Symbolic Methods and Tools for Discrete Event Dynamic Systems

Johan Gunnarsson

REGLERTEKNIK

AUTOMATIC CONTROL

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

Linköping 1997

Symbolic Methods and Tools
for
Discrete Event Dynamic Systems

Email: `johan@isy.liu.se`
WWW: `http://www.control.isy.liu.se`

Division of Automatic Control
Department of Electrical Engineering
Linköping University
S-581 83 Linköping
Sweden

ISBN 91-7871-917-8 ISSN 0345-7524

Printed in Sweden by Linköpings Tryckeri AB 1997.542

To Marit and Emma

Abstract

The interest in Discrete Event Dynamic Systems (DEDS) has increased during the last years, due to the lack of methods and tools that are capable of handling the complexity of problems and tasks present in industry today. In this thesis we will consider a framework based on relations over finite domains. The framework is used for modeling, analysis, and synthesis of DEDS.

Binary Decision Diagrams (BDDs) are used to represent relations, as well as the operations for modeling, analysis and synthesis of DEDS. To utilized the structure of integers and arithmetic operation, Integer Decision Diagrams (IDDs) are developed and implemented. Polynomials over finite fields are another type of representation that is used for the relational framework. Here Gröbner bases, and Integrated Monomial Diagrams (IMDs) are the tools that are used. IDDs and IMDs are both developed, by the author, to represent integer structures and arithmetic operations efficiently.

With tools for efficient relational representation, it possible to improve 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 relational model, represented by a 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. The landing gear controller serves as a benchmark test of BDDs and IDDs. The IDDs reduced the computation time by 50%.

To explore the ability and applicability of using a polynomial relational representation 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

Structure of the Thesis

This thesis consists of three separate parts:

Part I	Relational Representations
Part II	Applications
Part III	Modeling of DEDS - General Aspects

Parts II and III contain rewritten and reorganized material from the licentiate theses: Gunnarsson [49] and Plantin [99]. Parts III contains material from Gunnarsson [49].

How to Read this Document

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 2 introduces the relational framework, on which the thesis rests.

(ii) Part I can be read separately, but is motivated by Chapter 2 and Part II.

(iii) Part II consists of two application studies, which use theory and tools presented in Part I.

(iv) Read Chapter 8 in Part II before Part III, since Chapter 8 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.

An outline of this thesis can be found in Section 1.2, and a description of the contributions is given in Section 1.3.

Published Papers

The work presented in this thesis is based on the following technical reports and conference presentations.

[**41**] 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, Linköping University, June 1994.

[**42**] R. Germundsson, J. Gunnarsson, and J. Plantin. Symbolic algebraic discrete systems - applied to the JAS 39 fighter aircraft. Technical Report LiTH-ISY-R-1718, Linköping University, December 1994.

[**50, 51**] J. Gunnarsson. Algebraic methods for discrete event systems - a tutorial. In *Workshop on Discrete Event Systems*. IEE, August 1996.

[**52**] J. Gunnarsson. Symbolic algebraic discrete systems - applied to the JAS 39 fighter aircraft, part ii. Technical Report LiTH-ISY-R-1873, Linköping University, August 1996.

[**53, 54**] J. Gunnarsson and R. Germundsson. Dynamic verification of a large discrete system. In *Proc. of 35th IEEE Conference on Decision and Control*, Kobe, Japan, December 1996. IEEE.

[**55**] J. Gunnarsson and J. Plantin. Control law synthesis for a discrete event system. Technical Report LiTH-ISY-R-1651, Linköping University, August 1994.

[**56**] J. Gunnarsson and J. Plantin. Automatic synthesis for simultaneous supervision and control – a first example. In *American Control Conference*, The American Automatic Control Council, IEEE, June 1995.

[**57, 58**] J. Gunnarsson and J. Plantin. Synthesis of a discrete system using algebraic methods. In *Workshop on Discrete Event Systems*. IEE, August 1996.

[**59, 60**] J. Gunnarsson, J. Plantin, and R. Germundsson. Verification of a large discrete system using algebraic methods. In *Workshop on Discrete Event Systems*. IEE, August 1996.

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. Lennart knows supervision and control, and to be his student is certainly a privilege.

I would also like to thank a man with good taste in research subjects, Dr. Roger Germundsson, for the inspiration, support and visions he has shared with me. His research effort in the domain of DEDS made this thesis possible.

To my former research companion Jonas Plantin: Once again thank you. Your contributions are very important to this thesis also. I hope we will stay in touch in the future, both as friends and colleges.

I would like to thank Dr. Inge Klein, Magnus Larsson and Mats Jirstrand who have provided a forum for discussions in the absence of Roger and Jonas.

To be a member of a group like the Automatic Control Group in Linköping, has given me memories I never will forget. The fika[1]-room, junk-seminars, outrageous discussions round the fika, the floorball games and much more. Working in such an inspiring and warm[2] atmosphere makes this group special. Thank you all.

I would also like to thank Mats Jirstrand, Peter Lindskog, Jonas Plantin, Magnus Larsson, Inger Klein and Ulla Salaneck 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, Magnus Landberg and Ove Åkerlund at SAAB, who supplied me with valuable information on the JAS landing gear system.

Thanks to Peter Lindskog and Anders Stenman for supporting me with the LaTeX and *XEmacs* tools.

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 like to thank all friends to me and my family, and especially the Rosén family[3] for all support and help during this period of time.

Thanks to Lotta Bergmark and LTAB for the helping me out with the cover of this thesis.

