[σ, s'] | δ(s, σ) = s'} do
        if color(s') = WHITE then
            The DFS traversal just found a new state.
            if label_AU(P, f, s') then
                wff-state-restriction(f)(s)(σ) ← 1;
            else
                wff-state-restriction(f)(s)(σ) ← 0;
        else if color(s') = GREY then
            A "back edge" has been found. The transition is disabled
            wff-state-restriction(f)(s)(σ) ← 0;
        else if color(s') = BLACK then
            A "forward edge" has been found.
            A check is necessary to determine whether to disable
            the transition or not
            if Φ, s' ||= f then
                wff-state-restriction(f)(s)(σ) ← 0;
            else
                wff-state-restriction(f)(s)(σ) ← 1;
end
```

Figure 4.4: The procedure **label_AU_next**.

```
Procedure model_synth(S, P)
begin
    for all f ∈ topsort_sub_formulæ(S) do
        label_state_graph(P, f);
end
```

Figure 4.5: *The* ***model_synth*** *procedure.*

Supervisor Map Consolidation Phase

After the labeling phase is over, the map wff-state-restriction contains all the information needed to reconstruct the supervisor map for the CTL specification $\mathcal{S}$. However, the information is spread over the whole state-space and it needs to be consolidated.

As an example consider the simple specification $\mathcal{S} = \{p\}$. In this case, either we have $\Sigma^*, s_0(\mathcal{P}) \ \Vert\!\models \mathcal{S}$ or $\Phi, s_0(\mathcal{P}) \ \Vert\!\models \mathcal{S}$. However, the supervisor map assignment for a state $s \neq s_0$ may be completely blocked. Since what matters is only the language satisfying $\mathcal{S}$ at the initial state, somehow the supervisor map for s must be "fixed". This is what the *consolidation phase* does by a simple additional DFS traversal of the state-space.

4.1.2 Tradeoffs in the Treatment of Disjunctive Forms

In Chapter 3, it was shown that disjunctions have to be treated with care in constructing a supervisor map. Recall that the main problem here was caused by the inherent choice involved.

The MRSS algorithm restricts the logic used by imposing restrictions on the form of disjunctive formulæ. The restrictions simply impose a requirement on one of the disjuncts of

$$f \equiv f_1 \vee f_2.$$

During the implementation of Control-D, two kinds of restrictions were con-

sidered.

1. Distinguish between *local* and *path* disjuncts and restrict one of them to a local formula, where a local formula is either
   - A proposition in AP.
   - A conjunction, disjunction or negation of local formulæ.
2. Restrict as above one of the disjuncts to be a state formula, but treat the **EX** and **AX** operators in a special way.

The current implementation of Control-D implements the first limitation. Since state formulæ comprise all the propositional forms with no path operators, their controlled semantics is both

(a) Completely *local*.

(b) Either "all-or-nothing", in the sense that the supervisor map assignment, either enables or disables all of $\Sigma[s]$.

Therefore, the controlled semantics for disjunctions is preserved.

Note however that formulæ of the form

$$\mathbf{QX}^i(f), \quad 1 \leq i \leq n,$$

where each of the **QX** is either **AX** or **EX**, and f is a local formula (e.g. $\mathbf{EX}(\mathbf{AX}(\mathbf{AX}(p \wedge q)))$, with $p, q \in AP$), is really a local formula itself. This extension (to disjunction manipulation) has not been incorporated yet in

Control-D, and no analysis of its impact on the complexity of the algorithm has been carried out.

The restricted form of CTL used is more expressive than CTL$^-$

The restrictions imposed on CTL in order to achieve a linear time complexity for the MRSS algorithm are not the only ones possible. In [58], the authors propose the logic CTL$^-$ in order to tackle the problems of hierarchical verification of VLSI circuits. The logic CTL$^-$ is a syntactic restriction of CTL defined in the usual way.

(S1) A proposition $p \in AP$ is a CTL$^-$ formula.

(S2) If f_1 and f_2 are propositional CTL$^-$ formulæ, so are $\neg f1, f_2, \wedge f_2$, and $f_1 \vee f_2$.

(P1) If f_1 is a propositional formula and f_2 is a CTL$^-$ formula so are $\mathbf{A}[f_1 \,\mathbf{U}\, f_2]$, $\mathbf{E}[f_1 \,\mathbf{U}\, f_2]$, $\mathbf{AF}(f_2)$, and $\mathbf{EF}(f_2)$.

CTL$^-$ is therefore more restrictive than the logic manipulated by the MRSS algorithm. As already noted, the presence of **EX** and **AX** operators make the logic used by MRSS not usable for hierarchical synthesis and verification. However, since all the restrictions introduced are syntactic – both for the MRSS algorithm and for CTL$^-$ – it is still possible to modify the tool in order to recognize such special cases and to apply appropriate optimizations.

4.1.3 Comparison with Standard Supervisor Synthesis Algorithm

Control-D is not the first implementation of a discrete controller synthesizer for CDES. First of all, the effective computability of the synthesis procedure was established in [64]. The effective computability of the supervisor map in the CDES framework depends on the effective algorithmic constructibility of the supremal controllable sublanguage $K^{\uparrow}$. In the work mentioned, the authors first give a general argument for the constructibility of $K^{\uparrow}$ for regular languages, and then show an efficient algorithm for its construction (which will be referred to as *SCLS* algorithm – where SCLS stands for "Supremal Controllable Synthesis Algorithm") when the languages involved are prefix-closed[2].

The SCLS Algorithm

The SCLS algorithm works on prefix-closed regular languages. The core of the algorithm is a simple convergent doubly nested iteration that examines all the states of a FSM (which FSM will become clear in a while) and removes those not satisfying a given condition, called the *active event condition.* Its definition requires introduction of some terminology.

[2] Of course, if the specification language K is *controllable*, then no construction of its supremal controllable sublanguage is required.

Fixpoint Characterization of $K^{\uparrow}$. The SCLS algorithm computes the fixpoint of the operator

$$\Omega_{(K,\mathcal{P},\Sigma_u)}(L) = K \cap \{T \mid (T \subset \Sigma^*) \wedge (T = \bar{T}) \wedge (T\Sigma_u \cap \mathcal{P} \subset \bar{L}\},$$

where K is the "specification" language [64]. The fixpoint is the supremal controllable sublanguage $K^{\uparrow}$.

Definition of *Refinement* of FSM's. Given two FSM's $A = (\Sigma, S_A, \delta_A, a_0)$ and $B = (\Sigma, S_B, \delta_B, b_0)$, then B refines A if

$$\forall s, t \in \overline{\Lambda[B]}.(\delta_B(s, b_0) = \delta_B(t, b_0) \Rightarrow \delta_A(s, a_0) = \delta_A(t, a_0)).$$

Conversely B $\neg$refines A if

$$\exists s, t \in \overline{\Lambda[B]}.(\delta_B(s, b_0) = \delta_B(t, b_0) \wedge \delta_A(s, a_0) \neq \delta_A(t, a_0)).$$

If B refines A, then it can be shown that there exists a unique map $h : S_B \rightarrow S_A$ which satisfies

$$h(\delta_B(s, b_0)) = \delta_A(a, a_0), \quad \text{for } s \in \overline{\Lambda[B]}.$$

SCLS Applicability Conditions. Consider the sequence of languages

$$\begin{aligned} K_0 &\triangleq K, \\ K_{j+1} &= \Omega(K_j), \quad \text{for } j \geq 0. \end{aligned}$$

It can be proven that given an FSM A, such that $\Lambda[A] = \Lambda[\mathcal{P}]$ and a FSM $\mathcal{C}_j$, such that $\Lambda[\mathcal{C}_j] = K_j$ with the condition that $\mathcal{C}_j$ refines A, then

1. $w \in \Omega(K_j)$ if and only if $w \in K_j$ for each prefix u of w, $x = \delta_{\mathcal{C}_j}(u, c_{j,0})$ implies

   $$\Sigma[h(x)] \cup \Sigma_u \subset \Sigma_x.$$

   This last condition is the *active event condition* mentioned earlier.

2. $\Lambda[\mathcal{C}_{j+1}] = \Omega(K_j) = K_{j+1}$.

Given these results, the SCLS algorithm can be applied to build the supremal controllable sublanguage $K^{\uparrow}$ when the following conditions hold.

1. $K \subseteq \mathcal{P}$, and

2. it is possible to construct a $\mathcal{C}_0$ that refines the FSM for the language $\mathcal{P}$ and such that $\Lambda[\mathcal{C}_0] = K$.

It turns out that Condition 2 is satisfied by choosing the intersections of $\mathcal{P}$ and $M[K]$.

MRSS and SCLS Compared

The MRSS algorithm is slightly better than the synthesis algorithm for prefix closed languages. The reasons behind this statement derive from a series of observations on the SCLS and the MRSS algorithms. It will be argued that

- The usage of a temporal logic for the specification of the properties of the system being modeled results in a more usable system.

- The overall complexity of the MRSS algorithm is better than the SCLS one.

Use of a temporal logic for specifications. The SCLS algorithm implies a specification K given as either a *regular expression* ($\mathcal{R}(K)$) or as an FSM ($M[K]$). The version of CTL used by the MRSS algorithm has many advantages coming from its branching time nature and from its conciseness.

In all fairness, these advantages may be offset by the usage of *libraries of properties* in the style of those advocated for linear time logics and for automata based verification [46, 50].

Better overall complexity. The SCLS algorithm works by removing states that do not satisfy the active event condition from the representation of the recognizer FSM of the supervisor.

