On Modeling of Discrete Event Dynamic Systems, using Symbolic Algebraic Methods

Johan Gunnarsson

REGLERTEKNIK

AUTOMATIC CONTROL

Division of Automatic Control
Department of Electrical Engineering
Linköping University, Sweden
Email: `johan@isy.liu.se`

Linköping 1995

©1995 Johan Gunnarsson,
Department of Electrical Engineering,
Linköping University,
S-581 83 Linköping,
Sweden.

To Marit and Emma

Abstract

The interest in discrete event systems (DEDS) has increased during the last years, due to the lack of methods and tools that are capable to handle the complexity of problems and tasks present in industry today. In this thesis we will consider a symbolic and algebraic framework which will be used for modeling, analysis, and synthesis of DEDS.

We will use polynomials belonging to a polynomial ring over finite fields to represent finite quantities, functions, and relations of a DEDS system. The polynomials make it possible to improve efficiency and scalability of DEDS computations, as shown in this thesis by the modeling and analysis of the landing gear controller of the Swedish fighter aircraft JAS 39 Gripen. A polynomial model, represented by binary decision diagram (BDD), is automatically generated from a 1200 lines Pascal implementation, which contains 105 binary variables of which 26 are state variables. Function specifications expressed with temporal algebra, are verified using tools for dynamic analysis, which we also use to compute a polynomial representing the set of all reachable states in the model.

To explore the ability and applicability of the polynomial approach when doing synthesis, we use a tank system containing actuators (pump and valves) and sensors (the tank level and measurable disturbances). We propose a synthesis method that uses actuator priority, weighting of states, and Gröbner bases to compute explicit control laws for the actuators, fulfilling the control objectives even if one of the actuators (the pump) is defective.

Modeling aspects are emphasized further, by comparing the polynomial approach which we have used, with Boolean expressions and established DEDS approaches in the community of automatic control like Ramadge-Wonham, Petri nets, and COCOLOG. We discuss how to handle transformation between signals and events for DEDS and how to modularize DEDS to gain complexity advantages. Model description languages are discussed and desirable features are stated, using the experiences achieved from the modeling of the tank system and the landing gear controller.

Preface

This thesis consists of three separate parts:

- Part I Symbolic Algebraic Discrete Systems – Applied to the JAS 39 Fighter Aircraft.
- Part II Simultaneous Supervision and Control – A First Example.
- Part III General Aspects of Modeling of Discrete Event Dynamic Systems.

Parts I and II contain rewritten and reorganized material from

- R. Germundsson, J. Gunnarsson, and J. Plantin. Symbolic Algebraic Discrete Systems – Applied to the JAS 39 Fighter Aircraft. Technical Report LiTH-ISY-R-1718, Department of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, December 1994.
- J. Gunnarsson and J. Plantin. Control Law Synthesis for a Discrete Event System. Technical Report LiTH-ISY-R-1651, Department of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, August 1994.

Parts I and II have been co-authored with J. Plantin and are also found in [71].

The different parts in the thesis can be read independently and in any order, but to increase the benefit we suggest that the material is approached as follows:

(i) Chapter 3 lays the theoretical foundation on which the thesis rests. It is therefore essential for the understanding of all three parts.

(ii) Part I can be read independently but since it concerns modeling and analysis aspects, the reader may benefit from reading it before Parts II and III.

(iii) Read Parts II and III in order since Part II is the description of a first example of modeling and synthesizing the tank system which is used in Part III, where the modeling aspects are discussed and compared to other approaches within the field.

Acknowledgments

The work that has resulted in this thesis could not be performed without the support from several people.

First of all I would like to thank my supervisor, professor Lennart Ljung for the opportunity to be a member of his group, and for all encouragement and guidance he has given me. I would also like to thank a man with a good taste in research subjects, Roger Germundsson, for the inspiration and support he has shared with me. His research effort in the domain of DEDS made this thesis possible.

To be a member of a group like the Automatic Control Group in Linköping, has given me experiences I never forget[1]. Thank you all.

This work was supported by the Swedish Research Council for Engineering Sciences (TFR) and the Swedish National Board for Industrial and Technical Development (NUTEK), which is gratefully acknowledged.

I would also like to thank Mats Jirstrand for patiently proofreading the manuscript, Ulf Nilsson, IDA, for an excellent introduction to structured operational semantics, Jan-Erik Strömberg for being a bond graph guru, and Göran Backlund and Ove Åkerlund at Saab, who supplied me with valuable information on the JAS landing gear.

I would like to thank all friends to me and my family, and especially Annacarin[2], Berit, Hasse, Henrik and Ingrid, for all support and help during this period of time.

To my research companion Jonas: Thank you for your patience with my weird ideas, and the fun and friendship during these years. I congratulate the one who is going to share your whiteboard next.

To my parents Gunnar and Kerstin: Tank you for your love and support that you give me in this work, in spite that you have not the slightest idea of what I am doing.

At the end I give my greatest gratitude to my wife and daughter for their love, patience and encouragement. I love you.

[1] They are quite nice actually, even if they have a strange attitude to microwave ovens.

[2] Now you've got what you wanted.

Contents

[3]This part is co-authored with J. Plantin and can also be found in [71].

[4]This part is co-authored with J. Plantin and can also be found in [71].

Chapter 1

Introduction

The following question gives a motivation for this thesis:

> *"Why not try to bring mathematical techniques where they are not yet used, in order to improve formal guarantees and to reduce the size and combinatorial complexity of the resulting code?"*
>
> *Benveniste and Åström [5]*

The question above deals with the problem of developing control software systems in industry today. Control software is becoming more and more complex, as the hardware improves and the demands for the systems increase. To improve the development of software systems, tools like data bases, knowledge bases, and expert systems are used together with software development methods. This makes it possible to use higher abstraction in the development process.

In the community of automatic control the design of a system (controller) is based on models and objectives from which the system is derived, rather than designed using ordinary programming techniques. To make such derivations possible we need a formal framework (mathematics) by which it is possible to represent models of systems and control objectives. A powerful mathematical framework is the foundation of control theory. The use of control theory for continuous systems is well developed and widely used in applications. The control community is therefore inspired to find mathematics that is usable for systems where software techniques are used today.

The class of systems considered here is *discrete event dynamic systems* (DEDS). By a DEDS we mean a system with a discrete state space and with piecewise constant trajectories that evolve in response to certain abrupt input events. The research in the domain of DEDS is relatively new in the control community. Over the years many approaches have been introduced that in some sense give methods for modeling, analysis and synthesis for DEDS. Still, the most difficult and perhaps the most important feature to achieve in a DEDS framework, is the ability to reduce the combinatorial complexity that always tends to be insurmountable.

The theme of this thesis is modeling, analysis and synthesis of DEDS where a mathematical framework[1], developed by Germundsson [35], is used. The framework is based on symbolic and algebraic methods and tools that give efficient computations even for larger DEDS. This property will be exemplified in Part I in this thesis where the modeling and analysis of the landing gear controller of the Swedish fighter aircraft JAS 39 Gripen is presented. In Part I we also formally introduce the polynomial approach.

As mentioned above, the common approach for a design problem in control is to derive (or synthesize) a controller from a model (containing the information of the system behavior) and the control objectives (specifying the desired behavior). How this can be done within the polynomial approach is shown in Part II in this thesis, where the synthesis of a control law for a tank system is performed.

Part III contains three separate chapters. The first one gives a brief description of the DEDS approaches of today. In the second chapter we discuss more general aspects of modeling, using the polynomial approach. The last chapter makes a comparison between the polynomial approach and three other DEDS approaches: Ramadge and Wonham, Petri nets, and COCOLOG. This comparison is performed by remodeling the tank system from Part II.

[1] In this thesis we will refer to this framework as the *polynomial approach*

Part I

Symbolic Algebraic Discrete Systems – Applied to the JAS 39 Fighter Aircraft[1]

[1]This part is co-authored with J. Plantin and can also be found in [71].

Chapter 2

Introduction

Discrete event dynamic systems (DEDS) have become increasingly important in many industrial applications during the last 15 years. As examples we could mention integrated circuit design and design of telecommunication systems. Also from a control perspective there has been an increased interest in DEDS. One reason for this is that large parts of controller code consist of procedures handling switching between operating modes as well as different safety aspects.

A parallel trend is that the complexity of the systems has increased drastically. This has lead to a need for structured methods in the design process, which has not yet been satisfied. A common problem in industry today is the lack of structured methods and good software tools for design and analysis of DEDS. Our ultimate goal is to derive methods and tools for the entire design process for DEDS. This is however a long-term goal with many non-trivial problems to solve.

In this part of the thesis we will describe the modeling and analysis of an existing discrete subsystem of a modern fighter aircraft, the landing gear control system on the JAS 39 Gripen. This system was designed and implemented by Saab Military Aircraft without any formal methods or tools, as is usually the practice for discrete event dynamic systems in industry today [36].

The goal is to investigate the use of exact formal methods in the (industrial) design process of mainly discrete systems. By formal methods we mean methods where the results can be formally derived (proved) from the prerequisites (e.g. a system model and specification). The design process could roughly be grouped into the following steps:

Modeling: Obtaining a mathematical model of the process that has essentially the same behavior as the real process.

Analysis: Analyzing the behavior of the model.

Design: Modifying the behavior of the process model, through the use of a controller or supervisor model.

Implementation: Generating an implementation that has the same behavior as the model of the controller.

As mentioned, in a long term perspective the goal is to capture all of these categories in a way which is relevant in an industrial setting. Here we focus on the first two steps in the design process.

We will build a mathematical model of the landing gear control system and parts of its (natural language) specification and analyze its behavior w.r.t. this specification. The main focus is not on the specific system, but rather on the general methods that can be applied to discrete event dynamic systems of industrial size.

In the next chapter we will give an introduction to the polynomial representation used for modeling and analysis of DEDS. In Chapter 4 we introduce the methods and tools employed in this study. Chapter 5 contains a description of the landing gear system and the controller. In Chapter 6 we describe the modeling of the system and in Chapter 7 we analyze the system model with respect to function specifications. Finally we give some conclusions and ideas for future work in Chapter 8.

Chapter 3

Polynomial Representation

It is essential to have a powerful way to describe a system. For example, considering a time-continuous dynamic system, a natural mathematical description is

$$f(\dot{x}(t), x(t), u(t)) = 0 \tag{3.1}$$

denoting a set of general differential equations where $x(t)$ is the state, $\dot{x}(t)$ the time derivative of $x(t)$ and $u(t)$ the input. The model (3.1) gives $\dot{x}(t)$ implicitly, whereas the model

$$\dot{x}(t) = f(x(t), u(t)) \tag{3.2}$$

gives $\dot{x}(t)$ explicitly.

Analogously for discrete event dynamic systems we use the mathematical description

$$f(x^+, x, u) = 0 \tag{3.3}$$

which denotes a set of relations between inputs u, states x and the next state value x^+. If x^+ is given explicitly from inputs and the state of the system, (3.3) can be written as

$$x^+ = f(x, u) \tag{3.4}$$

and the output can be given as

$$y = h(x, u). \tag{3.5}$$

In an untimed DEDS all variables are discrete and the system is event driven, i.e. "real time" is not included in the description. Only the order of occurrence of events is taken into account.

If we have some relation between the variables x and y we will often write just

$$f(x, y) \tag{3.6}$$

saying that f is the relation or the model. The interpretation of this is that the relation between x and y is given by the solutions to

$$f(x, y) = 0. \tag{3.7}$$

We will use polynomials in a quotient ring (see Section 3.2) to represent relations as e.g. (3.6). Other formalisms for discrete event dynamic systems can be transformed to and from this polynomial representation. For example, the class of models described here includes any finite state based representation, which in its turn includes the regular language based models proposed by Ramadge and Wonham [74, 76].

We will give an introduction to the polynomial representation, starting with a brief review of concepts from commutative algebra. For a more thorough introduction to commutative algebra, see [26]. See also [35] and [30, 31, 32, 33, 34] for more details on the polynomial representation.

3.1 Basic Commutative Algebra

In this section we will introduce some basic concepts from commutative algebra that will be needed for the introduction of the polynomial representation. We begin by defining *abelian group*, *commutative ring* and *field.*

Definition 3.1 An *abelian group* G is a set of objects and a composition map (denoted by $*$) satisfying the following properties:

(i) *Closure.* For every $a, b \in G$, $c = a * b$ is in G.

(ii) *Associativity.* For every $a, b, c \in G$,

$$a * (b * c) = (a * b) * c.$$

(iii) *Commutativity.* For every $a, b \in G$,

$$a * b = b * a.$$

(iv) *Identity.* There is an identity element $e \in G$ that satisfies

$$a * e = e * a = a$$

for every $a \in G$.

(v) *Inverse.* If $a \in G$, then there is some element $\tilde{a} \in G$ such that

$$a * \tilde{a} = \tilde{a} * a = e.$$

□

An example of a finite abelian group is the set $\{0, 1, \ldots, 9\}$ under addition modulo-10. The identity element is 0.

From the abelian group we can construct a commutative ring by adding additional structure.

Definition 3.2 A *commutative ring* R is a set of objects obeying two composition maps, addition and multiplication, and satisfying the following properties:

(i) R is an abelian group under addition.

(ii) *Closure.* For any $a, b \in R$, the product ab is in R.

(iii) *Multiplicative identity.* There is an identity element f that satisfies

$$af = fa = a$$

for all a in R.

(iv) *Commutativity.* For every a, b in R

$$ab = ba.$$

(v) *Associativity.* For every a, b, c in R

$$a(bc) = (ab)c.$$

(vi) *Distributivity.* For every a, b, c in R

$$a(b + c) = ab + ac.$$

□

An example of a commutative ring is the set of all polynomials in x with real-valued coefficients under polynomial addition and multiplication. The multiplicative identity is the zero-degree polynomial, $p(x) = 1$.

From the commutative ring we can now define a field:

Definition 3.3 A *field* k is a commutative ring where every element except 0 has a multiplicative inverse, i.e.

$$(\forall a \in R, \exists \tilde{a} \in R \backslash \{0\})\ a * \tilde{a} = 1. \tag{3.8}$$

□

We say that a field is *finite* if the number of elements in the set is finite. A finite field is denoted $\mathbb{F}_q$, where q is the number of elements in the set.

An example of a finite field is the set $\{0, 1\}$ under modulo-two addition and modulo-two multiplication.

The following theorem states when it is possible to construct a finite field. For a more detailed investigation of this, see e.g. [61].

Theorem 3.1 If $q = p^m$ where p is prime and $m \in \mathbb{Z}_+$, then there exists a field $\mathbb{F}_q$ with q elements. □

We proceed by introducing the concepts of a polynomial ring and an ideal.

Definition 3.4 A *polynomial ring* (denoted $k[Z]$ or $k[z_1, \dots, z_n]$) is the set of all polynomials in the variables $z_1, z_2, \dots, z_n$ with coefficients from a field k. □

If the coefficients are from a finite field $\mathbb{F}_q$ we will use the notation $\mathbb{F}_q[Z]$ for the polynomial ring.

We can now define an *ideal*.

Definition 3.5 A subset $I \subset k[z_1, \dots, z_n]$ is an *ideal* if it satisfies:

(i) $0 \in I$.

(ii) If $f, g \in I$, then $f + g \in I$.

(iii) If $f \in I$ and $h \in k[z_1, \dots, z_n]$, then $hf \in I$.

□

The following lemma states that an ideal can be generated from a set of polynomials.

Lemma 3.1 ([26]) Let $f_1, \dots, f_s$ be polynomials in $k[z_1, \dots, z_n]$ and set

$$\langle f_1, \dots, f_s \rangle = \left\{ \sum_{i=1}^{s} h_i f_i \mid h_1, \dots, h_s \in k[z_1, \dots, z_n] \right\}. \tag{3.9}$$

Then $\langle f_1, \dots, f_s \rangle$ is an ideal of $k[z_1, \dots, z_n]$. □

We will call $\langle f_1, \dots, f_s \rangle$ the *ideal generated by* $f_1, \dots, f_s$, and call $f_1, \dots, f_s$ generators of the ideal. In this thesis we will only consider *finitely* generated ideals, i.e. s is finite.

The *variety* of the ideal, $\mathrm{V}(I)$, denotes the set of common zeros for all polynomials in I, i.e.

$$V(I) = \{z \in k^n : f(z) = 0 \text{ for all } f \in I\}. \tag{3.10}$$

This set is the same as all common zeros of the generator polynomials $f_1, f_2, \dots, f_s$.

The real importance of ideals is that they give us a possibility to compute with the varieties of the relations describing our systems.

Example 3.1 Let $I = \langle z_1^2 - 1,\ 5z_2,\ (z_1 - 1)(z_1 + 2)\rangle \subset \mathbb{Z}[Z]$. The variety V(I) is the set

$$V(I) = \{(z_1, z_2) : (1, 0)\}. \tag{3.11}$$

All relations in a model of a DEDS can be represented by polynomials in $\mathbb{F}_q[Z]$. This is a consequence of the fact that $\mathbb{F}_q[Z]$ is *functionally complete*, see [34]. One relation can, however, be represented by an infinite number of polynomials in $\mathbb{F}_q[Z]$. A one-to-one correspondence between relations and polynomials is desirable and this leads us to the introduction of a quotient polynomial ring in the next section.

3.2 The Quotient Polynomial Ring $R_q[Z]$

Before defining the quotient polynomial ring we need some introductory definitions.

Definition 3.6 ([26]) Let $I \subset k[z_1, \dots, z_n]$ be an ideal, and let $f, g \in k[z_1, \dots, z_n]$. We say that f and g are *congruent modulo I*, written

$$f \equiv g \bmod I.$$

if $f - g \in I$. □

It can be proven (see [26]) that if the polynomials $f, g \in k[Z]$ are such that $f \equiv g \mod I$ for the ideal $I \subset k[Z]$ then f and g define the same function on the elements of the variety $V(I)$ of the ideal I.

Example 3.2 Let $I = \langle z^3 - z\rangle \subset \mathbb{F}_3[z]$, $f(z) = z^4$ and $g(z) = z^2$, then

$$f(z) \equiv g(z) \bmod I. \tag{3.12}$$

The variety of the ideal is

$$V(I) = \{z : 0, 1, 2\}. \tag{3.13}$$

We can now check that $f(z) = g(z)$ for all $z \in V(I)$. The identity holds trivially for $z \in \{0, 1\}$. The value $f(2)$ can be computed as

$$f(2) = 2^4 \bmod 3 = 16 \bmod 3 = 1. \tag{3.14}$$

In the same way we get $g(2) = 1$.

It can also be shown that the congruence defines an equivalence relation, which means that the ring can be partitioned into a collection of disjoint subsets called equivalence classes. These equivalence classes are used to define the *quotient*.

Definition 3.7 ([26]) The *quotient* of $k[z_1, \ldots, z_n]$ modulo I, written

$$k[z_1, \ldots, z_n]/I$$

is the set of equivalence classes for congruence modulo I

$$k[z_1, \ldots, z_n]/I = \{[f] \mid f \in k[z_1, \ldots, z_n]\} \tag{3.15}$$

where the equivalence class $[f]$ is

$$[f] = \{g \in k[z_1, \ldots, z_n] \mid g \equiv f \bmod I\}. \tag{3.16}$$

□

From this we proceed to define the main object of the polynomial approach to DEDS, the quotient polynomial ring [35].

Definition 3.8 The *quotient polynomial ring* $R_q[Z]$ is defined as

$$R_q[Z] = \mathbb{F}_q[Z]/\langle z_1^q - z_1, \ldots, z_n^q - z_n\rangle. \tag{3.17}$$

□

We note that $R_q[Z]$ contains all polynomials in $\mathbb{F}_q[Z]$. However, the polynomials are partitioned into equivalence classes by the ideal

$$\langle z_1^q - z_1, \ldots, z_n^q - z_n\rangle.$$

It is shown in [35] that we can represent any relation between the variables Z (where $Z = \{z_1, \ldots, z_n\}$ and $z_i \in \mathbb{F}_q$) uniquely with a polynomial in $R_q[Z]$, i.e. there is a one-to-one correspondence between the relations and the equivalence classes in $R_q[Z]$. Each equivalence class of polynomials in $R_q[Z]$ can be represented by one of its members in all operations on relations. For every equivalence class there is a polynomial with degree less than q. Therefore the polynomial representing an equivalence class in $R_q[Z]$ can be chosen such that the degree and the "length" of the polynomial has an upper limit. This gives some complexity advantages.

Polynomials in $R_q[Z]$ will be the "language" we use to model DEDS. In the following we describe a number of properties that will be used in the modeling and control of DEDS.

3.3 Representing Functions with Polynomials

To illustrate how to represent a function with polynomials, the theorem of *functional completeness* from [34] will be presented.

Theorem 3.2 The polynomial ring $\mathbb{F}_q[z_1, \ldots, z_n]$ is *functionally complete.* $\square$

Proof Let $f(z_1, \ldots, z_n) : \mathbb{F}_q^n \to \mathbb{F}_q$ be any function. The corresponding polynomial $f_p(Z) \in R_q[Z]$ is computed as

$$f_p(Z) = \sum_{\xi \in \mathbb{F}_q^n} L_\xi(Z) f(\xi) \tag{3.18}$$

where $L_\xi(Z) = L_{\xi_1}(z_1) \cdots L_{\xi_n}(z_n) \in R_q[Z]$ and

$$L_{\xi_i}(z_i) = \frac{\prod_{\nu \in \mathbb{F}_q \setminus \{\xi_i\}} (z_i - \nu)}{\prod_{\nu \in \mathbb{F}_q \setminus \{\xi_i\}} (\xi_i - \nu)} = \begin{cases} 1 & , z_i = \xi_i \\ 0 & , z_i \neq \xi_i \end{cases} \tag{3.19}$$

is the *Lagrange interpolating polynomial*, see e.g. [79]. We then have $f(\xi) = f_p(\xi)$ for all $\xi \in \mathbb{F}_q^n$. $\square$

Example 3.3 Let the function $J(x)$ be defined by the table ($q = 7$):

x	$J(x)$
0	3
1	2
2	1
3	0
4	1
5	2
6	3

In this case we can write (3.18) as

$$J_p(x) = \sum_{\xi \in \mathbb{F}_7} L_\xi(x) J(\xi) \tag{3.20}$$

where $J(\xi)$ is given by the table above and $L_\xi(x)$ is given by (3.19). The polynomial

$$L_0(x) = \frac{(x-1)(x-2)(x-3)(x-4)(x-5)(x-6)}{(0-1)(0-2)(0-3)(0-4)(0-5)(0-6)} = 1 + 6x^6 \tag{3.21}$$

evaluates to 1 only for $x = 0$. Computing $L_1(x), \ldots, L_6(x)$ analogously, we get the resulting polynomial

$$J_p(x) = 3 + 3x + 6x^2 + x^3 + 2x^4 + 6x^5 + 2x^6 \tag{3.22}$$

where $J_p(x) \in R_7[x]$. Since $q = 7$, the degree of $J_p(x)$ cannot be higher than 6. To compute the values of $J_p(x)$, just substitute x with its value and reduce the result modulo 7.

3.4 Representation of Logical Expressions

In the previous section we showed how to represent a function $f : \mathbb{F}_q^n \to \mathbb{F}_q$ as a polynomial in $R_q[Z]$. It is often useful to be able to represent logical expressions and conditions in the formalism, i.e. to consider functions $b : \mathbb{F}_q^n \to \{\texttt{true}, \texttt{false}\}$. This could be done by interpreting the values of the polynomial $b(Z)$ as

$$\begin{aligned} b(Z) = 0 &\Rightarrow \texttt{true} \\ b(Z) \neq 0 &\Rightarrow \texttt{false} \end{aligned} \tag{3.23}$$

If $a(Z), b(Z) \in R_q[Z]$ represent two logical expressions, the result of AND-, OR- and NOT-operations can be computed by algebraically manipulating the polynomials a and b as

$$a(Z) \wedge b(Z) = 1 - (1 - a(Z)^{q-1})(1 - b(Z)^{q-1}) \tag{3.24}$$

$$a(Z) \vee b(Z) = a(Z)b(Z) \tag{3.25}$$

$$\neg a(Z) = 1 - a(Z)^{q-1} \tag{3.26}$$

where the NOT-operation maps zero on one. This gives a natural extension of the logical operations to ideals in $R_q[Z]$. See [32] for further details.

For convenience we will sometimes use the logical operations $\to, \leftrightarrow, \exists$ and $\forall$, which can be expressed using the logical operations defined above.

$$a(Z) \to b(Z) = \neg(a(Z) \wedge \neg b(Z)) \tag{3.27}$$

$$a(Z) \leftrightarrow b(Z) = (\neg a(Z) \wedge \neg b(Z)) \vee (a(Z) \wedge b(Z)) \tag{3.28}$$

$$\exists z_i \; a(z_1, \ldots, z_n) = \bigvee_{j=0}^{q-1} a(z_1, \ldots, z_{i-1}, j, z_{i+1}, \ldots, z_n) \tag{3.29}$$

$$\forall z_i \; a(z_1, \ldots, z_n) = \bigwedge_{j=0}^{q-1} a(z_1, \ldots, z_{i-1}, j, z_{i+1}, \ldots, z_n) \tag{3.30}$$

Example 3.4 Given the polynomial $a(x)$, let $b(x)$ be defined as

$$b(x) = \begin{cases} 1 & , a(x) \neq 0 \\ 0 & , a(x) = 0 \end{cases} \tag{3.31}$$

The polynomial $b(x)$ can be generated as

$$b(x) = \neg\neg a(x) \tag{3.32}$$

since the expression for NOT in (3.26) maps all zeros to ones.

For simplicity and clarity it is sometimes convenient to express polynomials as logical expressions. We can e.g. write that a polynomial $p \in R_3[Z]$ is

$$p(Z) = z_1 \wedge z_2. \tag{3.33}$$

Using the logical operation in (3.24), we interpret this as

$$p(Z) = z_1^2 + z_2^2 - z_1^2 z_2^2. \tag{3.34}$$

3.5 Gröbner Bases

For each ideal in a polynomial ring there are many possible sets of polynomials that generate the ideal. To be able to decide if two ideals are equal, we need a standard for choosing generators of an ideal. Gröbner bases[1] form one standard representation for ideals. The Gröbner bases are in a sense a nonlinear generalization of Gaussian elimination.

3.5.1 Gröbner Bases for General Polynomial Rings

Gröbner bases can be regarded as the "simplest" representation of an ideal w.r.t. some *term ordering*.

A monomial term ordering gives an order to all monomials in the polynomial ring, and a way to define the degree, $\deg(f(Z))$, of a polynomial $f(Z)$. For example by *lexicographic order* $y > x$ we mean a term ordering where e.g.

$$\deg(y^2x) > \deg(yx^5) > \deg(yx^4) > \deg(x^9).$$

The *leading term*, $LT(f(Z))$, is the term in $f(Z)$ with highest degree. The degree of a polynomial $f(Z)$ is

$$\deg(f(Z)) = \deg(LT(f(Z))). \tag{3.35}$$

The Gröbner basis of an ideal can be regarded as the set of generators which has the lowest possible degree w.r.t. a given term ordering.

Given a polynomial p and a polynomial set F, the *remainder polynomial* r is computed as

$$r = p - \sum_{f \in F} \alpha_f f \tag{3.36}$$

where α_f are chosen to give r the lowest possible degree. Note that $\deg(r) < \deg(f)$ for all polynomials $f \in F$. We say that p reduces to r w.r.t. F and denote this

$$p \longrightarrow_F r. \tag{3.37}$$

[1] Only a brief introduction is given in this report. Further details of Gröbner bases for ideals in a general polynomial ring $k[Z]$ can be found in [26].

If $p \longrightarrow_F p$ we say that p is *reduced* w.r.t. F. If F is *autoreduced* then all polynomials in F are reduced w.r.t. F.

$S(p_1, p_2)$ denotes the *S-polynomial* of p_1 and p_2, and is computed as

$$S(p_1, p_2) = h_1 p_1 - h_2 p_2 \tag{3.38}$$

where h_1 and h_2 are terms of lowest degree such that $LT(h_1 p_1) = LT(h_2 p_2)$.

The polynomial set $G = \{g_1, \dots, g_n\}$ is a Gröbner basis for the ideal I iff $G \subseteq I$, $\langle G \rangle = I$, and

$$S(g_i, g_j) \in I \quad \forall\, i \neq j \tag{3.39}$$

i.e. $S(g_i, g_j) \longrightarrow_G 0$. To avoid further details we can use this result as a definition of a Gröbner basis.

A polynomial p is a member of an ideal I iff

$$p \longrightarrow_G 0 \tag{3.40}$$

where G is a Gröbner basis of I.

To compute a Gröbner basis we can use Buchberger's algorithm (see [26]):

Algorithm 3.1

1. Take the generators of I as candidates for a Gröbner basis. Denote this set G.
2. If all s-polynomials $S(g_i, g_j) \longrightarrow_G 0$, $i \neq j$, then G is a Gröbner basis.
3. Otherwise add to G the remainder r computed as

$$S(g_i, g_j) \longrightarrow_G r \neq 0$$

4. Make G autoreduced and go to 2.

□

An autoreduced Gröbner basis G for an ideal is unique. If two ideals have the same autoreduced Gröbner basis w.r.t. to the same term ordering, the ideals are equal.

3.5.2 Gröbner Bases in $R_q[Z]$

Gröbner bases are used and defined in $R_q[Z]$ in analogy with the general polynomial ring $k[Z]$, except that in every arithmetic operation on the polynomials in $R_q[Z]$,

- degrees higher or equal than q are reduced as $z^q \to z$.

- coefficients only take values in $\mathbb{F}_q$.

The ring $R_q[Z]$ is a quotient ring. Therefore the term ordering is not well defined since the relation $\deg(fg) = \deg(f) + \deg(g)$ is not always fulfilled.

Example 3.5 Let $f_1 = y^4$, $f_2 = y^3 \in R_7[y]$. Then $fg = y$ and therefore $\deg(fg) \neq 7$.

To deal with this problem formally, the Gröbner basis for an ideal I is computed in the *free* (not quotient) polynomial ring $\mathbb{F}_q[Z]$ with the relations $z_1^q - z_1, \ldots, z_n^q - z_n$ (see Definition 3.8) included in the set of generators for the ideal I.

Definition 3.9 The Gröbner basis of the ideal $I = \langle f_1, f_2, \ldots, f_n \rangle \in R_q[Z]$ for some given term ordering is denoted

$$GB_q(f_1, f_2, \ldots, f_n). \tag{3.41}$$

□

As explained above the set $\{z_1^q - z_1, \ldots, z_n^q - z_n\}$ is added to the generator polynomials to give a true Gröbner basis. This makes a significant difference when computing a Gröbner basis of an ideal defined by a single polynomial[2].

3.6 Linear Variables

When we compute Gröbner bases we get a set of polynomials relating the variables to each other. When we use this for control design we are particularly interested in whether a variable can be computed as a function of the other variables in the system description.

Example 3.6 Let the ideals $I_1, I_2 \subseteq R_3[y, x_1, x_2]$ have the varieties

$$\begin{aligned} V(I_1) &= \{(y, x_1, x_2) : (1,2,3), (2,3,1), (1,3,0)\} \\ V(I_2) &= \{(y, x_1, x_2) : (1,2,3), (2,3,1), (1,3,1)\} \end{aligned} \tag{3.42}$$

respectively. Then for each value of $(x_1, x_2) \in V(I_1)$ we have a unique value of y. Therefore y can be written as a function of x_1, x_2. This is not true for I_2 since $V(I_2)$ have two different values of y for the same value of (x_1, x_2).

[2] All ideals in $R_q[Z]$ can be generated by a single polynomial, i.e. $R_q[Z]$ is a *principal* polynomial ring. See [34].

Knowing that a variable can be computed as a function, we will use the Gröbner basis to find the corresponding function. This leads us to the definition of *linear variables.*

Definition 3.10 Given an ideal $I \subseteq R_q[y, x_1, \dots, x_n]$ and the variety $V(I)$, which is a set of elements of the form $(y, x_1, \dots, x_n)$, we say that the variable y is *linear* w.r.t. I, if for each value of the variables $x_1, \dots, x_n$ in $V(I)$ there exists only one value of y. $\square$

Lemma 3.2 Given an ideal $I \subseteq R_q[y_1, \dots, y_n, x_1, \dots, x_m]$ there exist polynomials $p_1, \dots, p_n \in R_q[x_1, \dots, x_m]$ such that

$$\{y_1 - p_1, \dots, y_n - p_n\} \subseteq I \tag{3.43}$$

iff $y_1, \dots, y_n$ are linear variables w.r.t. I. $\square$

Proof The values of the linear variables $y_1, \dots, y_n$ in $V(I)$ are given as a function $f : \mathbb{F}_q^m \to \mathbb{F}_q^n$ which can be represented, using (3.18), as

$$y_1 = p_1, \dots, y_n = p_n. \tag{3.44}$$

Since (3.44) gives no constraints on $x_1, \dots, x_n$ we have

$$\begin{aligned} V(\langle y_1 - p_1, \dots, y_n - p_n \rangle) \supseteq V(I) \Leftrightarrow \\ \langle y_1 - p_1, \dots, y_n - p_n \rangle \subseteq I \Leftrightarrow \\ \{y_1 - p_1, \dots, y_n - p_n\} \subseteq I. \end{aligned} \tag{3.45}$$

$\square$

Theorem 3.3 Given an ideal $I = \langle f_1, \dots, f_l \rangle \subseteq R_q[y_1, \dots, y_n, x_1, \dots, x_m]$. The variables $y_1, \dots, y_n$ are linear w.r.t I iff the autoreduced Gröbner basis G of I with lex-ordering $y > x$ has the form

$$G = \{y_1 - h_1, \dots, y_n - h_n, v_1, v_2, \dots\} \tag{3.46}$$

where the polynomials $h_i, v_j \in R_q[x_1, \dots, x_m]$. $\square$

Proof If the Gröbner basis G has the form (3.46) the claim follows directly from Definition 3.10.