To my parents Gunnar and Kerstin: Thank you for your love and support. Forgive me, that you have not heard from me as often as I wish. I hope to improve on this matter.

To my brother and sisters with families: Thank you all for being the steady state in life, and all fun and joy.

Finally, I give my greatest gratitude to my wife, Marit, and daughter, Emma, for their love, support, patience and encouragement. This thesis is to you and because of you. I love you so much.

One of these nights in
Linköping, April 1997

Johan Gunnarson

[1]Fika is the coffee break culture.
[2]True in all interpretations.
[3]Dentists should be humble. ;-)

Acknowledgment for Acronyms

$\mathcal{A}\mathcal{M}\mathcal{S}$-TEX, BIBTEX, CMU BDD, Dell, euler, FBK, ftp, gcc, gdb, gnu, html, IBM, IDA, islandDraw, ISY, JAS, HP, LATEX, LiTH, *Mathematica*, Mathlink, Netscape, QDD, RRPE, Sun, SAS, SLITEX, SMF, SMU, tcsh, TEX, TFR, unix, vm, Win95, XEmacs, xfig.

Contents

1 Introduction

1.1 Motivation

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 [7]*

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 enhance 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 automatic control community 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.

1.2 Road Map for the Thesis

The theme of this thesis is representation and applications of DEDS where a relational framework[1], developed by Germundsson [40], is used. The framework is based on symbolic and algebraic methods as well as tools that give efficient computations also for larger DEDS.

The components of system theory in the relational framework such as modeling, analysis and synthesis will briefly be described in Chapter 2, "A Relational Framework for DEDS". There we will also introduce the idea of relations as a convenient tool for DEDS, both for algorithms and for our own understanding. We let relations be representations of finite sets but also of DEDS behavior. In the latter case we say that the relation is a model of DEDS that may be used for some analysis of the system behavior or as an input to a synthesis method. All the methods included in Chapter 2 will be defined and exemplified without concern for how these relations are represented.

The problem of finding efficient representations or implementations for the relations, is the main concern of Part I, "Relational Representations" in this thesis. There we will try to find representations for the relations on an algorithmic level, in contrast to the conceptual and abstract way we use the relations in Chapter 2. To make it more appealing, an overview of the different representations in Part I is given in Chapter 3. In Chapter 4, "Decision Diagrams", and in Chapter 5, "Polynomial Representation", two different approaches of representations in finite domains are described in more detail. At the end of Part I, Chapter 6 "IDD & IMD" our results are presented concerning relational representations. *Integer decision diagrams* (IDD) and *integrated monomial diagrams* (IMD) are both representations developed to increase computational performance for the methods of the relational framework.

Part II is devoted to applications where the relational system theory in Chapter 2 is used together with relational representations from part I. In Chapter 7 we demonstrate how to perform dynamic analysis on a complex system of industrial size. Here two variants of decision diagrams BDDs and IDDs from Chapter 4 are used as representations of a system model and analysis results. The system model is automatically generated by a compiler which translates a restricted part of the computer programming language Pascal into a relational model representing the

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

system behavior. The dynamic verification is performed on the closed loop system by using temporal logic as the specification language.

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 Chapter 8, where the synthesis of a control law for a tank system is performed by the use of polynomials over finite fields and Gröbner Bases as the computation tool.

Part III contains three separate chapters. The first 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 contains a comparison among the polynomial approach and three other DEDS approaches: Ramadge and Wonham [104], Petri nets [98], and COCOLOG [20]. This comparison is performed by remodeling the tank system from Chapter 8.

1.3 Contributions of the Thesis

The main contributions of this thesis are as follows.

(i) The core of Chapter 5 is the theory of polynomials over finite fields, developed by Germundsson [40]. The theoretic results regarding functional dependency w.r.t to ideals have been developed by the author together with Plantin [99]. We have developed the technique of the λ-polynomials to mix variables of different size. We found how to write λ-polynomials directly on a Gröbner bases form. These tools are essential for the synthesis of the tank application in Chapter 8.

(ii) Chapter 6 presents the development and the implementation of integer decision diagrams (IDDs), as well as complexity results. Performance measures of the IDDs are presented in Section 7.5.

(iii) In Chapter 6, we have defined and presented the integrated monomial diagram (IMD), which is an effective representation of polynomials over finite field.

(iv) Chapter 7 contains the main application of this thesis, where 1200 lines of Pascal code is automatically translated to a relational model. The chapter defines the compiler for a complete but restricted set of Pascal. We show that dynamic verification of the closed loop behavior using temporal logic is doable for complex problems.

(v) Chapter 8 presents a method for simultaneous supervision and control using polynomials over finite field, and Gröbner bases.

(vi) Part III presents general aspects on modeling of DEDS. The tank model form Chapter 8 is remodeled, using Ramadge-Wonham, Petri nets and COCOLOG. A characterization of the perfect modeling language is made. Events vs. signals is the topic for a discussion of different modeling paradigms for DEDS.

2

A Relational Framework for DEDS

The main theme of this thesis is finite domains and relations and how these are used to build a system theory of untimed DEDS. In fact all methods and tools in our mathematical framework as well as in our software are based on the principle that everything is relations over finite domains. Other frameworks of DEDS, such as *Ramadge-Wonham* [104], *Petri Nets* [97] and *COCOLOG* [20] also have some set of basic elements from which they build their frameworks. For Ramadge-Wonham we have the automata theoretic approach dealing with *events* and *formal languages*. For Petri Nets we have *places* and *transitions* as components to build a Petri Net graph. The *COCOLOG* approach depends on a set of *axioms* from which *theories* are derived. See Chapter 9 for an overview of DEDS frameworks.