As already noted, in order for this condition to be applicable, the supremal controllable sublanguage construction procedure must be provided with an "initial" FSM $\mathcal{C}_0$ which recognizes the language K *and* refines $\mathcal{P}$, and the condition $K \subseteq \Lambda[\mathcal{P}]$ must hold.

First of all, in the most general case, a test for language containment must be performed. This is usually equivalent to test either $K^c \cap \Lambda[\mathcal{P}] = \Phi$ or $K \cap \Lambda[\mathcal{P}]^c = \Phi$. This test requires $O(|K| \cdot |\mathcal{P}|)$ space and time. However, the containment test can be avoided by using some cleverness in the specification of the of K (as it is the case of many of the examples appearing in the

literature).

Apart from the language containment test, because of these requirements, the SCLS algorithm behaves in very different ways according to which of the following conditions holds.

1. *$M[K]$ does not refine $\mathcal{P}$.*

   In this case, a machine $\mathcal{C}_0$ that *does* refine $\mathcal{P}$ can be built by considering $\mathcal{S} \cap \mathcal{P}$. The space requirement becomes $O(|\mathcal{S}| \cdot |\mathcal{P}|)$ as well as the running time.

2. *$M[K]$ does refine $\mathcal{P}$.*

   The complexity of this case is bounded by the test for the relation "refine", which takes $O(|\mathcal{P}| \cdot \lg(|\mathcal{P}|))$.[3]

3. *$M[K]$ does refine $\mathcal{P}$ and $K \subseteq \Lambda[\mathcal{P}]$ by construction.*

   In the examples appearing in the literature this condition seems to be always true. In this case the SCLS algorithm requires only $O(M[K])$ space and time. However, the examples (e.g. the *cat and mouse*) correspond to only requiring the supervisor to enforce simple *invariant*

[3] By using the standard minimization procedure (or, better, the PARTITION procedure of Aho, Hopcroft, Ullman [2]) $B \neg$refines A can be tested in the following way: run the procedure on both machines (if they are already minimized, eventually the partitions as the same sets of states will result) splitting according to A. If whenever a partition for A is split, but B is not, then B does not refine A. This algorithm takes as long as the best PARTITION algorithm applied to "minimization" of FSM's. Ergo, circa $O(|A| \lg(|A|))$.

properties which are expressed as $\mathbf{AG}(p)$ or $\mathbf{AG}(\neg p)$ formulæ, with $p \in AP$. The MRSS algorithm works well in these cases.

The MRSS algorithm fares well against the SCLS algorithm in all three cases. In case 1, running time and space requirements are the same, though it can be argued that $|\mathcal{S}_{MRSS}| \leq |\mathcal{S}_{SCLS}|$ (where $\mathcal{S}_{MRSS}$ is a set of CTL formulæ, and $\mathcal{S}_{SCLS} = M[K]$) because of the CTL formulation.

In case 2, MRSS fares well in terms of time as long as $|\mathcal{S}_{MRSS} \leq \lg(|\mathcal{P}|)|$.

In case 3 the SCLS algorithm works better than the MRSS algorithm. Yet, the difference is minimal and the SCLS algorithm requires "non-automatable" cleverness to actually produce the initial $\mathcal{C}_0$[4].

Because of the above arguments and because of the greater expressiveness of the specification language employed (i.e. CTL), MRSS turns out to be superior to the SCLS algorithm. The "expressiveness" argument cannot of course be used if the more expressive ω-automata are used as a basis of CDES, as it is done in [65].

4.1.4 Open Problem: *Symbolic* Representation

It has been noted that the techniques of symbolic model checking based on BDD's have significantly improved the practical applicability of verification techniques to many problems.

[4]The *cat and mouse* example works fine by simply "removing" undesirable states from the plant $\mathcal{P}$.

It is clear that an application of these techniques would be beneficial to the MRSS algorithm and to the Control-D tool. To the best of the writer's knowledge, the only application of BDD techniques to the synthesis of supervisors controllers for CDES is in [13].

So far, no effort has been made to formulate the CTL supervisor synthesis problem with BDD's. A BDD library based on the implementation described in [17] is included in the Control-D tool, but it cannot be used yet for the supervisor synthesis procedure. This therefore remains an open problem.

4.2 Verification and Synthesis

The Control-D tool enjoys one more feature deriving from its use of CTL as a specification language. Since the MRSS algorithm is closely patterned after the model checking one, it is very easy to interpret its results as a "verification of the feasibility" of the specification $\mathcal{S}$. This claim is supported by the theoretical result of Section 3.2.5.

4.3 The Environment

The Control-D tool supports a design methodology which embodies the concept of "iterative design and debugging" phases. Therefore it provides the user with three basic tools.

- A "language" to describe FSM and CTL specifications.

- A graphical editor for describing and combining FSM's.
- A small "application programming interface" (API) to build graphical simulations of the system under consideration.

The user of Control-D can describe a model of a system by breaking it down into its subcomponents and by writing a set of CTL formulæ to be used for the synthesis. The synthesis algorithm is triggered from the graphical environment and it produces a supervisor map or a counterexample that shows why a certain formula would yield a null map.

Figure 4.6 shows a snapshot of the Control-D environment. Chapter 5 contains a description of the use of (an earlier version of) Control-D for the synthesis and simulation of the discrete part of a controller for a *walking machine*. Chapter 6 shows how the Control-D system has been used to replicate a part of the control system of a manufacturing line developed at Rutgers University. The next section contains a brief description of the Control-D tool components and user interface.

4.3.1 Control-D Components and User Interface

The Control-D tool is built on top of Common Lisp [72] and uses a Motif™ interface[5]. The use of Common Lisp allowed for a very rapid prototyping of the tool and provided a flexible environment for the implementation of

[5]The specific Common Lisp implementation used is the one created at CMU by the team headed by S. Fahlman. The system is known as CMUCL.

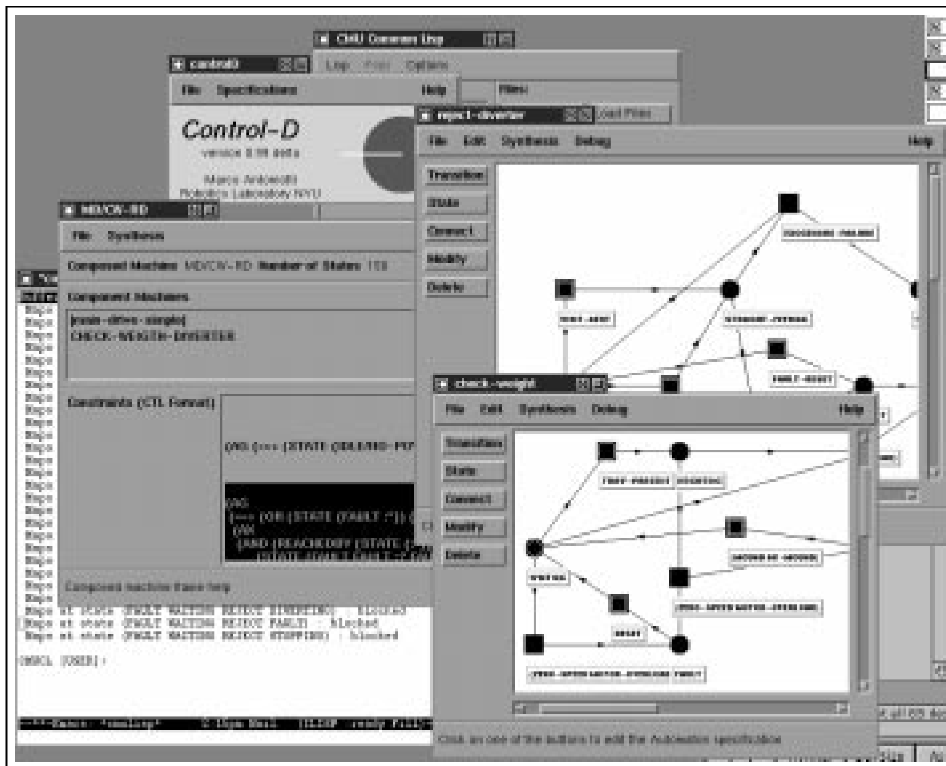

Figure 4.6: *A snapshot of a* Control-D *session. Two edit windows, a* CTL *specification window, and the main panel are visible. An Emacs frame containing the Common Lisp listener is in the background. The synthesis and verification algorithms can be controlled in their execution from the menus of the tool.*

the CTL manipulation module. Moreover, since the result of the synthesis problem is essentially a boolean table, nothing prevents the construction of back ends that produced either C/C++, Ada, Assembler, or any other desired language.

Control-D Languages

The specifications of the plant $\mathcal{P}$ and of the properties $\mathcal{S}$ are given in a Lisp-style syntax.

As an example the FSM describing the change of seasons is given as

```
(define-state-machine seasons
   :states (spring summer fall winter)
   :start  spring
   :alphabet (jun21 sep23 dec21 mar21)
   :uncontrollable (jun21 sep23 dec21 mar21)
   :delta ((spring jun21 summer)
           (summer sep23 fall)
           (fall dec21 winter)
           (winter mar21 spring))
   )
```

This is a very simple self explanatory example. All the components of a CDES are assumed present in the specification. In particular, note the `:uncontrollable` slot in the definition, which states that the mere mortals

do not have any control over the passing of the seasons.

It must be conceded that systems like SMV or COSPAN (see McMillan and Kurshan, op. cit.) can describe FSM's in a more concise and elegant way. However, the addition of a layer of "syntactic sugar" would not be difficult.

CTL is translated directly into Lisp forms following what has become standard practice for the manipulation of logic languages. The simple system `seasons` system is already a good testbed for a set of CTL specifications. E.g. the following properties hold.