Conversely, if $y_1, \dots, y_n$ are linear w.r.t. I, then by Lemma 3.2 we have

$$\{y_1 - p_1, \dots, y_n - p_n\} \subseteq I \tag{3.47}$$

for some p_i. Let $A = \{y_1 - p_1, \dots, y_n - p_n, f_1, \dots, f_l\}$. Then $I = \langle A \rangle$, and we can find a Gröbner basis for I by applying Buchberger's algorithm on A. After making A autoreduced we have

$$A = \{y_1 - \tilde{p}_1, \dots, y_n - \tilde{p}_n, \tilde{f}_1, \dots\} \tag{3.48}$$

where $\tilde{p}_i, \tilde{f}_j \in R_q[x_1, \ldots, x_m]$. For all s-polynomials computed from A we have

$$S(a_i, a_j) \longrightarrow_A r \in R_q[x_1, \ldots, x_m]. \tag{3.49}$$

Therefore no new polynomials containing the variable y_i can be included in the generator set produced by the algorithm. Knowing that an autoreduced Gröbner basis is unique and $G = GB_q(\langle f_1, \ldots, f_l \rangle) = GB_q(A)$, G has the form stated in (3.46). $\square$

3.7 Mixing Polynomials from Different Rings

Consider a polynomial $p(Z) \in R_q[Z]$ where a variable $z_i \in Z$ will not take values outside the interval $\{0, \ldots, r_i - 1\}$, where $r_i < q$. We say that for some values of z_i there are *"don't cares"*. This will help us to simplify[3] the polynomial $p(Z)$ by finding the simplest polynomial preserving the values of $p(Z)$ for all values of z_i that are not "don't care".

Let us compute a polynomial that is `true` only for those values of z_i that are not "don't care". We make the following definition.

Definition 3.11 Let $\lambda_q^{r_i}(z_i) \in \mathbb{F}_q[Z]$, $z_i \in Z$, $r_i < q$ be a polynomial such that

$$\lambda_q^{r_i}(z_i) = \begin{cases} \texttt{true} & , z_i = 0, \ldots, r_i - 1 \\ \texttt{false} & , z_i = r_i, \ldots, q - 1 \end{cases} \tag{3.50}$$

$\square$

As mentioned in Section 3.4 `true` is interpreted as 0, whereas `false` is interpreted as $\neq 0$.

How to use $\lambda_q^p(z_i)$ for simplifying $p(Z)$ is shown by an example.

Example 3.7 Let the variables $u, d \in \mathbb{F}_3$, be such that the variable u takes values only from $\{0, 1\}$. This means that $u = 2$ is a *"don't care"* value.

Let a polynomial represent the function given below.

d	u	$f(d,u)$
0	0	0
0	1	0
1	0	0
1	1	1
2	0	1
2	1	1

[3] Here simplify means to reduce the degree of the polynomial and to reduce the number of monomials.

One corresponding polynomial

$$f(d,u) \in R_3[d,u]$$

is

$$f(d,u) = d + 2d^2 + 2d^2u + 2du^2. \tag{3.51}$$

Since u is constrained to $\{0,1\}$ we have from (3.50)

$$\lambda_3^2(u) = u^2 - u \tag{3.52}$$

and the function $f(d,u)$ can be simplified as

$$f(d,u) \longrightarrow_{\mathcal{GB}_\lambda} d + 2d^2 + 2d^2u + 2du \tag{3.53}$$

where $\mathcal{GB}_\lambda = GB_3(\{\lambda_3^2(u)\})$

To collect all λ-polynomials corresponding to variables containing a "don't care" value we make the following definition.

Definition 3.12 Let Λ_q denote the set containing all $\lambda_q^{r_i}(z_i)$ corresponding to $z_i \in \{0,\dots,r_i-1\}$ where $r_i < q$. □

To compute the simplified polynomial, reduce $p(Z)$ w.r.t. $GB_q(\Lambda_q)$.

A polynomial defined as in Definition 3.12 can also be used to eliminate false solutions outside the valid domain. Given some relation $p(Z) \subset R_q[Z]$ and a set of λ-polynomials Λ_q we regard the ideal $\langle p(Z), \Lambda_q\rangle$ in order to obtain the correct set of solutions to $p(Z) = 0$.

Example 3.8 Consider the function from Example 3.7. Solving the equation

$$d + 2d^2 + 2d^2u + 2du^2 = 0 \tag{3.54}$$

we obtain the following set of solutions:

$$\{(d,u) : (0,0), (0,1), (0,2), (1,0), (1,2), (2,2)\}. \tag{3.55}$$

Since u is binary the solutions where $u = 2$ are obviously false. If we instead consider the ideal

$$I = \langle d + 2d^2 + 2d^2u + 2du^2, u^2 - u\rangle \tag{3.56}$$

we will get the variety

$$V(I) = \{(d,u) : (0,0), (0,1), (1,0)\} \tag{3.57}$$

which contains the correct solutions.

Chapter 4

Methods and Tools for DEDS

In this chapter we briefly describe the methods and software tools used in the modeling and analysis of the landing gear controller. We begin with some comments on the use of formal methods in industry today and the methods developed in [35]. We then focus on a technical description of the methods employed in this case study.

4.1 Current Methods and Tools

From a study of tools and procedures used in industry [36] we have concluded the following concerning *discrete* systems:

- The tools investigated in [36] can be categorized by their capabilities into: discrete system verification (NPCircuit), discrete modeling and simulation (ASA), high-level real-time programming and simulation (SAGA), hybrid systems modeling and simulation (SystemBuild and EASY5) and continuous systems modeling and simulation (Simulink and HOPSAN). In terms of analysis, these tools offer limited capabilities, i.e. only simulation or various types of static analysis of dynamic systems.

- The use of formal or symbolic methods is very limited, i.e. has not been integrated into the development process and the methods employed have very limited scope. This is of course strongly connected to the limited capabilities of the available tools.

In particular this means that there is a great potential for speeding up the development process as well as improving the quality of the end result through the use of symbolic methods.

The mathematics and algorithms developed to deal with discrete event dynamic systems are described in detail in [30, 31, 32, 33, 34, 35]. The methods generally fall into the same categories as we identified in Chapter 2 (modeling, analysis, design and implementation) and the symbolic representation and manipulation is based on polynomials over finite fields (see Chapter 3). Some of the crucial issues that have been addressed are:

- To have a framework that allows rigorous formulation of many analysis and design problems both for static and dynamic discrete systems.
- To have a framework that allows computational formulations of analysis and design problems that do scale to industrial size systems. These computations are symbolic (in most cases algebraic) as opposed to numeric and are generally *complete* in some sense.

The first item means that there is a solid foundation on which methods and algorithms are built. We essentially cover all finite state processes. The second item means that we can usually perform analysis and design for rather complex processes, i.e. processes having many system variables[1]. The second item also indicates that the computations usually also work as a proof, i.e. are complete. Some examples:

- If we have compiled a model from some description language (e.g. Pascal in this case study) into our polynomial framework, then the polynomial model has exactly the same behavior as the original description. Contrast this with a manual or semi-manual approach where we could only say that the *model* has a certain behavior since the manual procedure could have introduced a number of discrepancies.
- If we have verified that a behavior is impossible in a DEDS then it cannot occur. Contrast this with the result gained from extensive testing or simulation where we could only say that the undesired behavior is unlikely to occur, depending on the amount of testing and simulation.
- If we have computed that there is no controller that satisfies a given set of constraints, then there really exists none. Contrast this with a manual or heuristic approach where we could only say that the efforts so far has not produced a controller.

4.2 Modeling

By modeling we mean building a polynomial model of the system. In practical engineering work, several other model descriptions are used and we thus have to translate (or compile) from these descriptions to the form needed in order to do analysis.

[1] Equivalently: Processes having many possible inputs, states and outputs.

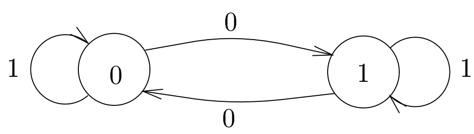
4.2.1 Polynomial Models

Polynomial models are used for computational purposes in Section 4.3 below and are of the form:

$$M(z, z^+) \tag{4.1}$$

where $z = [z_1, \ldots, z_l]$ are all the *system variables*, typically classified as *input*, *output* and *state* variables, but other groupings could be possible. Furthermore z^+ denotes the value of z one time instant into the future, i.e. if we use some index to indicate time we get $z(k)^+ = z(k+1)$. Finally, M is a polynomial in z and z^+ with coefficients from some finite field. In particular we will use the finite field $\mathbb{F}_2 = \{0, 1\}$ in this application. This means that the polynomials are essentially Boolean polynomials and that $z \in \mathbb{F}_2$. The main reason for modeling in $R_2[Z]$ is the application itself which contains a number of binary variables and that the (large scale) tools (see Section 4.4) we have developed so far operate on polynomials in $R_2[Z]$.

Example 4.1 Consider the simple dynamic system given by the graph:



This is a system that can be in either of the two states: 0 and 1. Furthermore in each (discrete) time instant the system changes state along one of the paths extending from the current state, depending on the value of the input (found on the arcs).

Let x_1 be a binary variable describing the state of the system ($x_1 = 0$ or $x_1 = 1$) and let u_1 be a binary variable describing the value of the input signal. If we let $z = [x_1, u_1]$ we can express the system relation as:

$$\begin{aligned} M(z, z^+) = {} & (x_1 - 0) \wedge (u_1 - 0) \wedge (x_1^+ - 1) \vee \\ & (x_1 - 0) \wedge (u_1 - 1) \wedge (x_1^+ - 0) \vee \\ & (x_1 - 1) \wedge (u_1 - 0) \wedge (x_1^+ - 0) \vee \\ & (x_1 - 1) \wedge (u_1 - 1) \wedge (x_1^+ - 1) \\ = {} & x_1^+ + 1 + x_1 + u_1 \end{aligned} \tag{4.2}$$

where $\vee$ denotes *or* and $\wedge$ denotes *and*, see Section 3.4 for the polynomial interpretation of these operators. The behavior of the system is given by the solutions to $M(z, z^+) = 0$. For example, if $x_1 = u_1 = 1$ we get $x_1^+ + 1 = 0$, which has the solution $x_1^+ = 1$.

When we have a simulation model or an actual implementation, the polynomial model typically is of the form:

$$x^+ = f(x,u), \quad y = g(x,u) \tag{4.3}$$

where $x = [x_1, \ldots, x_n]$ are the state variables, x^+ the state variables at the next time instant, $u = [u_1, \ldots, u_m]$ are the input signals and $y = [y_1, \ldots, y_p]$ are the output signals. The function $f(x,u) = [f_1(x,u), \ldots, f_n(x,u)]$ is the state transition function and $g(x,u) = [g_1(x,u), \ldots, g_p(x,u)]$ is the output function. This is just a special case of the model in equation (4.1) since if we let $z = [x; u; y]$ we get

$$M(z, z^+) = (x^+ = f(x,u)) \wedge (y = g(x,u)) \wedge (y^+ = g(x^+, u^+)). \tag{4.4}$$

The reason for including the relation $y^+ = g(x^+, u^+)$ is that $M(z, z^+)$ is a model of the system which contains variables representing two time steps. Static relations (relations defined in one time step) must of course hold in both time steps, which means that $y^+ = g(x^+, u^+)$ should be included in the model.

It is possible to prove that we can represent *any* finite state process with a polynomial model [34].

4.2.2 Model Description Languages

For most practical purposes, such as model input, model editing and model documentation, the representation in equation (4.1) is impractical. Some form of model description language should be used, possibly with accompanying graphical representation and graphical editor. There are a myriad of model description languages available and most tool vendors seem to have several proprietary versions just for good measure. There are however a couple that have been accepted as international standards: VHDL [49] for integrated circuits, SDL [18] for telecom applications and Grafcet [19] for sequential control. Even if these standards have been developed within standardization bodies for specific disciplines they are useful outside these disciplines.

If a process is described in terms of a model description language there should be a well defined behavior associated with that process, i.e. given input and initializations, it should be possible to generate unique outputs.[2] If the model is a finite state model, we should be able to translate this description to an equivalent polynomial model, in the sense that they have the same behavior. In practice it is usually both simpler and safer to write a general translation routine for the whole model description language or a

[2]This of course assumes explicit models, but all the mentioned cases conform to that.

subset thereof, i.e. a *compiler*[3]. The model compiler can then be seen as a function:

$$\phi : \text{Model Description Language} \rightarrow \text{Polynomial Model} \tag{4.5}$$

where the behavior is preserved. This is in effect what we have done in Chapter 6, where the model description language is Pascal and the polynomial model is a Boolean model. However, since a Pascal program can potentially have infinite state we have to restrict the compilation to a subset of Pascal. See Chapter 6 for more details.

4.3 Analysis – Verification

By analysis we mean that, given a polynomial model $M(z, z^+)$, we answer questions about the possible behavior of this model. There are two main types of analysis:

One Step Analysis where we look at a single time step of the system dynamics.

Multiple Step Analysis where we look at arbitrarily many time steps of the system dynamics.

It is important to note that the underlying algebra machinery that supports multiple step analysis has to deal with a far more complex situation than what is required in the one step analysis case.

We can use *temporal algebra* to formulate multiple step analysis questions (and, of course, as a special case of this, one step analysis questions).

4.3.1 One Step Analysis

By one step analysis we mean questions that can be resolved as equation systems of the form:

$$M(z, z^+) \wedge Q_1(z) \wedge Q_2(z^+) = 0 \tag{4.6}$$

where $M(z, z^+)$ is the process description and $Q_1(z)$ and $Q_2(z^+)$ are restrictions on z and z^+ respectively. The analysis consists of solving the system of equations or to prove that no such solution exists.

Example 4.2 Suppose we use the process model from Example 4.1 and want to know if there is a transition from 0 to 0? If we formulate this in algebraic terms we get

$$\begin{aligned} M(x, x^+) \wedge Q_1(x) \wedge Q_2(x^+) &= (x_1^+ + 1 + x_1 + u_1) \wedge x_1 \wedge x_1^+ \\ &= 1 + u_1 . \end{aligned} \tag{4.7}$$

[3]We will use compiler and translator interchangeably in the remainder of this document.

The solution to $1 + u_1 = 0$ is $u_1 = 1$, which means that there is a transition from 0 to 0 for the input $u_1 = 1$.

Unfortunately, solving Boolean equations is a fairly tough problem in general. In fact it is impossible to have a solution algorithm that behaves reasonably well on all polynomial equations. Partly, this is due to the potential size of the output, i.e. if we have polynomials in n variables over some finite field $\mathbb{F}_p$, there are p^n possible solutions, and in the Boolean case 2^n possible solutions. It is of course impossible to build an algorithm that works faster than it can output its answer. However, even if we formulate the restricted problem of only telling us whether or not there is a solution (without providing us with such a solution), the problem is in all likelihood almost as hard. Technically the solvability problem is known to be *NP complete*, see Hopcroft et. al. [47][4] for details.

The results quoted above states that if we are able to solve all possible equations in n variables, then the worst time complexity cannot be reasonable (polynomial in n). In practice one can deal with this problem by using methods that avoid the worst case complexity as often as possible. See Section 4.4 for details on how this is done.

4.3.2 Multiple Step Analysis

By multiple step analysis we mean analysis questions that take an arbitrary number of time steps of the system dynamics into account. An example is the set of states that are reachable in zero or arbitrarily many steps from some initial state. Given a process model $M(z, z^+)$ we can compute the set of states reachable in k steps or less from some initial set of states, described by $I(z) = 0$, as $R_k(z)$:

$$R_0(z) := I(z) \tag{4.8}$$

$$R_{k+1}(z) := R_k(z) \vee (\exists \tilde{z}\ (R_k(\tilde{z}) \wedge M(\tilde{z}, z))) \tag{4.9}$$

Since we are dealing with finite state systems this iteration will reach a fixed point, i.e. $R_{d+1}(z) = R_d(z)$ for some finite d. We will give some comments regarding this computation:

- The d for which we reach the fixed point above is the *depth* of the system. An interpretation of this is that in a maximum of d steps we can reach any reachable state in the system. In general, if we have n binary system variables ($z = [z_1, \ldots, z_n]$) then the maximal possible depth is 2^n. In most engineering applications the depth of a system seems to be far less than the maximal possible depth, but a simple process such as an n bit counter is one that has maximal depth. One

[4] Look for *satisfiability*.

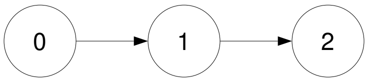
can also compare this to an n state linear system, where it is possible to reach any reachable state (from the origin) within n steps.

- In order to compute the set of reachable states (in arbitrarily many steps) we need to solve an equation after each iteration, i.e. when we check whether or not $R_k(z) = R_{k+1}(z)$. As mentioned, this problem is NP complete so we need to solve an NP complete problem in each iteration.

- We need to perform some form of data reduction in each step above, otherwise we will quickly overflow all available memory for all except trivial processes. This means that so called *proof theory* based methods (see e.g. [29]) as exhibited in NPCircuit [59] cannot be used in a multiple step setting. See Section 4.4 for how we have dealt with this problem.

- In theory we could compute the set of reachable states by using a simulation routine. However in practice this seems quite infeasible as we would have to keep track of all the reached states and then re-initiate the simulation from each of those states until we cannot reach new states any more. Even in moderately complex systems this is hardly feasible, and in the landing gear controller we have in the order of 10 000 reachable states out of 2^{26} potential reachable states. Hence running a few simulation scenarios usually says very little of the system behavior in general.

Example 4.3 Suppose we have the simple system below:



We can obtain a polynomial model by e.g. introducing two binary state variables x_1 and x_2 and encoding the states as follows

$\mathbf{x_1}$	$\mathbf{x_2}$	**State**
0	0	0
0	1	1
1	0	2

Using this encoding we get the polynomial model

$$M(x, x^+) = ((\neg x_1 \wedge \neg x_2) \wedge (\neg x_1^+ \wedge x_2^+)) \vee ((\neg x_1 \wedge x_2) \wedge (x_1^+ \wedge \neg x_2^+)) \tag{4.10}$$

We can now compute the set of reachable states from state 0 as

$$\begin{aligned}
R_0(x) &:= I(x) = \neg x_1 \wedge \neg x_2 \\
R_1(x) &:= ((\neg x_1) \wedge (\neg x_2)) \vee ((\neg x_1) \wedge x_2) \\
R_2(x) &:= ((\neg x_1) \wedge (\neg x_2)) \vee ((\neg x_1) \wedge x_2) \vee (x_1 \wedge (\neg x_2)) \\
R_3(x) &:= ((\neg x_1) \wedge (\neg x_2)) \vee ((\neg x_1) \wedge x_2) \vee (x_1 \wedge (\neg x_2))
\end{aligned} \tag{4.11}$$

We reach a fixed point for $k = 2$ steps, i.e. in two steps we can reach any reachable state. This can readily be seen from the graph above as well.

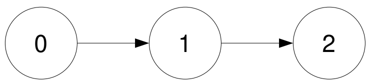
In this example we could not have found out that 2 is a reachable state by just one step analysis of $M(x, x^+)$ and the initial state information. In some cases this is important, since some undesirable action might be performed by the controller if it ever reaches state 2.

There are a multitude of other types of multiple step analysis that are possible and many of them are related to the idea of reachable states, either backward or forward in time.

4.3.3 Temporal Algebra and Verification

Since many specifications are written in something close to natural language, we could greatly simplify our analysis task if we could more or less directly translate this to a formal specification. In this application we have used *temporal algebra* [35] (or temporal logic since we use the binary Boolean algebra, see e.g. [20]) to achieve this task. In Table 4.1 some of the most common temporal algebra constructs are given. The expression "$Q(z)$ holds" should be interpreted as $Q(z) = 0$ in terms of polynomials. The *verification* is in general a multiple step analysis. For each temporal algebra expression $S(z)$ and process model $M(z, z^+)$ we compute the set of states from which the temporal algebra statement becomes true.

Example 4.4 Consider the process from example 4.3:



We wish to verify the specification:

"We should always be able to reach the safe state 2 as the next state."

In terms of temporal algebra this becomes:

$$\mathrm{EX}[x_1 \wedge \neg x_2]. \tag{4.12}$$

The verification is the computation

$$\begin{aligned}
\texttt{Verify}(M(x, x^+), \mathrm{EX}[x_1 \wedge \neg x_2]) &= \exists x^+ M(x, x^+) \wedge (x_1^+ \wedge \neg x_2^+) \\
&= (\neg x_1) \wedge x_2.
\end{aligned} \tag{4.13}$$

As expected this returns the state 1 in its encoded form, since this is the only state from which we can reach 2 in one step.

Suppose that we have the process *and* an initial state specified, then the above temporal algebra formula would be verified iff the returned set of states was a superset of the reachable states, i.e. we could reach 2 from every reachable state. For the system above this is clearly not the case if our initial state is 0, since the set of reachable states is $\{0, 1, 2\}$. Generally this extra level of reasoning is built into the verifier.

Temporal Algebra	**Natural Language**
$Q(z)$	$Q(z)$ holds in the initial state.
$\mathrm{EX}[Q(z)]$	$Q(z)$ can hold in the next time step.
$\mathrm{EU}[Q_1(z), Q_2(z)]$	$Q_1(z)$ will hold for finitely many steps and then $Q_2(z)$ can hold.
$\mathrm{EF}[Q(z)]$	$Q(z)$ can hold at some future time.
$\mathrm{EG}[Q(z)]$	$Q(z)$ can hold at all future times, i.e. from this point onwards.
$\mathrm{AX}[Q(z)]$	$Q(z)$ must hold in the next time step.
$\mathrm{AU}[Q_1(z), Q_2(z)]$	$Q_1(z)$ will hold for finitely many steps and then $Q_2(z)$ must hold.
$\mathrm{AF}[Q(z)]$	$Q(z)$ must hold at some future time.
$\mathrm{AG}[Q(z)]$	$Q(z)$ must hold at all future times, i.e. from this point onwards.

Table 4.1: Some of the most common temporal algebra constructs.

There are some remarks to be made regarding temporal algebra expressions and the verification of these:

- The constructs $P(z)$, $\mathrm{EX}[P(z)]$ and $\mathrm{AX}[P(z)]$ essentially denotes what we have termed one step analysis above.

- The remaining constructs enable multiple step analysis as seen from the user perspective. From the software tools perspective these constructs require the same type of fixed point computations as in the reachability analysis above.

- The temporal algebra statements can be *nested* with themselves as well as ordinary Boolean operations and thus provides great flexibility in expressing specifications.

Example 4.5 As an example of the flexibility of temporal algebra let us assume that we want to express the following specification:

"If A can happen then B must never happen."

Let $Q_A(z)$ be a polynomial describing the condition for the event A, and $Q_B(z)$ a polynomial describing the condition for the event B. We can then express the specification as

$$\mathrm{EF}[Q_A(z)] \rightarrow \mathrm{AG}[\neg Q_B(z)]. \tag{4.14}$$

For more details regarding temporal algebra (or temporal logic), see [20, 35].

4.4 Software Tools

In this section we will briefly describe the software tools used in the case study (see [35]for more details). The parts described in this document only constitute a small part of the whole software system.

The experimental system consists of *Mathematica* code [85] together with externally linked C code for critical operations through the MathLink structured communication protocol. *Mathematica* was chosen for its excellent programming model and plenty of numeric, symbolic and graphical algorithms well integrated into a consistent system.

4.4.1 Modeling

The modeling part follows the general outline provided in Section 4.2. In this case a new model compiler was written for the Pascal subset used in the controller implementation. This is described in Chapter 6.

4.4.2 Analysis

The analysis software makes use of an efficient implementation of Boolean algebra, known as *binary decision diagrams* (BDD), see [9, 13] for details[5]. Verification using the BDD representation together with temporal logic was initiated by McMillan in 1987 [62] and has lately been used for verification of large scale systems, see e.g. [21, 22]. Using an efficient symbolic algebraic computation engine is crucial to be able to analyze realistically sized examples. As pointed out earlier we cannot utilize a proof theory based computation engine if we want to perform multiple step analysis.

[5]Germundsson [35] has designed an implementation for general finite field polynomials (QDD), but the actual implementation was not available for this case study.

The basic idea used in binary decision diagrams is to rewrite Boolean expressions in a recursive form and reuse common subexpressions, a technique that has been used in compiler optimization for several decades. In the case of Boolean expressions this leads to highly efficient computations in most cases.

Suppose we have a Boolean expression $f(x_1, \ldots, x_n)$. We can then rewrite it using Shannon's expansion formula (see e.g. [28]):

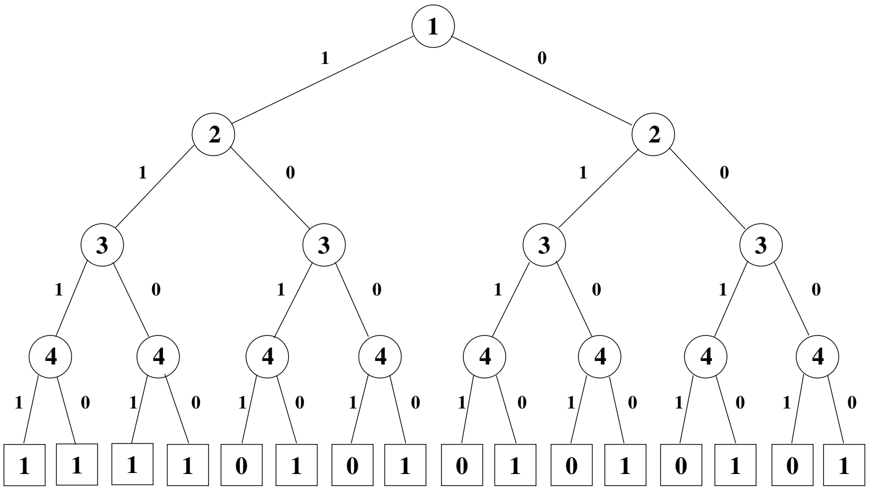
$$\begin{aligned} f(x_1, \ldots, x_n) =& ((\neg x_1) \wedge f(0, x_2, \ldots, x_n)) \\ & \vee (x_1 \wedge f(1, x_2, \ldots, x_n)). \end{aligned} \tag{4.15}$$

If we continue with this recursively for each of the functions $f(0, x_2, \ldots, x_n)$ and $f(1, x_2, \ldots, x_n)$ w.r.t. x_2 and then x_3 etc., we obtain

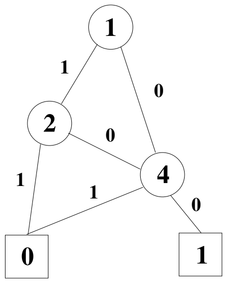
$$\begin{aligned} f(x_1, \ldots, x_n) = \neg x_1 \wedge & \underbrace{(\cdots (\underbrace{(\neg x_n \wedge \alpha)}_{g^{n-1}_{0,\ldots,0}(x_n)} \vee \underbrace{(x_n \wedge \beta)}_{g^{n-1}_{0,\ldots,1}(x_n)}))}_{g^1_0(x_2,\ldots,x_n)} \\ & \vee \\ x_1 \wedge & \underbrace{(\cdots (\underbrace{(\neg x_n \wedge \gamma)}_{g^{n-1}_{1,\ldots,0}(x_n)} \vee \underbrace{(x_n \wedge \delta)}_{g^{n-1}_{1,\ldots,1}(x_n)}))}_{g^1_1(x_2,\ldots,x_n)} \end{aligned} \tag{4.16}$$

where $\alpha, \beta, \gamma, \delta \in \{0, 1\}$. Furthermore we see that we obtain several subexpressions with progressively fewer variables. In fact all expressions g^i above are Boolean expressions in the variables $\{x_{i+1}, \ldots, x_n\}$. In case some of these expressions are equal we should not have to repeat this part more than once but instead substitute a reference to this common subexpression. The recursive Boolean expression form above can be visualized as a binary tree, where each node corresponds to the $\vee$ operator and essentially the g^i expressions as subtrees. This is the basis of the name BDD. The number of remaining nodes (or equivalently the number of *different* subexpressions) is a measure of the complexity of the given expression. This will be referred to as the number of nodes in Chapter 6. By changing the order in which we expand w.r.t. the various variables we usually get wildly different node counts. The ordering is termed *variable ordering* and plays a significant role in lowering the representational complexity of the landing gear system in Chapter 6.

Example 4.6 The Boolean function $f = x_1 x_2 + \neg x_4$ is represented by the following ordered but unreduced BDD if we use the variable ordering $x_1 > x_2 > x_3 > x_4$.



where the number in the nodes correspond to the index of the variables. After reduction we get



Chapter 5

The Landing Gear Process

The case study in Chapters 6 and 7 concerns the landing gear system on the Swedish fighter JAS 39 Gripen, depicted in Figure 5.1.

The landing gear system consists of a landing gear controller and three landing gears with corresponding doors. A simplified block description of the complete system is shown in Figure 5.2, where the arrows should be interpreted as signal vectors.

The objective of the study is to apply methods for formal verification of function specifications to the landing gear controller.

5.1 Description of the System

The landing gear controller on JAS 39 is a DEDS which is implemented in Pascal86 (an Intel version of Pascal). It consists of a module of code which is mainly self-contained but in some aspects communicates with the rest of the system code. The controller code consists of approximately 1500 lines, of which about 300 handles alarm functions and pilot information.

Besides the fact that the controller itself is discrete, the domains of all actuator and measurement signals are discrete. However, the underlying system is continuous and to fully understand the expected behavior of the controller, we need to know how the gears work.

Maneuvering of gears and doors is commanded electrically but actuated hydraulically. There is one hydraulic actuator for each gear as well as one actuator for each door. However, all gears are operated in parallel, as are the doors, i.e. it is not possible to close one door while keeping the other two open. In addition to the valves controlling gears and doors, there is a valve that shuts off hydraulic pressure in the system during flight with retracted gears.

There are basically three maneuver types, retraction, extension and emergency extension. Ordinary extension or retraction is commanded by a lever in the cockpit. During emergency extension the control signals are

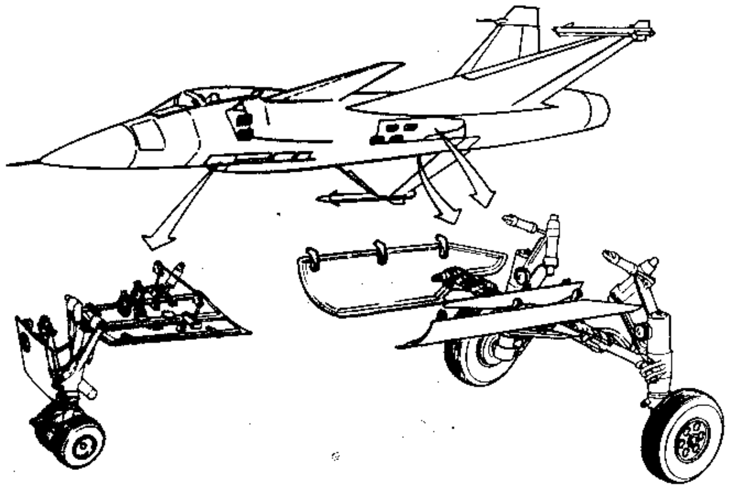


Figure 5.1: The fighter JAS 39 Gripen.

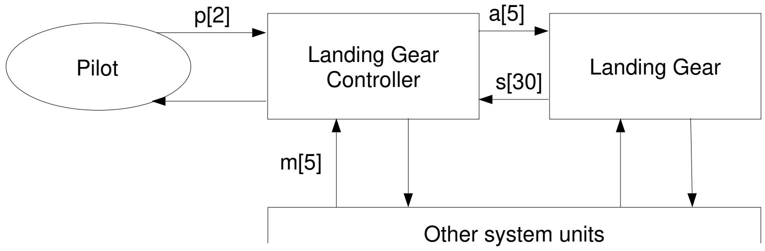


Figure 5.2: The landing gear system (number of signals in brackets).

generated by hardware logic and not by the landing gear controller. However, when emergency extension is initiated a signal is transmitted to the landing gear controller via other units in the aircraft. In emergency extension mode hydraulic power is not used to lower the rear gears. Instead they are extended by the air drag.

The feedback from the gears to the controller is generated by microswitches positioned on gears and doors. By these switches it is possible to detect certain discrete positions of the doors and gears, e.g. doors open, doors closed, gears retracted, gears extended, etc. This means that we cannot monitor the continuously changing position of the gears under e.g. extension, but only the discrete positions detected by the switches. Hence, the interface between the gears and the controller is discrete, which means that the models of the gears, which we will need for multiple step analysis, can be made discrete. The degree of refinement in the gear models will reflect the desired fidelity.

Signal Name	Short Name	Description
start_landn_panel.ut1	p_1	Extend gears (switch 1)
start_landn_panel.ut2	p_2	Extend gears (switch 2)

Table 5.1: Pilot input: Commands given by the pilot to control the landing gear.

Signal Name	Short Name	Description
utport_dd00[o_avst_vnt]	a_1	Hydraulic pressure on
utport_dd00[fall_in]	a_2	Retract gears
utport_dd00[fll_ut_st]	a_3	Extend gears
utport_dd00[s_lndst_lck]	a_4	Close doors
utport_dd00[o_lndst_lck]	a_5	Open doors

Table 5.2: Actuator signals: Commands to the hydraulic system.

5.2 The Signal Interface

As shown in Figure 5.2 there are basically three kinds of input to the landing gear controller: pilot commands, information from other units in the system and feedback from gears and doors.

The pilot command is detected by a double microswitch positioned on the extension lever. The system information consists of input from various other parts of the aircraft, e.g. power supply, hydraulic supply and motor. As mentioned, the gear feedback comes from microswitches on gears and doors. There are three microswitches on each gear and two on each door and each of the microswitches has two contacts. This makes a total of 30 binary signals from the gears and doors.

A detailed description of all input signals can be found in Tables 5.1, 5.3 and 5.4. All signals except m_3 and m_4 are binary. Note that only the signals that will be included in our model of the landing gear controller have been given a short name.

The output from the controller consists of actuator signals to the hydraulic valves, information signals to other aircraft units and signals used for status presentation in the cockpit. There are five (binary) signals to the actuators which are presented in Table 5.2. The other output signals will not be included in our model and are therefore not described in detail.

Signal Name	Short Name	Description
nodutf_stall	m_1	Emergency extension commanded
motorn_gaur	m_2	Motor is running
fpl_tillst	m_3	Aircraft status
mark_tillst	m_4	Aircraft status
plus28v_kritblk	m_5	An element in the variable so_info.fo.power that signals power down in AFPL
adc_t_afpl8	-	
ess_t_afpl30.fsw.mode	-	
hyd_larm_f	-	
paudrag	-	
subfas_inom_1hz	-	
update_cnt_ok	-	

Table 5.3: System state: Signals from other units in the aircraft.