There are more ways of describing DEDS, but here we will just note the fact that for finite DEDS all these frameworks may be interchanged with each other with respect to the behavior of the DEDS they represent. This is true since all finite DEDS can be represented by each framework[1]. What is then the core of DEDS which contains nothing but the necessary information of the behavior? In this thesis we regard relations as this core.

This section describes the fundament of sets and relations and introduces some notations. In Sections 2.2 to 2.4 we present the main ideas of a relational system theory for DEDS. The format chosen for this theory is the one from Germundsson [40]. Section 2A is devoted to propose methods to illustrate analysis results using our theory.

We will use common set theory and relations as presented, e.g., in [48], even though we use somewhat different interpretation and notations for convenience.

[1]However, note that Petri Nets can also represent DEDS with infinite state space.

2.1 Relations and Sets

2.1.1 A Shortcut to Understanding

Example 2.4 presents the relations and the relational operations that we will use in this thesis. Therefore we recommend to jump to that example for the understanding of how the relational framework is used for DEDS, relational representations, and applications.

The remainder of this section will give the formal definitions for relations, relation sets, and relational operations.

2.1.2 The Formal Route

Only finite domains will be considered in this thesis. Therefore we regard sets and relations as finite if nothing else is said.

For the sets S_1, S_2 we have the standard operations

$$\begin{aligned} \text{union:} \quad & S_1 \cup S_2 \\ \text{intersection:} \quad & S_1 \cap S_2 \\ \text{complement:} \quad & \overline{S_1} \\ \text{set minus:} \quad & S_1 \setminus S_2 \\ \text{cross product:} \quad & S_1 \times S_2 \end{aligned}$$

Elements of sets will be denoted by $[\cdot]$, e.g.,

$$[3,4] \in \{2,3\} \times \{1,4\} \tag{2.1}$$

Definition 2.1 Universe of Discourse
The *universe of discourse* (UoD), denoted $\mathcal{U}_n$, is the set

$$\mathcal{U}_n = S_1 \times \cdots \times S_n \tag{2.2}$$

where S_i, $1 \leq i \leq n$, are finite sets. □

The reason for defining the UoD is illustrated by the following example.

Example 2.1 Universe of Discourse
The complement $\overline{S}$ of the set S can be written as

$$\overline{S} = \mathcal{U}_n \setminus S \tag{2.3}$$

If $\mathcal{U}_n$ is not specified we cannot decide the interpretation of $\overline{S}$.

Which UoD we use in a given situation, is either explicitly stated or obvious from the context.

Subsets of $\mathcal{U}_n$ are usually called relations, [48], but we will instead use the term *relation set*, and use the term *relation* as an object representing a relation set.

If we regard a relation set as a piece of information, then the corresponding relation is a language, structure or formula that we use to *canonically* write this information down. By this we have that a relation is a unique representation of relation sets.

Definition 2.2 Relation Set
A *relation set*, denoted $\mathcal{R}$, is a subset of the cross product of n finite sets, i.e.,

$$\mathcal{R} \subseteq \mathcal{U}_n \tag{2.4}$$

is a relation set over the finite sets $S_i, 1 \leq i \leq n$.

Remark: A relation set can be defined over any finite number of sets, i.e., for all $n \geq 1$. □

Definition 2.3 Relation
A *relation* $R(x_1, \dots, x_n), R : \mathcal{U}_n \to \mathbb{B}$ is a representation of the relation set $\mathcal{R} \subseteq \mathcal{U}_n$, where x_i is a variable (argument) which can have elements from the set S_i as values for all $1 \leq i \leq n$.

If all variables x_i have values, i.e., $x_i = s_i \in S_i \; \forall 1 \leq i \leq n$ then

$$R(s_1, \dots, s_n) = \begin{cases} \texttt{true} & [s_1, \dots, s_n] \in \mathcal{R} \\ \texttt{false} & [s_1, \dots, s_n] \notin \mathcal{R} \end{cases} \tag{2.5}$$

where $[s_1, \dots, s_n]$ denotes an element in $\mathcal{R}$, and $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$.

Remark:

- By writing $x_i \in S_i$ we mean that the variable x_i takes an element from S_i as value.
- Note that we often interprete relations as a canonical representations of relations sets, i.e., the relation has one-to-one correspondence with the relation set. Therefore we say:

$$R(x_1, \dots, x_n) \equiv \texttt{true} \Leftrightarrow \mathcal{R} \equiv \mathcal{U}_n \tag{2.6}$$
$$R(x_1, \dots, x_n) \equiv \texttt{false} \Leftrightarrow \mathcal{R} \equiv \emptyset \tag{2.7}$$

□

The reason for separating relations and relation sets are somewhat similar as for separating syntax and semantics. The syntax is the relation and the semantics is the relation set. As we will see in Part I we will use several different syntaxes for relations but the semantics (the relation sets) remains the same.

We say that the relation is Boolean valued (denoted $\mathbb{B}$), i.e., the relation is `true` for elements included in the corresponding relation set and `false` otherwise. From this we have that operators similar to those in propositional logic are well defined on relations. These operators will correspond to the set theoretic operators for the relation sets. See Definition 2.6.

Example 2.2 Relation and Relation Set
We have some piece of information which consists of the pairs

$$[\text{red}, \text{apple}], \quad [\text{yellow}, \text{banana}] \tag{2.8}$$

If we put these pairs into a set we can regard this set as a relation set, which relates two fruits with two colors.

The immediate representation of this relation set is of course to store these pairs into a table or database.