- `(AG (implies winter (not (AX summer))))`:

  summer never follows winter (a "safety" property).

- `(AG (EF spring))`:

  spring always comes back (a "liveness" property) eventually.

Control-D Graphical Interface

The Control-D system can be started in a variety of ways, depending on the configuration of the underlying Common Lisp system. Once started, the Control-D's main window appears on the screen. This is just a repository for the "system menus". The main Specifications menu contains the following items.

- The Loaded Specifications simply lists the systems currently available for manipulation.

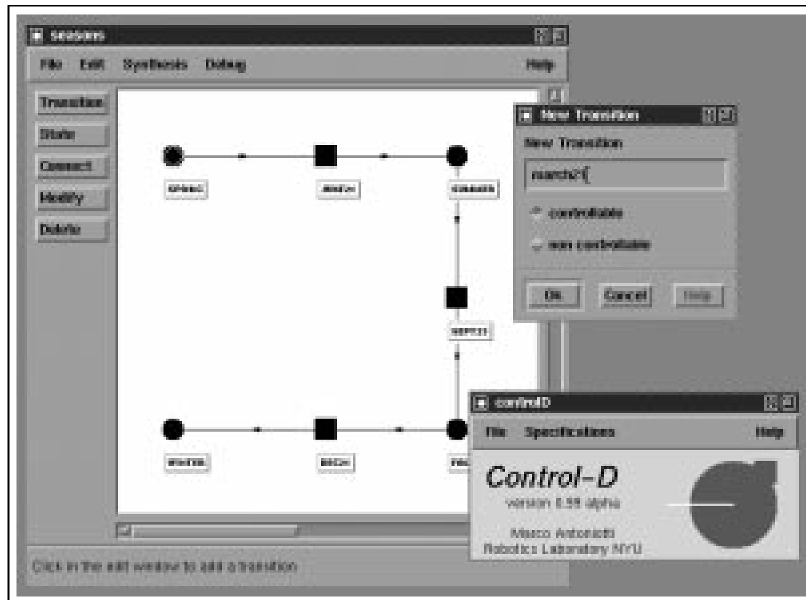

Figure 4.7: Control-D edit window with the `seasons` system.

- The Compose Specifications takes two FSM's and yields their composition.
- The Controller Synthesis calls the MRSS algorithm.
- Finally, the Dump Controller calls a "back end" to produce a "compiled version" of the synthesized controller.

Control-D manages two kinds of *working* windows: one for "simple" and the other for "composed" FSM's (i.e. FSM's obtained as the product of other FSM's). Figure 4.7 shows an edit window where the simple `seasons` system is being constructed.

A window for a "composed" specification does not contain a graphical representation for the underlying FSM since it would be unwieldy most of the time. These window contain a list of CTL formulæ that will serve as input to the MRSS algorithm.

Control-D Simulation Interface

In order to appreciate the results of the MRSS algorithm it may turn out to be useful to build simulations – mostly graphical – of the systems for which a controller is being built. These simulations may offer other insights into the actual behavior of a particular system. Control-D provides a very simple interface for this purpose.

The interface consists of one main macro `with-supervised-transition`, which realize the connection between the supervisor map obtained from the MRSS algorithm (presumably held in a variable) and a set of component FSM's to be controlled. In order to produce better debugging output, the writer of the simulation code can also control the actual timing of the transition firing in one of component FSM's by using the function `run-machine` (this function simply changes the state of a FSM according to the transition chosen).

The role of the Control-D system is also that of a sophisticated debugger. The capability of writing graphical simulations for specific systems proved to be a valuable asset. Chapter 5 contains a description of how, in the case of

the construction of a controller for a Walking Machine, this capability turned out to be crucial.

Chapter 5

Building a Discrete/Hybrid Controller for a Walking Machine

A *walking machine* is a robotic device capable of moving around over a rough terrain by emulating the limb (i.e. legs) movements of various multi-pedal animals. There are several arguments for the utility of such devices, and the goal of reproducing typical animal movements goes back in time. This chapter describes the problem of specifying in CTL and then automatically building a controller for a walking machine with the techniques developed in Chapter 3 and Chapter 4.

The problem proved to be very rich because of the variety of considerations that had to be taken into account in order to produce a reasonable behavior of the (simulated) walking machine.

This chapter is organized as follows. Section 5.1 contains a brief history

of walking machine devices, while Section 5.2 contains a description of the model used and of the typical problems to be treated. Finally, Section 5.3 contains a description of the construction of the discrete controller for the walking machine with Control-D.

5.1 A Brief History of Walking Machines

It is safe to say that the first attempts at building walking machines go back a few centuries. Leonardo da Vinci must have thought about the problem. However, the focus of this chapter is on the construction of a software controller for a walking machine, therefore the historical recount will be limited – and even incomplete – by any standard.

One of the first references to the problem of building a walking machine goes back to Chebyshev [62] in 1850. The application of control theory to this problem and its electronic implementation are obviously more recent. Some milestones are shown below:

1968 Frank & McGhee use simple electronic logic to drive, the Phony Pony [54],

1983 Sutherland's Hexapod

1986 Raibert's Hoppers (op. cit.),

1987 Donner's control of the Hexapod [34],

1994 Dante Hexapod for volcanic exploration (CMU Dante descends into an infernal environment).

5.2 Problems for Walking Machines

Walking machines provide several interesting problems that must be solved in order to achieve a reasonable locomotion movement. Three broad categories of problems can be individuated.

- *Gaits*: What kind of gaits can be achieved by the walking machine?
- *Stability*: How does one ensure that a walking machine does not fall?
- *Real-time control*: How can one devise a software capable of controlling the mechanics of the walking machine?

The problems in each category cannot be totally solved in isolation. Further complication is caused by the fact that some of these problems have a discrete nature, while others have a continuous one. A walking machine is therefore to be considered a *hybrid system* according to the definition employed recently by several researchers [38, 16].

Of course, walking machines must also be classified according to their morphology, where the *number* of legs and their disposition are the discriminating parameters. Different walking machine morphology leads to different problems to be solved. E.g. one, two, and three-legged machines can move

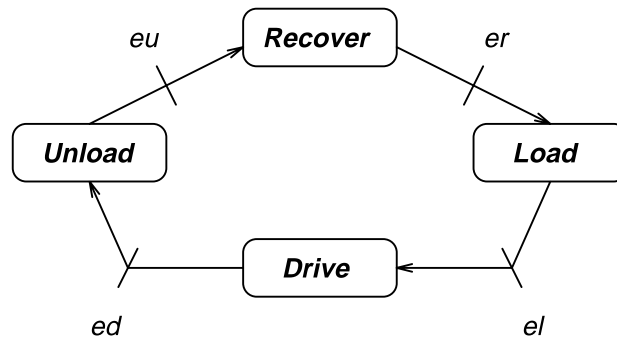

Figure 5.1: *State diagram representing the movement of a single leg.*

only in inherently "unstable" gaits, while four and more-legged machines can move in stable gaits.

5.2.1 Leg Behavior Models and Gaits

There seems to be some agreement among the researchers who study the problem of walking, that each leg – taken in isolation – goes through a sequence of phases which can be easily represented as a two or four state machines. Figure 5.1 shows a four state machine similar to the one in [34]. The four states respectively describe the following situations.

1. Drive: The "foot" of the leg is on the ground and exerting a force that moves the body.

2. Recover: The foot is swung "forward" in a "flying" motion.

3. Load: At the end of the Recover phase, the machine shifts (part of) its weight on the leg.

4. **Unload**: Symmetrically, at the end of the **Drive** phase the machine lifts (part of) its weight from the leg.

The distinction between these phases (or states) is justified by the fact that the dynamic equations describing the position and the torques of a leg are formulated differently during each of the phases. The variation is mostly due to changes in the boundary conditions of the system of dynamic equations (cfr. [29]).

The FSM model of the behavior of a leg constitutes a guideline for devising a high level discrete controller for a leg. In order to model walking machines with different morphologies the interleaving of several leg FSM models must be considered.

These "interleaved models" are hoped to eventually produce recognizable *gaits* for the walking machine, e.g. walking, trotting, galloping. A reasonable depiction of gaits is given by Gantt's chart. Figure 5.2 shows the so-called *tripod gait* for an insect-like hexapod (cfr. Sutherland's hexapod in [34]).

Hints from Biometrical Data

Biometrical data and models obtained from observing various animals (including humans) provide starting points for the construction of engineering models of robotic walking machine controllers (see [73, 62, 34] for references).

Of particular interest is information regarding:

- The decomposition of the models used for modeling walking,

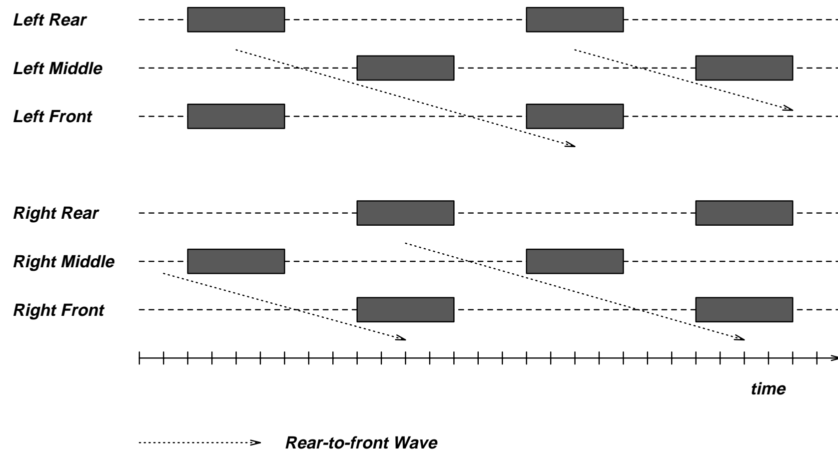

Figure 5.2: *Hexapod tripod gait. At each moment there are two legs on one side and one on the opposite side in a* Drive *state. This fact guarantees the* stability *of the gait.*

- Evidence about how the movement of legs is coordinated, and
- How gaits shift in walking machines with a given mass.