Signal Name	Short Name	Description
nosstall_inne	s_1^1, s_1^2	Nose gear retracted
nosstall_ute	s_2^1, s_2^2	Nose gear extracted
nosstall_infj	s_3^1, s_3^2	Weight on nose gear
nosstall_lucka_stangd	s_4^1, s_4^2	Nose door closed
nosstall_lucka_oppen	s_5^1, s_5^2	Nose door open
hstall_h_inne	s_6^1, s_6^2	Right gear retracted
hstall_h_ute	s_7^1, s_7^2	Right gear extracted
hstall_h_infj	s_8^1, s_8^2	Weight on right gear
hstall_lucka_h_stangd	s_9^1, s_9^2	Right door closed
hstall_lucka_h_oppen	s_{10}^1, s_{10}^2	Right door open
hstall_v_inne	s_{11}^1, s_{11}^2	Left gear retracted
hstall_v_ute	s_{12}^1, s_{12}^2	Left gear extracted
hstall_v_infj	s_{13}^1, s_{13}^2	Weight on left gear
hstall_lucka_v_stangd	s_{14}^1, s_{14}^2	Left door closed
hstall_lucka_v_oppen	s_{15}^1, s_{15}^2	Left door open

Table 5.4: Landing gear feedback: Output from microswitches on gears and doors.

Chapter 6

Modeling Based on the Controller Implementation

This chapter describes modeling based on the implemented Pascal code of the landing gear controller. As a first effort a model based on partial documentation of the landing gear controller was constructed. The partial documentation represented a good opportunity to fine tune the software used in the verification process. In fact, many packages were written and given their first real life test in this initial phase. This effort also showed some of the pitfalls using incomplete models. For example, the analysis results was not in correspondence with results obtained when using the implemented code for modeling.

In the case study presented below the implemented code is used for the model because the representation of the controller must be in exact correspondence with the implementation to ensure reliable verification results. In this case the only available representation of the implemented system is the code itself. Hence we have to use a compiler for the translation from the implemented code to the polynomial representation.

As will be shown below we perform the modeling by translating the code to Boolean algebra, which is efficiently represented by binary decision diagrams (see Section 4.4).

6.1 Restrictions in the Modeling

The modeling has been done using a restricted class of Pascal86. This section describes the restriction as well as the minor adjustments that we must make in order to fit the controller code into the finite state domain.

The allowed data types are integer and boolean. The integer range used is $\{0, \ldots, 15\}$, which is enough to represent all enumerable variables in the controller code. The controller code also makes use of linear arrays and abstract data types. It is possible to automatically represent these data

Comment	Syntax	Domains
Type	`BOOLEAN`	$\mathbb{B}$
	`INTEGER`	$\mathbb{I}$
Arithmetic expr.	`+`	$\mathbb{I}^2 \to \mathbb{I}$
	`-`	$\mathbb{I}^2 \to \mathbb{I}$
Relational expr.	`>`	$\mathbb{I}^2 \to \mathbb{B}$
	`<`	$\mathbb{I}^2 \to \mathbb{B}$
	`=`	$\mathbb{I}^2 \to \mathbb{B}$
	`NOT`	$\mathbb{B} \to \mathbb{B}$
	`AND`	$\mathbb{B}^2 \to \mathbb{B}$
	`OR`	$\mathbb{B}^2 \to \mathbb{B}$
Control	`IF THEN ELSE`	
	`CASE OF`	
	`BEGIN ... END`	
Miscellaneous	`:=`	$\mathbb{I}$ or $\mathbb{B}$
	`VAR`	$\mathbb{I}$ or $\mathbb{B}$
	`PROGRAM`	
	`PROCEDURE`	
	`FUNCTION`	

Table 6.1: Allowed Pascal primitives. $\mathbb{I}$ and $\mathbb{B}$ stands for integer and Boolean respectively.

types by integers and booleans, but in this case it has been done by hand.

Some of the Pascal primitives have also been excluded. For a list of allowed primitives, see Table 6.1. For code primitives such as `FOR`-loops and the `OTHERWISE` statement in conditionals, a manual translation was made where the `FOR`-loop was rewritten as a sequence of code (loop unrolling, see e.g. [1]) and `OTHERWISE` replaced by explicit arguments.

Timer variables and time conditions in the code have been replaced by binary state variables (flip flops) and corresponding input signals. A time condition becoming true in the original code corresponds to the timer input signal triggering the state variable. Once triggered, the state variable will be true until there is an explicit timer reset.

From the code module we have excluded the procedures concerning alarm handling and pilot information since they do not affect the other procedures in the controller directly. Since we have not had access to all values of signals from other units in the aircraft we have also defined some new input signals that are aggregations of the unknown signals.

6.2 Translating Pascal to Polynomial Relations

The translation from Pascal to a polynomial representation of the code is performed in two steps. First we parse the Pascal code to an intermediate representation in *Mathematica*, which we call MPascal. This *Mathematica* representation of the code is then automatically compiled into a representation in Boolean algebra. One formal way of describing the steps of parsing and compilation is to use structured operational semantics [72] and we will therefore begin with an introduction to this formalism. We will then describe the parsing and compilation of the landing gear controller code in some detail and comment on the resulting polynomial model.

6.2.1 Structured Operational Semantics

Structured operational semantics (SOS) is a way of describing the semantics of programming languages. This is done by combining the notion of *transition sequences* and *proof construction*. This essentially means that SOS consists of a set of transitions where each transition describes a semantical mapping and where a transition can be used only if its necessary conditions are proven to be true. The standard reference on SOS is [72]. A related formalism is *natural semantics* which is described in e.g. [50]. These and other formalisms are used alternately and therefore the notation has not fully converged. We will in the following use a notation based on SOS and natural semantics. In order to adapt the formalism to our application we will make our own definitions, even though some of them differ only marginally from the standard notation within the field.

With structured operational semantics we can express derivation rules for how a piece of code changes the state of the compiler. The state of the compiler is essentially a symbol table containing the present values of the symbols used in the code. When the whole code is processed we use the final state of the compiler to compute the polynomial representing the code.

The state of the compiler is described by a *binding environment*, σ. A binding environment is a partial function from identifiers (or variables) to values. We will write a binding environment as a set

$$\sigma = \{x_i \mapsto \eta_i\}, \quad i = 0, \dots n \tag{6.1}$$

where x_i is an identifier and η_i is the corresponding value.

We want to perform operations on the binding environments. For example we want to evaluate values of identifiers in a binding environment and alter these values. These operations are defined as:

Definition 6.1 Let σ be a binding environment containing the identifier x_i. Then

(i) $[\![\sigma(x_i)]\!]$ gives the value of x_i.

(ii) $\sigma[x_i \mapsto \eta_i]$ sets the value of x_i to η_i while all other variables in σ remains unchanged.

□

We introduce a new variable in the binding environment by setting its value. To simplify the notation we use $\sigma[\{x_1, x_2, \dots\} \mapsto \{\eta_1, \eta_2, \dots\}]$ for setting the value of several variables in σ. We illustrate the defined operations with an example.

Example 6.1 Let $\sigma_1 = \{x \mapsto 1, y \mapsto 2\}$. We can then evaluate x by

$$[\![\sigma_1(x)]\!] = 1$$

If we want to change the value of y to 3 we write

$$\sigma_1[y \mapsto 3]$$

This means that σ_1 is changed to the set $\{x \mapsto 1, y \mapsto 3\}$. By writing

$$\sigma_1[\{y, z\} \mapsto \{4, 5\}]$$

we alter the value of y and add a new identifier z to the binding environment. The value of σ_1 will now be $\{x \mapsto 1, y \mapsto 4, z \mapsto 5\}$.

Since we in the compilation represent all values in σ as Boolean polynomials, it will be useful to define the logical operations $\{\wedge, \vee, \neg\}$ on σ (see Section 3.4 for a polynomial interpretation of the logical connectives).

Definition 6.2 Let σ_1, σ_2 be two binding environments containing the *same* set of identifiers and b an arbitrary Boolean expression.

- $\sigma = \sigma_1 \ op\ \sigma_2$ is defined by

$$[\![\sigma(x_i)]\!] = [\![\sigma_1(x_i)]\!] \ op\ [\![\sigma_2(x_i)]\!], \quad i = 0, \dots, n$$

  for all $op \in \{\wedge, \vee\}$.

- $\sigma = \neg\sigma$ is defined by

$$[\![\sigma(x_i)]\!] = \neg[\![\sigma(x_i)]\!], \quad i = 0, \dots, n.$$

- $\sigma = \sigma_1 \ op\ b$ is defined by

$$[\![\sigma(x_i)]\!] = [\![\sigma_1(x_i)]\!] \ op\ b, \quad i = 0, \dots, n$$

  for all $op \in \{\wedge, \vee\}$.

□

Example 6.2 Let $\sigma_1 = \{x \mapsto a, y \mapsto b\}$, $\sigma_2 = \{x \mapsto c, y \mapsto d\}$ and $g \in \mathbb{B}$. Then

$$\begin{aligned} \sigma_1 \wedge \sigma_2 &= \{x \mapsto a \wedge c, y \mapsto b \wedge d\} \\ \sigma_1 \vee \sigma_2 &= \{x \mapsto a \vee c, y \mapsto b \vee d\} \\ \neg\sigma_1 &= \{x \mapsto \neg a, y \mapsto \neg b\} \\ \sigma_1 \wedge g &= \{x \mapsto a \wedge g, y \mapsto b \wedge g\} \end{aligned} \tag{6.2}$$

We also need to define a special kind of intersection between two binding environments.

Definition 6.3 Let σ_1, σ_2 be two binding environments, not necessarily containing the same set of identifiers. Then $\sigma = \cap_{\sigma_2}\sigma_1$ is defined by

$$[\![\sigma(x_i)]\!] = [\![\sigma_1(x_i)]\!]$$

for all x_i contained in both σ_1 and σ_2. □

Example 6.3 Let $\sigma_1 = \{x \mapsto a, y \mapsto b, z \mapsto c\}$ and $\sigma_2 = \{x \mapsto d, y \mapsto e\}$. Then

$$\cap_{\sigma_2}\sigma_1 = \{x \mapsto a, y \mapsto b\}. \tag{6.3}$$

Let a sequence of code primitives be denoted L. We restrict L to contain the primitives from Table 6.1 together with the empty string ϵ. With L and σ as basic elements we can now make the following definitions.

Definition 6.4 A *configuration* θ is a tuple (L, σ). □

Definition 6.5 A *transition* $\vdash$ is a mapping from a configuration $\theta = (L, \sigma)$ to a configuration θ' or to a binding environment σ',

$$\theta \ \vdash \ \theta' \quad \text{or} \quad \theta \ \vdash \ \sigma'. \tag{6.4}$$

□

Definition 6.6 A *transition relation* τ is a rule of the form

$$\tau = \frac{\mathcal{C}_1 \ \ldots \ \mathcal{C}_n}{\theta \vdash \theta'} \tag{6.5}$$

where the numerator is a set of conditions which are premises for the denominator. The interpretation of the notation in (6.5) is that θ' can be derived from θ if all conditions $\mathcal{C}_i$ $(1 \leq i \leq n)$ hold. □

For most of the transition relations presented in this thesis the conditions will be transitions but there are also other types of conditions. We illustrate the different types of conditions with an example.

Example 6.4

(i) The condition

$$\mathcal{C}_1 = \quad \theta \vdash \theta' \tag{6.6}$$

means that it must be possible to derive θ' from θ.

(ii) The condition

$$\mathcal{C}_2 = \quad N \in \mathbb{Z} \tag{6.7}$$

means that the value of N must be an integer.

(iii) The condition

$$\mathcal{C}_3 = \quad t = \texttt{MB} \tag{6.8}$$

means that t must equal the symbol `MB`.

Remark 6.1

(i) For transitions where the binding environment is obvious, we will write

$$L \ \vdash \ L' \tag{6.9}$$

to simplify the notation.

(ii) Unconditional transition relations will be written as

$$\tau = \quad \theta \ \vdash \ \theta'. \tag{6.10}$$

□

6.2.2 Input, Output and State Variables

The landing gear controller code is one part of the software loop in the aircraft system. This means that the state of the code is stored until next iteration of the code. If we want to write the system as in (4.3) we must determine which variables correspond to system state, next state, input and output. The equations in (4.3) can be represented by a block diagram as in Figure 6.1, where u, y and x are vectors for input, output and state variables respectively. Any part of the code (a single primitive or a complete program) can also be regarded as a function which computes and assigns

Variable Name	Short Name	Domain	Description
utf_pagar	x_1	$\mathbb{F}_2$	Extension maneuver
tid_latt_lte_block_tid_latt	x_2	$\mathbb{F}_2$	Timer condition
spin_start_lte_spin_tid	x_3	$\mathbb{F}_2$	Timer condition
utf_start_lte_block_tid_inf	x_4	$\mathbb{F}_2$	Timer condition
utf_start_gt_luck_oppn_tid	x_5	$\mathbb{F}_2$	Timer condition
init_flag	x_6	$\mathbb{F}_2$	Initialization flag
utf_mod	x_7	$\mathbb{F}_{11}$	Extension mode
inf_mod	x_8	$\mathbb{F}_8$	Retraction mode
man_akt_mod	x_9	$\mathbb{F}_6$	Maneuver action mode
fj_mod	x_{10}	$\mathbb{F}_3$	Take-off mode
man_komm_mod	x_{11}	$\mathbb{F}_8$	Maneuver command mode

Table 6.2: State variables in the code.

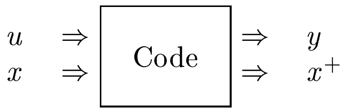


Figure 6.1: Signal interaction with the code.

values to output variables depending on input variables, see Figure 6.2. If we compare Figures 6.1 and 6.2, we find that the state variables are equal to the variables in the set $Inputs \cap Outputs$. In words we say that if there is an input variable which is reassigned in the code, it should be considered as a state variable. The state variables in the model of the controller are described in Table 6.2.

Temporary variables which are neither input nor output variables can be omitted in a model like the one in Figure 6.1, since we are only interested in the output behavior of the system. Still, the compiler must use the

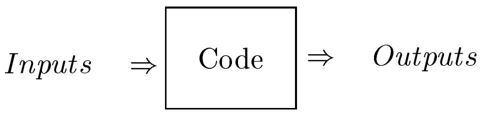


Figure 6.2: Input-output view of the code.

temporary variables in the code to compute the relations between input and output variables.

6.2.3 Parsing Pascal to MPascal

The first step in the translation is the parsing from Pascal to MPascal. The parsing is in this case straightforward and has been done by hand, but could of course be automated. MPascal represents the restriction of Pascal86 mentioned in Section 6.1.

Before we define the semantics of the parser formally, we give a simple example of the parsing of a piece of code.

Example 6.5 An `IF-THEN-ELSE` statement will be parsed to MPascal according to the figure below, where a, b, c, d and e are Pascal expressions.

```
IF a=b THEN
   y1 := c
ELSE
   BEGIN
       y1 := d
       y2 := e
   END;
```

$\rightsquigarrow$

```
MIfThen[
        MEqual[a,b],
        MAssign[y1,c],
        MBeginEnd[
                  MAssign[y1,d],
                  MAssign[y2,e]
                  ]
        ]
```

The parsing would in this case make use of the transition relations (6.18), (6.22), (6.25) and (6.31), defined below.

We are now ready to define the parser for the keywords of Table 6.1. In the parser we will not need the binding environment, which means that all transitions will be of the form (6.9). The parser consists of the following transition rules:

Constants

$$\texttt{TRUE} \vdash \texttt{MTrue}[] \tag{6.11}$$

$$\texttt{FALSE} \vdash \texttt{MFalse}[] \tag{6.12}$$

$$\frac{N \in \mathbb{Z}}{N \vdash N} \tag{6.13}$$

Arithmetic expressions

`+`

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 \texttt{+} e_2 \ \vdash \ \texttt{MPlus}[e_1', e_2']} \tag{6.14}$$

`-`

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 \texttt{-} e_2 \ \vdash \ \texttt{MMinus}[e_1', e_2']} \tag{6.15}$$

Relational expressions

`>`

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 \texttt{>} e_2 \ \vdash \ \texttt{MGreaterThan}[e_1', e_2']} \tag{6.16}$$

`<`

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 \texttt{<} e_2 \ \vdash \ \texttt{MLessThan}[e_1', e_2']} \tag{6.17}$$

`=`

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 \texttt{=} e_2 \ \vdash \ \texttt{MEqual}[e_1', e_2']} \tag{6.18}$$

`NOT`

$$\frac{e_1 \vdash e_1'}{\texttt{NOT}\, e_1 \ \vdash \ \texttt{MNot}[e_1']} \tag{6.19}$$

`AND`

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 \ \texttt{AND}\, e_2 \ \vdash \ \texttt{MAnd}[e_1', e_2']} \tag{6.20}$$

`OR`

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 \ \texttt{OR}\, e_2 \ \vdash \ \texttt{MOr}[e_1', e_2']} \tag{6.21}$$

Control primitives

`IF THEN ELSE`

$$\frac{c \vdash c' \quad s_1 \vdash s_1' \quad s_2 \vdash s_2'}{\texttt{IF}\, c\, \texttt{THEN}\, s_1 \texttt{ELSE}\, s_2 \ \vdash \ \texttt{MIfThen}[c', s_1', s_2']} \tag{6.22}$$

$$\frac{c \vdash c' \quad s_1 \vdash s_1'}{\texttt{IF}\, c\, \texttt{THEN}\, s_1 \ \vdash \ \texttt{MIfThen}[c', s_1', \epsilon]} \tag{6.23}$$

`CASE OF`

$$\frac{e \vdash e' \quad k_1 \vdash k_1' \;\ldots\; k_n \vdash k_n' \quad s_1 \vdash s_1' \;\ldots\; s_n \vdash s_n'}{\texttt{CASE}\; e \;\texttt{OF}\; k_1 : s_1 \,;\, \ldots \,;\, k_n : s_n \;\texttt{END} \;\vdash\; L} \tag{6.24}$$

where

$$L = \texttt{MCaseOf}[e', k_1', s_1', \ldots, k_n', s_n']$$

`BEGIN` s_1`;` s_2 `END`

$$\frac{s \vdash s'}{\texttt{BEGIN}\; s \;\texttt{END} \;\vdash\; \texttt{MBeginEnd}[s']} \tag{6.25}$$

$$\frac{s_1 \vdash s_1' \quad s_2 \vdash s_2'}{s_1 \;\texttt{;}\; s_2 \;\vdash\; s_1' \;\texttt{,}\; s_2'} \tag{6.26}$$

Procedures and functions

Declaration:

$$\frac{args \vdash args' \quad s \vdash s'}{\texttt{PROCEDURE}\; id(args)\; s \;\vdash\; \texttt{MProcedure}[id, \{args'\}, s']} \tag{6.27}$$

$$\frac{args \vdash args' \quad s \vdash s' \quad e \vdash e'}{\texttt{FUNCTION}\; id(args){:}t \;\texttt{BEGIN}\; s\texttt{;}\; id\,\texttt{:=}e \;\texttt{END} \;\vdash\; L} \tag{6.28}$$

where

$$L = \texttt{MFunction}[id, \{args'\}, \texttt{MBeginEnd}[s', \texttt{MReturn}[e']]]$$

Arguments of procedures and functions:

$$v_1, \ldots, v_k : t_1 \,;\, v_{k+1}, \ldots, v_n : t_2 \,;\, \ldots \;\vdash \\ v_1, \ldots, v_k, v_{k+1}, \ldots, v_n \tag{6.29}$$

Procedure and function call:

$$\frac{e_1 \vdash e_1' \;\ldots\; e_n \vdash e_n'}{id(e_1, \ldots, e_n) \;\vdash\; id[e_1', \ldots, e_n']} \tag{6.30}$$

Miscellaneous code primitives

`:=`

$$\frac{e \vdash e'}{v \;\texttt{:=}\; e \;\vdash\; \texttt{MAssign}[v, e']} \tag{6.31}$$

`VAR`

$$\frac{t_1 \vdash t_1' \quad t_2 \vdash t_2'}{\texttt{VAR}\, v_1, \ldots, v_k : t_1 \,\texttt{;}\, v_{k+1}, \ldots, v_n : t_2 \,\texttt{;}\, \ldots \;\vdash\; L} \tag{6.32}$$

where

$$L = \texttt{MVar}[\{t_1', v_1, \ldots, v_k\}, \{t_2', v_{k+1}, \ldots, v_n\}, \ldots]$$

$$\texttt{BOOLEAN} \;\vdash\; \texttt{MB} \tag{6.33}$$

$$\texttt{INTEGER} \;\vdash\; \texttt{MI} \tag{6.34}$$

`PROGRAM`

$$\frac{s \vdash s'}{\texttt{PROGRAM}\, s \;\vdash \texttt{MProgram}[s']} \tag{6.35}$$

By these transition rules we have defined the parsing step of the compilation of the restricted Pascal language defined in Table 6.1. To separate input and output variables in the code (see Section 6.2.2), the keywords `MInput` and `MOutput` are added to the MPascal language. These keywords are used with the same syntax as for `MVar`.

Input variables:

$$\frac{t_1 \vdash t_1' \quad t_2 \vdash t_2'}{\texttt{VAR}\, u_1, \ldots, u_k : t_1 \,\texttt{;}\, u_{k+1}, \ldots, u_n : t_2 \,\texttt{;}\, \ldots \;\vdash\; L} \tag{6.36}$$

where

$$L = \texttt{MInput}[\{t_1', u_1, \ldots, u_k\}, \{t_2', u_{k+1}, \ldots, u_n\}, \ldots]$$

Output variables:

$$\frac{t_1 \vdash t_1' \quad t_2 \vdash t_2'}{\texttt{VAR}\, y_1, \ldots, y_k : t_1 \,\texttt{;}\, y_{k+1}, \ldots, y_n : t_2 \,\texttt{;}\, \ldots \;\vdash\; L} \tag{6.37}$$

where

$$L = \texttt{MOutput}[\{t_1', y_1, \ldots, y_k\}, \{t_2', y_{k+1}, \ldots, y_n\}, \ldots]$$

Note that the state variables will be declared both as input and output variables. The second step in the compilation will treat variables declared with `MInput`, `MOutput` and `MVar` differently, see Section 6.2.4.

6.2.4 Compiling MPascal to a Polynomial Relation

In the second stage we translate MPascal to a Boolean relation $C(z, z^+)$. As mentioned earlier, any part of the code (a single primitive or a complete program) is essentially a function which computes and assigns values to output variables depending on input variables. Compiling these functions from MPascal to a Boolean relation can at least be made in two different ways.

One way is to first translate all primitives (the smallest parts) of the code into relations, and then combine these into larger relations. This would be a straightforward method which is easy to implement recursively. The disadvantage is that the combination of relations requires a substantial amount of substitutions and quantifications, which are time consuming operations in the present implementation of the software tools.

We have instead chosen to view the translation from the data perspective. First we separate all but the temporary variables into inputs $\mathcal{U}$ and outputs $\mathcal{Y}$. Then we go through the code, statement by statement, following the execution sequence[1] of the code. For each statement we store information of how the values of the output variables

$$\mathcal{Y} = [\tilde{y}_1, \dots \tilde{y}_{p+n}] = [y_1, \dots, y_p, x_1^+, \dots, x_n^+] \tag{6.38}$$

are affected by the input variables[2]

$$\mathcal{U} = [\tilde{u}_1, \dots, \tilde{u}_{m+n}] = [u_1, \dots, u_m, x_1, \dots, x_n]. \tag{6.39}$$

This information, which corresponds to a function $f_{\tilde{y}_i} : \mathcal{U} \to \tilde{y}_i$, is stored as a Boolean expression for each $\tilde{y}_i$. In each step the expressions $f_{\tilde{y}_i}$ is updated until we reach the final statement when $f_{\tilde{y}_i}$ is a complete description of how the output variable $\tilde{y}_i$ depends on the input variables $\mathcal{U}$.

At the end all these expressions together with the output variable symbols are combined into a final relation, which will be used for analysis.

Since we model with Boolean polynomials and the compiler must handle integers (in this case in the range $0, \dots, 15$) we will use a binary vector representation for the integers. We make the following definition

Definition 6.7 The vector space of *bit-represented integers*, denoted $\bar{\mathbb{I}}_n$, is

$$\bar{\mathbb{I}}_n = \{\{e_{n-1}, \dots, e_0\} | e_i \in \mathbb{B}[Z], \quad 0 \le i < n-1\} \tag{6.40}$$

where $\mathbb{B}[Z]$ is the set of all Boolean expressions. The position of e_0 is least significant. The elements of $\bar{\mathbb{I}}_n$ represent all integers from 0 to $2^n - 1$. □

[1] This is essentially the same as traversing a corresponding control flow graph, see e.g. [4].

[2] This corresponds to a data flow analysis for each output variable, see e.g. [4].

Consider the following example

Example 6.6 Let $n = 4$ and assume that the integer output variable y depends on the Boolean input variables u_1 and u_2. This dependence could for example be as in the following element from $\bar{\mathbb{I}}_4$:

$$\{\mathtt{false}, u_1, \neg u_2, u_1 \vee u_2\}.$$

For the different instantiations of the inputs we obtain the instantiation of the Boolean vector and the corresponding output value according to the table below.

u_1	u_2	**Boolean vector**	y
`false`	`false`	{`false`, `false`, `true`, `false`}	2
`false`	`true`	{`false`, `false`, `false`, `true`}	1
`true`	`false`	{`false`, `true`, `true`, `true`}	7
`true`	`true`	{`false`, `true`, `false`, `true`}	5

We also define a number of operations on the binary vectors in $\bar{\mathbb{I}}_n$.

Definition 6.8

(i) Let $Bit : \mathbb{Z}_{2^n} \to \bar{\mathbb{I}}_n$ be a function that converts an integer to its bit representation, e.g. $Bit(3) = \{\mathtt{false}, \mathtt{false}, \mathtt{true}, \mathtt{true}\}$ for $n = 4$.

(ii) Let $Int : \mathbb{B}^n \to \mathbb{Z}_{2^n}$ be a function that converts constant bit-vectors to integers (the notation $\mathbb{B}^n$ indicates that the bit-represented integer is constant).

(iii) Let $BAdd : \bar{\mathbb{I}}_n \times \bar{\mathbb{I}}_n \to \bar{\mathbb{I}}_n$ be a function that computes the sum of two bit-represented integers using Boolean operations on the Boolean expressions in the vectors.

(iv) Let $BSub : \bar{\mathbb{I}}_n \times \bar{\mathbb{I}}_n \to \bar{\mathbb{I}}_n$ be a function that computes the subtraction of one bit-represented integer from another. The Boolean relations are computed in a similar way as for $BAdd$.

(v) Let $BGreaterThan : \bar{\mathbb{I}}_n \times \bar{\mathbb{I}}_n \to \mathbb{B}[Z]$ be a function that computes the Boolean relation of two bit-represented integers x, y which is equal to `true` iff $x > y$.

(vi) Let $BLessThan : \bar{\mathbb{I}}_n \times \bar{\mathbb{I}}_n \to \mathbb{B}[Z]$ be a function that computes the Boolean relation of two bit-represented integers x, y which is equal to `true` iff $x < y$.

(vii) Let $BEqual : \bar{\mathbb{I}}_n \times \bar{\mathbb{I}}_n \to \mathbb{B}[Z]$ be a function that computes the Boolean relation of two bit-represented integers x, y which is equal to `true` iff $x = y$.

(viii) Let $\overset{s}{\leftrightarrow}: \bar{\mathbb{I}}_n \times \bar{\mathbb{I}}_n \to \mathbb{B}[Z]$ be a function defined as

$$\{x_{n-1}, \ldots, x_0\} \overset{s}{\leftrightarrow} \{y_{n-1}, \ldots, y_0\} = \bigwedge_{i=0}^{n-1} x_i \leftrightarrow y_i \tag{6.41}$$

□

The Boolean relations for adding two bit-represented integers can be computed recursively using a *full adder* [28], see the example below.

Example 6.7 Let two integers $n_1, n_2 \in \mathbb{Z}_4$ be bit-represented by the vectors $x = \{x_1, x_0\}$ and $y = \{y_1, y_0\}$ respectively. The sum $n_1 + n_2$ is represented as the vector

$$\begin{aligned} BAdd(x,y) = \{ & (y_0 \wedge ((y_1 \wedge ((x_0 \wedge x_1) \vee ((\neg x_0) \wedge (\neg x_1)))) \vee \\ & ((\neg y_1) \wedge ((x_0 \wedge (\neg x_1)) \vee ((\neg x_0) \wedge x_1))))) \vee \\ & ((\neg y_0) \wedge ((y_1 \wedge (\neg x_1)) \vee ((\neg y_1) \wedge x_1))), \\ & (y_0 \wedge (\neg x_0)) \vee ((\neg y_0) \wedge x_0)\}. \end{aligned} \tag{6.42}$$

For $n_1 = 2$ and $n_2 = 1$ we get the vectors $x = \{\mathtt{true}, \mathtt{false}\}, y = \{\mathtt{false}, \mathtt{true}\}$. From the summation vector above we get $BAdd(x,y) = \{\mathtt{true}, \mathtt{true}\}$.

Since no integer variable in the landing gear controller code takes values greater than 15 we use $\bar{\mathbb{I}}_4$ for representing integers in this case study.

As mentioned in Section 6.2.2 we support a special form of variable declaration which not only defines the data type for each variable, but also defines if the variable is an input, output, state or temporary variable. The compiler initiates all input variables with symbols that represent an arbitrary input value. All other variables are initiated with $\perp$. This symbol is used to indicate if there exist values of the input variables for which the value of some output variable is undefined.

Before defining the transformation rules for the compilation step formally we illustrate the compilation with a simple example.

Example 6.8 In the figure below we see how the values of the variables in the `IF-THEN-ELSE` statement would be represented as Boolean expressions. The compilation would in this case use the transition relations (6.57), (6.58) and (6.60).

All variables are Boolean and $f_{\bar{y}_2}^-$ defines the value of the variable `y2` before this piece of code is executed. If `y2` has not been assigned a value before, $f_{\bar{y}_2}^-$ will be equal to the symbol $\perp$, which we interpret as *undefined*.

```
MIfThen[
        q,
        MAssign[y1,c],
        MBeginEnd[
                  MAssign[y1,d],
                  MAssign[y2,e]
                  ]
        ]
```

$\rightsquigarrow$

Variables	Values
y1	$f_{\tilde{y}_1} = c \wedge q \vee d \wedge \neg q$
y2	$f_{\tilde{y}_2} = f^-_{\tilde{y}_2} \wedge q \vee e \wedge \neg q$

Finally all variables are connected to their values and combined into a single relation. In the example above we get

$$R(z, z^+) = (y1 \leftrightarrow (c \wedge q \vee d \wedge \neg q)) \wedge (y2 \leftrightarrow (\bot \wedge q \vee e \wedge \neg q)). \tag{6.43}$$

We can now give the transformation rules for the compilation from MPascal to polynomials:

Evaluation of expressions

Constants:

$$\texttt{MTrue}[] \ \vdash \ \texttt{true} \tag{6.44}$$

$$\texttt{MFalse}[] \ \vdash \ \texttt{false} \tag{6.45}$$

$$\frac{N \in \mathbb{Z}}{N \ \vdash \ Bit(N)} \tag{6.46}$$

Arithmetic expressions:

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MPlus}[e_1, e_2] \ \vdash \ BAdd(e_1', e_2')} \tag{6.47}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MMinus}[e_1, e_2] \ \vdash \ BSub(e_1', e_2')} \tag{6.48}$$

Relational expressions:

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MGreaterThan}[e_1, e_2] \ \vdash \ \mathit{BGreaterThan}(e_1', e_2')} \tag{6.49}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MLessThan}[e_1, e_2] \ \vdash \ \mathit{BLessThan}(e_1', e_2')} \tag{6.50}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2' \quad (e_1', e_2' \in \bar{\mathbb{I}}_4)}{\texttt{MEqual}[e_1, e_2] \ \vdash \ \mathit{BEqual}(e_1', e_2')} \tag{6.51}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2' \quad (e_1', e_2' \in \mathbb{B}[X])}{\texttt{MEqual}[e_1, e_2] \ \vdash \ e_1' \leftrightarrow e_2'} \tag{6.52}$$

$$\frac{e \vdash e'}{\texttt{MNot}[e] \ \vdash \ \neg e'} \tag{6.53}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MAnd}[e_1, e_2] \ \vdash \ e_1' \wedge e_2'} \tag{6.54}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MOr}[e_1, e_2] \ \vdash \ e_1' \vee e_2'} \tag{6.55}$$

Compiling statements

Sequences of statements:

$$\frac{(s_1, \sigma_1) \vdash \sigma_2}{(\ s_1 \texttt{,} s_2\ \texttt{,} \sigma_1) \ \vdash \ (s_2, \sigma_2)} \tag{6.56}$$

Assignment:

$$\frac{(e, \sigma) \vdash (e', \sigma)}{(\texttt{MAssign}[v, e], \sigma) \ \vdash \ \sigma[v \mapsto e']} \tag{6.57}$$

Control statements:

$$\frac{(c, \sigma) \vdash (c', \sigma) \quad (s_1, \sigma) \vdash \sigma_1 \quad (s_2, \sigma) \vdash \sigma_2}{(\texttt{MIfThen}[c, s_1, s_2], \sigma) \ \vdash \ \sigma_1 \wedge c' \vee \sigma_2 \wedge \neg c'} \tag{6.58}$$

$$\frac{(e, \sigma) \vdash (e', \sigma) \quad k_1 \vdash k_1' \ \ldots \ k_n \vdash k_n' \quad (s_1, \sigma) \vdash \sigma_1 \ \ldots \ (s_n, \sigma) \vdash \sigma_n}{(\texttt{MCaseOf}[e, k_1, s_1, \ldots, k_n, s_n], \sigma) \ \vdash \ \theta} \tag{6.59}$$

where

$$\theta = \sigma_1 \wedge (e' \overset{s}{\leftrightarrow} k_1') \vee \ \cdots \ \vee \sigma_n \wedge (e' \overset{s}{\leftrightarrow} k_n') \\ \vee \sigma \wedge \neg((e' \overset{s}{\leftrightarrow} k_1') \vee \ \cdots \ \vee (e' \overset{s}{\leftrightarrow} k_n'))$$

$$\frac{(s_1 \texttt{,} \ \ldots \ \texttt{,} s_n, \sigma_1) \vdash \sigma_2}{(\texttt{MBeginEnd}[s_1, \ldots, s_n], \sigma_1) \ \vdash \ \sigma_2} \tag{6.60}$$

Variable declarations[3]

Temporary variables:

$$\frac{t = \texttt{MB}}{(\texttt{MVar}[\{t, v_1, \ldots, v_n\}, \mathit{vardecl}], \sigma) \ \vdash \ \theta} \tag{6.61}$$

where

$$\theta = (\texttt{MVar}[\mathit{vardecl}], \sigma[v_1 \mapsto \bot, \ldots, v_n \mapsto \bot])$$

$$\frac{t = \texttt{MI}}{(\texttt{MVar}[\{t, v_1, \ldots, v_n\}, \mathit{vardecl}], \sigma) \ \vdash \ \theta} \tag{6.62}$$

where

$$\theta = (\texttt{MVar}[\mathit{vardecl}], \sigma[v_1 \mapsto \{\bot, \bot, \bot, \bot\}, \ldots, v_n \mapsto \{\bot, \bot, \bot, \bot\}])$$

Input variables:

$$\frac{t = \texttt{MB}}{(\texttt{MInput}[\{t, u_1, \ldots, u_n\}, \mathit{vardecl}], \sigma) \ \vdash \ \theta} \tag{6.63}$$

where

$$\theta = (\texttt{MInput}[\mathit{vardecl}], \sigma[u_1 \mapsto u_1[b], \ldots, u_n \mapsto u_n[b]])$$

$$\frac{t = \texttt{MI}}{(\texttt{MInput}[\{t, u_1, \ldots, u_n\}, \mathit{vardecl}], \sigma) \ \vdash \ \theta} \tag{6.64}$$

where

$$\begin{aligned}\theta = (\texttt{MInput}[\mathit{vardecl}], \sigma[u_1 &\mapsto \{u_1[3], u_1[2], u_1[1], u_1[0]\}, \ldots, \\ & u_n \mapsto \{u_n[3], u_n[2], u_n[1], u_n[0]\}])\end{aligned}$$

Output variables:
The transitions for the declaration of output variables (`MOutput`) are similar to the transitions for `MVar`, see (6.61) and (6.62).