If we let the variables f and c correspond to fruit and color respectively, then the following logic formula

$$(f == \text{apple}) \wedge (c == \text{red}) \vee (f = \text{banana}) \wedge (c == \text{yellow}) \tag{2.9}$$

is the same relation as the table since it corresponds to the same relation set. (Details of the logical operators in the formula above will be given the Section 2.1.3 below.)

If we encode (or label) fruits and colors by integers we can use polynomial equations to represent the relation, see Chapter 5.

We will continue by defining projection and embedding of relation sets.

Definition 2.4 Projection
For $\mathcal{R} \subseteq \mathcal{U}_n = S_1 \times \cdots \times S_i \times \cdots \times S_n$, $\pi_{S_i} : \mathcal{U}_n \to S_i$ $1 \leq i \leq n$ is defined by

$$\pi_{S_i}(\mathcal{R}) = \{ [s_i] \mid [s_1, \ldots, s_n] \in R \} \tag{2.10}$$

and called the *projection of* $\mathcal{R}$ *on* S_i, where the set elements $[s_1, \ldots, s_i, \ldots, s_n] \in \mathcal{U}_n$ and $[s_i] \in S_i$.

Remark: Projection on several finite sets, denoted $\pi_{S_{n_1} \times S_{n_2}}(\mathcal{R})$ is defined in the obvious way. □

Definition 2.5 Embedding
For $\mathcal{R} \subseteq \mathcal{U}_n = S_1 \times \cdots \times S_n$, $\epsilon_{S_{n+1}} : \mathcal{U}_n \to \mathcal{U}_n \times S_{n+1}$ is defined by

$$\epsilon_{S_{n+1}}(\mathcal{R}) = \mathcal{R} \times S_{n+1} \tag{2.11}$$

and called the *embedding of* $\mathcal{R}$ *in* $\mathcal{U}_{n+1}$ (or in $\mathcal{U}_n \times S_{n+1}$).

Remark:

- Instead of appending S_{n+1} to $\mathcal{U}_n$ we can put the new set in any position of $\mathcal{U}_n$, even though the index notation will be less clear.
- The definition above can easily be generalized to embedding of relation sets in any number of sets, by using recursion.

□

2.1.3 Relational Operations

To manipulate relation sets we use the ordinary set operations. But for relations we will use logic operations for notational reasons. Logic operations and set operations share the same axiomatic laws (see pages 52 and 109 in [48]) and can therefore be used in a similar way but for different domains.

We will use the notation X for the variables $x_1, \ldots, x_n$, i.e., by $R(X)$ we mean $R(x_1, \ldots, x_n)$.

Definition 2.6 Relational Operations
For the relational sets $\mathcal{R}_1$ and $\mathcal{R}_2$, and the corresponding relations $R_1(X)$ and $R_2(X)$ we have:

Relation	**Relation set**
$R_1(X) \wedge R_2(X)$	$\mathcal{R}_1 \cap \mathcal{R}_2$
$R_1(X) \vee R_2(X)$	$\mathcal{R}_1 \cup \mathcal{R}_2$
$\neg R_1(X)$	$\overline{\mathcal{R}_1}$
$R_1(X) \rightarrow R_2(X)$	Condition for $\mathcal{R}_1 \subseteq \mathcal{R}_2$
$R_1(X) \leftrightarrow R_2(X)$	Condition for $\mathcal{R}_1 = \mathcal{R}_2$

□

We also will use the following rules for $\rightarrow$ and $\leftrightarrow$.

$$a \rightarrow b = \neg a \vee b \tag{2.12}$$

$$a \leftrightarrow b = a \wedge b \vee \neg a \wedge \neg b \tag{2.13}$$

Definition 2.7 Relational Quantifier Operations
The quantifier operations $\exists$ and $\forall$ are defined as

$$\exists x_i .\, R(x_1, \ldots, x_n) = \bigvee_{x_i \in S_i} R(x_1, \ldots, x_n) \tag{2.14}$$

$$\forall x_i .\, R(x_1, \ldots, x_n) = \bigwedge_{x_i \in S_i} R(x_1, \ldots, x_n) \tag{2.15}$$

Remark: Note that we interpret $\exists$ and $\forall$ as operations mapping one relation to another, i.e., $\exists$ and $\forall$ return relations. This differs from the interpretation used in propositional logic [48] but is very common in other research areas such as *Quantifier Elimination Theory*, [4, 31, 91]. □

Working with relations, it is convienient to have a "solver" that gives the corresponding relation set of a relation. The problem of solving a relation is highly dependent on how we represent the relations. This topic will be discussed further in Part I of this thesis.

We define a special function that gives *one* random solution of a relation, which will be used in Section 2A.

Definition 2.8 Pick One Solution
Let the relation set $\mathcal{R} \subseteq S_1 \times S_2$ be represented by the relation $R(x, y)$ where $x \in S_1$ and $y \in S_2$.

We define the function $\mathsf{Pick}(R(x, y), x)$ to return a randomly chosen element of the relation set $\pi_{S_1}(\mathcal{R})$ corresponding to the relation

$$\exists y.\ R(x, y) \tag{2.16}$$

Remark: In other words; $\mathsf{Pick}(R(X, Y), X)$ returns one solution for the variables X. □

Equations of finite functions are also relations. We will introduce the infix operator $\cdot\!\!==\!\cdot$ to denote the relation representing an equation.