Walking Models Decompositions. For non bipedal animal-like walking machines[1], biometrical data show that the left "train" of legs operates quite independently from the right train[2]. This suggests a way to build a model for a four or six legged walking machine by treating separately the left and right trains of legs.

[1]Most walking animals – from small sized insects to large mammals – show a "left/right" symmetry in their body morphology.

[2]See the references on *the spinal cat* in [34]. Not for cat lovers.

Rear-to-Front Wave Coordination. Another piece of evidence gathered from biometrical data is that non bipedal animals move their trains of legs in a so-called *rear-to-front wave*. During this movement, an animal (or a walking machine) with n legs on either the left or right train, numbered from front to rear, starts recovering leg i *before* leg $i-1$ (for $i = 1, 2, \ldots n$). This sort of movement is particularly evident in millipedes, where more than one wave can be observed. Though less evident, the rear-to-front wave is also observable in quadrupedal animals, like cats. Figure 5.2 shows the rear-to-front waves for the six legged tripod gait.

Gait Shifting. Many animals can move with different gaits. The difference among these gaits is caused by the mechanics of the animal's body. It has been noted that for insects it does not make sense to speak of different gaits. Insects exhibit an almost continuous range of gaits where the only variable is the frequency of the leg movements.

Different gaits appear only in animals of greater mass than insects. The reason for these differences is accounted for by the way energy is stored in the skeleton and muscles of the animal. Speed "quantizes" the number of gaits that minimize the energy expenditure.

Shifting between gaits poses interesting controller design problems, since it is necessary to make provisions for an overhaul of one model used by the software with another one. I.e. this requires some sort of a *meta controller* which gets executed in place of to the "current" one, subject to change in

ambient conditions[3] (e.g. terrain, friction, speed, gravity etc.).

5.2.2 Stability of Walking Machines

The problem of stability of a walking machine – i.e. how to prevent the machine from collapsing on the ground – has been widely studied. E.g. the cited work by Raibert [62] studies the "active/dynamic" balance of *hopping machines* with one, two or four legs.

The studies of stability involve exact forms of feedback control scheme in the framework of control theory. The models employed can become rather complicated even for simple robots.

5.2.3 Synchronization and Real-time Control

The implementation of the feedback schemes requires some form of sampling loop within a program that realizes the model, e.g. the stability model of the walking machine. The FSM representation of the phases of each leg (cfr. Figure 5.1) is built in the software controlling the system.

This embedding of the FSM model in the control software is usually done in an "ad hoc" fashion. It would seem that decoupling the discrete and the continuous models would be beneficial from the "software engineering" point of view. As already noted, McCarragher and Asada propose a similar

[3]This approach may be similar in spirit to the one proposed by Brook's with his *subsumption architecture* for robotics [19, 43].

course of action in [53]. However this separation of the discrete level from the continuous one is rather difficult to achieve. The case of walking machines presents one example of such difficulties.

The control of a walking machine would in fact benefit from the separation of the issues regarding the actual generation of gaits – a problem which involves the generation of proper sequences of phases in the FSM representing the interleaving of several legs – from the issues regarding the generation of the proper torques in order to position a limb in a desired location. Yet, this separation is not easy to obtain, since the state transitions in the FSM model obviously *depend* on the actual position of the limbs and vice-versa.

Ideally, the resulting architecture of a continuous/discrete controller for a walking machine (or for a more general hybrid system) would look like the one proposed by McCarragher and Asada. Figure 5.3 shows an example of such hybrid controller architecture.

5.3 Building Walking Machines Controllers with Control-D

The robotics laboratory of Courant Institute of Mathematical Science has developed an inexpensive yet powerful technology of *mini actuators* [74]. The design and construction of direct drive walking machines out of mini-actuators have provided inspiration for the creation of the Control-D system.

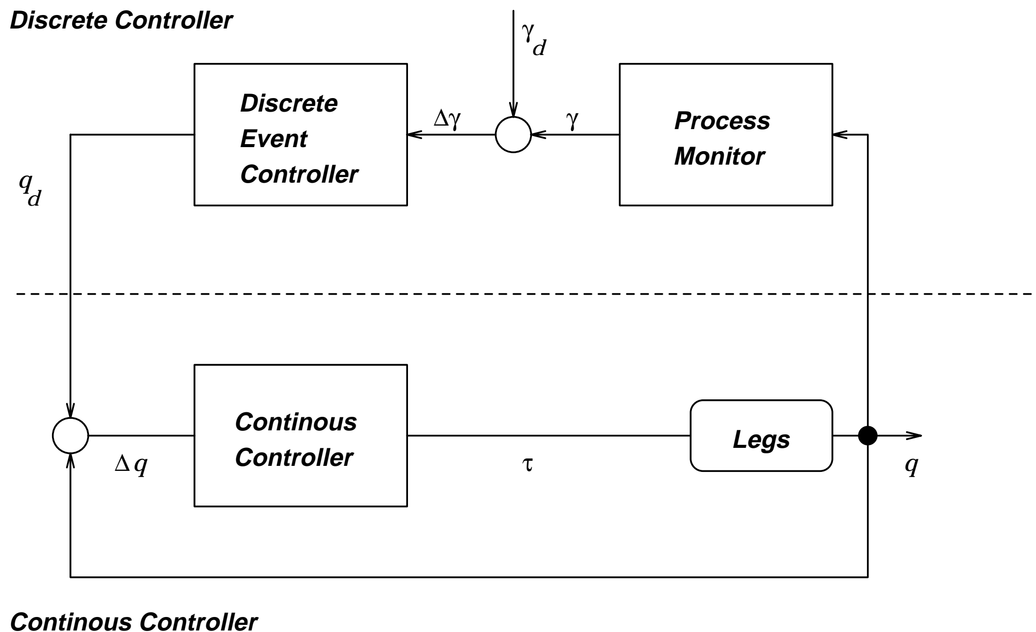

Figure 5.3: *Hybrid Controller Architecture for Walking Machine.*

Figure 5.4 shows a prototype of the leg joints.

A four-legged walking machine was studied as an example of the use of the Control-D system [11, 12]. The model of the device assumed that the controller would be divided into a continuous and a discrete part as the one depicted in Figure 5.3. The continuous controller for each leg of the walking machine was further assumed to be solved separately (i.e. a controller capable of controlling at least the position of the leg was assumed to be available).

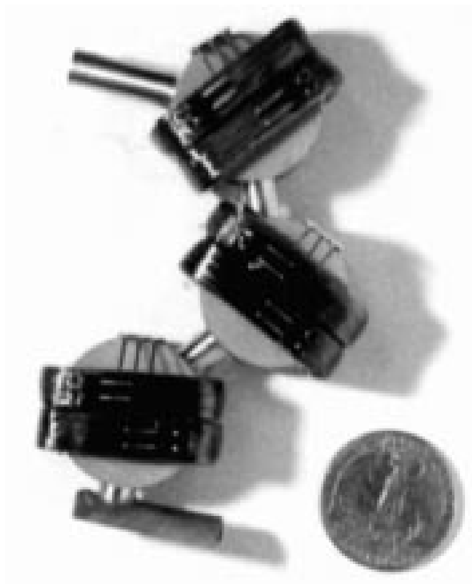

Figure 5.4: *Prototypes of the mini actuator links for the legs of the Walking Machine built by R. Wallace and F. Hansen. (Photo by G. Kondogianis).*

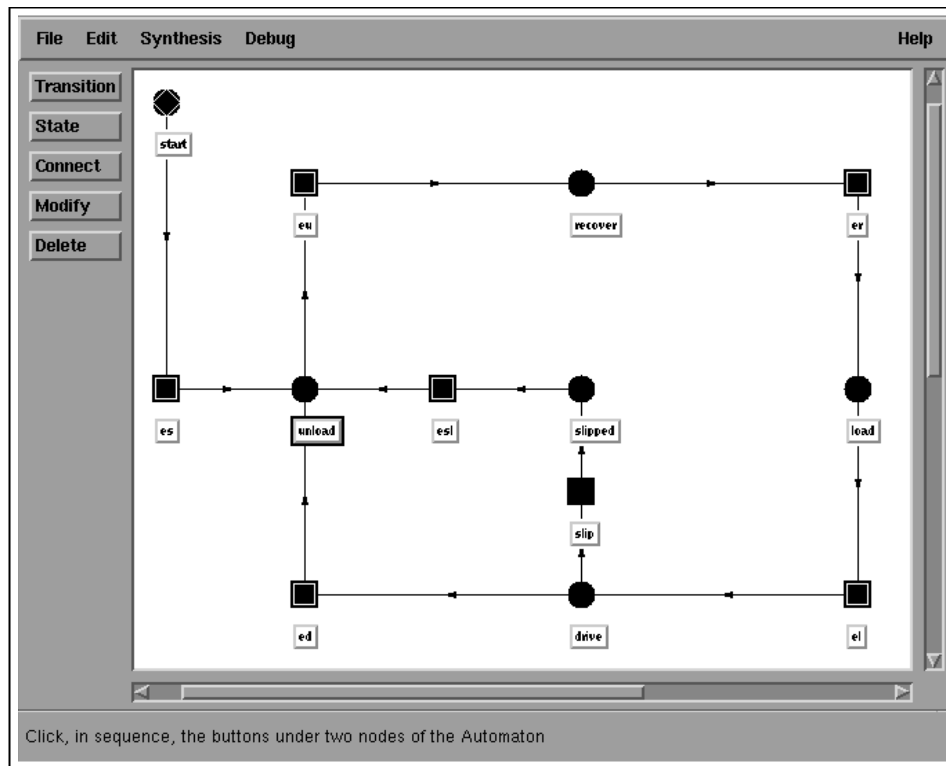

Figure 5.5: *The FSM model of a Leg with uncontrollable event slip.*

5.3.1 Discrete Controller for the Walking Machine

The standard FSM model of a leg behavior does not make any distinction between *controllable* and *uncontrollable* events. In order to make the walking machine FSM model more robust via the CDES techniques, the transitions of the model were thought to be controllable, while an extra, uncontrollable, transition was inserted. The new transition called slip represents the possibility of a leg "slipping" while pushing on the ground. Figure 5.5 shows the complete leg model after it was edited with Control-D.