Procedures and functions

Declaration:

$$(\texttt{MProcedure}[\mathit{id}, \{\mathit{args}\}, s], \sigma) \ \vdash \ \sigma[\mathit{Arg}(\mathit{id}) \mapsto \mathit{args}, \mathit{Code}(\mathit{id}) \mapsto s] \tag{6.65}$$

$$(\texttt{MFunction}[\mathit{id}, \{\mathit{args}\}, s], \sigma) \ \vdash \ \sigma[\mathit{Arg}(\mathit{id}) \mapsto \mathit{args}, \mathit{Code}(\mathit{id}) \mapsto s] \tag{6.66}$$

where $\mathit{Arg}(\mathit{id})$ and $\mathit{Code}(\mathit{id})$ are place holders for the argument symbols and the code of the procedure or function named id in the binding environment σ.

[3] Note that these rules are defined recursively.

Execution:

$$\frac{E \quad ([\![\sigma_1(Code(id))]\!], \sigma_1[[\![\sigma_1(Arg(id))]\!] \mapsto \{e'_1, \ldots, e'_n\}]) \vdash (e_r, \sigma_2)}{(id[e_1, \ldots, e_n], \sigma_1) \ \vdash \ (e_r, \cap_{\sigma_1}\sigma_2)} \tag{6.67}$$

where

$$E = (e_1, \sigma) \vdash (e'_1, \sigma) \ldots (e_n, \sigma) \vdash (e'_n, \sigma)$$

$$\frac{(e, \sigma) \vdash (e', \sigma)}{(\texttt{MReturn}[e], \sigma) \ \vdash \ (e', \sigma)} \tag{6.68}$$

The `PROGRAM` keyword

$$\frac{s \vdash \epsilon}{(\texttt{MProgram}[s], \emptyset) \ \vdash \ (\epsilon, \sigma_{\mathit{final}})} \tag{6.69}$$

where $\emptyset$ denotes the empty binding environment, and σ_{final} denotes the final binding environment (final state of the compiler).

Using these transition rules we automatically compile MPascal to a polynomial relation $C(z, z^+)$. For this application we let integer variables take values in the range $\{0, \ldots, 15\}$, but for most of the integer input variables this range is too large. It is therefore important to reduce the range to avoid false input cases. Let $\Lambda(u)$ denote a Boolean expression that includes all range conditions for all input variables (compare to the set of all λ-polynomials, Λ_q, described in Section 3.7). Then we can compute the final BDD relation as

$$\tilde{C}(z, z^+) = C(z, z^+) \wedge \Lambda(u) \wedge \Lambda(u^+). \tag{6.70}$$

The valid range for each integer variable is specified in MPascal and the Λ-expression is automatically computed by the compiler.

6.2.5 The Controller Model

Let $\tilde{C}(z, z^+)$ denote the relation representing the output and state variables of the 1200 lines of controller code. This relation has approximately 320 000 nodes when represented as a BDD. It contains 105 binary variables, of which 26 are state variables. The time required for the compilation is approximately 35 minutes on a SPARCstation 10.

In order to make the BDD representation efficient it is critical to choose a good variable ordering. The basic rule is to rank the inputs lowest and the outputs highest with temporary variables in between. This is intuitive since the output variables depend on both temporary and input variables and should be found higher up in the tree structure. Also the ordering among the variables in each of the three groups affect the size of the BDD relation.

As a help in choosing ordering we have studied how the BDD grows when combining the variables. A big increase in the size when adding a variable to the relation suggests that this variable should have been ordered higher. To improve the efficiency it would be of interest to introduce an automatic choice of variable ordering in the parsing of the code.

We have thus parsed and compiled the implemented Pascal86 code into a polynomial relation whose behavior is in exact correspondence with the original code. In the next section we will use this polynomial representation to analyze the landing gear controller.

Chapter 7

Analysis of the Landing Gear Controller

In this chapter we describe the analysis of the landing gear controller with respect to some function specifications. The purpose is to show that it is possible to perform formal verification of function specifications for a system of industrial size. We begin by computing the reachable states of the system and then proceed to one step analysis and multiple step analysis of the system.

7.1 Computation of Reachable States

A part of the Pascal code initializes the controller. Using this initialization and the system relation $\tilde{C}(z, z^+)$ it is possible to compute all reachable states of the controller. It turns out that there are 10 015 reachable states and that they are all reachable in no more than five iterations of the code. This should be compared with the 2^{26} states that could be represented by the 26 binary state variables used in the code.

By combining the system relation with the relation describing all reachable states it is possible to reduce the size of our model. If $R_{states}(x)$ denotes the reachable states, the reduced model[1] is computed as

$$C_{red}(z, z^+) = \tilde{C}(z, z^+) \wedge R_{states}(x) \wedge R_{states}(x^+). \tag{7.1}$$

In this case, the complexity of the model is reduced considerably. Using the reduced model it is possible to draw conclusions about the behavior of the system.

[1]The size will decrease in most cases.

7.2 One Step Analysis

By one step analysis (see Section 4.3.1) it is for example possible to ask questions of the type

> "Is it possible to simultaneously give the hydraulic commands extend gears and retract gears?"

The answer to such a question can be `true` (or `false`) in which case the statement holds (is false) for all combinations of the input signals and states. There is also a possibility that we get a relation as the answer. This relation represents all input combinations for which the statement holds.

Using our model of the controller for verification of the statement above we get the answer `false`. This result can also trivially be deduced from the original Pascal code.

In order to try our machinery on a set of more interesting (and realistic) questions we have used material from an earlier verification project within Saab Military Aircraft. In this project Saab verified certain one step properties of the landing gear controller with the use of NPCircuit [59], a commercial tool which essentially is a Boolean equation solver.

In these tests Saab assumed that the hardware (switches etc.) works as intended and this gives additional restrictions on the input variables. These restrictions, which we also will use, are as follows (see Tables 5.1, 5.2, 5.3 and 5.4 for an explanation of the variable names):

1. Both switches on the extension lever in the cockpit have the same value.
$$p_1 \leftrightarrow p_2$$
2. Emergency extension is not activated.
$$\neg m_1$$
3. All feedback switches on gears and doors have two contacts. These have the same value.
$$s_j^1 \leftrightarrow s_j^2, \quad j = 1, \dots, 15$$
4. The switches on the doors does not give unreasonable values (open and closed at the same time).
$$\neg(s_4^i \wedge s_5^i) \wedge \neg(s_9^i \wedge s_{10}^i) \wedge \neg(s_{14}^i \wedge s_{15}^i), \quad i = 1, 2$$
5. The switches on the gears does not give unreasonable values (retracted and extended at the same time).
$$\neg(s_1^i \wedge s_2^i) \wedge \neg(s_6^i \wedge s_7^i) \wedge \neg(s_{11}^i \wedge s_{12}^i), \quad i = 1, 2$$

6. A gear with weight on is extended.

$$(s_3^i \to s_2^i) \wedge (s_8^i \to s_7^i) \wedge (s_{13}^i \to s_{12}^i), \quad i = 1, 2$$

7. Power supply is working.

$$\neg m_5$$

8. The variables m_3 and m_4 have values according to their definitions.

$$\begin{aligned}(m_3 = 1) &\leftrightarrow (s_3^1 \wedge s_8^1 \wedge s_{13}^1)\\ (m_3 = 0) &\to (\neg s_3^1 \wedge s_8^1 \wedge s_{13}^1)\\ (m_4 = 0) &\to (m_3 = 1)\end{aligned}$$

These restrictions can be regarded as a first model of how the landing gear works and they can be combined into one relation, which we denote $P_1(u)$.

One Step Analysis I We want to verify the statement

> "If extension of gears is performed (a_3), then opening of doors is performed (a_5) simultaneously."

under the following conditions:

- Hardware operates correctly in accordance with items 1–8 above.
- Nose door is closed, i.e. $(s_4^1 \vee s_4^2)$.
- Nose gear is retracted, i.e. $(s_1^1 \vee s_1^2)$.
- Previous gear command was not "extend gears", i.e.

$$(x_7 = 0 \vee x_7 = 1 \vee x_7 = 2).$$

In terms of Boolean algebra the statement can be expressed as

$$\begin{aligned}(C_{red}(z, z^+) \wedge P_1(u) \wedge P_1(u^+) \wedge (s_4^1 \vee s_4^2) \wedge (s_1^1 \vee s_1^2) \wedge \\ (x_7 = 0 \vee x_7 = 1 \vee x_7 = 2)) \to (a_3 \to a_5).\end{aligned} \quad (7.2)$$

The result of the analysis is `true`, which means that the statement is true for any combination of inputs and states. This corresponds to the result obtained earlier by Saab.

One Step Analysis II We want to verify the statement

"Retraction of gears is not performed ($\neg a_2$)."

under the following conditions:

- Hardware operates correctly in accordance with items 1–8 above.
- Nose door is closed, i.e. $(s_4^1 \vee s_4^2)$.
- Nose gear is extended, i.e. $(s_2^1 \vee s_2^2)$.
- Previous gear command was not "retract gears", i.e.

$$\neg(x_8 = 4) \wedge \neg(x_8 = 5).$$

- In the previous iteration the nose gear was not retracted, i.e.

$$\neg(x_8 = 7).$$

In terms of Boolean algebra this is expressed as

$$(C_{red}(z, z^+) \wedge P_1(u) \wedge P_1(u^+) \wedge (s_4^1 \vee s_4^2) \wedge (s_2^1 \vee s_2^2) \wedge \\ \neg(x_8 = 4) \wedge \neg(x_8 = 5) \wedge \neg(x_8 = 7)) \rightarrow \neg a_2. \quad (7.3)$$

The result of the analysis is a BDD relation, with approximately 50 000 nodes, describing for which combinations of inputs and states the statement holds. We have not analyzed this result further. However, it could be in correspondence with the verification performed by Saab, since they also found that there were cases when the statement did not hold.

7.3 Multiple Step Analysis

With multiple step analysis (see Section 4.3.2) we take an arbitrary number of steps in the controller into account. This means that we can verify statements that for example say that certain events never (or always) will take place. The specifications are given in terms of temporal logic.

As in the case with one step analysis the answer will be either `true`, `false` or a relation describing all input and state combinations for which the statement holds. If we try to verify behaviors of the controller with the input signals totally uncorrelated with the actions of the gears and doors there will be a number of "false" input-state combinations in the answer that will be hard to interpret.

In order to reduce the number of irrelevant answers, we build a discrete model of the landing gear itself. The gears are modeled by a discrete finite automaton, see Figure 7.1. A similar automaton is used to model the doors.

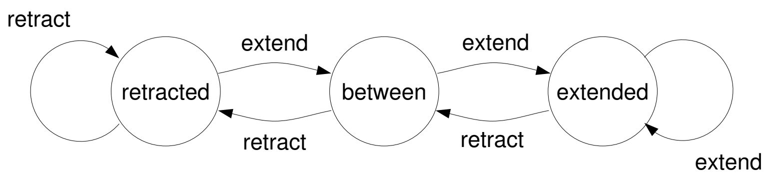


Figure 7.1: Gears modeled as an automaton.

These automata can be expressed as a Boolean relation, $P_2(z, z^+)$. By combining the plant and the controller we get a somewhat more sophisticated model to perform verification on. Let $M(z, z^+)$ denote this model of the closed loop system. We then have

$$M(z, z^+) = C_{red}(z, z^+) \wedge P_1(u) \wedge P_1(u^+) \wedge P_2(z, z^+). \tag{7.4}$$

Without detailed knowledge of the desired behavior of the system it is difficult to formulate relevant specifications concerning the multiple step behavior. We are currently in the process of completing the model of the closed loop system and finding relevant specifications. No specific results on multiple step properties except for the computation of reachable states have therefore been achieved yet.

Chapter 8

Conclusions and Future Work

The goal of this case study was to investigate if polynomials over a finite field can be used for modeling and verification of discrete event dynamic systems of industrial size. The application chosen as an example was the landing gear controller of the Swedish JAS 39 fighter aircraft. In this section we will give some general conclusions and directions of future work.

8.1 Conclusions

In this case study we have given an example of how one may verify a discrete dynamic control system by building a model of the whole process:

$$M(z, z^{+}) := C(z, z^{+}) \wedge P(z, z^{+}) \tag{8.1}$$

where $C(z, z^{+})$ is the controller and $P(z, z^{+})$ is the plant. Furthermore $M(z, z^{+})$ is a polynomial over a finite field, which in this work has been the Boolean field.

We can also build a simple model of the function specification $S(z)$ using temporal algebra. Using the closed loop system model $M(z, z^{+})$ and the specification $S(z)$ we can then either *verify* or *falsify* the system behavior w.r.t. the specification. In case we falsify the system behavior we can also generate a sequence of inputs that exhibits the failing behavior. This can then be independently verified in a system simulator and the error should be characterized well enough for modification of the controller.

Below we briefly list some of the overall conclusions regarding this application of symbolic algebraic discrete systems theory to an industrial scale problem.

- The developed methods and tools allow us to analyze industrial scale discrete systems. In particular this allows us to prove (or disprove) that the system behaves according to its specification.

- The system should be automatically translated from an internal model description language to the polynomial format, which is suitable for analysis. This procedure eliminates potential discrepancies between documents and the actual system.
- For dynamic systems, dynamic analysis will ultimately be needed and hence an algebraic computation engine that can handle dynamic analysis is necessary.

8.2 Future Work

Let us conclude this part by giving some examples of future work concerning the modeling and analysis aspects of discrete event dynamic systems.

We need more research on model description languages aimed at symbolic analysis (e.g. verification) rather than simulation or code generation. There are several candidate languages to examine, some are international standards and some are industry specific model description languages. Furthermore we need to better understand the relation between model complexity, as seen in some model description language, and model complexity in the polynomial representation. This is of crucial importance when doing symbolic analysis and design and the problem would greatly benefit from a more focused study.

In the analysis part there are a number of important issues concerning complexity. In particular, in performing the multiple step analyses there are several equivalent computational formulations that have wildly different space and time complexities. These ought to be investigated more thoroughly.

In order to reach our ultimate goal of capturing the entire design process from analysis to implementation, we should also begin to investigate the problem of design and implementation for industrial scale examples.

Part II

Simultaneous Supervision and Control – A First Example[1]

[1]This part is co-authored with J. Plantin and can also be found in [71].

Chapter 9

Introduction

In industry, control and supervision have often been regarded as separate problems, where the supervision has been dealt with in an ad hoc manner. With the increasing complexity of discrete event dynamic systems there is a need for structured methods for control of DEDS. There are several results in the area of supervisory control, initiated by Ramadge and Wonham [76]. Two good surveys are [82] for the automata-theoretic approach to supervisory control and [44] for the Petri net approach to supervisory control

The earlier approaches within the field, mentioned above, have concentrated on the pure supervision problem. It would however be desirable to find a design method, formally handling both control and supervision aspects in a discrete setting. By supervision we mean avoiding forbidden states and the control problem is that of achieving a desired behavior among all allowed behaviors.

We propose a method where we use polynomials in a quotient ring to represent the system and the controller. By working through an example we investigate whether it is possible to automatically synthesize the control law for a discrete event dynamic system using this representation. We describe the process of controller design for the example, but the method is not specific for this example and general conclusions are drawn about the method used.

We begin by describing the model of the water tank. In Chapter 11 we describe the design criteria, deal with the computation of the control laws, using Gröbner bases, and analyze the resulting controller. In the last chapter we discuss the results achieved and draw some general conclusions. Since the method makes extensive use of polynomials in a quotient ring, which are described in Chapter 3, the reader may benefit from beginning with this chapter.

Chapter 10

Modeling the Water Tank

The example system chosen for controller design is a water tank (see Figure 10.1) in which we want to control the water level. The tank has one inlet and two outlets, which are controlled by valves that are either on or off. The inlet is supplied by a pump that is either on or off and in one of the outlets there is a measurable but uncontrollable flow out of the tank. Apart from the normal control of the tank we also want to be able to handle a possible pump failure at the inlet.

The supervisory objective is to prevent the tank from drying up or overflowing. In addition we have a control objective of keeping the level as close to the middle of the tank as possible. We want to compute a control law that uses only the pump as long as that is sufficient. When a pump failure occurs, however, we have to use the valves to fulfill the objective.

This configuration is not the result of modeling a physical system. Instead, we have tried to generate a good test example which is not entirely trivial but where it is possible to analyze the computed control laws by hand. Even if this is a simple system some important features in the process of control design are clearly visible. It should also be noted that the example does not have to be much larger before it becomes difficult to solve by hand.

10.1 Notation

Let u_1 represent the binary control signal to the pump with *off* corresponding to $u_1 = 0$ and *on* corresponding to $u_1 = 1$. Let u_2, u_3 and u_4 be the binary control signals to the valves, where an open valve corresponds to $u_i = 1$, while a closed valve is represented by $u_i = 0$. We let ϕ be the net flow in the tank with $\phi = 0$ corresponding to a negative flow, $\phi = 1$ to no net flow and $\phi = 2$ to a positive flow. The level in the tank is denoted x and is discretized to take values between zero and six.

The flow at the outlet is modeled as a measurable binary disturbance w,

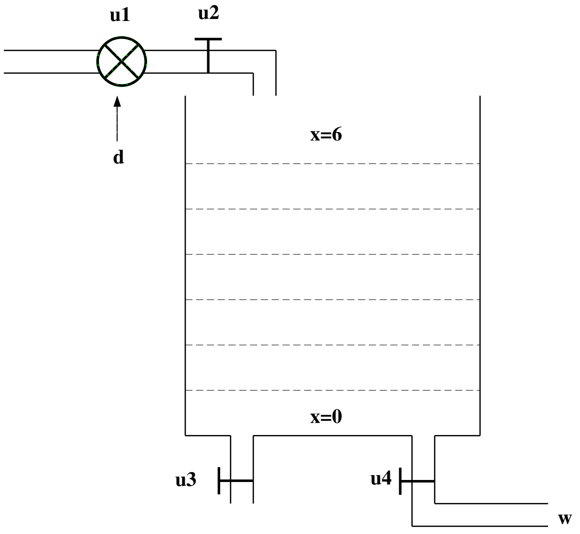


Figure 10.1: The water tank.

with $w = 0$ corresponding to no flow in the outlet and $w = 1$ corresponding to a flow out of the tank.

The pump failure is modeled as a three-valued disturbance, d, acting on the pump. The signal d represents two failure states and one normal state of the pump. The effect of the disturbance on the pump is described in Table 10.1, where $\tilde{u}_1$ represents the effect on the flow into the tank. In other words, when $d = 0$ the pump is stuck and when $d = 2$ the pump is running, regardless of the value of u_1. Only when $d = 1$, the pump obeys the control signal, u_1. The disturbance, d, is assumed to be measurable.

The different variables and their domains are shown in Table 10.2.

10.2 Deriving the Model

To reduce complexity it is essential to divide the system into subsystems, if possible. In this case we can use the net flow ϕ to write the tank model as

$$\begin{aligned} x^+ &= f_1(x, \phi) \\ \phi &= f_2(u_1, u_2, u_3, u_4, w, d) \end{aligned} \tag{10.1}$$

where x^+ denotes the next value of the state. Note that this is an event-driven system, the "real time" is not included in the description. This means that the model will not contain time information as e.g. how fast the water level increases or decreases. The model just defines the order of the events interacting with the tank.

In Table 10.2 we see that x is the variable with the largest domain ($\{0, \ldots, 6\}$). Therefore we have to represent all signals by polynomials in

u_1	d	$\tilde{u}_1$
0	0	0
1	0	0
0	1	0
1	1	1
0	2	1
1	2	1

Table 10.1: The effect of the disturbance d.

Variable	**Domain**	**Quantity**
x	$\mathbb{F}_7$	Water level
u_1	$\mathbb{F}_2$	Pump signal
$u_2 - u_4$	$\mathbb{F}_2$	Valve signals
ϕ	$\mathbb{F}_3$	Net flow
w	$\mathbb{F}_2$	Outflow disturbance
d	$\mathbb{F}_3$	Pump failure

Table 10.2: Variables with corresponding domain.

$R_7[Z]$, where Z is the variable set $\{x, \phi, u_1, u_2, u_3, u_4, w, d\}$. The variables that are two- or three-valued can be embedded in $R_7[Z]$ using the following relations (see Section 3.7)

$$\begin{aligned}
\lambda_7^2(u_i) &= u_i^2 - u_i, \quad i = 1, \dots, 4 \\
\lambda_7^2(w) &= w^2 - w \\
\lambda_7^3(d) &= 3d - d^2 - 2d^3 \\
\lambda_7^3(\phi) &= 3\phi - \phi^2 - 2\phi^3
\end{aligned} \tag{10.2}$$

The set of these λ-polynomials will, according to Definition 3.12, be denoted Λ_7.

When deriving the polynomial f_1 in (10.1) it is important to keep in mind the physical aspects of the tank. When $x = 6$ and we have a positive net flow into the tank ($\phi = 2$), it will overflow. This means that x^+ should still be six. A similar problem has to be accounted for when the level is $x = 0$ and the net flow $\phi = 0$. Because of this, the polynomials describing the system become fairly complicated.

One way of describing the behavior of the system is to write down two tables, one specifying how ϕ depends on u_1, u_2, u_3, u_4, w and d, and one specifying how x^+ depends on x and ϕ. These tables then specify ϕ and

x^+ as *functions*. Using the Lagrange interpolation polynomial introduced in Section 3.3 we can derive the polynomial description of the system. The two polynomials f_1 and f_2 in (10.1) become

$$\begin{aligned} f_1(x,\phi) = x &+ 3\phi - 3\phi^2 - 3\phi x + 3\phi^2 x + 3\phi x^2 - 3\phi^2 x^2 - \\ &- 3\phi x^3 + 3\phi^2 x^3 + 3\phi x^4 - 3\phi^2 x^4 - 3\phi x^5 + 3\phi^2 x^5 - \\ &- x^6 + \phi x^6 \end{aligned} \tag{10.3}$$

and

$$\begin{aligned} f_2(u_1,u_2,u_3,u_4,w,d) = 1 &+ 3du_2 - 3d^2u_2 + 2du_1u_2 - d^2u_1u_2 - \\ &- u_3 - u_4w + u_3u_4w - 3du_2u_3u_4w + \\ &+ 3d^2u_2u_3u_4w - 2du_1u_2u_3u_4w + \\ &+ d^2u_1u_2u_3u_4w. \end{aligned} \tag{10.4}$$

where $f_1, f_2 \in R_7[z, z^+]$ with z being the set of system variables.

Here, f_1 is a description of the next state, when we know x and ϕ. The polynomial f_2 tells us how ϕ, in its turn, depends on the control signals and the disturbances. Together these polynomials tell us how x^+ depends on the control signals and the disturbances. It is of course easy to verify that the behavior is as desired by substituting values for the u_i, w and d and computing the value of the next state modulo 7.

Example 10.1 Let us for example assume that the pump works normally ($d = 1$) and is on ($u_1 = 1$), that the disturbance w is equal to one and that the valves are positioned according to

$$u_2 = 1 \quad u_3 = 0 \quad u_4 = 0$$

Substituting these values into (10.4) the result is 2, which means that we have a net flow into the tank. Let us assume that the current state is $x = 3$. Substituting values for ϕ and x in (10.3) we see that $f_1 = 4$. Thus the next state will be $x^+ = 4$.

In (10.3) and (10.4) we have a mathematical model of the water tank. The question now is how to specify the control objectives and how to compute the desired control law.

Chapter 11

Controller Design

In this chapter we will describe the process of controller design for the water tank modeled in Chapter 10. We begin by formulating the design criteria in terms of polynomials in Section 11.1 and then we derive the control laws using successive Gröbner basis computations in Section 11.2. Finally we analyze the designed controller in Section 11.3.

11.1 Design Criteria

The supervisory objective is to avoid the case when the tank dries up or overflows. In terms of the level x, we want to avoid $x = 0$ and $x = 6$. Given the present level and the disturbances w and d, we want to find a control law that guarantees that we never reach the forbidden levels, specified by the polynomial

$$p(x) = 1 + x + 6x^2 + x^3 + 6x^4 + x^5 + 5x^6. \tag{11.1}$$

This polynomial can be generated from e.g. a table, using the Lagrange interpolating polynomial described in Section 3.3. The polynomial is equal to zero for all values of x, except $x = 0$ and $x = 6$, where it is equal to one.

By formulating only a supervisory objective we get a set of solutions. For example, since we only want to avoid $x = 0$ and $x = 6$, we know that there are no constraints on ϕ at all, unless the level, x, is in a neighborhood of the forbidden levels. In addition to this, we have four actuators to choose between at every instant, some of them giving the same control behavior.

One way of finding one of the possible control laws, is to reduce the set of solutions by imposing more requirements on the system. A weighting function could then be used, both to obtain our control objective and to reduce the solution set. Using a weighting function, we could formulate our control objective as reducing the weight of the next value of x compared to the present value. In other words, we steer the level towards the middle of the tank. If we can still guarantee that the level never reaches $x = 0$

or $x = 6$, this is just one way of picking a single solution. The weighting function that we have chosen is given by Example 3.3.

If we represent the weighting function as a polynomial, we have from (3.22):

$$J(x) = 3 + 3x + 6x^2 + x^3 + 2x^4 + 6x^5 + 2x^6 \tag{11.2}$$

where $J(x) \in R_7[x]$. Use this polynomial to weight the new values of the state variable:

$$J(x^+) = J(f_1(x, \phi)). \tag{11.3}$$

We want to find the value of ϕ that minimizes $J(x^+)$ with respect to the ordering $0 < 1 < \cdots < 6$, i.e.

$$\phi = \arg \min_{\phi \in \mathbb{F}_3} J(f_1(x, \phi)). \tag{11.4}$$

Now let the polynomial $p_1(x, \phi, m)$ be defined as

$$p_1(x, \phi, m) = J(f_1(x, \phi)) + m - J(x). \tag{11.5}$$

If there exists a $\phi \in \mathbb{F}_3$ that decreases the weight of the state with m steps, that value of ϕ is defined as the solution to

$$p_1(x, \phi, m) = 0. \tag{11.6}$$

This solution is valid only in the case when $m < q - J_{max}$ ($q = 7$ here), where $J_{max} = \arg \max_{x \in \mathbb{F}_7} J(x)$ and m, q and J_{max} and the max- and $<$-operations are regarded as in $\mathbb{N}$. The ordering will otherwise be destroyed, and there will be false solutions.

By this construction we will find a solution, ϕ, if one exists. If we can express ϕ explicitly, the values of the actuator signals are the solutions to

$$p_2(\phi, u_1, u_2, u_3, u_4, w, d) = 0 \tag{11.7}$$

where $p_2(\phi, u_1, u_2, u_3, u_4, w, d)$ is defined as

$$p_2(\phi, u_1, u_2, u_3, u_4, w, d) = f_2(u_1, u_2, u_3, u_4, w, d) - \phi. \tag{11.8}$$

The explicit expressions for the polynomials p_1 and p_2 are

$$\begin{aligned} p_1(x, \phi, m) = {} & -3\phi + 3\phi^2 + m + x + \phi x - 2\phi^2 x - 2x^3 - 2\phi x^3 - \\ & - 3\phi^2 x^3 - 2\phi x^4 + 2\phi^2 x^4 + 2x^5 - 2\phi x^5 \end{aligned} \tag{11.9}$$

$$\begin{aligned} p_2(\phi, u_1, u_2, u_3, u_4, w, d) = {} & 1 - \phi + 3du_2 - 3d^2u_2 + 2du_1u_2 - \\ & - d^2u_1u_2 - u_3 - u_4w + u_3u_4w - \\ & - 3du_2u_3u_4w + 3d^2u_2u_3u_4w - \\ & - 2du_1u_2u_3u_4w + d^2u_1u_2u_3u_4w \end{aligned} \tag{11.10}$$

11.2 Computation of the Control Laws

The two polynomials $p_1(x, \phi, m)$ and $p_2(\phi, u_1, u_2, u_3, u_4, w, d)$ express relations between the variables, implicitly describing the control laws. We want to find explicit control laws

$$\begin{aligned} u_1 &= K_{u_1}(x, w, d) \\ u_2 &= K_{u_2}(x, w, d) \\ u_3 &= K_{u_3}(x, w, d) \\ u_4 &= K_{u_4}(x, w, d) \end{aligned} \tag{11.11}$$

where the u_i are functions of x, w and d. For every combination of these variables, the control laws should guarantee that we never enter the forbidden area and that the additional control objective is met.

In order to do this, we first need to express ϕ as a function of x, using the polynomial p_1. One way of doing that is to compute a Gröbner basis with lexicographic ordering, where we rank ϕ the highest and use m as a parameter. The Gröbner basis will then contain a polynomial that is linear in ϕ (see Definition 3.10), since the choice of ϕ is unambiguous for every x.

11.2.1 Computing the Desired Net Flow

We know that the level x in the tank cannot increase or decrease by more than one in each step. Therefore it is only possible to reduce the weight by at most one ($m = 1$) in each time step, and it will not be possible to find control laws for $m > 1$.

Begin by computing the Gröbner basis

$$\mathcal{GB}_a = GB_7\left(\left\{p_1(x, \phi, m), \lambda_7^3(\phi)\right\}|_{m=1}\right) \tag{11.12}$$

with lexicographic order $\phi > x$. In the general case a Gröbner basis contains a number of polynomials

$$\mathcal{GB} = \{g_1(\phi, x),\ g_2(x), \dots, g_i(x)\}\,. \tag{11.13}$$

In this case the Gröbner basis only contains two polynomials ($i = 2$), so we have

$$\mathcal{GB}_a = \{g_1(\phi, x),\ g_2(x)\}\,. \tag{11.14}$$

By the construction of p_1 from (11.5) we have no ambiguity for $m = 1$, i.e. ϕ is a linear variable w.r.t. $\langle p_1, \lambda_7^3\rangle$ and the first polynomial in $\mathcal{GB}_a$ can be written (see Section 3.6)

$$\phi - k_{\phi_a}(x) \tag{11.15}$$

The other polynomials in $\mathcal{GB}_a$ define where this solution is valid. Let the polynomial $v_{\phi_a}(x)$ denote the *valid area* for $k_{\phi_a}(x)$. Generally we would compute $v_{\phi_a}(x)$ as

$$v_{\phi_a}(x) = g_2(x) \wedge g_3(x) \wedge \ldots \wedge g_i(x). \tag{11.16}$$

Here we only have one polynomial except g_1, so $v_{\phi_a}(x)$ is

$$v_{\phi_a}(x) = g_2(x). \tag{11.17}$$

In this case it turns out that the solution is valid for all x, except $x = 3$. This is of course due to the fact that we have required the weight of the state to decrease by one and this is not possible when $x = 3$. Instead, we have to be satisfied as long as the weight does not increase. This means that we have to search for another solution in the case when $v_{\phi_a} \neq 0$. Compute

$$\mathcal{GB}_b = GB_7\left(\left\{p_1(x,\phi,m), \lambda_7^3(\phi), \neg v_{\phi_a}(x)\right\}|_{m=0}\right) \tag{11.18}$$

with lexicographic order $\phi > x$ to get

$$\phi - k_{\phi_b}(x) \tag{11.19}$$

with corresponding polynomial $v_{\phi_b}(x)$, denoting the valid area for $k_{\phi_b}(x)$.

We can now compute ϕ as a function of x as

$$\phi = K_\phi(x) = k_{\phi_a}(x)\neg v_{\phi_a}(x) + k_{\phi_b}(x)\neg v_{\phi_b}(x) \tag{11.20}$$

with valid area

$$V_\phi = v_{\phi_a} \vee v_{\phi_b}. \tag{11.21}$$

It turns out that $V_\phi \equiv \texttt{true}$. Therefore the valid area covers all values of x and there is a solution in ϕ for every $x \in \mathbb{F}_7$. The explicit expression for $K_\phi(x)$ is

$$K_\phi(x) = 2 - x - 3x^2 - 3x^3 + 2x^4 - 2x^5. \tag{11.22}$$

If we are able to compute the u_i as functions of ϕ, w and d, using p_2, we can express the actuator signals in the desired form of equation (11.11) by using equation (11.22).

11.2.2 Finding an Unambiguous Control Law

In order to compute the u_i as functions of ϕ, w and d we need the solution to equation (11.7) to be unambiguous. Obviously, this is not the case with the water tank. There are several control actions that, given values of w and

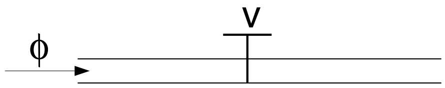
d, give the same net flow ϕ. Therefore we need to make a priority among the actuators. In this case it is natural to try to control the tank by using the pump, if possible, and only use the valves if necessary.