Definition 2.9 Equal Operator
The *equal operator*, used as

$$f(X) == g(X) \tag{2.17}$$

returns the relation representing the equation $f(X) = g(X)$, where $f(X)$ and $g(X)$ are finite functions. □

Example 2.3 Equal Operator
The equation set

$$\begin{aligned} a + b &= 1 \\ c + d &= 2 \end{aligned}$$

is represented by the relation

$$(a + b == 1) \wedge (c + d == 2)$$

To conclude this section, where the notions of relations and relation sets have been introduced, we will give an example that illustrates the operators and functions for relations and relation sets.

Example 2.4 Relation and Operators
Let $S_1 = \{0, 1, 2\}$ and $S_2 = \{3, 4\}$ be finite sets with integer elements. The UoD will be the cross product of S_1 and S_2 containing six elements.

$$\begin{aligned} \mathcal{U}_2 &= S_1 \times S_2 \\ &= \{[0, 3], [0, 4], [1, 3], [1, 4], [2, 3], [2, 4]\} \end{aligned}$$

Let $\mathcal{R}_1 = \{[0,4],[1,3]\}$ and $\mathcal{R}_2 = \{[1,3],[2,3]\}$. The corresponding relations will be written as

$$R_1(x_1,x_2) = (x_1{=}0)\wedge(x_2{=}4) \vee (x_1{=}1)\wedge(x_2{=}3) \tag{2.18}$$

$$R_2(x_1,x_2) = (x_1{=}1)\wedge(x_2{=}3) \vee (x_1{=}2)\wedge(x_2{=}3) \tag{2.19}$$

For the relation operations in Definition 2.6 we get

$$R_1(x_1,x_2) \wedge R_2(x_1,x_2) = (x_1{=}1)\wedge(x_2{=}3) \tag{2.20}$$

$$\begin{aligned} R_1(x_1,x_2) \vee R_2(x_1,x_2) = {} & (x_1{=}0)\wedge(x_2{=}4) \vee \\ & (x_1{=}1)\wedge(x_2{=}3) \vee \\ & (x_1{=}2)\wedge(x_2{=}3) \end{aligned} \tag{2.21}$$

corresponding to $\mathcal{R}_1 \cap \mathcal{R}_2 = \{[1,3]\}$ and $\mathcal{R}_1 \cup \mathcal{R}_2 = \{[0,4],[1,3],[2,3]\}$.

The complement $\overline{\mathcal{R}_1} = \{[0,3],[1,4],[2,3],[2,4]\}$ corresponds to negating the corresponding relation $R_1(x_1,x_2)$.

$$\neg R_1(x_1,x_2) = \neg\big((x_1{=}0)\wedge(x_2{=}4) \vee (x_1{=}1)\wedge(x_2{=}3)\big) \tag{2.22}$$

$$\begin{aligned} &= (x_1{=}0)\wedge(x_2{=}3) \vee (x_1{=}1)\wedge(x_2{=}4) \vee \\ &\quad (x_1{=}2)\wedge(x_2{=}3) \vee (x_1{=}2)\wedge(x_2{=}4) \end{aligned} \tag{2.23}$$

Note that it is not obvious at this point how to algebraically derive expression (2.23) from expression (2.22). In fact we have not yet decided the canonical form of relations. Canonical forms and algebraic manipulations will be discussed further in Part I. For now we just observe that (2.22) and (2.23) will have the same value (`true` or `false`) for the same values of x_1 and x_2.

By construction we know that $\mathcal{R}_1 \not\subseteq \mathcal{R}_2$ in this case. This can be checked by

$$\begin{aligned} R_1(x_1,x_2) \rightarrow R_2(x_1,x_2) &= \neg R_1(x_1,x_2) \vee R_2(x_1,x_2) \\ &= /\text{All elements in the UoD } \mathcal{U}_2 \text{ except } [0,4]/ \\ &= \neg\big((x_1{=}0)\wedge(x_2{=}4)\big) \end{aligned}$$

The result above is not identically `true` which means that $\mathcal{R}_1$ is not included in $\mathcal{R}_2$.

To test if $\mathcal{R}_1$ is equal to $\mathcal{R}_2$ we compute $R_1(x_1,x_2) \leftrightarrow R_2(x_1,x_2)$ which in this case will be equal to `false`.

The projection of $\mathcal{R}_1$ on S_1 is

$$\pi_{S_1}(\mathcal{R}_1) = \{[0],[1]\} \tag{2.24}$$

which corresponds to the existential quantification

$$\exists x_2.\, R_1(x_1,x_2) = \bigvee_{x_2 \in S_2} R_1(x_1,x_2) \tag{2.25}$$

$$= (x_1{=}0) \vee (x_1{=}1) \tag{2.26}$$

which is a relation only in the variable x_1.

The universal quantifier would in this case be

$$\forall x_2.\, R_1(x_1,x_2) = \texttt{false} \tag{2.27}$$

since there is no single value of x_1 for all values of x_2 in the relation set $\mathcal{R}_1$.

From the projection (2.24) we make the embedding

$$\epsilon_{S_2}(\mathcal{R}_1) = \{[0,3],[0,4],[1,3],[1,4]\} \tag{2.28}$$

which is represented by the same relation from equation (2.26)

$$(x_1 \Longleftrightarrow 0) \lor (x_1 \Longleftrightarrow 1) \tag{2.29}$$

Finally, we use the solver function Pick on $R_1(x_1, x_2)$.

$$\mathsf{Pick}(R_1(x_1,x_2),x_2) = [4] \tag{2.30}$$

There are two possible solutions for x_2, [3] and [4] where the last one was randomly chosen.