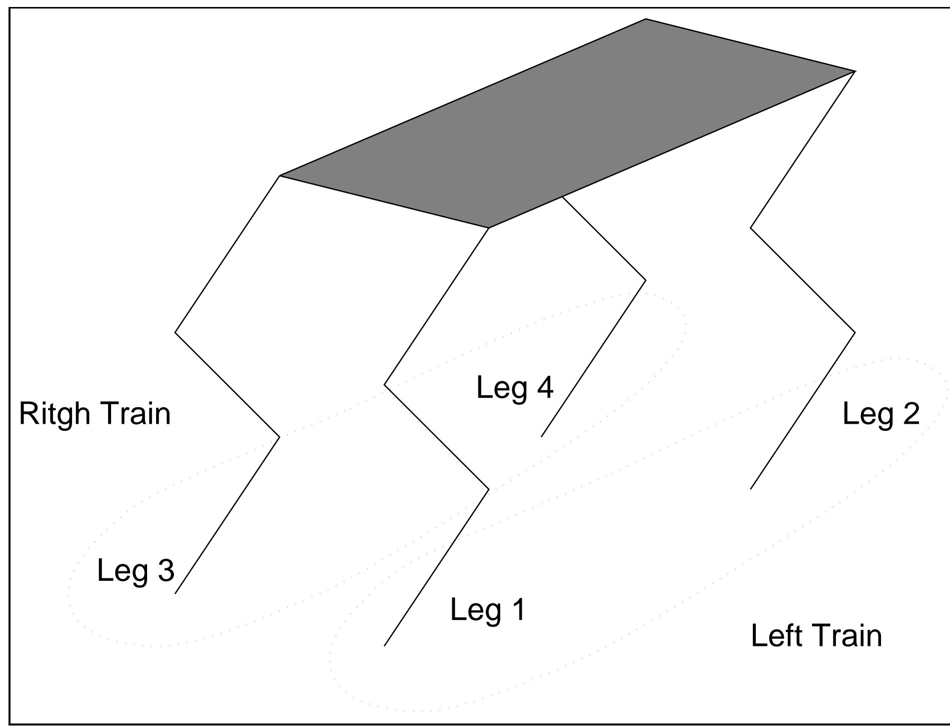

Figure 5.6: *Schematic representation of the walking machine.*

The complete discrete model for the walking machine is broken down into three parts: one for each trains of legs (left and right) and a third one modeling the alternation between the leg pairs. A schematic of the walking machine used for the modeling exercise is shown in Figure 5.6.

By following the standard practice of CDES modeling, the $\mathcal{P}$ specification for a train of legs is obtained via a composition of two instances (say 1 and 2) of the FSM, one for each leg[4]. The specification $\mathcal{S}$ must be thought out

[4]This "copying operation" is still quite cumbersome in Control-D. COSPAN and SMV provide better linguistic constructs to obtain this effect.

carefully in order to achieve a desired gait. A single train of legs was required to have an alternation of Drive and Recover phases between the two legs. This approach resulted in the following simple specification:

$$\begin{aligned} \mathbf{AG}(\ & \neg\mathtt{state}([\mathsf{Drive}_1, \mathsf{Drive}_2]) \\ & \wedge\ \neg\mathtt{state}([\mathsf{Recover}_1, \mathsf{Recover}_2]) \\ & \wedge\ \neg\mathtt{state}([\mathsf{Slipping}_1, \mathsf{Slipping}_2])), \end{aligned} \tag{5.1}$$

where the syntax $\mathtt{state}([\mathit{state},\mathit{state}])$ indicates the composed state of the train specification.

The Formula (5.1) is simply the statement of a *state avoidance problem.* The system must avoid any state where both legs of a train are both recovering, or driving, or (worse) slipping.

The *rear-to-front wave* can be represented by the following CTL formula.

$$\begin{aligned} & \mathbf{AG}(\mathtt{state}([\mathsf{Drive}_1, \mathsf{Recover}_2]) \\ & \quad \Rightarrow\ \neg\mathbf{EX}.(\mathtt{state}([\mathsf{Unload}_1, \mathsf{Recover}_2]))). \end{aligned}$$

The meaning of this formula is that whenever the rear leg (number 2) is *recovering*, the front leg (number 1) cannot start unloading.

5.3.2 Continuous Control Constraints

The "desired behavior" of the Walking Machine is obviously not completely specified by the constraints imposed at the discrete level. The transitions between states are governed by measurements taken from sensors. Only

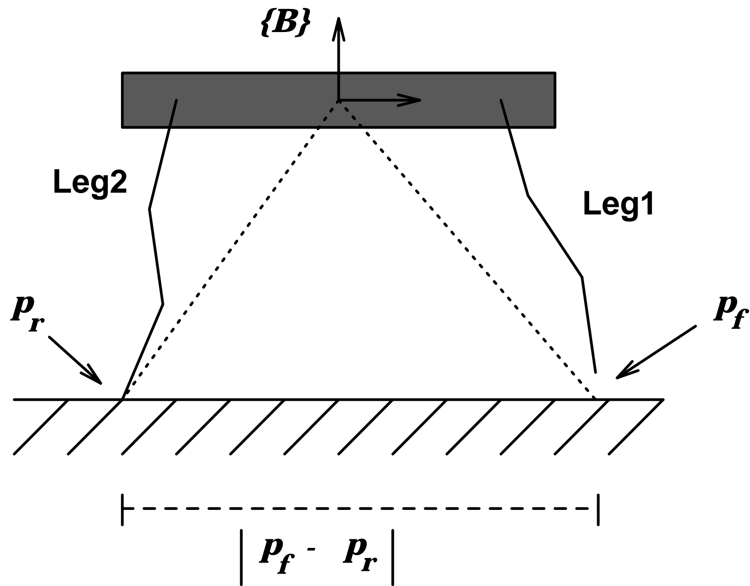

Figure 5.7: *Simplified Geometric Model of the Walking Machine. $\{B\}$ is a coordinate frame set in the body. All measurements are taken with respect to it.*

position information was used in order to allow the transition from one state to the other of the discrete control. This is sufficient to get nice simulations and already poses interesting problems for the control synthesis procedure.

The geometric model used for the Walking Machine is depicted in figure 5.7. A discrete supervisor that ensures the satisfiability of the specification $\mathcal{S}$ is readily obtained. Unfortunately, it turned out that such supervisor also allows for

$$\mathbf{EX}(\texttt{state}([\mathsf{Load}_1, \mathsf{Load}_2]))$$

to be true for one train of legs[5]. In this state, the supervisor has to choose which transition to make next: i.e. to either state $[\mathsf{Load}_1, \mathsf{Drive}_2]$ or $[\mathsf{Drive}_1, \mathsf{Load}_2]$. Since both transitions are controllable and not forbidden by the supervisor, the system might end up "taking a step longer than the leg" by cycling one too many times through the $[\mathsf{Load}_1, \mathsf{Load}_2]$ state [6]. In Petri Net terminology, this is a *conflict* and it really represents a situation where "extra information" is needed (or assumed) by the system.

In order to solve this problem, the control algorithm must identify these "conflict states" in order to reduce the actual behavior of the system to a "geometrically acceptable" one. This operation is analogous to the one described in [53].

Continuous Conditions and Choice Points

For a train of two legs, the transition er1 for the front leg (leg 1) in state $[\mathsf{Recover}_1, \mathsf{Drive}_2]$ causes the difference in the position of the feet

$$\Delta(p_{feet}) = |p_f - p_r|$$

[5]With respect to the "start" state. Actually it can be proved

$$\mathbf{EG}(\mathtt{state}([\mathsf{Load}_1, \mathsf{Load}_2]) \Rightarrow \mathbf{EX}(\mathbf{EF}(\mathtt{state}([\mathsf{Load}_1, \mathsf{Load}_2])))).$$

[6]This argument applies also when two legs in alternation – left and right – and *virtual legs* (cfr. [62]).

to change in the following way

$$\Delta(p_{feet})[\mathsf{Load}_1, \mathsf{Drive}_2] =$$

$$\Delta(p_{feet})[\mathsf{Recover}_1, \mathsf{Drive}_2] + \frac{1}{2}\mathtt{step},$$

if the rear leg is assumed to have moved a "very small" distance,

$$\Delta(p_{feet})[\mathsf{Load}_1, \mathsf{Drive}_2] =$$

$$\Delta(p_{feet})[\mathsf{Recover}_1, \mathsf{Drive}_2] + 2\,\mathtt{step}$$

if both legs have moved (almost) the full `step` distance.