Therefore we start by computing a Gröbner basis for the case when we only use the pump for control, setting the other actuators to default values. In some cases the choice of default values is quite natural, but we can of course define them any way we want. Here we let the default values of the valves be as in normal operation (the pump working), that is: u_2 open, u_3 closed and u_4 open. The default value for the pump is chosen to be $u_{1,def} = 0$.

The first Gröbner basis computation gives us a valid area for the first control law. We then continue with the computation of a second Gröbner basis, using u_1 and u_2 for the control. In this way we finally get four expressions for each u_i, valid in four different areas and these can be combined into one control law.

Due to physical causes, there is a possibility that some of the actuators can not affect the behavior of the system. In terms of the Gröbner basis this means that the value of the corresponding variable will be unspecified. In order to guarantee that there is only one solution to (11.7) in these cases, we need extra constraints in the Gröbner basis computations. This is illustrated by a simple example.

Example 11.1 Consider a pipe with one valve that can be either open or closed, see the figure below.



An open valve corresponds to $v = 1$, while a closed valve is represented by $v = 0$. The flow into the pipe is represented by ϕ and can be either $\phi = 1$ (flow) or $\phi = 0$ (no flow).

Obviously, when $\phi = 0$, it does not matter what we do with the valve. To get a unique control strategy we can decide that we let $v = 1$ be the default value in this case. Using polynomials this can be expressed by the condition

$$c(\phi, v) = (\phi - 1)(v - 1). \tag{11.23}$$

If we include a polynomial like this in the Gröbner basis computations the result will be that when $\phi = 0$, v is forced to the value 1, while the condition is satisfied no matter what v is, when $\phi = 1$.

In the case of the water tank we have, for example, that when $d = 0$ or $d = 2$ we cannot use u_1 for control, so for these values of d we assign the default value to u_1. There are three similar cases to account for and all these have to be expressed by extra constraints. These extra constraints will be specified in the next section.

11.2.3 Computation of the Actuator Signals

First define default values for the actuators

$$\begin{aligned} u_{1,def} &= 0 \\ u_{2,def} &= 1 \\ u_{3,def} &= 0 \\ u_{4,def} &= 1 \end{aligned} \tag{11.24}$$

The extra constraints are given by

$$\begin{aligned} c_1(u_1, d) &= (d-1)(u_1 - u_{1,def}) \\ c_2(u_2, d) &= (d-2)(u_2 - u_{2,def}) \\ c_3(u_3, w) &= w(u_3 - u_{3,def}) \\ c_4(u_4, w) &= (w-1)(u_4 - u_{4,def}) \end{aligned} \tag{11.25}$$

Let $k_{u_i,j}$ denote the control law for actuator i in valid area v_j, where $i = 1, \dots, 4$ and $j = 1, \dots, 4$.

Start by computing

$$\mathcal{GB}_1 = GB_7(\{p_2, c_1, \Lambda_7\}\,|_{u_2=1, u_3=0, u_4=1}) \tag{11.26}$$

with lexicographic order $u_1 > \phi > d > w$. This gives us

$$u_1 - k_{u_1,1}(\phi, d, w) \tag{11.27}$$

as the first polynomial in $\mathcal{GB}_1$, and a valid area for this solution which we denote $v_1(\phi, d, w)$. In this valid area, we let the other actuators take their default values, that is

$$\begin{aligned} k_{u_2,1} &= 1 \\ k_{u_3,1} &= 0 \\ k_{u_4,1} &= 1 \end{aligned} \tag{11.28}$$

Next use both u_1 and u_2. Since this control action should not be allowed in v_1 we add $\neg v_1$ in the Gröbner basis computation. That is, compute

$$\mathcal{GB}_2 = GB_7(\{p_2, c_1, c_2, \neg v_1, \Lambda_7\}\,|_{u_3=0, u_4=1}) \tag{11.29}$$

with lexicographic order $u_1 > u_2 > \phi > d > w$.

From this we get $k_{u_1,2}$ and $k_{u_2,2}$, while $k_{u_3,2}$ and $k_{u_4,2}$ take their default values. We then continue by computing

$$\mathcal{GB}_3 = GB_7(\{p_2, c_1, c_2, c_3, \neg v_1 \wedge \neg v_2, \Lambda_7\}\,|_{u_4=1}) \tag{11.30}$$

and

$$\mathcal{GB}_4 = GB_7\left(\{p_2, c_1, c_2, c_3, c_4, \neg v_1 \wedge \neg v_2 \wedge \neg v_3, \Lambda_7\}\right) \tag{11.31}$$

with lexicographic order $u_1 > u_2 > u_3 > \phi > d > w$ and $u_1 > u_2 > u_3 > u_4 > \phi > d > w$ respectively.

Since the valid areas in the four cases are disjoint by construction, we can compute the total control law for each u_i as

$$K_{u_i}(\phi, w, d) = \sum_{j=1}^{4} k_{u_i,j} \neg v_j. \tag{11.32}$$

This control law will have the total valid area $V_u = v_1 \vee v_2 \vee v_3 \vee v_4$. If we express the control laws and the valid area explicitly, we get

$$\begin{aligned}
K_{u_1}(\phi, w, d) &= -d\phi - 3d^2\phi + d\phi^2 - 3d\phi w - 2d^2\phi w - \\
&\quad - 2d\phi^2 w + d^2\phi^2 w \\
K_{u_2}(\phi, w, d) &= 1 - 3d + 3d^2 - 3\phi + 3d\phi + 3\phi^2 - 3d\phi^2 - d\phi w + \\
&\quad + d^2\phi w - 3d\phi^2 w + 3d^2\phi^2 w \\
K_{u_3}(\phi, w, d) &= 1 + 2\phi - 3\phi^2 - w - 2\phi w + 3\phi^2 w \\
K_{u_4}(\phi, w, d) &= 1 - 3\phi + d\phi + 2d^2\phi + 3\phi^2 - d\phi^2 - 2d^2\phi^2 - 2\phi w + \\
&\quad + 2d\phi w - 3d^2\phi w + \phi^2 w + 3d\phi^2 w - d^2\phi^2 w
\end{aligned}$$

$$V_u(\phi, w, d) = 1 - 3\phi + d\phi + 2d^2\phi + 3\phi^2 - d\phi^2 - 2d^2\phi^2 \tag{11.33}$$

As we will see later, the valid area will not cover all possible combinations of x, d and w.

We have now computed all the control laws and could substitute (11.22) into (11.33) to get the equations in the desired form (11.11). However, to reduce complexity, it is essential to keep the modularity of the system. In the analysis we therefore avoid the substitution as long as possible.

11.3 Analysis of the Design

After the design phase we need to make sure that both the control objective and the supervisory objective are achieved. We will focus on

- *Controllability.* Is it possible to fulfill the control objectives by actions on the inputs?
- *Supervisability.* Given an initial state which is allowed, can we guarantee that we will never reach a forbidden state?

11.3.1 Controllability

The polynomial V_u in (11.33) represents the values of the variables ϕ, w, d, for which we have an appropriate control law. If this polynomial is `false` for some values of ϕ, w, d we know that for those values no control law has been computed. This is an indication that the system is not controllable everywhere for the design configuration chosen. If the system was controllable everywhere, this design method would find control laws for all values of ϕ, w, d.

Evaluating the polynomial (11.33) for all values of ϕ, w and d, we find that V_u is `false` only for the case

$$\begin{cases} \phi = 2 \\ d = 0 \end{cases} \tag{11.34}$$

(independent of w). This corresponds to the case when there is a pump failure and, at the same time, the level in the tank is required to increase. Looking at the physical system in Figure 10.1 there is no doubt that it is impossible to fulfill that requirement.

A `false` value forces us to do a re-engineering of either the system itself (adding some actuators) or the objectives stated for the control design. In this example we choose the latter.

With an uncontrollable system we cannot trust that the control laws will give us the expected behavior. Therefore we have to define a *true* net flow, ϕ_0, which corresponds to the true net flow for the real tank. The true net flow ϕ_0 may differ from the *desired* net flow ϕ when we have an uncontrollable system. The desired net flow ϕ is used in the computations of the actuator values but may not always be obtained because of the uncontrollability.

11.3.2 The Closed Loop System

From (10.3), (10.4), (11.22) and (11.33) we have the model and the control law equations from which we shall derive a closed loop description.

The closed loop system must be properly defined, which means that the control laws also must be defined for the non valid area. Let us do that in two ways:

1. Let the control laws take their default values also outside the valid area (a simple and intuitive strategy when the physical causes of the non valid area are unknown).

$$u_{1,def} = 0,\ u_{2,def} = 1,\ u_{3,def} = 0,\ u_{4,def} = 1 \tag{11.35}$$

2. It seems smarter and more careful to let the valve u_4 be closed. (To be sure that the level is not decreasing.)

$$u_{1,def} = 0,\ u_{2,def} = 1,\ u_{3,def} = 0,\ u_{4,def} = 0 \tag{11.36}$$

These two ways of handling the control laws will generate the closed loop descriptions $G_{c1}(x, w, d)$ and $G_{c2}(x, w, d)$ respectively.

The computation of the closed loop system $x^+ = G_c(x, w, d)$ is straightforward using simple substitutions

$$x^+ = f_1(x, \phi_0) \tag{11.37}$$
$$\phi_0 = f_2(u_1, u_2, u_3, u_4, w, d) \tag{11.38}$$
$$u_1 = K_{u_1}(\phi, w, d) \tag{11.39}$$
$$u_2 = K_{u_2}(\phi, w, d) \tag{11.40}$$
$$u_3 = K_{u_3}(\phi, w, d) \tag{11.41}$$
$$u_4 = K_{u_4}(\phi, w, d) \tag{11.42}$$
$$\phi = K_\phi(x, w, d) \tag{11.43}$$

where we substitute (11.43) into (11.42) and so on. To formulate the closed loop system in one polynomial in the general case is of course rather heavy (even for a computer). Still, it is possible to do this in our small example, and it will make the analysis convenient for us.

The resulting polynomials for the two closed loop systems are

$$\begin{aligned} G_{c1}(x, d, w) = {} & -2d + 3d^2 - 2x + 3dx - d^2x - 2wx + 3dwx - \\ & - d^2wx - dx^2 - 2d^2x^2 + 3wx^2 - dwx^2 - 2d^2wx^2 - \\ & - x^3 - 3dx^3 + d^2x^3 + 2wx^3 - 3dwx^3 + d^2wx^3 + \\ & + 3dx^4 - d^2x^4 - 2wx^4 + 3dwx^4 - d^2wx^4 + x^5 - \\ & - dx^5 - 2d^2x^5 + 3wx^5 - dwx^5 - 2d^2wx^5 + 3x^6 - \\ & - dx^6 - 2d^2x^6 + 2wx^6 - 3dwx^6 + d^2wx^6 \end{aligned} \tag{11.44}$$

$$\begin{aligned} G_{c2}(x, w, d) = {} & -2d + 3d^2 - 2x + 3dx - d^2x - dx^2 - 2d^2x^2 - \\ & - x^3 - 3dx^3 + d^2x^3 + 3dx^4 - d^2x^4 + x^5 - dx^5 - \\ & - 2d^2x^5 + 3x^6 - dx^6 - 2d^2x^6 \end{aligned} \tag{11.45}$$

Note that the polynomial G_{c2} is independent of w which means that the control laws eliminate the influence of the disturbance signal w. This in turn leads to a simpler polynomial.

These polynomials are hard to interpret as they are. One way of gaining insight would be to substitute all possible values of the variables and derive a table of the closed loop system. For larger systems such a method would be of little use, and as we will see in the next section it is possible to analyze the closed loop behavior, using the polynomial description.

11.3.3 Supervisability

One way of checking supervisability is to test *backward reachability* for our closed loop systems. If we from all forbidden states move backwards one step with all possible input signals, and the states reached are all forbidden, we know that it is impossible to reach a non forbidden state in any number of backward steps. Thus the forbidden states are not reachable from a non forbidden state. If the polynomial describing the allowed states is simpler, it would of course be better to test forward reachability from the allowed states instead.

The forbidden states are described by the polynomial $\neg p(x) = 0$, where $p(x)$ is as defined in (11.1). Now consider the ideal

$$A = \langle \neg p(x^+),\ x^+ - G_c(x, w, d),\ \Lambda_7 \rangle. \tag{11.46}$$

Let the values of x in the variety $V(A)$, specifying the *one step* backward reachable states from the states defined by $\neg p(x^+)$, be contained in a set denoted B_x. We have that if $B_x \subseteq V(\neg p(x))$, a properly initialized system cannot reach the forbidden area, in other words the objectives are fulfilled.

Let us compute the Gröbner bases for the two cases with lexicographic order $d > w > x^+ > x$.

(i)

$$GB_7\left(\{\neg p(x^+), x^+ - G_{c1}(x, w, d), \Lambda_7(w, d)\}\right) = \\ \{\, d,\ -w + w^2,\ -x + wx,\ x^+,\ 3x + 4x^2\} \tag{11.47}$$

(ii)

$$GB_7\left(\{\neg p(x^+), x^+ - G_{c2}(x, w, d), \Lambda_7(w, d)\}\right) = \\ \{\, d,\ -w + w^2,\ 2x + x^+,\ x^2\} \tag{11.48}$$

The backward reachable states are specified by the last polynomial in the Gröbner bases above, since x has the lowest rank. For G_{c1} we have the polynomial $3x + 4x^2$ which has $\{\, x = 0,\ x = 1\}$ as roots, and we see that this system does not fulfill our objectives, since $x = 1$ is in the non forbidden area. For G_{c2} we have $x = 0$ as the only root, showing that no non forbidden states are reached. Therefore G_{c2} is a robust design for this model of disturbances and choice of forbidden states. The supervisability analysis shows that it is important to deal appropriately with the control behavior outside the valid area.

To conclude the discussion about the closed loop behavior the complete closed loop description is presented in Figure 11.1.

d	w	x: 0	1	2	3	4	5	6
	0	0	1	2	3	3	4	5
0	1	0	0	1	3	3	4	5
	0	1	2	3	3	3	4	5
1	1	1	2	3	3	3	4	5
	0	1	2	3	3	3	4	5
2	1	1	2	3	3	3	4	5

d	w	x: 0	1	2	3	4	5	6
	0	0	1	2	3	3	4	5
0	1	0	1	2	3	3	4	5
	0	1	2	3	3	3	4	5
1	1	1	2	3	3	3	4	5
	0	1	2	3	3	3	4	5
2	1	1	2	3	3	3	4	5

Figure 11.1: Values for $x^+ = G_{c1}(x, d, w)$ and $x^+ = G_{c2}(x, d, w)$ respectively.

Chapter 12

Conclusions

The example has shown that polynomial methods can be used to formally handle both control and supervision aspects in a discrete setting. Even if some steps were manipulated by hand in this example, the results indicate that the design process could be stated as an algorithm. This means that it would be possible to generate control code automatically, given a DEDS model and control/supervisor objectives.

12.1 The Design Method

The design method used produces control laws that are *functions* of the measurable variables. In order to achieve this we must have a way to choose one of, possibly, many control laws fulfilling the specification. Using ideas from LQG-design in the continuous domain, we propose the use of a weighting function, where the forbidden states are given the highest weight. This method allows assignment of weights to groups of states and it also opens the possibility to specifications of another kind than "forbidden states".

We also propose a priority among the actuators. This priority sometimes is a natural priority, otherwise we can regard it as a design criteria. It could for example be used to distinguish emergency actions from normal control, i.e. handle supervision aspects.

To further reduce the number of possible control laws we impose constraints that eliminate all control actions that have no effect on the system. Since these constraints seem to be physically intuitive it is possible that they can be generated algebraically in the modeling process.

The functions representing the control laws can be analyzed symbolically. Another advantage is that we get a valid area for our control laws, which makes it possible to examine the controllability of the design. If there is a need for re-engineering the controllability analysis indicates what parts need to be re-engineered.

12.2 Computational Aspects

The key issue in the design process is to reduce the complexity as much as possible. This is done by careful modeling and by imposing requirements on the behavior of the system that reduces the number of possible control laws.

In the modeling it is very important to divide the system into subsystems with as few interacting variables as possible. An open question is how the choice of the finite field affects the complexity of the model.

The computations described in this paper are done by a straightforward implementation of Buchberger's algorithm [26] in *Mathematica*[85]. The seven Gröbner bases needed in this example were computed in a total time of half an hour on an LX SparcStation, which is acceptable considering that the algorithm is designed for general polynomials. Since the problem domain is discrete there are a lot of optimizing possibilities to consider, e.g. extending existing algorithms for Boolean equations.

Even though it might be time consuming to compute many Gröbner bases it is important to notice that in each Gröbner basis calculation we get an indication if we have specified the design criteria enough. If the Gröbner basis equals identity, we know that we have imposed too many constraints and that there is no solution. Similarly, if the unknown variable is not linear we need to impose more design constraints.

An important aspect is that the control laws computed, easily can be translated into executable code. This means that once the system description and the design criteria are decided, the controller code could be generated automatically. Since the computation of the actuator signals only consists of evaluating a polynomial for the measured values of the system variables, the computation can be made very fast and is therefore suited for real time controllers.

Part III

General Aspects of Modeling of Discrete Event Dynamic Systems

Chapter 13

Approaches to Describing Discrete Event Dynamic Systems

The area of *discrete event systems* (DEDS) has increased in importance during the last years. The reason is the growing need of tools which support the development of the complex DEDS of today. The nature of DEDS is that complexity often grow very fast, and they are therefore difficult to analyze and verify.

Research of DEDS became a topic within the automatic control community in the beginning of the eighties. Since then several different approaches have been developed for DEDS, but none of these approaches has proven to be the ultimate choice. Therefore the research area is still open for new suggestions.

In chapter we will briefly discuss the main candidates of these approaches for DEDS. Three of them: Ramadge-Wonham, Petri nets and COCOLOG will also be described in more details in Chapter 15.

13.1 The Automata Theoretic Approach

The notion of *finite automata* was introduced by McCulloch and Pitts [60] in the mid-forties and since then the area of finite automata has been developed further by Moore and Mealy [64, 63] in the fifties.

In the area of automatic control, Ramadge and Wonham [76] initiated the *supervisory control theory* which is used for correctness issues in the control of DEDS. This theory was set in a simple, abstract framework of formal languages [47] and finite automata, chosen for conceptual simplicity and generality rather than modeling efficiency or computational tractability. See [76] for a survey of supervisory control theory. Properties of DEDS such as controllability [86] and observability [58] have been investigated in the theory of supervisory control.

To reduce complexity of synthesizing a control law for DEDS, horizontal [73] and vertical (hierarchical) [88] decomposition of the overall control task have been studied.

The framework of untimed DEDS has been extended with real time features. In this framework, *timed transition models* with *real time temporal logic* [67] are used for formal verification. See [66] for a survey of formal models and specifications for real-time systems.

Another extension to the area of DEDS is the notion of *infinite strings* and *Büchi automata* [75]. These are used for modeling infinite behavior and for analyzing e.g. absence of *deadlock* properties of these behaviors.

To further develop the modeling process of finite automaton, *statecharts* were introduced by Harel [42]. Statecharts support aggregation of state-machine models without combinatorial explosion. For a special class of the statechart formalism: *hierarchical state machines* (HSM), Brave and Heymann [11] have a suggestion of an algorithm solving the reachability problem of HSMs.

For a survey of the area discussed above, see [82] and [55].

13.2 The Petri Net Approach

Petri nets were introduced by Petri in 1962 [70] and have since then been used for modeling and analysis of DEDS. Petri nets were first used to give a theoretical framework for communication between concurrent systems e.g. asynchronous communication between computers, see Holt et al. [46]. For an introduction to Petri nets, see Peterson [69] and Murata [65].

Not until recently have Petri nets been used for analysis and synthesis of control laws for DEDS. Krogh and Holloway [52] and Ichikawa and Hiraishi [48] have introduced *controlled Petri net* (CPN) which is a Petri net extension, from which they have formulated theories and methods for supervisory control.

The dynamic of Petri nets can be viewed from a perspective analogous to linear algebra, where the dynamics are represented by a linear matrix equation. This approach has been formed in a control context by Giua, DiCesare and Silva [39], and by Li and Wonham [56, 57].

The notation of Petri nets is often extended to supports new types of systems [69], e.g. *Timed-Event-Graphs* (TEG) [2] is a class of Petri nets where time delays are included in the notation. To represent TEGs algebraically Cohen, Dubois, Quadrat and Viot have introduce the *max-plus* algebra [25]. Max-plus is a dioid algebra [2] which is the mathematical framework of the system theory of TEGs models. The TEG theory has many analogies of the theory for linear systems, but the applications of the theory are mainly concerned with performance and optimizations issues rather than feedback control. For a tutorial of max-plus and a survey of the system theory of TEG, see [24] and [2].

13.3 The COCOLOG Approach

In 1990 Caines and Wang [15] introduced COCOLOG, a conditional observer and controller logic for finite machines. COCOLOG is designed to adaptively compute control actions from observed outputs and inputs of the system. A COCOLOG controller can be regarded as an adaptive controller for finite machines.

The COCOLOG framework is based on first order logic. In COCOLOG *axioms* of the model, the measurements of the process, and the control strategies are used to generate *theories*. From these theories control action can then be derived. All computations are done with a *theorem prover* [8] called Blitzensturm [14].

For a recent introduction see [15] and the application of controlling an elevator [27].

13.4 Simulation and Perturbation Analysis

Perturbation analysis [40] deals with the problem of minimizing a function $J(\theta)$ which is some performance measure of the behavior of a DEDS, where θ denotes e.g. model parameters and control actions. Minimizing $J(\theta)$ is an optimization problem with large complexity in general, and since θ often takes discrete values optimization methods based on differentiating $J(\theta)$ are not useful. However, perturbation analysis extracts J's dependence on θ by analyzing a sample path (one behavior) of the system and how this path is "perturbed" for small variations on θ.

Perturbation analysis is used to analyze and optimize DEDS, see [16, 43]. For a research survey of the area see [17].

Chapter 14

Aspects of Modeling using Algebraic Methods

14.1 Introduction

In the previous parts of this thesis, we have used polynomials over finite fields [33, 35] to represent different quantities in modeling and analysis of DEDS. These quantities involve for example finite relations that encode system behavior in a model, or representations of results from some analysis applied to models. Polynomials are also used to represent objective specifications of closed loop behaviors when synthesizing a control law. This demonstrates that algebraic methods can be used in the theory and computation of DEDS.

Still, the complexity problem of DEDS cannot be overcome in general. In fact, some problems of DEDS are proven to be NP-complete [47], and therefore we cannot reduce the worst case complexity, regardless of representation.

As shown in [3] and in Part I in this thesis, there are cases where large and realistic complexity can be managed and both analysis and synthesis can be performed. Therefore we know that there are cases where it is possible to handle complex system with a satisfactory result. By using a representation that reduces memory and performance demands we will be able to handle a large class of systems. The polynomial approach developed by Germundsson [33, 35] has the potential to be a representation of DEDS that in most cases reduce complexity and at the same time be well suited for computer implementation.

In this chapter we will deal with some aspects of modeling using polynomials over finite fields. The purpose is to give some guidelines of how to make a DEDS model using polynomials. Before making a DEDS model, we must choose a suitable finite field $\mathbb{F}_q$. We should also try to separate the systems into subsystems to lower complexity. Guidelines that hopefully will make these choices easier will be presented in this chapter.

We will also reason about the differences of event and signal interaction with a DEDS. As shown in the previous parts of this thesis it is rather straightforward to represent signals with variables of a finite field. But it is not straightforward to convert signals to events and vice versa. We will give some useful hints of the process of converting system descriptions between event and signal interactions, see Section 14.4.

To do modeling directly with polynomials is not a realistic approach in everyday engineering work, and to make modeling more convenient we will have to use some kind of modeling language. At the end of this chapter we will point out the features we would like to see in a modeling language, but we will not make any suggestions about a new modeling language.

14.2 Polynomial Representation of DEDS

We will start this section by discussing the λ-polynomials from Section 3.7 more thoroughly, and then remodel the tank from Chapter 10 using different choices of finite field $\mathbb{F}_q$ to see how different choices affect the complexity of the resulting model.

14.2.1 Enclosing Subranges Variables in $\mathbb{F}_q$

In Definition 3.11 we introduced the polynomial $\lambda_q^p(z)$ as a polynomial in $R_q[Z]$, which is `true` only for values of $z \in \{0, \ldots, p-1\}$ where $p < q$.

As mentioned in Section 3.7 the λ-polynomial restricts the possible values for a variable z representing some quantity of the model for which only the first p values are valid. This is essential since there must not be any false relations in a polynomial model, i.e. relations not corresponding to the physical system. It is also important to be able to model systems with variables of different ranges.

Another important aspect is that the λ-polynomials often reduce degree of the polynomial representation of the model, which often means lower complexity. To be able to show this we will present a way to construct the λ-polynomials.

One way of constructing the λ-polynomial is to multiply factors representing the p first values of $\mathbb{F}_q$ as:

$$\lambda_q^p(z) = z(z-1)\cdots(z-p+1) \tag{14.1}$$

From (14.1) we see that

$$\deg(\lambda_q^p(z)) = p. \tag{14.2}$$

which is the minimum degree of a polynomial which has p zeros.

All λ-polynomials for variables $z_i, z_j, \ldots$ of smaller ranges are collected into the polynomial set $\Lambda_q(Z) = \{\lambda_q^{p_i}(z_i), \lambda_q^{p_j}(z_j), \ldots\}$.

Before modeling the behavior of a system into a polynomial we compute $\Lambda_q(Z)$ for the system. After that, all results from polynomial computations will be reduced with respect to the ideal generated by $\Lambda_q(Z)$. This corresponds to symbolic computations in the quotient polynomial ring $\tilde{R}_q[Z]$ defined as:

$$\tilde{R}_q[Z] = R_q[Z]/\langle \Lambda_q(Z) \rangle \tag{14.3}$$

For a variable z of smaller range we know from (14.2) that the highest degree of z in the polynomials in $\tilde{R}_q[Z]$ will be $p-1$. Therefore the λ-polynomials will reduce the degree of the subrange variables in the polynomials in $\tilde{R}_q[Z]$, which also reduces the complexity, see Example 3.7. If we replace (14.1) by some other method of constructing λ-polynomials we get the polynomial say $\tilde{\lambda}_q^p(z)$. The zeros of $\tilde{\lambda}_q^p(z)$ are of course the same as for λ_q^p but the degree might be greater than p.

The Gröbner basis $\mathcal{GB}_q(\tilde{\lambda}_q^p(z))$ will return the polynomial in $R_q[z]$ which represents these zeros of $\tilde{\lambda}_q^p(z)$ with the minimum degree, which is p_i. This means that we can use any method to construct a polynomial with the appropriate set of zeros and then use the Gröbner basis to get the corresponding λ-polynomial.

14.2.2 Aspects of Choosing $\mathbb{F}_q$

In the previous parts of this thesis we have modeled systems using polynomials in $R_7[Z]$. We have also used Boolean relations which can be regarded as polynomials in $R_2[Z]$. Modeling DEDS with Boolean equations is a well known technique from Moore and Mealy [64, 63].

To be able to indicate some conceptual differences between modeling a system in a smaller or larger field we remodel a simplified version of the tank from the Section 10.2 twice. First we do a more thorough modeling of the tank using $R_7[Z]$, compared to Chapter 10. Then we do the same using Boolean expressions.

By doing this remodeling manually we hope to see if there are some advantages using $R_7[Z]$ compared to the Boolean expressions. Even if polynomials is not realistic and practical for manual use, this comparison might indicate the potential in a modeling tool based on polynomials.

In this section we will use the simplified tank shown in Figure 14.1.

Example 14.1 For the tank in Figure 14.1 we will model the relation between the level of the tank x and the net flow into the tank ϕ. The definitions of x and ϕ are analogous to Section 10.2.

Since the level of the tank only takes seven different values we can let the variable x belong to the field $\mathbb{F}_7$ which is a natural choice of field since we then can

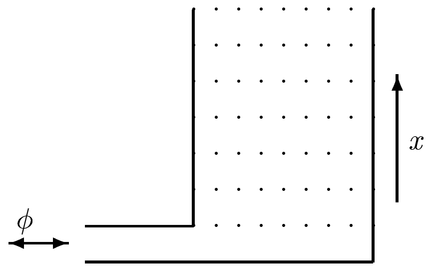


Figure 14.1: Simplified tank

express the level with only one state variable. Basically we now have the following state equation

$$x^{+} = x + (\phi - 1) \tag{14.4}$$

where $(\phi - 1)$ takes values from $\{-1, 0, 1\}$. (ϕ is defined as in Section 10.2.)

Equation (14.4) works fine for all levels except at the top ($x = 6$) and the bottom ($x = 0$), where x^{+} will take wrong values if ϕ is 2 or 0 respectively. To prevent this we add a condition $C(x, \phi)$ to the state space equation as

$$x^{+} = x + (\phi - 1) * C(x, \phi) \tag{14.5}$$

where $C(x, \phi)$ is a polynomial which is equal to 1 (`false`) for all values of x and ϕ except for the cases $\{x = 6, \phi = 2\}$ and $\{x = 0, \phi = 0\}$ for which $C(x, \phi)$ equals 0 (`true`). The condition $C(x, \phi)$ is defined as

$$C(x, \phi) = (x = 6) \wedge (\phi = 2) \vee (x = 0) \wedge (\phi = 0) \tag{14.6}$$

The polynomial representing $C(x, \phi)$ can now be computed using the rules described in Section 3.4.

$$\begin{aligned}
C(x, \phi) &= (x = 6) \wedge (\phi = 2) \vee (x = 0) \wedge (\phi = 0)\\
&= (x - 6) \wedge (\phi - 2) \vee (x) \wedge (\phi)\\
&= (1 - (1 - (x - 6)^6)(1 - (\phi - 2)^6)) \vee (1 - (1 - x^6)(1 - \phi^6))\\
&= (1 - (1 - (x - 6)^6)(1 - (\phi - 2)^6))(1 - (1 - x^6)(1 - \phi^6))\\
&= 5\,\phi + 3\,\phi^2 + 4\,\phi\,x + 3\,\phi^2\,x + 3\,\phi\,x^2 + 4\,\phi^2\,x^2 + 4\,\phi\,x^3 + 3\,\phi^2\,x^3 +\\
&\quad 3\,\phi\,x^4 + 4\,\phi^2\,x^4 + 4\,\phi\,x^5 + 3\,\phi^2\,x^5 + x^6 + 5\,\phi\,x^6 + \phi^2\,x^6
\end{aligned} \tag{14.7}$$

If we plug (14.7) into (14.5) and expand the result we get

$$\begin{aligned}
x^{+} =& x + 3\,\phi - 3\,\phi^2 - 3\,\phi\,x + 3\,\phi^2\,x + 3\,\phi\,x^2 - 3\,\phi^2\,x^2 - 3\,\phi\,x^3 + 3\,\phi^2\,x^3 +\\
& 3\,\phi\,x^4 - 3\,\phi^2\,x^4 - 3\,\phi\,x^5 + 3\,\phi^2\,x^5 - x^6 + \phi\,x^6
\end{aligned} \tag{14.8}$$

which is the same expression as (10.3) in Section 10.2.

Even if the resulting polynomial is complex and difficult to handle manually, the modeling method used is rather straightforward. In this case we formed the state space equation (14.4) simply by adding x and ϕ together. This was correct except for two cases for which the model was adjusted.

To compare the modeling performed above with modeling using Boolean expressions consider the following example.

Example 14.2 For the tank in Figure 14.1 we will model the relation between the level of the tank x and the net flow into the tank ϕ. The definitions of x and ϕ are analogous to Section 10.2.

Since the level of the tank takes 7 different values we must at least represent these levels with 3 Boolean state variables. We could also represented each level by one Boolean variable, which makes a total of 7 Boolean state variables. But if we want to reduce the number of variables in the model we choose the former approach.

Let the level x and the net flow ϕ be bit represented in $\bar{\mathbb{I}}_3$ (see Definition 6.7) as $x = [x_3, x_2, x_1]$ and $\phi = [\phi_2, \phi_1]$ where $x_i, \phi_i \in \mathbb{B}$. The levels of the tank are encoded as $[\mathtt{false}, \mathtt{false}, \mathtt{false}]$ for bottom level ($x = 0$) up to $[\mathtt{true}, \mathtt{true}, \mathtt{false}]$ for the top level ($x = 6$).

For the net flow ϕ we hope that by letting ϕ_1 and ϕ_2 correspond to one direction each we will simplify the manual composition of the expressions of the model. That is $\phi = [\mathtt{true}, \mathtt{false}]$ corresponds to increasing the level whereas $\phi = [\mathtt{false}, \mathtt{true}]$ corresponds to decreasing the level of the tank. With $\phi = [\mathtt{false}, \mathtt{false}]$ the level is constant.