As equation (2.29) indicates the same relation may represent different relation sets depending on which UoD is considered. When counting the number of solutions of a relation we usually decide the UoD from the variables that are included in the relation. In the case of relation (2.29) we have a univariate relation with two solutions. If we regard (2.29) as a result from an embedding into S_2 then there are four solutions to the relation.

2.2 Modeling

Ramadge-Wonham [104] has formulated a definition of DEDS in one sentence:

Definition 2.10 DES (DEDS)

> "A Discrete Event System (DES) is a dynamic system that evolves in accordance with the abrupt occurrence, at possibly unknown irregular intervals, of physical events."

□

To stress the dynamics of these systems even further they are often referred to as Discrete Event Dynamic Systems (DEDS) which is the term we will use.

From the definition above we have that a DEDS takes events as inputs. The dynamic of the DEDS depends on events to appear if any transition of the system should take place.

In this thesis we take a slightly different approach. We let the DEDS react on signals, i.e., the inputs and outputs of a DEDS are values changing as time elapses. The DEDS is *measuring* the values at the input at the time instants when the system makes a transition. When the transitions occur and in what pace are not determined by the environment of the system, but by the system on its own. This approach is more natural if we, e.g., regard DEDS in which a computer is polling some sensors and reacting on these measures in some way. The setups of control

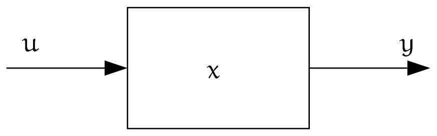


Figure 2.1: Input, output and state variables of a DEDS.

systems are often made in this way, even though there are cases where the event driven approach is more appropriate. The approaches can be translated to each other. Chapter 10 contains further discussion on this subject from the modeling perspective.

2.2.1 Relational Model

Modeling frameworks of a DEDS can vary in syntax and features. Common frameworks for modeling DEDS are, e.g., *finite automata* and *Petri Nets*, which have properties that make them easy to use and to interprete due to their convenient graphical appearances. See Part III.

Our purpose here is not to impose new modeling syntaxes or languages to model DEDS. We will instead focus on the mathematics behind DEDS and how to represent DEDS in a more mathematical way. The simple mathematical object that we will use throughout this thesis is the relation.

Definition 2.11 Elements of DEDS
Consider Figure 2.1. The *input* $u \in U$ is a received signal to the DEDS, and the *output* $y \in Y$ is a produced signal from the DEDS, where U is the *input domain* and Y the *output domain.*

The extra information that is necessary to compute the output y from the present input u, is called the *state*, $x \in X$, where X is the *state domain* or *state space.* □

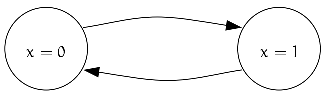
Perhaps the most important entity of dynamic systems is the state, which contains the "memory" necessary for describing the system behavior.

To represent the dynamics of a system we need a *transition function* that describes how the dynamics evolves. The transition function maps the present state x to the next state denoted by x^+.

Definition 2.12 Transition Function/Relation
The *transition function* $f : X \times U \to X$, $x^+ = f(x, u)$ and the *transition relation* $R_f : X \times U \times X \to \mathbb{B}$, $R(x, u, x^+)$ each represent the dynamics of the DEDS, i.e., how the *next state* x^+ depends on x and u. □

Consider the following example:

Example 2.5 Flip Flop



The transition function $f : \mathbb{B} \to \mathbb{B}$ of the toggling flip flop in the figure above is

$$x^+ = f(x) = \neg x \tag{2.31}$$

We are now ready to introduce the object that will be used to represent (or model) the DEDS in this thesis.

Definition 2.13 Relational Model
The *relational model*, denoted

$$M(z, z^+) \tag{2.32}$$

is a relation representing a mapping from *system variables* z to the next system variables z^+.

Remark:

- We let z and z^+ denote a set of variables as

  $$z = \{z_1, z_2, \dots, z_n\} \tag{2.33}$$
  $$z^+ = \{z_1^+, z_2^+, \dots, z_n^+\} \tag{2.34}$$

  if the variables are not partitioned in input, output and state variables.

  Another example is

  $$z = \{x, u, y\} \tag{2.35}$$
  $$z^+ = \{x^+, u^+, y^+\} \tag{2.36}$$

  where x, u and y in order denote the state, the input and the output.

- All variables in z and z^+ do not have to be specified by the relation in $M(z, z^+)$. In that case they are free to take any value, i.e., the value of z_i can be any element in S_i if z_i is not specified by $M(z, z^+)$. The relation set of $M(z, z^+)$, where z and z^+ each contains n variables, will be a subset of $\mathcal{U}_{2n}$.

$\square$

2.2.2 Deterministic/Nondeterministic DEDS

A *deterministic* system has only one possible behavior, given present state and inputs. A *nondeterministic* system may take any of several possible trajectories given present state and inputs. Compare with deterministic/nondeterministic finite automata (DFA/NFA) [68].

Deterministic DEDS with inputs and outputs can be modeled by an explicit state space form such as

$$\begin{aligned} x^+ &= f(x,u) \\ y &= g(x,u) \end{aligned} \tag{2.37}$$

where f is the *transition function* and g is the *output function*.

An explicit state space form can be encapsulated into a relation model, by reformulating (2.37) into two equations which are then combined by an $\wedge$-operator.

$$M(z,z^+) = (x^+ == f(x,u)) \wedge (y == g(x,u)) \tag{2.38}$$

where $z = \{x, y, u\}$.