This argument can be repeated for all the other states. This "interval" computation for the transitions can be reconstructed from the description of the state in which it is taking effect, hence a simple graph traversal could mark the states where a given constraint *could* (but not necessarily would) be violated. In this specific case the simple constraint we would like to maintain is

$$\Delta(p_{feet}) \leq \ell,$$

where ℓ is derived from the geometry of the Walking Machine.

The graph traversal simply maintains for each node traversed a possible maximum and minimum value for $\Delta(p_{feet})$ while following only the controllable transitions enabled by the Supervisor. Whenever there are two or more such transitions outgoing from a state s and one of the reachable states (or the state itself) possibly violates the constraint, then s is marked as a "choice point".

Eventually, it will be possible to equip the runtime component of the system with appropriate tests to avoid the controllable transitions which in specific occasions (usually after a few tours around a cycle in the state space) would violate the constraint.

Chapter 6

Ensuring Failure Behavior for the CRAMTD Manufacturing Line

Manufacturing system are obvious targets for the the application of the techniques developed for CDES. Examples of this trend can be found in [13] and [75] (though this last reference contains a variation of the approach).

This chapter contains a description of an application of the Control-D system to the specification of a controller for a *tray packing line* built for the *Combat Ration Advanced Manufacturing Technology Demonstration* (CRAMTD) of Rutgers University.

The chapter is organized as follows. Section 6.1 describes the tray packing line of the CRAMTD project. Section 6.2.2 discusses the application of the Control-D tool to CRAMTD. Some concluding remarks are finally made about the experience gathered from the project.

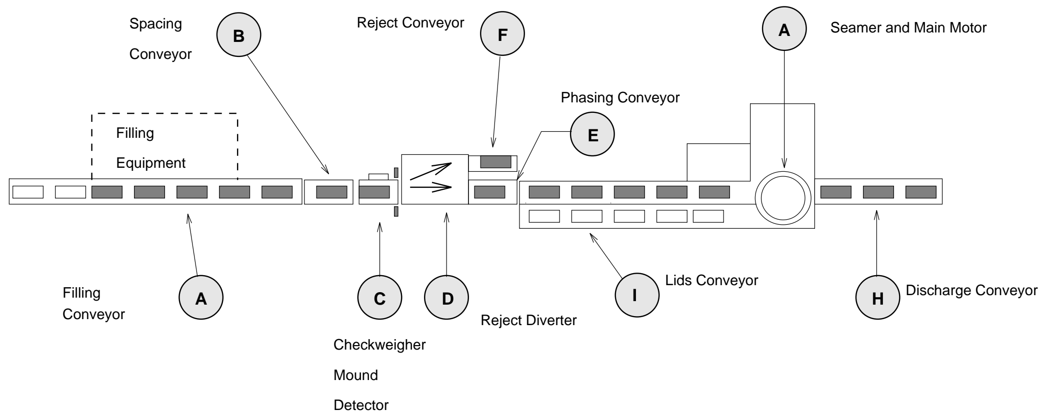

Figure 6.1: *Schematic drawing of the tray pack line of the CRAMTD project.*

6.1 CRAMTD Project Tray Packing Model

The aim of the CRAMTD project is to build an advanced dual-purpose (civilian and military) food processing manufacturing facility. The project comprises several manufacturing lines, among which is the tray packing line [5], chosen for the application of the Control-D tool. Figure 6.1 shows a schematic layout of the line.

The function of tray packing line is to take empty trays (from a tray stock), fill them with food rations, check for standardized requirements of weight (and perhaps discard a "non standard" tray), seal the tray, and finally discharge them to the distribution facility.

The tray pack line consists of the following *modules* as shown in Figure 6.1.

A. *Filling Conveyor*: The conveyor that takes the trays from the *filling station*.

B. *Spacing Conveyor*: The conveyor that ensures that the trays are properly spaced.

C. *Check Weight Station*: Where each tray is weighed and checked for "mounding" (in order to ensure that it could be closed with a lid).

D. *Reject Diverter*: If a tray weight is outside some predefined parameters or if its content is "mounded", then the tray is rejected and diverted to a different line.

E. *Phasing Conveyor*: Where lids and trays are synchronized.

F. *Rejected Tray Conveyor*: Where the rejected trays are collected.

G. *Seamer and Main Motor*: Where the trays are closed with the lids. Also the motor of this component actually drives the whole tray pack line.

H. *Discharge Conveyor*: Where the sealed trays are collected.

I. *Lid Conveyor*: Where the tray lids are fed to the system.

6.2 Failure Behavior Control

The control software for the tray packing line was built using industry standard *Programmable Control Logic* (PLC) devices (e.g. see [15]). These PLC's are programmed by defining *ladder diagrams* which represent boolean circuits

and are descendants of the old *electro-mechanical relay logic* used until a few decades ago in many industrial applications.

There are essentially two functions that the control software performs.

- Data gathering for statistical and tracking software. The relative efficiency of the line must be measured; this is essentially a function of the number of trays that get rejected. Also, each tray is eventually labeled so that its origin can be determined in case of problems with the quality of the content. If a tray in a batch is "rotten", then whole batch must be retired from distribution.

- "Graceful" fail-stop of the entire line in presence of various error conditions detected by a number of sensors distributed along the length of the line.

The application of the CDES/CTL modeling methodology with the Control-D tool concentrates on the reproduction of the fail-stop behavior of the PLC software.

This is a typical application of CDES. The possible failures are classified as uncontrollable events and the main requirements on the behavior of the system is that it stops "gracefully" when one (or more) of such failures occurs.

6.2.1 Modeling the Tray Packing Line

The tray packing line has 9 logical modules. Each of these modules has to communicate some information to the other ones (namely the movement of trays). The interleaving of the FSM models of these modules would result in a rather large state-space, therefore a simpler modeling strategy was adopted.

The modules are actually arranged "in series" and (apart from the *main motor*) they communicate only with the preceding and the following module. It was therefore deemed safe to break down the model into smaller sub-models involving only contiguous modules.

As an example, consider the Check Weight Station (Figure 6.2) and Diverter (Figure 6.3) subsystems[1]. The transitions in the Check Weight Station FSM must influence the transitions in the Diverter FSM. The interaction between the two subsystems is modeled by means of a "communication variable" (figure 6.4). Moreover, this modules are attached to the Main Motor of the line (which is actually located in the Seamer module).

Therefore the plant $\mathcal{P}$ consists of the Check Weight Station, the Diverter, the state variable and the Main Motor.

6.2.2 **CTL** Specification of Behavior

The specification $\mathcal{S}$ for the sub-modules considered is rather straightforward. The notation used indicates the combined state of the system with the

[1] The circle with the inscribed square indicate a duplicate node.

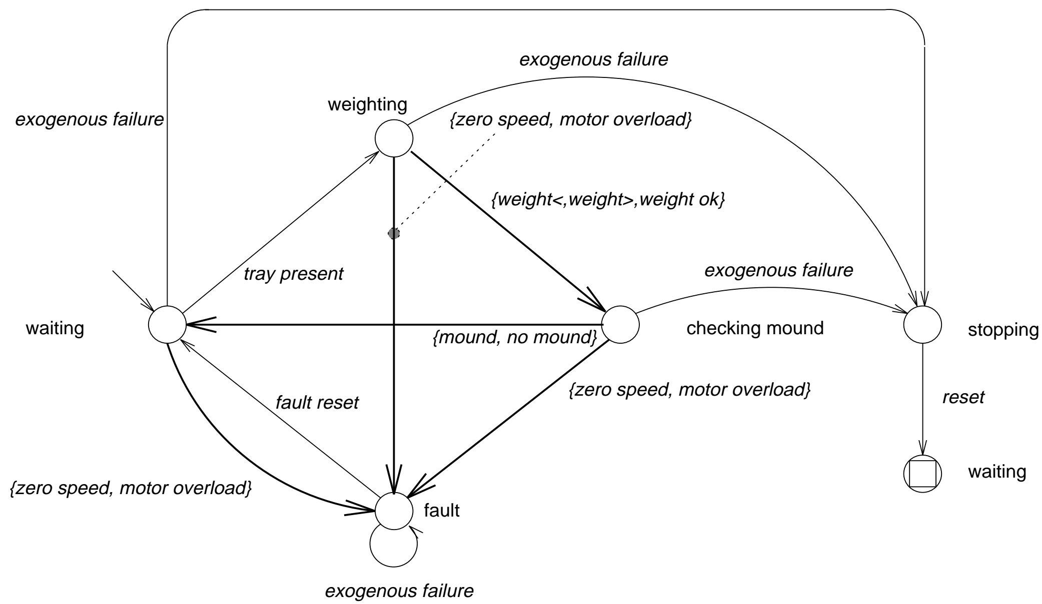

Figure 6.2: *Model of the discrete transitions for the Check Weight Station. The* uncontrollable *events are* zero speed, motor overload *and a generic* exogenous fault.