We want to find the functions f_1, f_2, f_3 in the state space equation below.

$$[x_3^+, x_2^+, x_1^+] = [f_3(x_3, x_2, x_1, \phi_1, \phi_2), f_2(x_2, x_1, \phi_1, \phi_2), f_1(x_1, \phi_1, \phi_2)] \tag{14.9}$$

As in the previous example we will start by considering the increasing and decreasing behavior of the model. By adding two bit representations like $[x_3, x_2, x_1] + [\mathtt{false}, \mathtt{false}, \phi_1]$ we will get the appropriate expressions for the increasing behavior, and since -1 is represented by $[\mathtt{true}, \mathtt{true}, \mathtt{true}]$ we get the decreasing behavior by $[x_3, x_2, x_1] + [\phi_2, \phi_2, \phi_2]$. Therefore the expressions for both increasing end decreasing can be computed by $[x_3, x_2, x_1] + [\phi_2, \phi_2, \phi_1 \vee \phi_2]$. To convert the integer plus to Boolean expressions we use the function $BAdd$ from Definition 6.8,

$$[\tilde{f}_3, \tilde{f}_2, \tilde{f}_1] = BAdd([x_3, x_2, x_1], [\phi_2, \phi_2, \phi \vee \phi_2]) \tag{14.10}$$

which gives the following expressions:

$$
\begin{aligned}
\tilde{f}_3 =& (\phi_1 \wedge \phi_2 \wedge x_1 \wedge x_2 \wedge x_3) \vee (\phi_1 \wedge \phi_2 \wedge x_1 \wedge (\neg x_2) \wedge x_3) \vee \\
& (\phi_1 \wedge \phi_2 \wedge (\neg x_1) \wedge x_2 \wedge x_3) \vee (\phi_1 \wedge \phi_2 \wedge (\neg x_1) \wedge (\neg x_2) \wedge (\neg x_3)) \vee \\
& (\phi_1 \wedge (\neg \phi_2) \wedge x_1 \wedge x_2 \wedge (\neg x_3)) \vee (\phi_1 \wedge (\neg \phi_2) \wedge x_1 \wedge (\neg x_2) \wedge x_3) \vee \\
& (\phi_1 \wedge (\neg \phi_2) \wedge (\neg x_1) \wedge x_2 \wedge x_3) \vee (\phi_1 \wedge (\neg \phi_2) \wedge (\neg x_1) \wedge (\neg x_2) \wedge x_3) \vee \\
& ((\neg \phi_1) \wedge \phi_2 \wedge x_1 \wedge x_2 \wedge x_3) \vee ((\neg \phi_1) \wedge \phi_2 \wedge x_1 \wedge (\neg x_2) \wedge x_3) \vee \\
& ((\neg \phi_1) \wedge \phi_2 \wedge (\neg x_1) \wedge x_2 \wedge x_3) \vee ((\neg \phi_1) \wedge \phi_2 \wedge (\neg x_1) \wedge (\neg x_2) \wedge (\neg x_3)) \vee \\
& ((\neg \phi_1) \wedge (\neg \phi_2) \wedge x_1 \wedge x_2 \wedge x_3) \vee ((\neg \phi_1) \wedge (\neg \phi_2) \wedge x_1 \wedge (\neg x_2) \wedge x_3) \vee \\
& ((\neg \phi_1) \wedge (\neg \phi_2) \wedge (\neg x_1) \wedge x_2 \wedge x_3) \vee ((\neg \phi_1) \wedge (\neg \phi_2) \wedge (\neg x_1) \wedge (\neg x_2) \wedge x_3) \\
\tilde{f}_2 =& (\phi_1 \wedge \phi_2 \wedge x_1 \wedge x_2) \vee (\phi_1 \wedge \phi_2 \wedge (\neg x_1) \wedge (\neg x_2)) \vee \qquad (14.11) \\
& (\phi_1 \wedge (\neg \phi_2) \wedge x_1 \wedge (\neg x_2)) \vee (\phi_1 \wedge (\neg \phi_2) \wedge (\neg x_1) \wedge x_2) \vee \\
& ((\neg \phi_1) \wedge \phi_2 \wedge x_1 \wedge x_2) \vee ((\neg \phi_1) \wedge \phi_2 \wedge (\neg x_1) \wedge (\neg x_2)) \vee \\
& ((\neg \phi_1) \wedge (\neg \phi_2) \wedge x_1 \wedge x_2) \vee ((\neg \phi_1) \wedge (\neg \phi_2) \wedge (\neg x_1) \wedge x_2) \\
\tilde{f}_1 =& (\phi_1 \wedge \phi_2 \wedge (\neg x_1)) \vee (\phi_1 \wedge (\neg \phi_2) \wedge (\neg x_1)) \vee \\
& ((\neg \phi_1) \wedge \phi_2 \wedge (\neg x_1)) \vee ((\neg \phi_1) \wedge (\neg \phi_2) \wedge x_1)
\end{aligned}
$$

The expressions $\tilde{f}_1, \tilde{f}_2, \tilde{f}_3$ correspond to an ordinary counter in the interval 0-7. To adjust the behavior at the bottom an top level we add appropriate constraints to ϕ_1, ϕ_2 by substituting the entries of ϕ_1, ϕ_2 in $\tilde{f}_1, \tilde{f}_2, \tilde{f}_3$. The substitution rules are

$$\rho = \left\{ \begin{array}{l} \phi_1 \rightarrow \phi_1 \wedge \neg(x_3 \wedge x_2 \wedge \neg x_1) \\ \phi_2 \rightarrow \phi_2 \wedge \neg(\neg x_3 \wedge \neg x_2, \wedge \neg x_1) \end{array} \right\} \qquad (14.12)$$

and with ρ we can now compute the final state space expressions

$$[f_3, f_2, f_1] = [\tilde{f}_3, \tilde{f}_2, \tilde{f}_1]|_\rho \qquad (14.13)$$

We will not present f_3, f_2, f_1 explicitly here, but instead make the observation that f_3, f_2, f_1 will be rather complex and impenetrable, and to compute (14.11) without using a computer is not a tempting task.

It is not fair to draw any precise conclusions of the modeling of one system only. But we have seen that in this case the polynomial approach with $R_7[Z]$ gives simpler computations and model. This indicates that there are cases were a larger field is more appropriate to use.

In Section 11.1 we presented the function $J(x)$ as a weighting function that assigned different weights to the states of the plant. With this function it was possible to compute a control law that reduced the present weight to a minimum. To introduce a similar weighting function in a model based on Boolean expressions will be much harder. It seems that the arithmetic operations following from the use of a lager field, are essential to simplifying both the modeling and design efforts.

Theoretically we know that there are always cases which have the worst case complexity independently of the choice of field. Still, we hope that most of the real applications are not worst cases, and that they are modeled efficiently using appropriate polynomial approach with a good choice of the finite field $\mathbb{F}_q$.

14.3 Modeling Description Language

14.3.1 Introduction

The mathematical domain that we use for modeling is polynomials over finite fields. To model systems directly with polynomials is not realistic except for trivial cases, since the polynomial representation is hard to interpret manually. As stated in Section 4.2.2 the modeling is best performed in an appropriate *model description language* (MDL) which preserves the behavior of the system correctly.

We will discuss what necessary and preferable properties we would like to have in an MDL, by discussing MDLs in general and by some examples where we use Pascal and Petri nets. This is a large research area with many results and suggestions of MDL aspects, see the following overview. In this section though, we will only reason from the experiences received from the modeling of the landing gear controller and the tank in Part I & II in this thesis, and the section can therefore be regarded as an introduction to future research and studies in the area.

No MDL will be nominated as the best choice in this thesis. Instead we would like to focus on the perspective of modeling languages, where the modeling aspects are more important then design or synthesis aspects although we have to take precaution to include these aspects as well.

14.3.2 An MDL Overview

There are many suggestions of MDLs, and the number of proposals are increasing. The suggestions originates from a variety of applications and disciplines like applications in manufacturing industries, telecommunication and integrated circuits.

Some of the MDLs like Grafcet [19], Simulink, Pascal and C are used as engineering tools for developing control systems and are widely used in the industries. Other MDLs like VHDL [49], ASA [83], SDL [18] are supported by tools performing verification of designs modeled in the languages. The verification can be performed both formally and by performance measures.

There are also MDLs originating from the research area like Statecharts [42], SIGNAL [6], ESTRELLE [7], and LUSTRE [41]. Results on object-orientation for modeling can be found in Zeigler [87]. For a comparative survey of different MDLs applied to a design problem see [23].

MDLs are used in all stages in development, e.g. ASA for specification, Simulink for testing and analysis and Pascal for implementation. See [36] for a study of avaliable MDL tools at Saab Military Aircraft and how these tools support the development process in different stages.

All these MDLs support different features that are important for a good modeling environment. But the main objective of an MDL is to give an exact description (or model) of the behavior of the system. If this is fulfilled and if the model is a finite system, i.e. that all quantities of the model are of finite domains, then it is possible to translate the MDL model into the polynomial domain without introducing discrepancies [35]. The mapping from an MDL to a corresponding polynomial representation must capture the system behavior described by the MDL representation exactly.

14.3.3 Modeling of Physical DEDS

In order to compare MDLs, we need to specify which type of systems are interesting in the perspective of modeling. There are systems where modeling is not necessary since the systems we want to model already are defined in some sort of formal language. Therefore it is not useful to remodel these to some intermediate modeling language. The language defining a system might not be the most suitable for translation but these difficulties must be solved even if an intermediate language is used.

In the case of the tank from Chapter 10 we have another situation. The tank is a physical system with physical quantities. In general and for the tank in particular, a physical system is best described by a continuous model. The water level and moving parts of pump and valves in the tank system follows physical laws in the continuous domain. Therefore the exact behavior of the tank is best modeled by methods of continuous or heterogenous domains.

However, if we have the discrete perspective of the tank we know that the discretization of this system is straightforward due to the discrete nature of the pump and valves. Therefore the system can be viewed from the controller as a pure discrete system. Still the discretization will not give us a formal description of the discrete behavior of the landing gear and we are therefore forced to do modeling.

In the rest of this section we will use the term *physical DEDS* (PDEDS) which denotes continuous systems well suited for discretization.

Pascal

In Chapter 6 we used a piece of Pascal code to describe the controller of the landing gear of the fighter aircraft JAS 39 Gripen. As pointed out in Section 8.1 Pascal is not suited for describing a DEDS. What are the main characteristics in Pascal that makes it a bad choice as an MDL for DEDS? This questions can at least be answered in three ways:

1. *Pascal is not a pure language for finite domains.*
   This is of course a trivial observation, since we in Pascal have data types for real valued variables (approximated with a bounded representation), which is not a member of a finite domain.

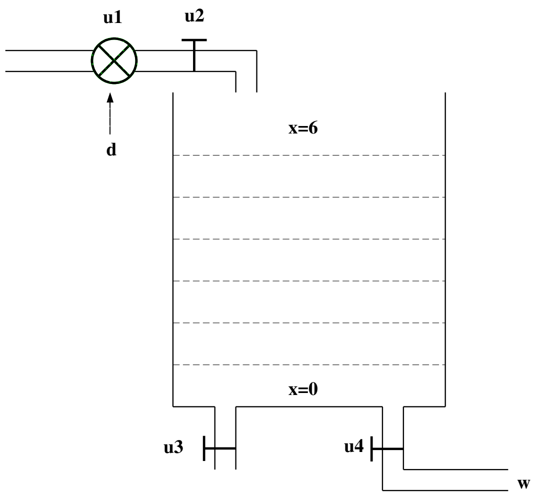
2. *Pascal is a language that is meant to be used for synthesis of systems.*
   Many computer languages like Pascal or C are used for system development and not as a modeling tool. There are very few, if any, languages that are intended purely for modeling.

3. *Pascal is a sequential language.*
   As a consequence of the synthesis nature of Pascal the statements are written down in a sequence which have some practical engineering advantages, but when doing modeling of a PDEDS we find it rather confusing since these systems often contains concurrent events.

These observations of problems using Pascal as an MDL give us some motivations for necessary features of more useful MDLs.

To illustrate essential criteria for a good MDLs we use the following example.

Example 14.3



If we write a model of the tank from Chapter 10 using the subset of Pascal presented in Section 6.2.3, one solution is:

```
CASE d OF
        0 : pump := FALSE;
        1 : pump := u1;
        2 : pump := TRUE
END;
```

```
inflow :=  pump AND u2;

outflow := u3 OR ( u4 AND w);

phi := 1;

IF inflow AND (Not outflow) THEN
        phi := 2
ELSE
        IF (NOT inflow) AND outflow
                phi := 0;

CASE phi OF
        0 : IF x > 0 THEN
                x :=  x - 1;
        1 : ;
        2 : IF 6 > x THEN
                x :=  x + 1;
END;
```

The Pascal code gives the same behavior and interacts with the signals in the same way as the real process shown in the figure above.

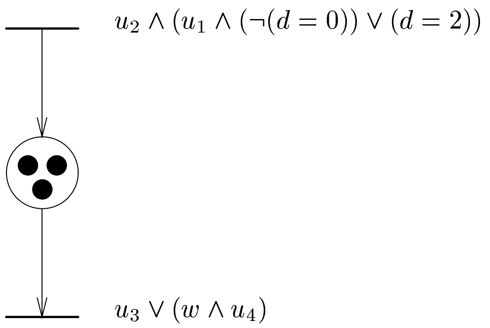
From Example 14.3 above we can conclude that it is very difficult to see any structural similarities between the Pascal code and the physical system. If we only had access to the Pascal code as a description of the system it would be hard to see that the actual system is a tank.

Petri Nets

The problem mentioned above might be solved with a graphical MDL, where graphical symbols represent different parts of the system. Note that specifying the structure of a system using ordinary block diagrams will not be classified as an MDL, since these diagrams must be used together with another language to specify the behavior. In general it is not easy to find a graphical language that gives good structural and behavioral descriptions at the same time.

Consider the example below where we model the tank using Petri nets [69].

Example 14.4 To simplify the modeling of the tank using Petri nets we assume that the tank is infinitely high. The resulting Petri net model will then be like the figure below. The figure shows the Petri net model with initial level $x_0 = 3$ (marked by three tokens). The transitions (bars at the top and bottom) represent inputs to the Petri net and are assigned to logical conditions. These conditions depend in turn on the input signals of the tank. Every time the condition for a transition becomes true a token will pass through the transition.



Further details of how to model the tank using Petri nets is presented in Section 15.3.

This example shows that the graphical language of Petri nets can in this case model the tank in a more straightforward way than Pascal. The structure of the system and the physical interpretation can at least with some amount of imagination be notified in the Petri net model. Water is coming from above into the tank controlled by some logical conditions, and the outflow is controlled by a condition below the tank.

The logical conditions that control the in- and outflow are modeled directly using the interpretation of the valves, the pump and the disturbances as presented in Chapter 10.

A problem though with the Petri net in Example 14.4 is that the physical information of how the valves and the pump are connected is lost. Moreover, if we for instance modify the tank by connecting an extra pipe u_5 between the pump and the valve u_2 (see Figure 14.2), it is not clear how to modify the Petri net model in example 14.4.

Modifications in systems should be easy to apply in the MDL describing the system. In case of Figure 14.2 we added a device to the system which would correspond to adding two models together in the MDL. This feature must be fulfilled by a good MDL.

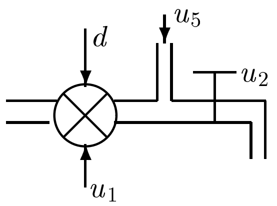


Figure 14.2: Modified inflow construction.

Bond Graphs

The perfect MDL should be a description that is in some sense as close to the physical system as possible to make modeling easier and making the model support simpler. The same needs for an MDL as presented here can be identified in the domain of modeling continuous systems. The most common way of modeling a continuous system today is to identify the mathematics behind the primitive elements in the system and then to compose these into a complete mathematical model. In this way all physics behind the mathematics is hard to interpret in the model, and the model is not easily adjusted for changes in the physical system.

In this area the *bond graph language* [68, 51] is a suggestion of how to model a system without going into the mathematical details at once. The bond graph language can be automatically compiled to a mathematical form. This is essentially the same as to compile an MDL model for DEDS to the polynomial domain.

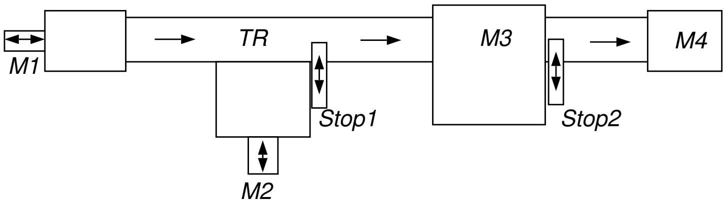
The bond graph language is composed of a small number of primitive elements that are connected in a graph. The functionality of the system is described by this graphical language at the same time as the overall structure and topology of the system can be preserved. In this sense the bond graphs gives inspiration to what features we would like to have for the perfect MDL.

Bond graphs have been used with great success for continuous systems [12]. Strömberg et al. [81] introduced switched bond graphs in order to handle discontinuities. This was done by adding a new primitive element *the switch* to the bond graphs language. Each switch element introduces a binary state taking the values *effort* and *flow* that can be interpret as on/off for switch. Actually, a bond graph consisting of only switches can be regarded as a model of a DEDS. This means that a pure switched bond graph is an MDL.

The pure switched bond graph may be a starting point for the search for the perfect MDL in the sense that principles of the bond graphs should be adopted even in the MDL. But it would not be practical to use primitive elements of such a low level as binary switches. The ability of the MDL in modeling complex systems will then be lost.

The main idea of bond graphs is that subsystems are connected by interchanging energy, and that causalities for the energy interchanging subsystems are decided first after the interconnection. This means that we can have a general description of a subsystem that is valid for all different types of surrounding systems. The corresponding feature in the MDL case is that we can build a general model for a DEDS subsystem independently of the context in which this model is going to be used. We will give an example of this by using a PDEDS consisting of a production line.

Example 14.5 At the Dept. of AC. in Linköping there is a LEGO car factory laboratory process [80]. This factory assembles parts of a car in a specified sequence. The control sequences are synthesized directly and written into a Grafcet editor that compiles the code into PLC code. For an overview of the factory see the figure below.

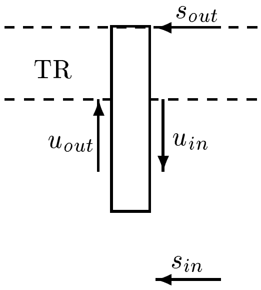


The factory consists of four machines, namely *M1*, *M2*, *M3* and *M4* working along an assembly line *TR*. There are two stoppers *Stop1*, *Stop2* positioned after *M2*, *M3* respectively. The production of a car follows the sequence:

1. The machine *M1* delivers a car to the conveyor *TR*.
2. When the car reaches the front of *M2*, *Stop1* blocks the conveyor and *M2* puts a roof on the car.
3. The stopper *Stop1* releases the car.

Similar actions will be taken by *M3* and *Stop2* when *M3* fixes the roof on the car. The machine *M4* is a storage of completed cars.

The factory consists of several actuators working together and influencing each other depending of the situation. Let us look closer at one of the stoppers.



The signals interacting with the stopper are the inputs u_{in} and u_{out} and the sensors s_{in} and s_{out}.

The stopper is essentially analogous to the valve in a pipe since it stops the flow (of cars) on the assembly line. But this analogy will not hold for the case when the stopper smashes into a bypassing car. This case is an error case and if we want a good model of the stopper this behavior must be included in the model. This would not be the case in the context of fluid material.

The example shows that interaction between the stopper and its environment decides how the interacting models should be mapped to mathematical description of polynomials. Compare this with acausal property of bond Graphs were the computational causality is first decided by global rules applied on the bond graph (see [68, 81]).

Two stoppers positioned right after each other will have the same effect as two valves on the same pipe. If we are to compute control laws as in Section 11.2 for such a system we need an extra constraint that excludes one of the combinations of the stoppers states to give an unambiguous control law (see 11.2.2 for further details). The problem of finding extra constraints automatically for a complete model will be as complex as to compute a single explicit control law directly. Instead it might be possible to compute local extra constraints between two subsystems connected to each other in the MDL model. Combining[1] local extra constraints together will then give the necessary reduction of the solutions space to give an unambiguous control law. This shows that to be able to automate the modularization of analysis and synthesis, it is necessary for MDL to preserve the structure of the system. For MDLs that preserves the structure of the system, it will be easier to do analyzing and synthesizing modularized to reduce complexity.

To conclude this section we will repeat the desirable features[2] of an MDL.

(i) The MDL should be based on finite domains.

(ii) The MDL should be graphically represented.

(iii) The MDL should preserve the structure of the modeled system.

(iv) Sequential description of the behavior should not be necessary for the MDL.

(v) Abstraction and modularity should be supported in the MDL.

(vi) The MDL should support a smart context dependent interconnection of general devices.

(vii) The MDL should support a user interface intended for system modeling, not for system design and synthesis.

Further research will show if there is a defined language that fulfills the specification above or if it has to be invented or if such a language cannot exist.

14.4 Event and Signal Models

In this section we will discuss events and event driven models compared to models where inputs and outputs are signals. Even though the system

[1] Compare this with computing local control laws from local specifications.
[2] This is a wish list.

behind the models is the same, an event driven model and a signal model might look quite different. The reason for this will be discussed below.

First we will define what we mean by signals and events.

Definition 14.1

(i) A *signal* is a quantity that has a value. This value is measurable at all times. Changes of the signal value can occur at any time instant with no notification.

(ii) An *event* is a notification (message) present only at a time instant. Events can contain information.

□

Typically, events give information of changes, whereas signals keep track of the present values.

We will in the following discuss transformation of systems from signal models to event models and vice versa.

14.4.1 Going from Signals to Events

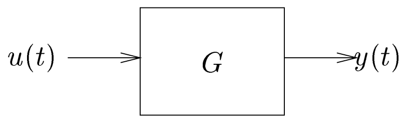


Figure 14.3: A continuous system.

We will make some analogies with continuous systems. Consider the system G in Figure 14.3, where $u(t)$ is the continuous input signal and $y(t)$ is the continuous output signal.

If we want to transform G to a model G' that instead of signals interacts with signals changes, we will have a the system presented in Figure 14.4.

We see in Figure 14.4 that by adding an integrator to the inputs and a differentiator to the outputs we can derive G' from G. Note that the number of states will increase with the number of input signals. These states can be regarded as input observers for the system G.

For DEDS systems we can reason the same way. If we want to create an event driven model M' for a DEDS system represented by a signal model M, we will have to add finite automata for every input signal that can observe events corresponding to the signals.

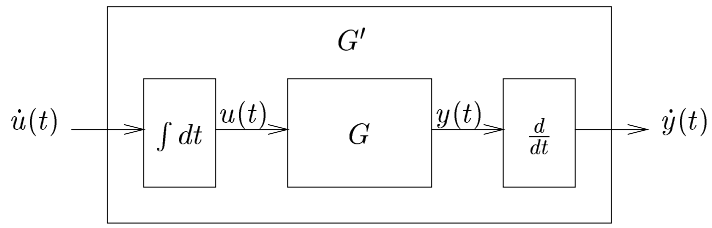
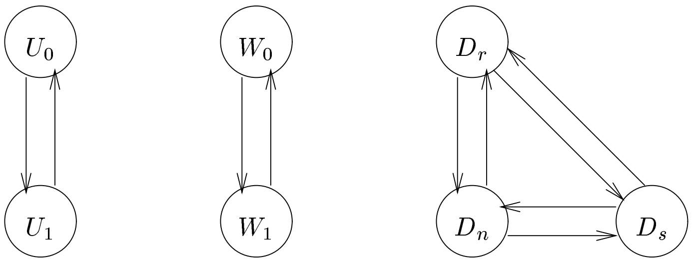


Figure 14.4: A system interacting with signal changes.

Example 14.6 The tank in Chapter 10 have three input signals u_1, d and w. To make an event model we have to represent each possible value change in these signals by a corresponding event. These events are presented in the following table:

Signal	**Events Σ_I**	**Event States**
u_1	$u_1(\mathit{off}), u_1(\mathit{on})$	U_0, U_1
w	$w(\mathit{stop}), w(\mathit{open})$	W_0, W_1
d	$d(\mathit{stuck}), d(\mathit{normal}), d(\mathit{run})$	D_s, D_n, D_r

The states needed to observe the input events are presented in the third column of the table above. The resulting automaton will have the structure as shown in the figure below:



where the labels of the transitions are omitted but can easily be deduced from the table above.

We see that the number of states are increased by a factor 12 ($2 \times 2 \times 3$) for the event driven model.

To transform model outputs from signals to events might in some cases decrease the number of states in the model, since the only purpose of these states is to capture the output events and generate the output signals. However, if the outputs from the event driven model of the tank in Example 14.6 are the inflow and outflow events, the output states (level in the tank) are still necessary since the output events depends on the level.

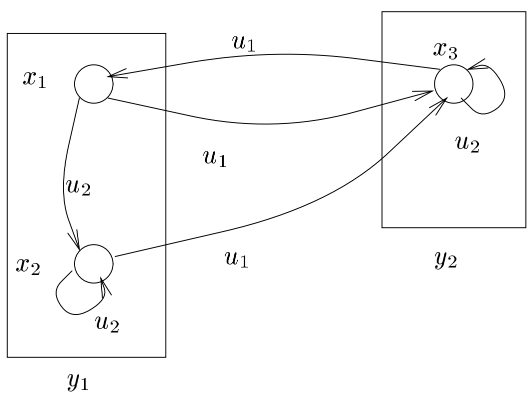
Example 14.6 indicates that the transformation from signal models to event driven models, increases the state dynamics for the inputs in the systems. The effect of this is also shown in sections 15.2.2 and 15.3.2 where finite automata and Petri nets are used to model the tank system.

14.4.2 Going from Events to Signals

We will now discuss the opposite problem: How to represent event driven models by signal models. In fact, we have already done this in Section 4.2 where finite automata were modeled by polynomials.

Consider the following example:

Example 14.7 Let the Moore machine [64] M_1 be defined by the figure below



where the variables x_1, x_2, x_3 are Boolean state variables, u_1, u_2 are Boolean input variables representing the input events and y_1, y_2 are Boolean output variables.

The state transition relation is given by

$$\begin{aligned} f(x^+, x, u) = x_1^+ &\leftrightarrow (x_3 \wedge u_1) \wedge \\ x_2^+ &\leftrightarrow (x_1 \wedge u_2 \vee x_2 \wedge u_2) \wedge \\ x_3^+ &\leftrightarrow (x_1 \wedge u_1 \vee x_2 \wedge u_1 \vee x_3 \wedge u_2) \end{aligned} \tag{14.14}$$

and the output relation is given by

$$\begin{aligned} g(y, x) = y_1 &\leftrightarrow (x_1 \vee x_2) \wedge \\ y_2 &\leftrightarrow x_3 \end{aligned} \tag{14.15}$$

where $x = [x_1, x_2, x_3], u = [u_1, u_2]$ and $y = [y_1, y_2]$.

The relations $f(x^+, x, u)$ and $g(y, x)$ are relations of variables that take the values `true` and `false`. In this sense f and g represent a signal model of a event driven system. How is this possible?

The input variables of the model represent one input event each. Each one of these variable has a value that corresponds to “no event”. Therefore it is possible to evaluate the model even for the case of no events.

By using $R_3[Z]$ instead of the Boolean representation for the system in Example 14.7, we would get the input variable $u' \in \mathbb{F}_3$ where the values correspond to events as

u'	**Event**
0	No event
1	u_1
2	u_2

This means that if we evaluate the model with the input $u' = 0$ no state transitions will be made.

To include symbols for “no events” is often recommendable in general. It is particularly important if the model shall be implemented in a simulator that evaluates the model periodically independent of event occurences.

14.5 Modeling Methodology

14.5.1 Introduction

In this section we will discuss modeling aspects for systems with actuators and sensors that might give failures which alter the behavior of the systems. For supervisory control, proper failure models are essential since we are interested in control laws that guarantee that certain forbidden states never are reached in spite of possible failures in the system. In fact, most efforts in developing control systems today is to assure as much as possible the robustness of failure cases [5].

To achieve complexity advantages it is necessary to divide the models into submodels as much as possible. How to do this modularization depends of course in the use of the models and on the structure of the system that is modeled.

14.5.2 The EFG-Model

In the following we will focus on how to modularize a system related to sensor and actuator failures. Consider the system in Figure 14.5 where y and u denote the output and input of the system. The signal v represents all types of unmeasurable and/or uncontrollable events of the system. This means that we regard u as controllable inputs, i.e. we have actuators for u. For the same reasons y is regarded as measurable outputs from the system.

We will try to divide the system disturbances v into smaller subsets, by separating the actuator and sensor failures from the system. To do this we

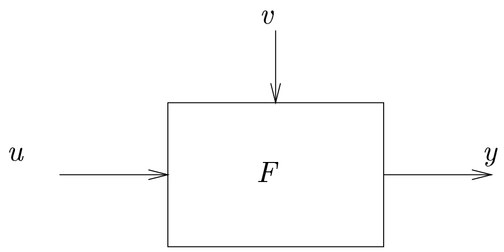


Figure 14.5: A system with disturbances.

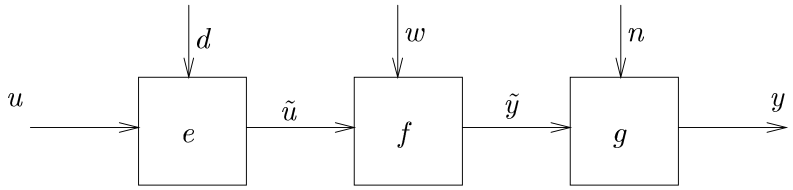


Figure 14.6: The EFG-model

propose a modularization of the system F into a model consisting of three sub systems e, f and g shown in Figure 14.6. This model structure will be denoted EFG-model.

The input u and the output y are the same as for F in Figure 14.5. The f-system represents the *nominal* system, which includes the behavior of F when no failures on actuators and sensors are present. The e-system is a filter that includes a model representing the possible disturbances on the actuators. The best example of this is the pump in the tank system from Chapter 10 where we assume that the pump has two failure modes: stuck and running. The dynamics of the e-system is driven by the input of failure events d, and the output from the e-system is the disturbed control value $\tilde{u}$. The g-system filters the outputs from the nominal system and includes models for how the failures on sensors affect the outputs from the system F. The input of failure events n drives the dynamics of the g-system similarly as for the e-system.

14.5.3 Modular Synthesis

By the EFG modularization we specify explicitly what we regard as the nominal system f, sensor failure model g and actuator failure model e. From the EFG-model we can now specify our control objectives for the nominal system and make local analysis and synthesis of the nominal model to deduce the effects of the uncontrollable events in w. Then when an appropriate closed loop behavior is found for the nominal system we add the e or g systems and redo the analysis of the system. Hopefully, this way of solving smaller problems first to simplify the larger problems will ease the computational load in general.

14.5.4 Failure Observers

The structure of the EFG-model encourages us to find a method to observe the disturbances d, w and n. These observations would drastically improve the power of the feedback control. The observer for an EFG-model should include the models for the e, f and g systems, and from observation on u and y be able to generate at each time instant a set of possible values of the disturbances d, w and n.

There are similar approaches for observers, e.g. by Sampath et al. [77], and by the COCOLOG framework [15], see Section 15.4. Since observers must be implemented online in the real time environment, an efficient observer based on the polynomial approach will be preferable.

This will be the topic of further research.

Chapter 15

Comparative Reviews of the Tank Model

15.1 Introduction

As described in Chapter 13 there are several theories and methods for modeling, analysis and synthesis in the domain of DEDS. Even though these approaches are different, they often share concepts since they often solve similar type of problems.

In this chapter we will focus on three different DEDS approaches from the area of automatic control: Ramadge-Wonham [76], Petri nets [70] and COCOLOG [15]. By remodeling the tank from Chapter 10 using these approaches, we get a measure of how appropriate it is to use polynomial methods for modeling of DEDS, compared to these three established approaches. The purpose is not to make the comparison complete in all details, e.g. we will not deal with aspects of computational performance, but to concentrate on the main features of methods in these approaches.

The common issue of the chosen approaches is that their main objectives are analysis and synthesis of supervisory control laws for DEDS, i.e. control laws that prevents the system to reach forbidden states. The system (or the model) is often assumed to be given in an appropriate way for the framework of the theory. This assumption makes sense since translations of a DEDS models from one framework to another can often be performed directly. Still, the process of building models of physical systems is not always that obvious.

In this chapter we will start from a physical system (the tank from Chapter 10) and make a model that can be applied to control law synthesis if possible for each framework mentioned above. In particular we will see that modeling is not always straightforward in various formalisms.

15.2 The Ramadge-Wonham Approach

Ramadge and Wonham (RW) [76] introduced the supervisory control theory for DEDS in the mid-eighties. This theory is based on finite automata and formal language theory and supports methods and algorithms for solving correctness issues on control of DEDS.

The tank from Chapter 10 will be modeled as a *finite automata* (FA) [47] following principles of converting signals to events that we discussed in Section 14.4. We will also reason about the problem of modeling behavior of a system that is influenced by real time. Since we are only using theory for logical DEDS (DEDS with no time description) we have to use other techniques.

Before we can go further into these details we give a short introduction of the theory of Ramadge-Wonham.

15.2.1 Introduction to the RW Theory

In a logical model of a DEDS we are interested in the sequence of events which the process can generate as outputs or accept as inputs. All these sequences of events form the *language* of the DEDS. The theory of Ramadge-Wonham contains different characteristics of these languages of events that aim to build control laws that can disable certain events from occuring.

A more formal overview of common quantities follows.

Definition 15.1

(i) Let Σ denote finite set of events (alphabet) that labels the state transitions of the model.

(ii) Let Σ^* denote the set of all finite strings of elements of Σ include the empty string ϵ.

(iii) The set of all admissible, i.e. physically possible strings of a system is denoted L. The term sample path is often used instead of strings. Since L contains all possible sample paths for a system, L specifies or defines the behavior of the system.

(iv) A string z is a *prefix* of a string $v \in \sigma^*$ if for some $w \in \Sigma^*$, $v = zw$.

(v) The *prefix closure* of $L \subseteq \Sigma^*$ is the language

$$\overline{L} = \{z : zv \in L \quad \text{for some } v \in \Sigma^*\}$$

(vi) If $\overline{L} = L$ we say that L is prefixed closed.

□

If L is a prefixed closed language then all prefixes of the strings in L are members of the language L.

We will only model DEDS represented by a prefixed closed language L over the alphabet Σ, where each $z \in L$ represents a possible (partial) event sample path of the DEDS.

Let the automaton or "generator" representing the plant be denoted $\mathcal{M}$.

Definition 15.2

(i) Let Q denote the set of all states of $\mathcal{M}$.

(ii) Let $\delta : \Sigma^* \times Q \to Q$ be a transition function defined as

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

whenever $\tilde{q} = \delta(w, q)$ and $\delta(\sigma, \tilde{q})$ are defined.

(iii) Let $\delta(w, q)!$ be an abbreviation for "$\delta(w, q)$ is defined". In terms of the graph of $\mathcal{M}$, $\delta(w, q)!$ simply means that there is a path in the graph starting from q that is labeled by the consecutive elements of the string w.

(iv) The closed behavior of $\mathcal{M}$ is defined to be the prefixed closed language

$$L(\mathcal{M}) = \{w : w \in \Sigma^* \quad \text{and } \delta(w, q_0)!\}.$$

□

To introduce a control mechanism Ramadge and Wonham postulate that certain events of the system can be disabled (i.e. prevented from occuring) when desired.

Definition 15.3

(i) Partition the set of events Σ into *uncontrollable* and *controllable* events: $\Sigma = \Sigma_u \cup \Sigma_c$, Σ_u and Σ_c are mutually exclusive. The events in Σ_c can be disabled at any time, while those in Σ_u cannot.

(ii) A *control input* for $\mathcal{M}$ consists of a subset $\gamma \subseteq \Sigma$ satisfying $\Sigma_u \subseteq \gamma$. If $\sigma \in \gamma$, then σ is *enabled* by γ (permitted to occur), otherwise σ is *disabled* by γ (prohibited from occuring).