To go the other way around, we start by a deterministic relational model $M(z,z^+)$. We can then extract the equations in (2.37) if the transition function f and the output function g are defined for all values of x and u. In this case we say that f and g are *non partial*.

$$\exists y . M(z,z^+) = (x^+ == f(x,u)) \tag{2.39}$$

$$\exists x^+ . M(z,z^+) = (y == g(x,u)) \tag{2.40}$$

For nondeterministic systems there are no functions f and g as in (2.37) describing the behavior. Instead we use the relations

$$R_f(x^+, x, u) \tag{2.41}$$

$$R_g(y, x, u) \tag{2.42}$$

where R_f and R_g are transition and output relations, respectively. The encapsulation into a relational model is immediate:

$$M(z,z^+) = R_f(x^+, x, u) \wedge R_g(y, x, u) \tag{2.43}$$

where $z = \{x, u, y\}$.

The extraction of R_f and R_g from $M(z,z^+)$ is similar as for deterministic relation models, see (2.39) above.

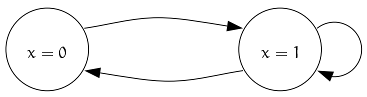
Example 2.6 Deterministic/Nondeterministic DEDS

The toggling flip flop system from Example 2.5 is a deterministic system which can be encapsulated into the relational model

$$M(z,z^+) = (x^+ == \neg x) = (x^+ \leftrightarrow \neg x) \tag{2.44}$$

where $z = \{x\}$ and we use the fact that x is Boolean in the last equality.

The relational model for the nondeterministic system



where $0 = \texttt{false}$ and $1 = \texttt{true}$ can be derived as follows

$$\begin{aligned}
M(z,z^+) &= (x^+ {=\!=} \texttt{true}) \wedge (x {=\!=} \texttt{false}) \vee (x^+ {=\!=} \texttt{false}) \wedge (x {=\!=} \texttt{true}) \vee \\
&\quad (x^+ {=\!=} \texttt{true}) \wedge (x {=\!=} \texttt{true}) \\
&= (x^+ \leftrightarrow \texttt{true}) \wedge (x \leftrightarrow \texttt{false}) \vee (x^+ \leftrightarrow \texttt{false}) \wedge (x \leftrightarrow \texttt{true}) \vee \\
&\quad (x^+ \leftrightarrow \texttt{true}) \wedge (x \leftrightarrow \texttt{true}) \\
&= x^+ \wedge (\neg x) \vee (\neg x^+) \wedge x \vee x^+ \wedge x \\
&= x^+ \wedge (\neg x) \vee x \\
&= x^+ \vee x \qquad (2.45)
\end{aligned}$$

2.2.3 Symmetric Models

Relational models contain a relation representing a mapping between the system variables z and the next system variables z^+. In addition, relational models also contain relations over variables from the same time instant. As an example we have the output relation $R_g(y,x,u)$ in (2.42) which relates the system variables x, u and y to each other. These variables will always obey this relation for all time instants, and therefore we might add the relation $R_g(y^+,x^+,u^+)$ as well to the relational model. We will then have the same "information" in the variables y, x and u in two time instants.

Imposing this seemingly redundant information is desirable for some analysis methods where it is convenient to have a symmetrical representation, see Section 2.3.2.

For example, backward simulation is common in some analysis methods. In this case we will not have all information about the variables in the present state.

Example 2.7 Backward System
Let $M(z,z^+)$ be the deterministic relational model

$$M(z,z^+) = (x^+ {=\!=} f(x,u)) \wedge (y {=\!=} g(x,u)) \qquad (2.46)$$

The backward system $M_b(z,z^+)$ is then obtained by reversing the order of z and z^+:

$$M_b(z,z^+) = M(z^+,z) \qquad (2.47)$$

which results in the following relation

$$M_b(z,z^+) = (x {=\!=} f(x^+,u^+)) \wedge (y^+ {=\!=} g(x^+,u^+)) \qquad (2.48)$$

As we see from (2.48) there is no output relation at the present time instant. To add a relation $R(y)$, constraining the present output, to $M_b(z, z^+)$ will not have an effect since y is a free variable in $M_b(z, z^+)$.

We will define *symmetric relational models* which contain the same information on system variables in both the present and the next time instant. A symmetric relational model $M(z, z^+)$ has the same projection on z as on z^+, with respect to the relation set of $M(z, z^+)$.

Definition 2.14 Symmetric Relational Model
For a relational model $M(z, z^+)$, let

$$R(z) = \exists z^+.\, M(z, z^+) \tag{2.49}$$

and

$$R^+(z^+) = \exists z.\, M(z, z^+). \tag{2.50}$$

The relational model $M(z, z^+)$ is *symmetric* iff $R(z) = R^+(z)$. □

To make a non symmetric model $M(z, z^+)$ symmetric we do the following

$$M_s(z, z^+) = M(z, z^+) \wedge R(z^+) \wedge R^+(z) \tag{2.51}$$

where M_s is the symmetric version of M, and R and R^+ according to Definition 2.14 above.

There must be transitions going to and from each state in a symmetric relational model, since all states must be included in the intersection of the domains for z and z^+.

To compose a symmetric model from an explicit state space form we add the output equation in both the present and the next instant:

$$M(z, z^+) = (x^+ {==} f(x, u)) \wedge (y {==} g(x, u)) \wedge (y^+ {==} g(x^+, u^+)) \tag{2.52}$$

This works only if f and g are non partial, see (2.39). If this is not fulfilled we can write the transition and output relations as

$$(x^+ {==} f(x, u)) \wedge P_f(x, u) \tag{2.53}$$

and

$$(y {==} g(x, u)) \wedge P_g(x, u) \tag{2.54}$$

where P_f and P_g are constraints for x and u represented by relations.