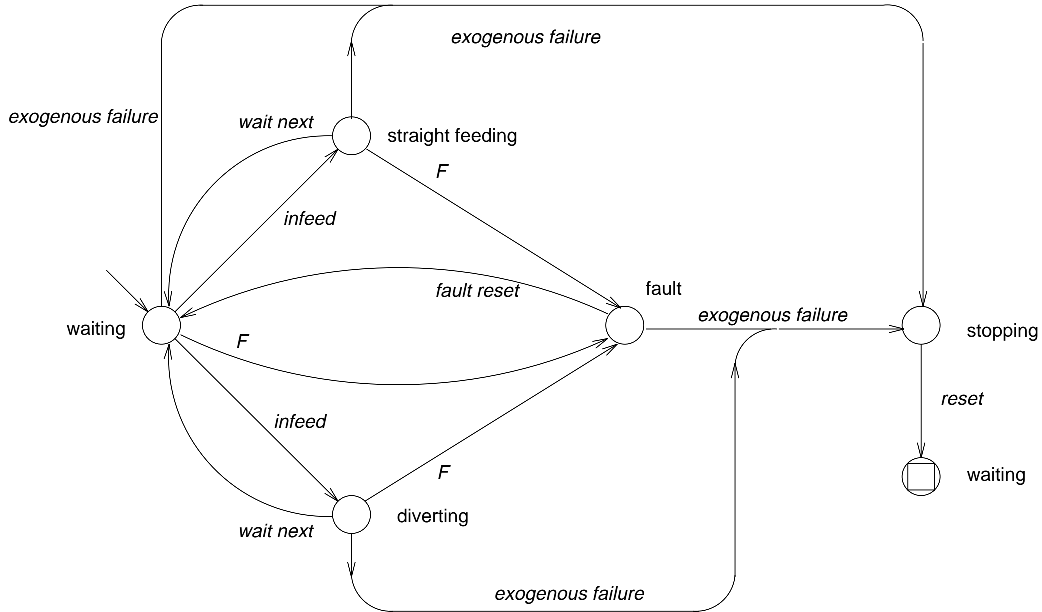

Figure 6.3: *Model of the discrete transitions for the Reject Diverter. The* uncontrollable *events are* zero speed, motor overload *and a generic* exogenous fault.

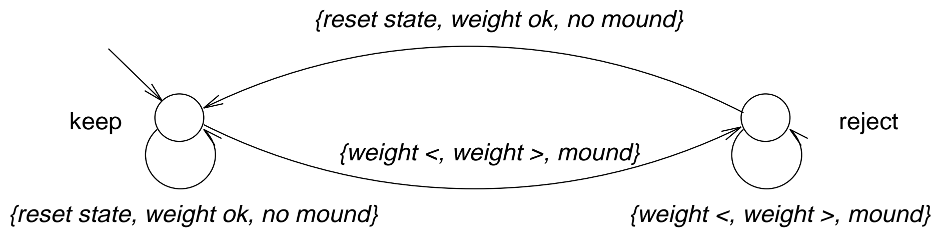

Figure 6.4: *States of the measurements done by the Check Weight Station. The Reject Diverter will make its transitions accordingly.*

form $[\mathsf{state\ id}_i]$[2]. Also, the notation $\mathsf{transition\ label} \mapsto [\mathsf{state\ id}_i]$ indicates that a state is reached through a transition marked label, i.e. $\exists s.(\delta(\mathsf{label}, s) = [\mathsf{state\ id}_i])$.

A. Nothing moves until the power in on (or after the power goes off):

$$\mathbf{AG}([\mathsf{main\ drive\ idle}] \Rightarrow \mathbf{AX}([\mathsf{main\ drive\ running}]).$$

B. If a fault occurs in one of the subsystems, then the fault must be propagated to the other ones as well:

$$\begin{aligned} &\mathbf{AG}([\mathsf{main\ drive\ fault}] \vee [\mathsf{weighter\ fault}] \vee [\mathsf{diverter\ fault}] \\ &\quad \Rightarrow \quad \mathbf{AX}(\mathsf{exogenous\ fault} \mapsto \\ &\qquad\qquad [\mathsf{main\ drive\ fault}, \mathsf{weighter\ fault}, \mathsf{diverter\ fault}])). \end{aligned}$$

C. If the weight station determines that a tray is non acceptable, the diverter must take the appropriate action:

$$\begin{aligned} \mathbf{AG}([\mathsf{keep}] &\Rightarrow \mathbf{AX}(\neg[\mathsf{diverting}]), \\ \mathbf{AG}([\mathsf{reject}] &\Rightarrow \mathbf{AX}(\neg[\mathsf{straight\ feeding}]). \end{aligned}$$

We use this negative convention in order to take into consideration the possibility of faults.

---

[2] We do not use the full form used in the implementation here, because of space constraints. Actually the form of a state should be

$$[\mathsf{diverter}, \mathsf{weight\ station}, \mathsf{weight\ result}, \mathsf{main\ drive}].$$

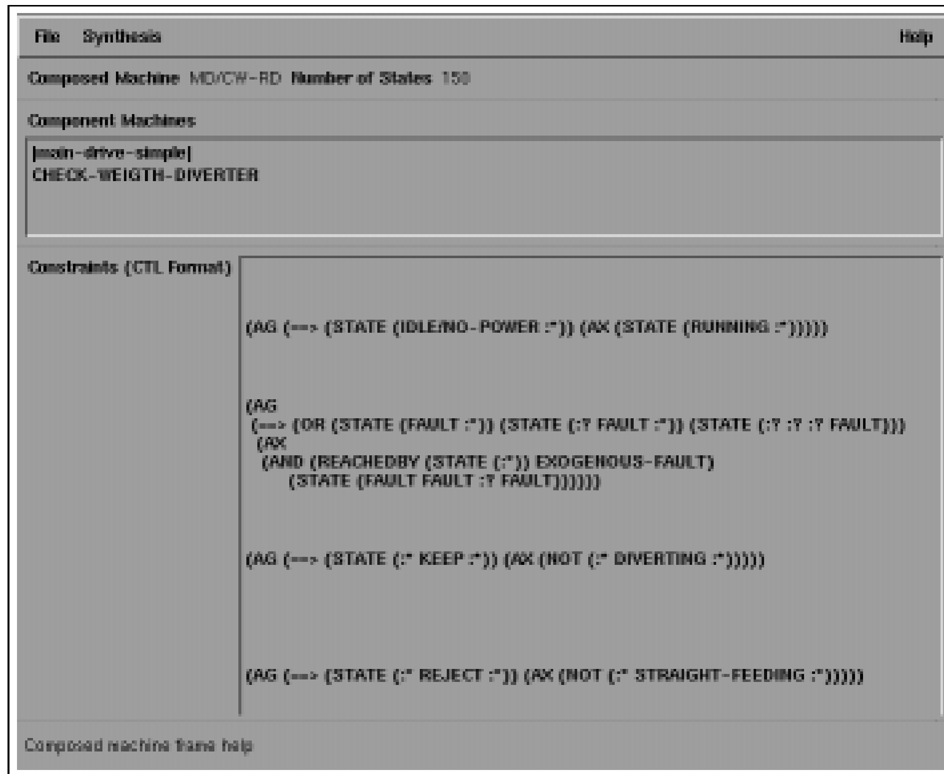

Figure 6.5: Control-D window with CRAMTD constraints.

These constraints represent safety, liveness, and "mandatory" properties that the weight station plus diverter subsystem must comply with. More precisely, the listed constraints model (1) a portion of the standard operations of the system (constraints A and C) and (2) the desired fault detection behavior of the system (constraint B). Figure 6.5 shows the Control-D "composed specification" window containing the internal Lisp form of the CTL constraints.

As an illustration of the the interplay between the fault detection requirements and the regular operations of the station, constraint C could not be

represented as $\mathbf{AG}([\mathsf{keep}] \Rightarrow \mathbf{AX}([\mathsf{straight\ feeding}]))$, because the uncontrollable transitions representing the faults must be taken into account.

6.3 Concluding Remarks

The CRAMTD system turned out to be not as difficult to model as the Walking Machine. This is simply due to the fact that timing constraints of various nature were not taken into consideration. This was justified by a rather safe assumption that the fail-stop behavior to be implemented can be considered "instantaneous", while all the other conditions are boolean in nature, simply because the PLC specification was taken as a starting point for the modeling process.

Chapter 7

Conclusion and Future Work

This thesis discussed how to use the branching-time temporal logic CTL as a basis for a discrete event system supervisor synthesis problem similar to the one defined by Ramadge and Wonham. It was shown that by modifying some of the basic assumptions of Ramadge and Wonham's model, it was possible to construct a semantically sound algorithm capable of solving the synthesis problem. The use of CTL as a specification language permits the expression of quite complex behaviors, while retaining a simple syntax. Moreover, this increased expressiveness does not increase the time or the space complexity of the synthesis algorithm with respect to similar schemes.

For the future there are two main directions of research. The Control-D system requires a more stable implementation of the MRSS algorithm, possibly with the inclusion of *symbolic encoding* techniques, like BDD's. Moreover, while CTL constitutes a very good specification language, it is necessary to

devise a better description language for the finite state models. SMV (op. cit.), COSPAN (ibid.) and PROMELA [40] are good candidates.

From a more theoretical viewpoint, there are three major open problems. The MRSS algorithm restricts the behavior of the plant to a language $L \subseteq \Lambda[\mathcal{P}]$. However, the *size* of this language is not known. There is a conjecture that the language produced is maximal, but no proof was found yet.

Another conjecture regards the treatment of disjunctions of path formulæ. The example in Section 3.2.2 leads to a restriction on the form of the disjunctions treatable by the Control-D system. The conjecture states that the supervisor synthesis problem for CTL with "full" disjunctions is $\mathcal{NP}$-complete.

Finally, the state-space explosion problem is still open. I.e. the problem of *decomposing* a system into parts to be treated in isolation is still a major obstacle on the widespread use of verification and synthesis systems. The book by Kurshan (op. cit.) contains some results in this direction. The Control-D system can treat a restricted form of decomposition by limiting the specifications to formulæ in CTL^{-}. While restrictive, this approach may prove satisfactory for a wide variety of systems.