(iii) Let $\Gamma \subseteq 2^{\Sigma}$ denote the set of control inputs.

(iv) A *supervisor* is a map $f : L \to \Sigma$ specifying the control input $f(w)$ for each event in all strings $w \in L$.

□

If we apply the supervisor map f to the system we get the closed loop system.

Definition 15.4 The language L_f of the closed loop system is defined as

(i) $\epsilon \in L_f$ and

(ii) $w\sigma \in L_f$ iff $w \in L_f$, $\sigma \in f(w)$, and $w\sigma \in L$.

□

To define the control objectives, Ramadge and Wonham use a sublanguage to define the specification of the closed behavior.

Definition 15.5

(i) Let the sublanguage $K \subseteq L$ denote the desired closed loop behavior.

(ii) The sublanguage K is said to be *controllable* if the condition

$$\overline{K}\Sigma_u \cap L \subseteq \overline{K} \tag{15.1}$$

is satisfied.

□

When composing a desired closed loop behavior it is often useful to intersect languages of subspecifications.

Lemma 15.1 ([76]) If the languages $K_1, K_2 \in \Sigma^*$, then

$$\overline{K_1 \cap K_2} \subseteq \overline{K_1} \cap \overline{K_2} \tag{15.2}$$

and if K_1, K_2 are *prefix closed* languages then

$$\overline{K_1 \cap K_2} = \overline{K_1} \cap \overline{K_2} \tag{15.3}$$

□

Ramadge and Wonham has found a necessary and sufficient condition to determine when it is possible to find a supervisory control law f.

Theorem 15.1 ([76] and [74]) For nonempty $K \subseteq L$ there exists a supervisor f such that $L_f = K$ iff K is prefix closed and controllable. □

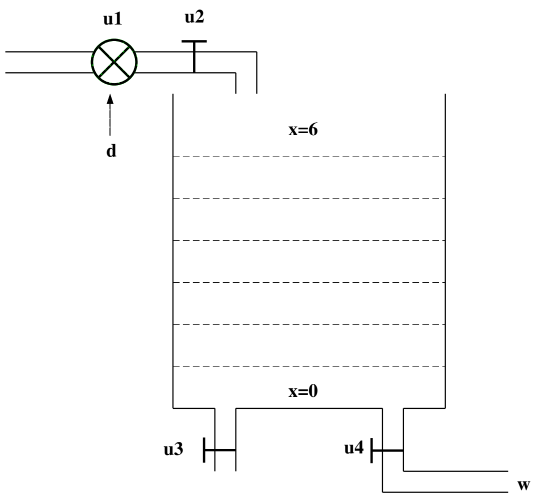


Figure 15.1: The tank

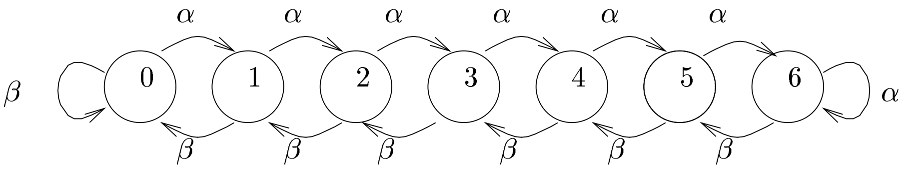


Figure 15.2: FA model of the tank.

15.2.2 The FA Model of the Tank System

To demonstrate the Ramadge-Wonham approach and to point out some special aspects that might occur in modeling a DEDS, we remodel the tank in Figure 15.1 using the RW approach.

When we did the control law synthesis of the tank using the polynomial approach (see Section 11.1) we introduced a priority among the actuators. The priority was set so that the valves u_2, u_3 and u_4 only were used in emergency cases. In this modeling we will exclude the valves represented by the signals u_2, u_3 and u_4. Instead we hope that some of these valves will be determined as necessary in the control law synthesis for fulfilling our objectives, i.e. the some of the valves are needed to get a controllable system with respect to the states control objectives.

The actual synthesis will not be presented in this thesis, but for comparative view of synthesis in the approach of RW and the polynomial approach see the thesis of Plantin [71].

The notations and interpretation of FA models can vary to some extent, and it is not intuitively clear which type of FA is the best for modeling the tank or for the RW framework. Throughout this section we will try simple things first and if necessary adjust the models or the notation.

In the previous efforts of tank modeling (see Section 10.2) we have considered the inputs to the system to be regarded as signals. Keeping the notion of signals we could make the automaton shown in Figure 15.2 where

$$\begin{aligned} \alpha &= u_2 \wedge ((d=2) \vee u_1 \wedge \neg(d=0)) \\ \beta &= u_3 \vee (u_4 \wedge w) \end{aligned} \tag{15.4}$$

are predicates that must be true to make a state transition possible. This model is not yet a pure event driven model since (15.4) are functions of signals and therefore α and β also are signals. (See Section 14.4.) We also need a more detailed description of how the the tank interacts with the input signals if the model should be used for synthesizing a control law.

To make a pure event driven FA model we have to change the interpretation of the inputs. Following the guidelines in Section 14.4 we associate each value of each signal with a unique event that represents a signal change to that value. For the tank we denote the set of these events as Σ_I (see Table 15.1).

Signal	**Events** Σ_I	**Event States**
u_1	$u_1(\mathit{off}), u_1(\mathit{on})$	U_0, U_1
w	$w(\mathit{stop}), w(\mathit{open})$	W_0, W_1
d	$d(\mathit{stuck}), d(\mathit{normal}), d(\mathit{run})$	D_s, D_n, D_r

Table 15.1: Events and states of tank input signals.

The events presented above represent signal changes. This means that the model must remember the most recent event for each signal. Therefore the model must include a FA for each signal. For the three signals u_1, w, d we get the FAs in Figure 15.3.

With the three FAs specified by the Figure 15.3 and Table 15.1 we now have a model for the inputs to the tank. In order to proceed we must decide how to specify the behavior that these FAs give to the system.

For each combination of states of the FAs in Figure 15.3, at most one of the events α and β is possible. To state the exact behavior of the tank we form one FA including the total behavior of the input events using the *cross product* (see [47]) on the FAs in Figure 15.3. This FA will have 12 states and is illustrated as a table of state combinations of the FA for u_1, w and d (see Table 15.2). The table also indicates for which state combinations the level of the tank increases or decreases (denoted by α and β respectively) following the predicates in (15.4). Table 15.2 describes all 12 states of the total input

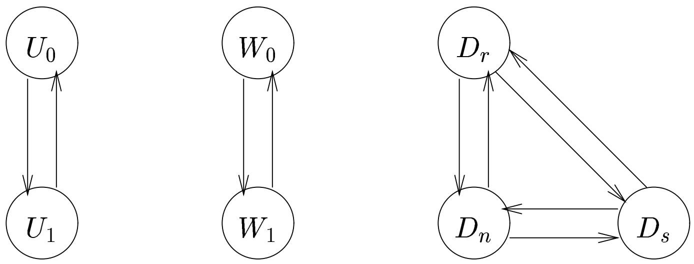


Figure 15.3: FAs modeling the tank input. The arc corresponds to the events give in Table 15.1.

FA, where no information of how the arcs are connected is presented, but the structure of the arcs is trivially deduced from Figure 15.3.

States of d	**States of** u_1 **and** w			
	U_0		U_1	
	W_0	W_1	W_0	W_1
D_r	α	-	α	-
D_n	-	β	α	-
D_s	-	β	-	β

Table 15.2: Total state space for input FAs where α and β denotes the net flow in the tank.

In the Table 15.2 we have indicated for which state the events α and β should happen. We must now decide how these events shall be represented properly by the input FA.

Let us assume that the events α, β are generated from the FA of Table 15.2 formed as a Mealy machine [47, 63] in the way shown in Figure 15.4, where we take a closer look at the pair of states $\{D_n, U_0, W_0\}$ and $\{D_n, U_1, W_0\}$.

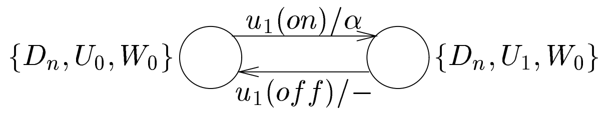


Figure 15.4: Mealy FA sending event α in one of the transitions. The labels of the transitions are *input/output*.

If we label each arc going to a state that increases the level of the tank with an event α (and conversely for decreasing) we will get the behavior which increases the tank level every time we go into an increasing state. But what will happen if we remain in an increasing state? The physical behavior of the tank is that when the input states give an increasing net flow, the tank will eventually be full. How fast or slow this happens is essentially a matter of time, and we could of course use a time event called *tick*[1] to represent a certain elapse of time necessary to increase the level one unit. For an α-state the tick events will make transitions towards state 7 in Figure 15.2, but for a β-state the *tick*-events will make transitions toward state 0.

We have until now tried to create a FA that generates the events α, β. This is the most intuitive approach since α, β are internal events in the tank that are generated from events on the actuator of the tank. But for RW this is not the most appropriate way to model the behavior. The RW approach regards all events (external or internal) to be generated by some "universal" event generator. From where an event is generated is not the issue for RW. Instead RW uses FAs to specify which strings of events are accepted by the system. It is therefore only interesting to specify the possible sequences of events in a system. This is done by using incomplete FAs, i.e. FAs where some transitions of events are missing.

We use the input FA to enable or disable α and β. In this way we do not have to actively generate α and β. We only specify for which input states α and β are allowed to happen. For the pair of states in Figure 15.4 we get the result showed in Figure 15.5, where only α are allowed to happen in the state $\{D_n, U_1, W_0\}$, and none of α, β are allowed to happen in state $\{D_n, U_0, W_0\}$.

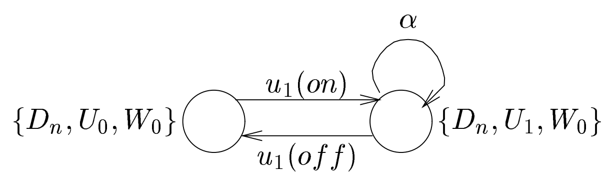


Figure 15.5: FA enabling event α in one of the states.

By adding an arc as in Figure 15.5 for each state in Table 15.2 that is indicated with an α or a β, we finally get a pure event FA describing how α and β depend on the input events. The resulting FA is denoted $\mathcal{M}_I$.

We modify the tank level FA in Figure 15.2 by adding all input events Σ_I as possible events for each state. This will not make this FA defined for

[1]The tick events are introduce in framework of "Timed Discrete Event Systems" [10].

all possible events. If this modification is not done this FA will forbid all occurences of events in Σ_I. The resulting FA is denoted $\mathcal{M}_T$ and is shown in Figure 15.6.

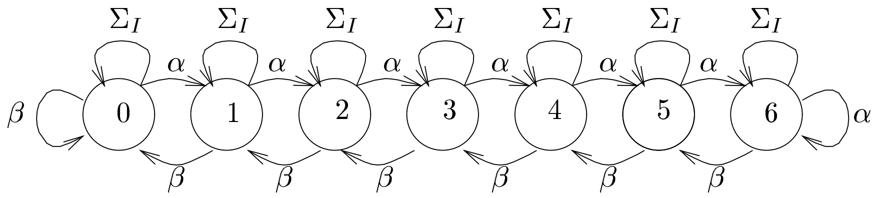


Figure 15.6: $\mathcal{M}_T$: Final FA model for the level dynamics. Σ_I denotes the set of all input events.

We can now consider the FA models $\{\mathcal{M}_I, \mathcal{M}_T\}$ as FA descriptions of the system in Figure 15.1. It would be possible to combine these FAs using cross product to a single FA denoted $\mathcal{M}$ with a total number of 84 states[2], since $\mathcal{M}_I$ has 12 states and $\mathcal{M}_T$ has 7 states. But we will not gain any immediate advantages by doing that and it is also hard to present $\mathcal{M}$ graphically.

We know from the the modeling using polynomials in Section 10.2 that by finding ways to separate systems into subsystems often reduces complexity. In fact, in this case $\{\mathcal{M}_I, \mathcal{M}_T\}$ correspond in a sense to the polynomials $\{f_1, f_2\}$ respectively, whereas $\{\alpha, \beta\}$ represent the net flow variable ϕ.

The language of the events describing the behavior of the tank represented by the total FA $\mathcal{M}$ can now be generated using the FAs $\{\mathcal{M}_I, \mathcal{M}_T\}$.

15.2.3 The Language Model and Control Objective of the Tank

A language for a DEDS contains all strings of events that influence the system. From the section above we have two FAs $\{\mathcal{M}_I, \mathcal{M}_T\}$ from which we shall deduce the language L describing all the possible system events. In the following we will find that only $\mathcal{M}_I$ is needed for L, whereas $\mathcal{M}_T$ gives inspiration to the specification of the closed loop behavior K. More about this later in this section.

Deducing L

The alphabet of events of the tank system is

$$\Sigma = \Sigma_I \cup \{\alpha, \beta\} \tag{15.5}$$

[2]The cross product result in a FA where the number of states are equal to the product of number of states for the input FAs.

From this alphabet event strings are generated that run the FAs $\mathcal{M}_I, \mathcal{M}_T$. For $\mathcal{M}_T$ we have transitions for all events in Σ at all states. This means that the language of $\mathcal{M}_T$ is

$$L(\mathcal{M}_T) = \Sigma^* \tag{15.6}$$

which is the language of all possible combinations of events in Σ. Therefore, M_T has no restrictions on the language and is in this case not interesting.

The language accepted by $\mathcal{M}_I$ cannot accept α, β in all states, and therefore $L(\mathcal{M}_I) \subset \Sigma^*$. On the other hand M_I has no restrictions on events in Σ_I which means that we can make the following observation

$$\Sigma_I^* \subset L(\mathcal{M}_I). \tag{15.7}$$

Since $L(M_I) \subset L(M_T)$ we can specify the language L accepted by the FA $\mathcal{M}$ to be

$$L = L(\mathcal{M}) = L(\mathcal{M}_I) \tag{15.8}$$

In general the language L depends on the initial state of the system, but in this case L is independent of the initial state of the tank. However, the initial state becomes important when specifying the desired behavior K, see the following.

The property $\overline{L} = L$ is fulfilled for our tank since we have

$$uv \in L \Rightarrow u \in L.$$

Therefore $\overline{L} = L$ which means that L is a *prefix closed* language.

The alphabet Σ can be divided in controllable and uncontrollable events as

$$\Sigma_c = \{u_1(\mathit{off}), u_1(\mathit{on})\} \tag{15.9}$$

$$\Sigma_u = \Sigma \setminus \Sigma_c \tag{15.10}$$

Deducing K

To express our control objectives of the tank from Section 11.3.3 as a language we use a FA. The FA $\mathcal{M}_T$ now becomes important since we can easily apply the constraints saying that the level of the tank must not reach $x = 0$ or $x = 6$ (using the notations from Section 11.3.3). By simply removing the end states and the corresponding arcs from $\mathcal{M}_T$ we get the FA denoted M_K shown in Figure 15.7.

The FA $\mathcal{M}_K$ restricts the maximum length of consecutive events α or β. But we also realize that the set of allowed strings of $\mathcal{M}_K$ depends strongly of the initial state of $\mathcal{M}_K$. We let the initial state be $q_0 = 3$ which means that we always start with the level in the middle.

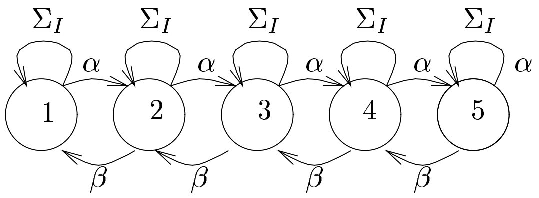


Figure 15.7: FA $\mathcal{M}_K$ specifying the non forbidden behavior.

We actually skipped some details above when we specified L, since we did not specify the initial state. Nevertheless the language L will be independent of which initial state in M_T we choose, since the set of accepted strings of events will be independent of the initial state of the tank and an initial value of M_I is not appropriate since the language should include any initial state of the input signals to the tank. Therefore (15.8) is a valid definition of L independent of the initial state of the tank. Still, the closed loop system must be initialized properly, i.e. we have to start the controller with the middle level in the physical tank, if K is specified from the initial state $q_0 = 3$.

The language of desired behavior of the tank is the sublanguage $K \subseteq L$ accepted by the FA $\mathcal{M}_K$ initialized with $q_0 = 3$, i.e.

$$K = L(\mathcal{M}_K|_{q_0=3}) \cap L \tag{15.11}$$

Since both $L(\mathcal{M}_K)$ and L are closed languages we know from Lemma 15.1 that

$$\overline{K} = \overline{L(\mathcal{M}_K|_{q_0=3})} \cap \overline{L} \tag{15.12}$$

which gives $\overline{K} = K$, i.e. K is a *prefix closed* language. This property of K is essential if we want to find a control law for the tank.

To conclude the modeling of the tank we note that the quadruple

$$\{L, K, \Sigma_c, \Sigma_u\} \tag{15.13}$$

specifies the language of the tank, the objective of the desired behavior, controllable events and uncontrollable events. From (15.13) it is possible to make synthesis of a supervisory control law. Note that the first thing to do in the synthesizing process is to check whether (15.13) is controllable or not. In fact, in this case we will find, by using the constraint (15.1) in Definition 15.5, that we need to make some of the uncontrollable events in Σ_u controllable. This is of course obvious from the physical system, since we cannot assure our control objectives without using some sort of security valves added to the system.

15.2.4 Summing Up

By doing a model intended for the Ramadge-Wonham approach we get some understanding of how it works, and how this modeling correlates with the polynomial methods used in Chapter 10. The main difference between these approaches depends on the difference between signals and events. As we have seen above the effect of transforming a signal model to an event model is that events represents signal changes and therefore every signal must be represented with a FA as an event observer. This may of course increase the number of states rapidly. For the polynomial approach we measure the signal values directly without input observers. The complete FA $\mathcal{M}$ for the tank contains 84 states compared to 7 for the polynomial approach.

If we use the polynomial approach to model a pure event driven system instead, we can always keep the number of states the same, i.e. the number of states in the polynomial model and in the event driven FA will be equal. In this case the values of input signals represent the events of the system, and each evaluation of the polynomial model corresponds to an occurrence of an event. In this case the transformation from the polynomial model to a FA needs no observer FAs of the input signals, since the signal already represents events. But these characteristics of the signal are not true for the physical tank model.

A language $L(\mathcal{M})$ of a FA $\mathcal{M}$ contains strings of sequential events. If two events may occur at the same time instant, we have to denote this event couple to a new event symbol if the assumption that all events can be ordered sequentially shall hold. For the polynomial approach it is possible to let different input variables change values independently of each other. This may in some case be an advantage in modeling non-sequential system.

Neither Ramadge-Wonham nor the polynomial approaches are intended to be used without powerful computer tools. But the manual modeling shown here using FAs is not, at least for the tank, easier to do and to understand compared to the polynomial modeling presented in Section 14.2.

15.3 Petri Nets

Petri nets were introduced by Petri in 1962 [70] and have since then been used for modeling and analysis of DEDS. Petri nets were first used to give a theoretical framework for communication between concurrent systems e.g. asynchronous communication between computers, see Holt et al. [46].

Today the theory of Petri nets is used for general types of DEDS systems. Even DEDS with an infinite state space can be represented by a finite number of Petri net primitives. This is one of the main advantages of Petri nets in modeling.

The notation of Petri nets is often extended to supports new types of systems [69], e.g. *Timed-Event-Graphs* [2] is a class of Petri nets where time delays are included in the notation.

Not until recently have Petri nets been used for analysis and synthesis of control laws for DEDS. Krogh and Holloway [52, 45] have introduced *controlled Petri nets* (CPN) which is a Petri net extension, from which they in the domain of CPNs have formulated theories and methods for supervisory control.

As mentioned in Chapter 13 there are many restricted classes of Petri nets designed for different classes of systems and problems. Powerful theoretical results can often only be derived for these restricted Petri nets and there are few theoretical tools for general Petri nets.

In this section we will introduce the notion of general Petri nets and then use these to remodel the tank from Chapter 10. This modeling will be compared to the modeling performed in Section 14.2.2 where we used the polynomial approach.

15.3.1 Introduction to Petri Nets

We begin by a formal introduction of Petri nets, adapting the notion of Peterson [69].

Definition 15.6 A *Petri net* (PN) is a four-tuple $C = (P, T, I, O)$.

$P = \{p_1, \ldots, p_n\}$ is a finite set of *places*, $n \geq 0$.

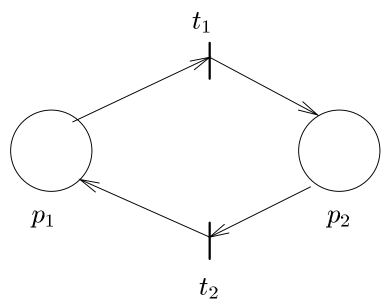
$T = \{t_1, \ldots, t_m\}$ is a finite set of *transitions*, $m \geq 0$.

The set of places and the set of transitions are disjoint, $P \cap T = \emptyset$. $I : T \to P^\infty$ is the *input* function, a mapping from transitions to bags[3] of places. $O : T \to P^\infty$ is the *output* function, a mapping from transitions to to bags of places. □

A Petri net is a graph where the nodes represent the places, the bars represent the transitions and the arcs represent the inputs and outputs of the transitions. The graph is similar as a standard FA (finite automaton) [47].

Example 15.1 A switch with two states (on,off) is represented by the Petri net $C_1 = (P, T, I, O)$ show in the figure below.

[3]A bag is a generalization of sets, which allows multiple occurences of elements. The set of all bags of places is denoted P^∞.



where $P = \{p_1, p_2\}$ and p_1, p_2 representing "on" and "off" respectively and $T = \{t_1, t_2\}$ and t_1, t_2 representing the events "switching off" and "switching on" respectively. For the input and output function we get:

$$
\begin{aligned}
I(t_1) &= \{p_2\} \\
I(t_2) &= \{p_1\} \\
O(t_1) &= \{p_1\} \\
O(t_2) &= \{p_2\}
\end{aligned}
\tag{15.14}
$$

The state of a PN is given by its *marking*, which is the distribution of *tokens* in the state places. The tokens are used to define the execution of a PN. A token denoted $\bullet$ can be viewed as an object that jumps from one place to another (or to the same) by passing through one of the transitions in T.

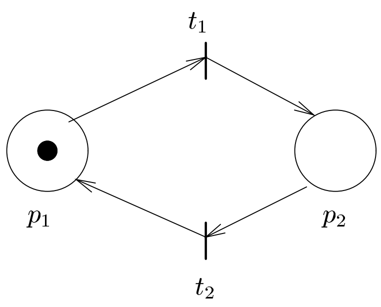
The marking of a PN gives the number of tokens for every place in the PN. Formally this is defined as

Definition 15.7 The *marking of a place* in a PN $C = (P, T, I, O)$ is a function $m : P \to \mathbb{N}$, where $m(p)$ is the number of tokens in place p. □

Definition 15.8 The *marking of the Petri net* $C = (P, T, I, O)$ is a function $M : P \to \mathbb{N}^n$, where $M(P)$ is the vector $[m(p_1), \ldots, m(p_n)]$.

$\mathcal{M}$ denotes the set of all markings of a PN. □

Example 15.2 Initiating the PN C_1 of the switch in Example 15.1 with a token like



means that the switch is "on".

The marking of the PN will be

$$M(P) = [1, 0] \tag{15.15}$$

If the transition t_1 *fires* ("switching off") then we will have the marking $M(P) = [0, 1]$.

As mentioned in the example above a Petri net changes marking by firing transitions. The firing of a transition corresponds to the occurrence of an event in the standard DEDS terminology.

A transition in a general Petri net can fire if it is *state enabled*, which means that all state places which are inputs to a transition must contain at least as many tokens as there are arcs between the place and the transition. When a transition fires, tokens are removed from the input places and added to the output places.

This will be described more formally by the following rules:

(i) A transition $t \in T$ is state enabled and allowed to fire if

$$m(p) \geq \#(p, I(t)) \quad \text{for all } p \in I(t) \tag{15.16}$$

where the function $\#(p, I(t))$ gives the number of occurrence p in the bag $I(t)$.

(ii) A firing of a transition $t \in T$ will decrease the number of tokens in the input places as

$$m'(p) = m(p) - \#(p, I(t)) \quad \text{for all distinct } p \in I(t) \tag{15.17}$$

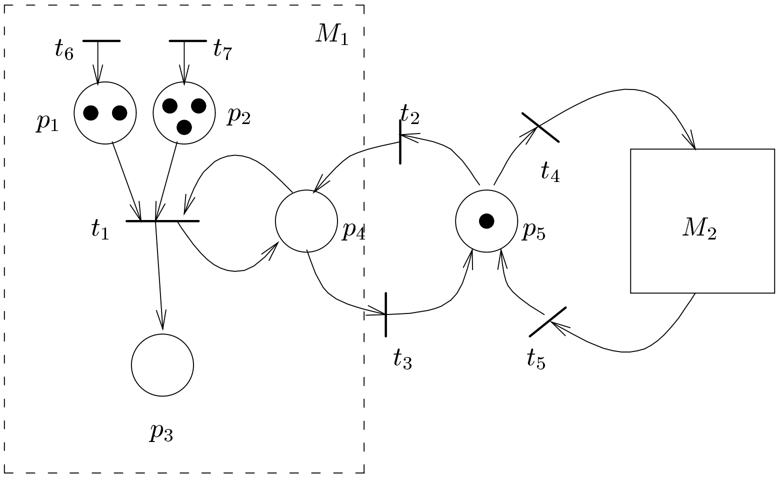
where $m'(p)$ denotes the new number of tokens in place p.

(iii) A firing of a transition $t \in T$ will increase the number of tokens in output places as

$$m'(p) = m(p) + \#(p, O(t)) \quad \text{for all distinct } p \in O(t) \tag{15.18}$$

To illustrate executions of Petri nets we use an example.

Example 15.3 Consider two equal transmitters M_1 and M_2 charing the same communication channel using mutual exclusion. The transmitters receive messages and message labels (identification of messages) and combine these to a message package that is transmitted on the channel. To transmit a package the transmitter must have a message, a message label, and access to the channel. A Petri net model of this system is shown in the figure below where the details of the the transmitter M_2 are omitted.



p_1	Arrived messages for M_1
p_2	Arrived message labels for M_1
p_3	Packages send from M_1
p_4	M_1 has access to the channel
p_5	The channel is idle
t_1	Package sent by M_1
t_2	M_1 allocates the channel
t_3	M_1 deallocates the channel
t_4	M_2 allocates the channel
t_5	M_2 deallocates the channel
t_6	A message arrives for M_1
t_7	A message label arrives M_1

The Petri net is initialized with 2 messages and 3 message labels arrived at M_1, and the channel is initialized to idle.

A place can contain any number of tokens in a Petri net. As show in the previous example this can be used to model, e.g. buffering of an arbitrary amount of objects (in this case messages). This means that the state space may not be finite for a Petri net and therefore there are systems that can be modeled by Petri nets but not by methods using finite domains like the polynomial approach.

Still, the description of Petri nets is finite since it consists of a finite number of places and transitions. This means that we can use polynomials over finite fields to represent the Petri net, instead of representing the behavior of the system modeled by the Petri net. By doing this it is possible to formulate algorithms for Petri net graphs in an algebraic framework and thereby gain computational advantages. There are other algebraic approaches for Petri nets or for extended Petri nets, e.g. Max-Plus algebra [2].

In Example 15.3 above we modeled the incoming of messages and message labels with transitions without input places. Such transitions can in a sense be regarded as the input of events to the system modeled by the Petri net. This construction of system inputs described by [69] will be used consequently in the modeling of the tank later in this section.

We make the following definition.

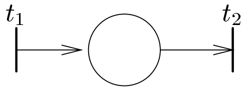
Definition 15.9

(i) A *source transitions* $t_u \in T$ is a transition such that $I(t_u) = \emptyset$, i.e. t_u shall not have any input places.

(ii) A *sink transition* $t_y \in T$ is a transition such that $O(t_y) = \emptyset$, i.e. t_y shall not have amy output transitions.

□

Example 15.4 In the figure below



t_1 is a source transition and t_2 a sink transition.

With source and sink transitions defined we now have all we need to make a model of the tank in Figure 15.8 using general PN.

15.3.2 The Petri Net Model of the Tank

We will make a Petri net model of the tank from Chapter 10 shown in Figure 15.8.

As in the modeling presented in Section 15.2.2 we will exclude the valves represented by the signals u_2, u_3 and u_4, since these are emergency actuators which should be included as a consequence of control law synthesis.

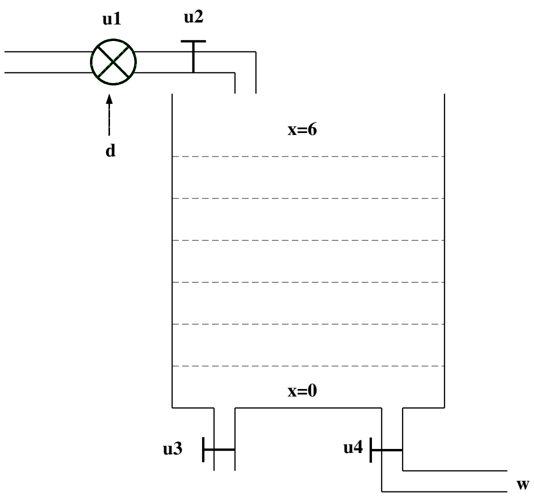


Figure 15.8: The tank

In analogy with previous modelings of the tank we will start by the modeling of the increasing and decreasing of the water level.

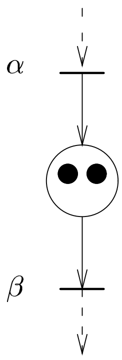


Figure 15.9: PN model of infinite tank.

Since a Petri net place can model accumulating buffers as described above, we will try to model the tank as in Figure 15.9 where α is the event of an inflow one unit (one token) and β is the event of an outflow one unit. Figure 15.9 above indicates that the level is initiated to 2. The dashed arrows indicate the connection with the rest of the PN model, which will be discussed later.

The Petri net in the figure above gives a model of tank which can accumulate an infinite amount of water. But since the level of the tank has an upper limit, we must find a way to maximize the number of possible tokens in the PN to 6. The inspiration to the solution can be found in Example 15.3.

The tokens in the place representing the channel can be viewed as the number of available channel resources, i.e. if we instead have n channels we simply initiate the place with n tokens to get an appropriate PN model. With the same reasoning for the tank we can use a place representing the available space in the tank. The PN is now as in Figure 15.10 which is initiated with the level 2 ($m(x) = 2$) and with 4 available units ($m(\overline{x}) = 4$) in the tank. By this construction we get a model with the maximum level 6 which corresponds to the tank in Figure 15.8.

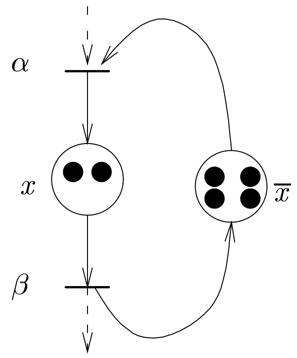


Figure 15.10: PN model of a finite tank.

We have derived a model for the dynamics of the level in the tank. This model shall now be connected to PN models of the pump and the uncontrollable outflow w.

Peterson [69] presents a modeling method for PN. The method concentrates on two primitive concepts: *events* and *conditions*. Events are actions that occur in the system and the state of the system can be described as a set of conditions. A model of a system shall describe when an event is allowed to occur i.e. decide the conditions corresponding to the event. These are the *preconditions* of the event. The occurrence of the event may cease the precondition to hold and may cause other conditions, *postconditions* to hold. Conditions are associated with places in a PN and events are associated with transitions.

Given the set of conditions and events for a system, the PN model for the system can easily be derived. (To find the appropriate conditions and events for a system may of course not be that easy.)

We will explain this further by modeling the tank outflow w, using this method. First we make a list of conditions (labeled $a, b, \dots$) and events (labeled $1, 2, \dots$).

The conditions for the outflow are:

Outflow condition	
a	Outflow
b	No outflow
c	Outflow requested
d	No outflow requested

The events for the outflow are:

Outflow event	
1	Outflow starts
2	Outflow stops

From the lists above we can deduce the preconditions and postconditions for every event as:

Event	**Precondition**	**Postcondition**
1	b, c	a
2	a, d	b

By associating each condition with a place and each event with a transition with input and output places given by the table above, we will get the Petri net graph in Figure 15.3.2 initiated to the state "no outflow".

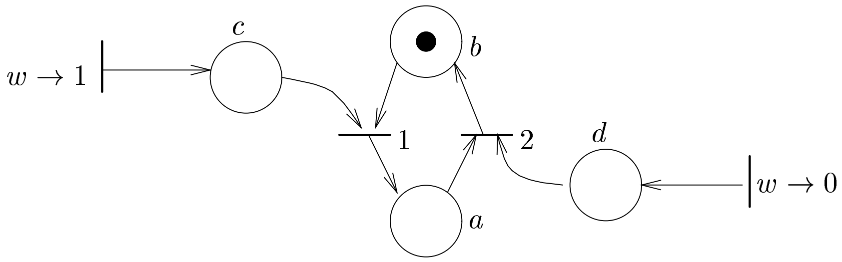


Figure 15.11: PN model for the outflow.

The PN model in Figure 15.3.2 corresponds to the switch discussed in Example 15.2. The main difference is that we have a well defined input interface to the model consisting of two source transitions labeled $w \to 0$ and $w \to 1$ and the places c and d. The labels for the source transitions indicate when the transitions fires, e.g. the label $w \to 0$ means the event when w goes from 1 to 0.

The modeling method used above will be even more useful when modeling the pump. The pump is controlled by the binary signal u_1, but the pump is also disturbed by the three valued signal d. In this case we will have three conditions for each mode of the disturbance and two conditions corresponding to "pumping" and "not pumping" when the pump is in a non failure mode. We will also have request conditions similarly as to the PN of the outflow w made above.

The set of conditions for the pump are:

Pump conditions	
e	Pump ON
f	Pump OFF
g	Pump idle
h	Pump stuck
i	Pump running
j	Pump ON request
k	Pump OFF request
l	Pump idle request
m	Pump stuck request
n	Pump running request

The events for the pump are:

Pump events	
3	Pump turned ON
4	Pump turned OFF
5	Pump is repaired
6	Pump failure stuck
7	Pump failure running

From the lists above we can derive the preconditions and the postconditions for the pump as:

Event	**Precondition**	**Postcondition**
3	f, j	e
4	e, k	f
5	$(l, h) \lor (l, i)$	g
6	g, m	h
7	g, n	i

This derivation is similar as for the outflow w except for event number 5 that must be represented by two transitions corresponding to the logical OR-operation. The OR-operation indicates that the pump can be repaired in two ways, i.e. repaired from stuck failure mode or repaired from running failure mode.

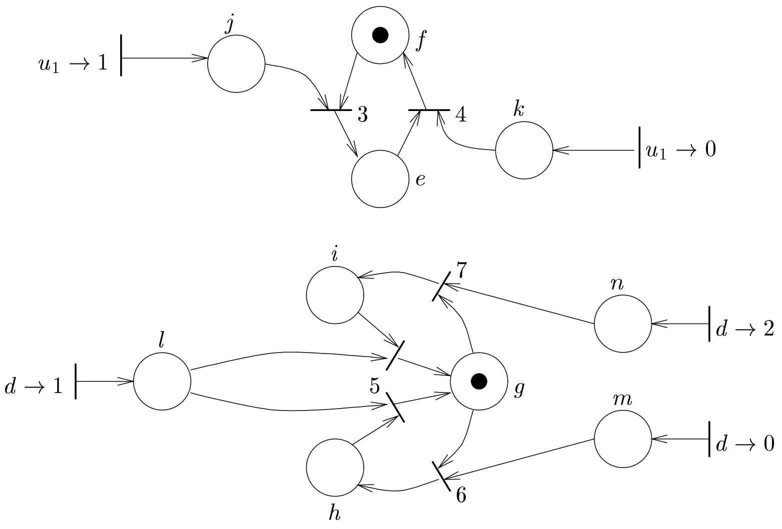


Figure 15.12: Two separate PN models. The first represent the the pump behavior of events on the actuator u_1. The second represents failure behavior of the pump.

.

Figure shows the Petri net of the pump derived from the table above.

As we can see in Figure 15.3.2 the Petri net is divided into two separate nets. The reason for this is that we have not modeled the actual inflow. We have only considered the ON and OFF states and the failure modes of the pump. But we have not connected these conditions to decide for which cases the pump contributes with an inflow into the tank.

We will now perform the modeling of the this inflow, and thereby connect the Petri net models for the tank, outflow and the pump. It seems reasonable that new conditions and events are not necessary. In fact, one major feature of Petri nets is that it is easy to combine Petri nets together using the transitions as an interface.

The conditions needed for the modeling of the inflow and the outflow are:

Conditions	
a	Outflow
e	Pump ON
g	Pump idle
i	Pump running
x	Level in tank
$\overline{x}$	Available in tank

which are previously defined.

The events for the inflow and outflow are as before:

Events	
α	Increasing the level
β	Decreasing the level

From the tables above we get the following pre- and postconditions:

Event	Precondition	Postcondition
α	$(e, g, \overline{x})_1 \vee (i, \overline{x})_2$	$(e, g, x)_1 \vert (i, x)_2$
β	a, x	$a, \overline{x}$

The α-event in the table above is divided in two transitions since we have an OR-operation. This means that the inflow events can be generated in two ways, i.e. one when the is pump in normal mode and one when the pump is failure mode.

Since transitions consume tokens from the input places, we have to include preconditions that must hold after the event in the set of postconditions. This means that for the α event in the table above, each precondition in an OR-statement corresponds to a postcondition. The precondition $(e, g, \overline{x})_1$ corresponds to the postcondition $(e, g, x)_1$ which means that if the α-event was enabled due to that the pump was ON and idle then the pump remains ON and idle after the firing of the α-event. The same is true for i with the precondition $(i, \overline{x})_2$ and the corresponding postcondition $(i, x)_2$.

The final Petri net model of the tank system will now be presented in Figure 15.13 where two sink transitions are added to represent the output from the state of the system. The sink transitions indicate if the level increases or decreases.

The same system was modeled using automaton (FAs) in Section 15.2.2 where the resulting model included 84 states. The Petri net model in Figure 15.13 has 18 places, but 9 of them correspond to source and sink transitions. The remaining 9 places correspond therefore to the 84 states in the FA model, which we can verified since the sum of possible markings of these 9 places is 84. This shows the use of Petri nets power to capture modularities in systems.

15.3.3 Petri Net Synthesis

Having a Petri net model for the tank in Figure 15.8 we can hopefully use this model to synthesize a control law following the specifications stated in Section 11.1. We will discuss the possibilities for our model to fit in some synthesis methods in the area of Petri nets. For a tutorial survey see Holloway and Krogh [44].

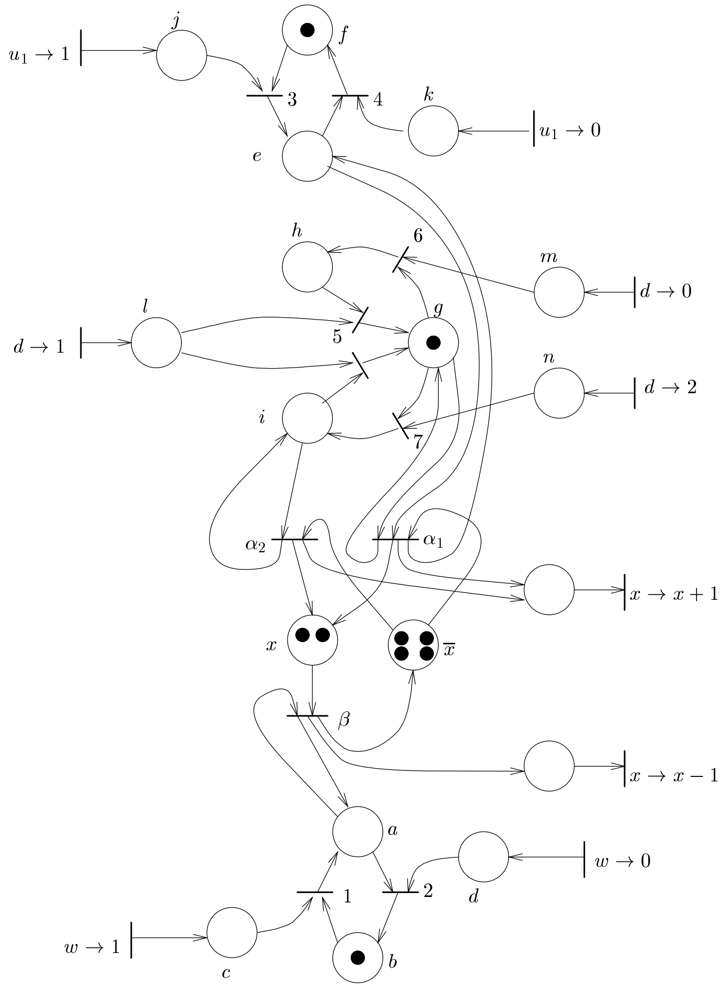


Figure 15.13: The final Petri net model of the tank.

To be able to apply synthesis methods to Petri nets an extension of Petri nets has been defined called *Controlled Petri Nets* [48, 52]. For controlled Petri nets the primitive *control places* are introduced, which is a places with two modes: enable or disable. The modes in control places can be assigned from an external controller. By this construction the model of the plant is separated from the controller model which is necessary when doing synthesis.

Transitions which have a control place as an input can only be state enabled if its control place is enabled. The controller of the system can use control places to prevent events from occuring. If this type of control shall be possible on the plant the control places must correspond to some actuators of the plant. In the case of the tank in Figure 15.8 we have two valves u_2 and u_4 that prevent inflow and outflow respectively. The valve u_2 corresponds to a control place which would be input to the transitions α_1 and α_2 in Figure 15.13, whereas the valve u_4 corresponds to a control place which would be input to the transition β. By adding these control places to our Petri net model of the tank the model is usable for synthesis.

The area of controlled Petri nets can be divided into two major approaches: state feedback and event feedback. The state feedback approach assumes that the marking (state) of a system can be observed and that the controller can compute a control law from these observations that acts on the control places. The event feedback approach assumes that the firings of transitions are observable as events of the plant and that it is possible to compute control from these observations [37].

State feedback policies for controlled Petri nets have been investigated by a number of researchers, see the overview in [44]. One approach to perform supervisory control of controlled Petri nets is simply to create the equivalent controlled automaton. This is a matter of generating the *reachability graph*, i.e. a tree containing all markings reachable from an initial marking. Since the reachability graph can grow exponentially with respect to the size of the controlled Petri net model [54], this cannot be done in general. Alternative methods are *linear integer programming* approach (Giua et al. [38, 39] and Li and Wonham [56, 57]) and *path based* algorithms (Krogh et al. [54, 53, 45].

Krogh and Holloway present in [45] a method for state control for a subclass of Petri nets, *cyclic controlled marked graphs* (CMG). CMG are controlled Petri nets where:

- Each place is the output of exactly one transition and the input of exactly one transition.
- Every place is contained in a cycle[4].
- Every cycle contains at least one marked place.
- Every place is contained within some cycle which has exactly one marked place.

For this class of Petri nets the method by Krogh and Holloway computes a supervisory control law efficiently. Unfortunately though, the Petri net model of the tank in Figure 15.13 is not a CMG and we cannot use this method on the tank.

The generality of Petri nets is an advantage for modeling issues but a disadvantage for analysis and control synthesis. Indeed, it is shown that controllability is undecidable for the most general Petri net languages [78]. The Petri net model of the tank corresponds to the FA model in Section 15.2.2 from which it is possible to create a supervisory control law

[4]A cycle in a controlled Petri net is a directed graph beginning and ending at the same node with all other nodes in the path occuring only once.

using the Ramadge-Wonham approach. Therefore we can conclude this discussion that the control problem of the Petri net model in Figure 15.13 is decidable in spite that it will not fit into one of the Petri net subclasses specialized for synthesis.

15.3.4 Summing Up

We have used Petri nets to model the tank system shown in Figure 15.8, by adapting the modeling method in [69] which uses identification of the events and conditions in the system to generate a Petri net model of the system. Since Petri nets uses events as a representation of inputs and outputs extra modeling effort is needed to convert signals to events, as described in Section 14.4.

The resulting Petri net model cannot be used for synthesizing a supervisory control law with the approach introduced by Holloway and Krogh [45]. The reason for this is that the synthesizing method is based on a restricted class of Petri nets, and the final Petri net model of the tank cannot be tranformed to that class.

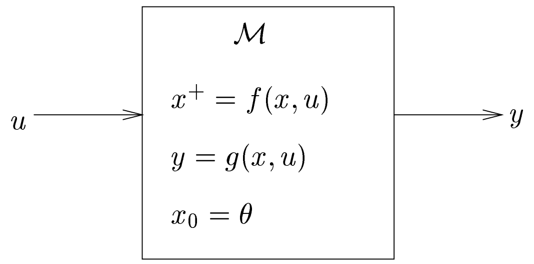
15.4 COCOLOG

In 1990 Caines and Wang [15] introduced COCOLOG, a conditional observer and controller logic for finite machines. COCOLOG is designed to adaptively compute control actions from observed outputs and inputs of the system. A COCOLOG controller can be regarded as an adaptive controller for finite machines (FA).

In this section we first give a brief and informal introduction to COCOLOG. Then we will focus on the modeling aspect through an example where we make a COCOLOG model of the tank.

15.4.1 Introduction to the COCOLOG Approach

Consider a finite state machine $\mathcal{M}$ defined by the functions f and g as shown in the figure below.



The variables x, y, u are state, output and input variables of $\mathcal{M}$ and the initial state x_0 is set to some state θ. Assume that we want to control $\mathcal{M}$ using state feedback, and that we have a state feedback function K such that $u = K(x)$ gives us the desired closed loop behavior.

To use the feedback $K(x)$ we must have access to the state of the system x, i.e. that we must be able to measure x directly. If x is not directly accessible there will be problems. But inspired by identification and state estimation methods in the area of continuous linear systems, we might hope that there is a way to estimate the current state of the machine. The state estimate $\hat{x}$ could then be used for control by using the feedback $u = K(\hat{x})$.

However, finite domains have in general no metric that can be used to measure the distance between two quantities the the domain. For the purpose of state estimation this means that the only reliable state estimate $\hat{x}$ is the one which fulfills $\hat{x} = x$, i.e. the same as to measure x directly. Therefore we will not be able to find methods for finite systems in analogy with feedback from estimated state methodology for linear systems.

Within the framework of COCOLOG, another approach has been proposed for feedback control based on the set of observed input and output denoted $\mathcal{U}, \mathcal{Y}$ respectively. COCOLOG considers finite systems where dynamic and output relations are known (f, g), but where the initial state x_0 is unknown. The COCOLOG observer of the system computes from samples of the inputs $\mathcal{U}$, outputs $\mathcal{Y}$ and the model f, g the set of possible states $\mathcal{X}$ in which the present state of the system is included As the number of observations increase, the set of possible states will in the generic case decrease. This set of possible states will then be used to compute a control law. See Figure 15.14.

The COCOLOG control law is a set of mutually exclusive conditions corresponding to control actions (values of u). The set is called *the set of conditional control rules* and is of the form

$$
\begin{array}{llll}
\texttt{IF} & C_1 & \texttt{THEN} & u = U_1 \\
\texttt{IF} & \neg C_1 \wedge C_2 & \texttt{THEN} & u = U_2 \\
 & \vdots & & \vdots \\
\texttt{IF} & \bigwedge_{j=1}^{m-1} \neg C_j \wedge C_m & \texttt{THEN} & u = U_m \\
\texttt{IF} & \bigwedge_{j=1}^{m} \neg C_j & \texttt{THEN} & u = U_*
\end{array}
$$

where $U_i, (0 \leq i \leq m)$ and U_* are control actions and $C_j, (0 \leq j \leq m)$ are conditions on $\mathcal{X}, \mathcal{U}$ and $\mathcal{Y}$. The control action U_* is used when there are no conditions C_i that can be fulfilled.

The conditional control rules are by definition mutually exclusive, since they can be regarded as an algorithm that returns the control action corresponding to the first fulfilled condition, starting from C_1. By this con-

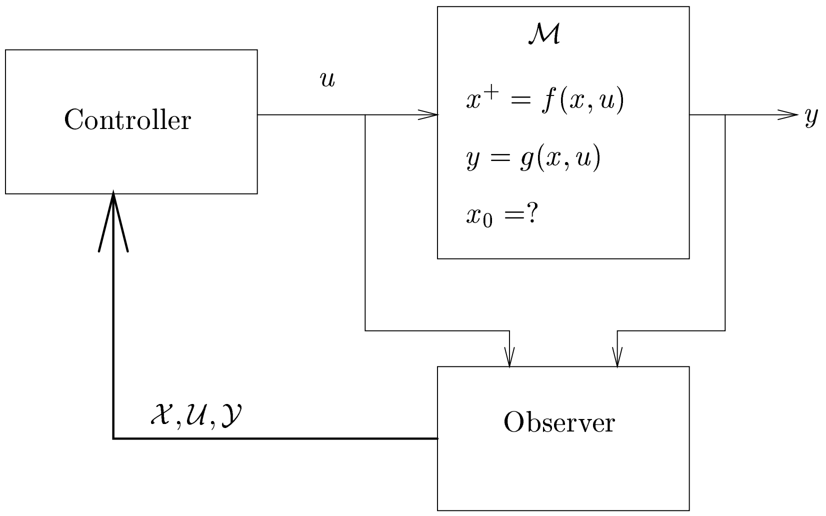


Figure 15.14: Structure of a COCOLOG controller.

struction we will always get a unambiguous control action as result of the observed inputs and outputs.

By finding[5] the appropriate conditions C_j and control actions U_j that give the desired closed loop behavior it is possible by the use of COCOLOG to implement a state feedback control law even though the initial state of the system is unknown.

The COCOLOG Framework

Quantities and relations are in COCOLOG expressed as *axioms* of first order logic, e.g. the system model, observations, and conditional control rules among others are expressed as axioms. From these axioms new *theorems* can be derived and added to the set of axioms. The set of all theorems that can be derived from the axioms is called a *theory* and represents all knowledge that can be derived from the axioms.

To be more concrete; consider the set of all initially known relations for a controlled system, i.e. the system model and the conditional control rules. These relations are the first part of the axioms in the initial theory Th_0 which in a sense represents all knowledge of the system before any measurements are done. When the first measurements of the inputs and outputs are made, new axioms are added to Th_0 to derive the new theory Th_1. This continues for every step, i.e. for the time instant k we will have the theory Th_k containing the axioms for the observed data for all time

[5]This can be a delicate matter of synthesizing by hand.

instances up k. The control action at time instant k is derived by computing the conditional control rules within the theory Th_k.

Since the COCOLOG framework is based on first order logic there are axioms needed in the theories to make the COCOLOG framework complete.

Some of these are:

- Logic axioms, defining the standard set of axiom schematas for the first order logic.
- Equality axioms, defining equality and substitution axioms like $Eq(t, t') \rightarrow Eq(t', t)$ which defines the symmetric property of equality.
- Finite arithmetic axioms, defining arithmetical operations like $+$ and $-$ on a subset of symbols $\{0, \ldots, N\}$ representing the natural numbers.

These axioms are needed to make computation possible in the theories. It is possible to add any kind of axioms designed for some special purpose, e.g. axiom for computing the set of reachable states of a model.

The computations in COCOLOG are performed by the *automatic theorem prover* Blitzensturm [14] which is a software tool that can prove or refute theorems from a set of axioms. In general, theorem proving is very complex, with computation time proportional to the exponential of the number of axioms. In addition, a COCOLOG controller must be built upon a theorem prover suitable for real time use, since the controller actions must be computed by proving conditional control rules after each data observation.

To reduce the complexity of COCOLOG the *Markovian fragment theories* are introduced in [84]. Markovian fragment theories contain the initial theory Th_0 and a finite number of axioms of the most recent observations. This means that the Markovian fragment theories do not increase the number of included axioms at every time instant. It can be shown that provided certain conditions are satisfied, the Markovian fragment has the same power to make control decisions as the original COCOLOG, see [84].

15.4.2 Modeling the Tank with COCOLOG

We will in this section model the tank in Figure 15.15 using the COCOLOG approach.

Before we can state the axioms for the initial theory Th_0 we will derive logical expressions of the dynamics in the tank. Using the notion of increasing α and decreasing β stated in previous sections, we can express the rules of the dynamics as

$$\texttt{IF}\ \alpha \wedge \neg(x = 6)\ \texttt{THEN}\ x^{+} = x + 1 \tag{15.19}$$

$$\texttt{IF}\ \beta \wedge \neg(x = 0)\ \texttt{THEN}\ x^{+} = x - 1 \tag{15.20}$$

$$\alpha = u_2 \wedge ((d = 2) \vee u_1 \wedge \neg(d = 0)) \tag{15.21}$$

$$\beta = u_3 \vee (u_4 \wedge w) \tag{15.22}$$

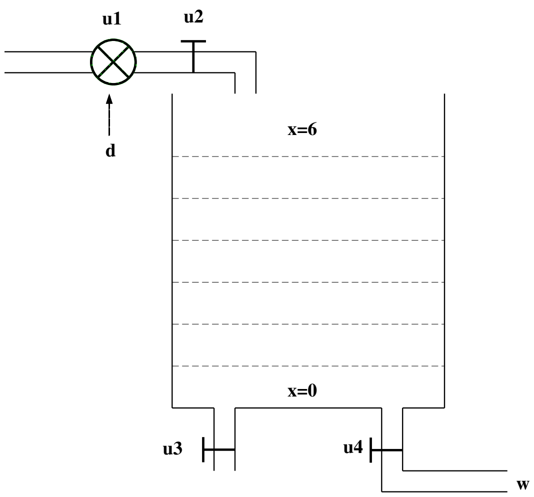


Figure 15.15: The Tank

From the description of the dynamics above we can define the state transition functions as

$$\begin{aligned}\Phi(x,\alpha) &= \alpha \wedge \neg Eq(x,6) \rightarrow x +_6 1\\ \Phi(x,\beta) &= \beta \wedge \neg Eq(x,0) \rightarrow x -_6 1\end{aligned} \tag{15.23}$$

where the operations $+_6, -_6$ will be defined below.

The set of state transition axioms are then defined as

$$\begin{aligned}AXM^{dyn} \equiv \{&Eq(\Phi(x,\alpha),x),\\ &Eq(\Phi(x,\beta),x),\\ &\alpha \leftrightarrow u_2 \wedge (Eq(d,2) \vee u_1 \wedge \neg Eq(d,0)),\\ &\beta \leftrightarrow u_3 \vee (u_4 \wedge w)\}\end{aligned} \tag{15.24}$$

The axioms in (15.24) capture all information in the system and therefore (15.24) is the complete model in a sense. But, as mentioned above, more axioms are needed to be defined for the initial theory Th_0. In this section we will only describe the set of arithmetic axioms.

The arithmetic axioms are in this case based on the set of integers represented by the symbols

$$I_6 = \{0,1,2,\ldots,5,6,6+1\} \tag{15.25}$$

where the symbol $6+1$ is used as an arithmetic overflow indicator. Then the set of arithmetic axioms by the symbols in $(I_6, +_6, -_6)$ are give by the following addition and subtraction operations:

$$a +_6 b = \begin{cases} a + b & \text{if } a + b \leq 6 \\ 6 + 1 & \text{if } a + b > 6 \end{cases} \tag{15.26}$$

$$a -_6 b = \begin{cases} a - b & \text{if } a - b \geq \text{ and } a \neq 6 + 1 \text{ and } b \neq 6 + 1 \\ 6 + 1 & \text{if } a - b < 0 \text{ or if } a = 6 + 1 \end{cases} \tag{15.27}$$

15.4.3 Summing up

COCOLOG is a framework based on theorem proving in the first order logic domain. The COCOLOG approach to control is based on conditional observer and control that performs control on finite systems with unknown initial state.

COCOLOG controllers use theorem provers to compute control actions from observed input and output signals of the system. This gives general and flexible controllers, but the computational complexity is a problem. To deal with this the Markovian fragment theories are introduced.

The modeling of the tank was easy to perform in COCOLOG. The disadvantages with the COCOLOG modeling is perhaps the administration in defining axioms for logical, arithmetic operations. However, except for the arithmetic operations $+, -, *, /$ we must do the same in the polynomial approach. If we for example need the conditional relation $>$ defined for a set of integers, we will have to define this operations axiomatically in both COCOLOG and the polynomial approach.

The most important advantages of the polynomial approach are the ability of reducing complexity and that it can be supported by efficient software tools.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: an Algebra for Discrete Event Systems.* John Wiley & Sons, 1992.

[3] S. Balemi, G. Hoffman, P. Gyugyi, H. Wong-Toi, and G. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Control*, 38(7):1040–1059, July 1993.

[4] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, second edition, 1990.

[5] A. Benveniste and K. J. Åström. Meeting the challenge of computer science in the industrial applications of control: An introductory discussion to the special issue. *IEEE Transactions on Control*, 38(7):1004–1010, July 1993.

[6] A. Benveniste, P. Le Borgne, and C. Jacquemot. The SIGNAL software environment for real-time system specification, design, and implementation. In *1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design (CACSD)*, pages 41–49, 1989.

[7] F. Boussinot and R. de Simone. The ESTRELLE language. another look at real time programming. *Proceedings of IEEE*, 79(9):1293–1304, 1991.

[8] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[9] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[10] B. A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. Technical Report 9210, System Control group, Dept. of Electl. and Comp. Engrg., University of Toronto, August 1992.

[11] Y. Brave and M. Heymann. Control of discrete event systems modeled as hierarchical state machines. *IEEE Transactions on Automatic Control*, 38(12):1803–1819, December 1993.

[12] P. C. Breedveld, R. C. Rosenberg, and T. Zhou. Bond Graph Bibliography. *Special Issue Journal of the Franklin Institute on "Current Research in Graph Modeling"*, 328(5/6):1067–1109, November/December 1991.

[13] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[14] P. Caines, T. Mackling, and Y.J.Wei. Logic control via automatic theorem proving: COCOLOG fragments implemented in Blitzensturm 5.0. In *Proceedings of the American Control Conference*. San Francisco, California, June 1993.

[15] P. E. Caines and S. Wang. COCOLOG: A conditional observer and controller logic for finite machines. In *Proceedings of the 29th IEEE Conference on Decision and Control*. IEEE, 1990.

[16] C. G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis.* Irwin, 1993.

[17] C. G. Cassandras. Sample-pathbased continuous and discrete optimization of discrete event systems: From gradient estimation to "Rapid Learrning". In G. Cohen and J.-P. Quadrat, editors, *11th International Conference on Analysis and Optimization of Systems - Discrete Event Systems*, number 199 in Lecture Notes in Control and Information Sciences, pages 388–400. Springer-Verlag, June 1994.

[18] CCITT. *CCITT Recommendation Z.100: Specification and Description Language SDL.* CCITT, ITU General Secretariat – Sales Section, Places des Nations, CH-1211 Geneva 20, Switzerland, 1988. Blue Book, Volumes X.1–X.5.

[19] CEI-IEC. Preparation of function charts for control systems. Standard 848, IEC, 1988. First edition.

[20] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–63, April 1986.

[21] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium*

on Computer Hardware Description Languages and their Applications. North-Holland, 1993.

[22] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflection and Pespectives. REX School/Symposium Proceedings*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer-Verlag, 1994.

[23] T. L. Claus Lewerentz. *Formal Development of Reactive Systems, Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.

[24] G. Cohen. Dioids and discrete event systems. In G. Cohen and J.-P. Quadrat, editors, *11th International Conference on Analysis and Optimization of Systems - Discrete Event Systems*, number 199 in Lecture Notes in Control and Information Sciences, pages 223–236. Springer-Verlag, June 1994.

[25] G. Cohen, D. Dubois, J. P. Quadrat, and M. Viot. A linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing. *IEEE Transactions on Automatic Control*, AC-30(3):210–220, March 1985.

[26] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer-Verlag, 1992.

[27] D. N. Dyck and P. E. Caines. The logical control of an elevator. *IEEE Transactions on Automatic Control*, 40(3):480–486, March 1995.

[28] E. D. Fabricius. *Modern Digital Design and Switching Theory.* CRC Press, 1992.

[29] J. H. Gallier. *Logic for Computer Science: Foundations for Automatic Theorem Proving.* Harper & Row, 1986.

[30] R. Germundsson. Basic results on ideals and varieties in finite fields. Technical Report LiTH-ISY-I-1259, Dept. of Electrical Engineering, Linköping University, Sweden, September 1991.

[31] R. Germundsson. Logic proofs $\equiv$ ideal inclusions. Technical Report LiTH-ISY-I-1286, Dept. of Electrical Engineering, Linköping University, Sweden, November 1991.

[32] R. Germundsson. A polynomial view of logics. Technical Report LiTH-ISY-I-1301, Dept. of Electrical Engineering, Linköping University, Sweden, November 1991.

[33] R. Germundsson. Analysis of polynomial dynamical systems over finite fields. Technical Report LiTH-ISY-I-1347, Dept. of Electrical Engineering, Linköping University, Sweden, March 1992.

[34] R. Germundsson. Symbolic and algebraic methods for modeling, analysis, design and implementation of discrete systems. Technical Report LiTH-ISY-R-1477, Dept. of Electrical Engineering, Linköping University, Sweden, May 1993.

[35] R. Germundsson. *Symbolic Algebraic Systems: Theory, Computation and Applications.* PhD thesis, Linköping University, September 1995. Forthcoming thesis.

[36] R. Germundsson, J. Gunnarsson, A. Jansson, P. Krus, M. Morin, S. Nadjm-Tehrani, J. Plantin, M. Sethson, and J.-E. Strömberg. Complex hybrid systems I: a study of available tools and specification of planned work. Technical Report LiTH-IDA-R-94-29, Dept. of Computer Science, Linköping University, Sweden, June 1994.

[37] A. Giua and F. DiCesare. Blocking and controllability of petri nets in supervisory control. *IEEE Transactions on Automatic Control*, 39(4):818–823, April 1994.

[38] A. Giua and F. DiCesare. Petri net structural analysis for supervisory control. *IEEE Transactions on Robotics and Automation*, page ??, April 1994.

[39] A. Giua, F. DiCesare, and M. Silva. Generalized mutual exclusion constraints on nets with uncontrollable transitions. In *Proceedings of 1992 IEEE Intenational Conference on Systems, Man, and Cybernetics*, pages 974–979. Chicago, October 1992.

[40] P. Glasserman. *Gradient Estimation via Pertubation Analysis.* Kluwer, 1991.

[41] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. IEEE Special Issus on Real Time Programming.

[42] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[43] Y. Ho and X. R. Cao. *Pertubation Analysis of Discrete Event Dynamic Systems.* Kluwer, 1991.

[44] L. Holloway and B. Krogh. Controlled Petri nets: A tutorial survey. In G. Cohen and J.-P. Quadrat, editors, *11th International Conference*

on Analysis and Optimization of Systems - Discrete Event Systems, number 199 in Lecture Notes in Control and Information Sciences, pages 158–168. Springer-Verlag, 1994.

[45] L. E. Holloway and B. H. Krogh. Synthesis of feedback control logic for a class of controlled Petri nets. *IEEE Transactions on Automatic Control*, 35(5):514–523, May 1990.

[46] A. Holt, H. Saint, R. Shapiro, and S. Warshall". Final report of the information system theory project. Technical Report RADC-TR-68-305, Rome Air Development Center, Griffis Air Force Base, New York, September 1968.

[47] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[48] A. Ichikawa and K. Hiraishi. Analysis and control of discrete event systems represented by Petri nets. In *Discrete Event Systems: Models and Applications*. IIASA Conference, Sopron, Hungary, August 3–7, 1987, Springer-Verlag, 1988.

[49] IEEE. *The IEEE Standard VHDL Language Reference Manual*. IEEE, 1987. The IEEE 1076–1987 standard.

[50] G. Kahn. Natural semantics. In *Proc. STACS'87*, number 247 in Lecture Notes in Computer Science, pages 22–39. Springer-Verlag, 1987.

[51] D. Karnopp, R. Rosenberg, and D. Margolis. *System dynamics - A unified approach*. John Wiley & Sons, second edition, 1990.

[52] B. H. Krogh. Controlled Petri nets and maximally permissive feedback logic. In *Proceedings of 25th Annual Allerton Conference*. University of Illinois, Urbana, September 1987.

[53] B. H. Krogh and L. E. Holloway. Synthesis of feedback control logic for discrete manufacturing systems. *Automatica*, 27(4):641–651, July 1991.

[54] B. H. Krogh, J. Magott, and L. E. Holloway. On the complexity of forbidden state problems for controlled marked graphs. In *Proceedings of the Conference on Descision and Control*. Brighton, UK, December 1991.

[55] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic Publishers, 1995.

[56] Y. Li and W. Wonham. Control of vector discrete-event systems I – the base model. *IEEE Transactions on Automatic Control*, 38(8):1214–1227, August 1993.

[57] Y. Li and W. Wonham. Control of vector discrete-event systems II – controller synthesis. *IEEE Transactions on Automatic Control*, 39(3):512–531, March 1994.

[58] F. Lin and W. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(2):173–198, 1988.

[59] Logikkonsult NP AB. Using NP-circuit to model and verify full-scale systems. Preprint, 1994.

[60] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophysics*, 5:115–133, 1943.

[61] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1987.

[62] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[63] G. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[64] E. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton University Press, 1956.

[65] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[66] J. S. Ostroff. Formal methods for the specification and design of real-time safty-critical systems. *Journal of Systems and Software*, 18:33–60, April 1992.

[67] J. S. Ostroff and W. M. Wonham. A framework for real-time discrete-event control. *IEEE Transactions*, 35(4):386–397, 1990.

[68] H. M. Paynter. *Analysis and design of egineering systems.* MIT Press, Cambridge Mass., 1961.

[69] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[70] C. Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, 1962.

[71] J. Plantin. Algebraic methods for verification and control of discrete event dynamic systems. Licentiate thesis LiU-TEK-LIC-1995:33, Dept. of Electrical Engineering, Linköping University, June 1995.

[72] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, Aarhus, Denmark, September 1981.

[73] P. Ramadge and W. Wonham. Modular feedback logic for discrete event systems. In *Large Scale Systems: Theory and Applications 1986 Selected Papers from the 4th IFAC/IFORS Symposium. Geering, H.P.; Mansour, M.*, pages 93–98, 1987.

[74] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, January 1987.

[75] P. J. Ramadge. Some tractable supervisory control problems for discrete-event systems modeled by Büchi automata. *IEEE Transactions on Automatic Control*, 34(1):10–19, January 1989.

[76] P. J. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[77] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of Discrete Event Systems. In G. Cohen and J.-P. Quadrat, editors, *11th International Conference on Analysis and Optimization of Systems - Discrete Event Systems*, number 199 in Lecture Notes in Control and Information Sciences, pages 73–79. Springer-Verlag, June 1994.

[78] R. S. Sreenivas. A note on deciding the controllability of a language k with respect to a langauge l. *IEEE Transactions on Automatic Control*, pages 658–662, April 1993.

[79] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag, 1980.

[80] J.-E. Strömberg. Styrning av lego-bilfabrik. Technical report, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 1991. Manual for control laboratory session.

[81] J.-E. Strömberg. *A mode switching modelling philosophy.* PhD thesis, Dept. of Electrical Engineering, Linköping University, 1994.

[82] J. G. Thistle. Logical aspects of control of discrete-event systems: A survey of tools and techniques. In G. Cohen and J.-P. Quadrat, editors, *11th International Conference on Analysis and Optimization of Systems - Discrete Event Systems*, number 199 in Lecture Notes in Control and Information Sciences, pages 3–15. Springer-Verlag, June 1994.

[83] VERILOG. *ASA, Automata and Structured Analysis, Reference Manual.* VERILOG SA, October 1991.

[84] Y. Wei and P. Caines. On Markovian fragments of COCOLOG for logic control systems. In *Proceedings of 31st IEEE Conference on Decision and Control*, pages 2967–2972. Tuscon, December 1992.

[85] S. Wolfram. *Mathematica. A System for Doing Mathematics by Computer.* Addison-Wesley, second edition, 1991. ISBN 0-201-51502-4.

[86] W. Wonham and P. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, May 1987.

[87] B. P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models.* Academic Press, Boston Mass., 1990.

[88] H. Zhong and W. M. Wonham. On the consistency of hierarchical supervision in discrete-event systems. *IEEE Transactions on Automatic Control*, 35(10):1125–1134, October 1990.