Bibliography

[1] M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8:277–295, 1989.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] S. B. Akers. Binary Decision Diagrams. *IEEE Transaction on Computers*, c-27(6):509–516, June 1978.

[4] J. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence Journal*, (23):123–154, 1984.

[5] G. Alpan, T.O. Boucher, and D. Livingston. Level 2 automation of the tray pack line. CRAMTD Technical Working Paper (TWP) 88, Department of Industrial Engineering, Rutgers University, New Jersey Agricoltural Experimental Station, New Brunswick, New Jersey, 08903, USA, August 1994.

[6] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking for Real-Time System. In *International Symposium on Logic in Computer Science*, number 5, pages 414–425. IEEE Computer Press, 1990.

[7] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid Automata: An Algorithmic Approach to the Specificatio and Verification of Hybrid Systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[8] R. Alur and D. Dill. The Theory of Timed Automata. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice (REX Workshop)*, volume 600 of *Lecture Notes in Computer Science*, pages 45–73. Springer-Verlag, 1992.

[9] R. Alur and T. A. Henzinger. Real-time Logics: Complexity and Expressiveness. In *International Symposium on Logic in Computer Science*, number 5, pages 390–401. IEEE Computer Press, 1990.

[10] M. Antoniotti, M. Jafari, and B. Mishra. Applying Temporal Logic Verification and Synthesis to Manufacturing Systems. In *IEEE International Conference on Systems, Man and Cybernetics*, 1995.

[11] M. Antoniotti and B. Mishra. Automatic Synthesis Algorithms for Supervisory Controllers (Preliminary Report). In *International Conference on*

Computer Iintegrated Manufacturing and Automation Technology, pages 151–156. IEEE, October 1994.

[12] M. Antoniotti and B. Mishra. Discrete Event Models + Temporal Logic = Supervisory controller: Automatic Synthesis of Locomotion Controllers. In *IEEE International Conference on Robotics and Automation*, 1995.

[13] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory Control of a Rapid Thermal Multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, July 1993.

[14] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, March 1991.

[15] J. G. Bollinger and N. A. Duffie. *Computer Control of Machines and Processes*. Addison Wesley, 1988.

[16] A. Bouajjani and O. Maler, editors. *Second European Workshop on Real-time and Hybrid Systems*. VERIMAG Laboratoire, Grenoble, France, May 1995.

[17] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *27th Design Automation Conference*, pages 40–45. ACM/IEEE, 1990.

[18] B. A. Brandin, W. M. Wonham, and B. Benhabib. Manufacturing Cell Supervisory Control – A Timed Discrete Event System. In *IEEE International Conference on Robotics and Automation*, pages 931–936. IEEE, 1992.

[19] R. A. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[20] M. Browne, E. M. Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, c-35(12):1035–1044, 1986.

[21] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[22] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *5th LICS*, pages 428–439, 1990.

[23] C. L. Chang and C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[24] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.

ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

[25] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. Technical Report 114, School of Computer Science, Carnegie Mellon University, February 1994.

[26] E. M. Clarke, O. Grumberg, H. Hirashi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ Cache Coherency Protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.

[27] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional Model Checking. Technical Report CMU-CS-89-145, School of Computer Science, Carnegie Mellon University, April 1989.

[28] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 1990.

[29] J. J. Craig. *Introduction to Robotics*. Addison-Wesley, 1986.

[30] E. Davis. Branching Continuous Time and the Semantics of Continuous Action. Technical Report 666, Courant Institute of Mathematical Sciences, New York University, July 1994.

[31] M. D. Davis, R. Sigal, and E. J. Weyuker. *Computability, Complexity, and Languages*. Academic Press, 1994.

[32] T. L. Dean and M. P. Wellman. *Planning and Control*. Morgan Kaufmann, 1991.

[33] A. R. Deshpande. *Control of Hybrid Systems*. PhD thesis, University of California at Berkeley, 1994.

[34] M. D. Donner. *Real-Time Control of Walking*, volume 7 of *Progress in Computer Science*. Birkhäuser, 1986.

[35] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. MIT Press, 1990.

[36] E. A. Emerson and E. M. Clarke. Using Branching Time Temporal Logic To Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2(3):241–266, December 1982.

[37] M. Genesereth and N. Nilsson. *Logical Foundation of Artificial Intelligence*. Morgan Kaufmann, 1987.

[38] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[39] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. In *7th Annual IEEE Symposium on Logic in Computer Science*, pages 394–406. IEEE, IEEE Computer Society Press, June 1992.

[40] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[41] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[42] G. E. Hughes and M. J. Cresswell. *An introduction to Modal Logic.* University Paperbacks. Routledge, 1968. Original edition by Methuen and Co., Current edition, 1990.

[43] J. L. Jones and A. M. Flynn. *Mobile Robots.* A K Peters, 1993.

[44] R. E. Kalman, , P.L. Falb, and M. A. Arbib. *Topics in Mathematical System Theory.* McGraw-Hill, 1965.

[45] J. Košecká and L. Bogoni. Application of Discrete Event Systems for Modeling and Controlling Robotic Agents. In *IEEE International Conference on Robotics and Automation*, pages 2557–2562. IEEE, 1994.

[46] R. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[47] Y. Li and W. M. Wonham. Control of Vector Addition Discrete-Event Systems I – The Base Model. *IEEE Transactions on Automatic Control*, 38(8):1214–1227, August 1993.

[48] Y. Li and W. M. Wonham. Control of Vector Addition Discrete-Event Systems II – Controller Synthesis. *IEEE Transactions on Automatic Control*, 39(3):512–531, March 1994.

[49] J.-Y. Lin and D. Ionescu. Analysis and Synthesis Procedures of Discrete Event Systems in a Temporal Logic Framework. In *International Symposium on Intelligent Control*. IEEE, 1992.

[50] Z. Manna and A. Pnueli. Tool and Rules for the Practicing Verifier. In R. F. Rashid, editor, *CMU Computer Science, 25th Anniversary Commemorative*, Anthology Series, pages 125–159. ACM Press, 1991.

[51] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[52] Z. Manna and P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. In D. Kozen, editor, *Logics of Programs*, number 131 in Lecture Notes in Computer Science, pages 253–281. Springer-Verlag, 1981.

[53] B. J. McCarragher and H. Asada. A Discrete Event Approach to the Control of Robotic Assembly Tasks. In *IEEE International Conference on Robotics and Automation*, pages 331–336. IEEE, 1993.

[54] R. B. McGhee. Robot Locomotion. In *Neural Control of Locomotion*, pages 237–264. Plenum, 1976.

[55] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[56] R. Milner. Operational and Algebraic Semantics of Concurrent Processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 19, pages 1201–1243. MIT Press, 1990.

[57] B. Mishra. Notes on "Advanced Robotics". Manuscript based on a course held at Courant Institute of Mathematical Sciences, New York University, Spring semester 1994.

[58] B. Mishra and E. M. Clarke. Hierarchical Verification of Asynchronous Circuits Using Temporal Logic. *Theoretical Computer Science*, 38:269–291, 1985.

[59] A. Nerode and W. Kohn. Models for Hybrid Systems: Automata, Topologies, Ccontrollability, Observability. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[60] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An Approach to the Description and Analysis of Hybrid Systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 149–178. Springer-Verlag, 1993.

[61] A. Prior. *Past, Present and Future.* Clarendon Press, Oxford, 1967.

[62] M. H. Raibert. *Legged Robots That Balance.* MIT Press, 1986.

[63] P. J. Ramadge and W. M. Wonham. Modular Feedback Logic for Discrete Event Systems. *SIAM J. Control and Optimization*, 25(5):1202–1218, September 1987.

[64] P. J. Ramadge and W. M. Wonham. On the Supremal Controllable Sublanguage of a Given Language. *SIAM J. Control and Optimization*, 25(3):637–659, May 1987.

[65] P. J. G. Ramadge. Some Tractable Supervisory Control Problems for Discrete-Events Systems Modeled by Büchi automata. *IEEE Transactions on Automatic Control*, 34(1):10–19, 1989.

[66] P. J. G. Ramadge and W. M. Wonham. Supervisory Control of a Class of Discrete Events Processes. *SIAM J. Control and Optimization*, 25(1):206–230, 1987.

[67] P. J. G. Ramadge and W. M. Wonham. The Control of Discrete Events Systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[68] K. Rudie and W. M. Wonham. Think Globally, Act Locally: Decentralized Supervisory Control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[69] Y. Shoham. *Reasoning about Change: Time and Causation form the Standpoint of Artificial Intelligence.* MIT Press, 1988.

[70] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logic. *Journal of the ACM*, 32:733–749, 1985.

[71] E. D. Sontag. *Mathematical Control Theory*, volume 6 of *TAM*. Springer-Verlag, 1990.

[72] G. L. Steele Jr. *Common Lisp the Language.* Digital Press, 2 edition, 1990.

[73] D. J. Todd. *Walking Machines.* Kogan Page, 1985.

[74] R. S. Wallace. Miniature Direct Drive Rotary Actuators. *Robotics and Autonomous Systems*, 11:129–133, 1993.

[75] R. A. Williams, B. Benhabib, and K. C. Smith. A Hybrid Supervisory Control System for Flexible Manufacturing Workcells. In *IEEE International Conference on Robotics and Automation*, pages 2551–2556. IEEE, 1994.

[76] P. Wolper. The Tableau Method for Temporal Logic: an overview. *Logique at Analise*, (28):119–136, 1985.

[77] W. M. Wonham. *Linear Multivariate Control – A Geometric Approach.* Springer–Verlag, 3rd edition, 1985.