The symmetric relational model will then be

$$\begin{aligned} M(z, z^+) = {} & (x^+ {==} f(x, u)) \wedge (y {==} g(x, u)) \wedge (y^+ {==} g(x^+, u^+)) \wedge \\ & P_f(x, u) \wedge P_g(x, u) \wedge P_f(x^+, u^+) \wedge P_g(x^+, u^+) \end{aligned} \tag{2.55}$$

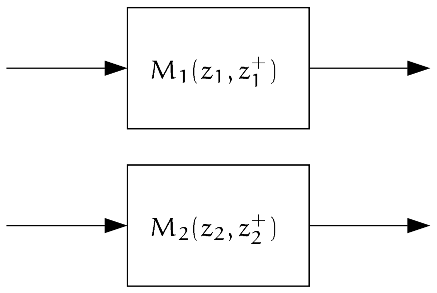


Figure 2.2: Two DEDS.

2.2.4 Composition of Models

When combining models to a single model we form a *product* of the models.

Consider the two DEDS $M_1(z_1, z_1^+)$ and $M_2(z_2, z_2^+)$ in Figure 2.2. We want to combine these models into one model $M(z, z^+)$. This can be done in three different ways which differ in the ordering of the transitions of the systems: *synchronously*, *asynchronously* and *interleaved*

Before we present these three products we will specify how to combine the system variables. If $M(z, z^+)$ is the total model including $M_1(z_1, z_1^+)$ and $M_2(z_2, z_2^+)$ then $z = z_1 \cup z_2$ and $z^+ = z_1^+ \cup z_2^+$.

Synchronous Product

Synchronous composition means that the transitions (or evaluations) of $M_1(z_1, z_1^+)$ and $M_2(z_2, z_2^+)$ are synchronous.

Definition 2.15 Synchronous Product
The *synchronous product* ($\|$) of $M_1(z_1, z_1^+)$ and $M_2(z_2, z_2^+)$ is

$$M_1(z_1, z_1^+) \parallel M_2(z_2, z_2^+) = M_1(z_1, z_1^+) \land M_2(z_2, z_2^+) \tag{2.56}$$

□

This is the most common composition performed in this thesis and will often be used without any remarks.

Example 2.8 Synchronous Product
Let the relational models $M_1(z_1, z_1^+)$ and $M_2(z_2, z_2^+)$ represent the finite automata in Figure 2.3. We regard the labels of the automata as inputs and let each symbol be represented by a Boolean variable. State variables for M_1 and M_2 are x_1 and x_2, respectively.

The relational models for M_1 and M_2 are

$$\begin{aligned} M_1(z, z^+) = (x{=}A) \land (x^+{=}B) \land a \lor \\ (x{=}B) \land (x^+{=}A) \land b \end{aligned} \tag{2.57}$$

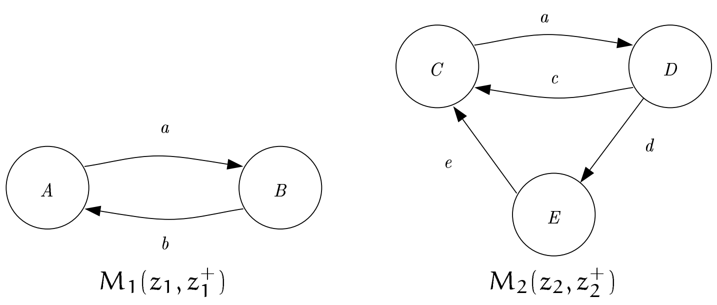
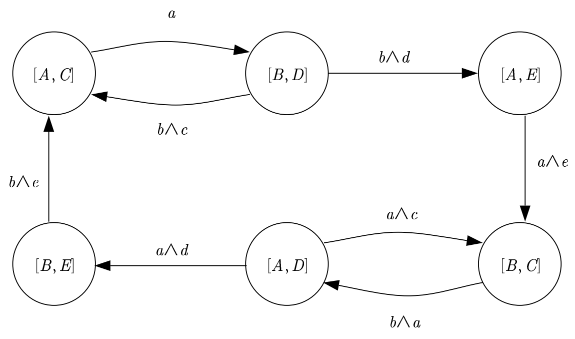


Figure 2.3: Two finite automata where *A,B,C,D,E* are states and the transition conditions *a,b,c,d,e* are inputs. Note that the input *a* is common for both automata.

and

$$\begin{aligned} M_2(z, z^+) = (x{=}C)\wedge(x^+{=}D)\wedge a\ \vee \\ (x{=}D)\wedge(x^+{=}C)\wedge c\ \vee \\ (x{=}D)\wedge(x^+{=}E)\wedge d\ \vee \\ (x{=}E)\wedge(x^+{=}C)\wedge e \end{aligned} \tag{2.58}$$

The synchronous composition is represented by the automaton



The synchronous composition forces both M_1 and M_2 to have synchronous transitions. This will in turn impose constraints on the inputs of the two systems. As indicated on the transition labels we have constraints on pairs of inputs. One input from M_1 and one from M_2 must be `true` simultaneously.

Asynchronous (Parallel) Product

Asynchronous or parallel composition means that the transitions of $M_1(z_1, z_1^+)$ and $M_2(z_2, z_2^+)$ are asynchronous. This means that one system can have a transition while the other system rests, but simultaneous transitions are allowed also.

A first suggestion to such a composition might be

$$\mathsf{M}_1(z_1, z_1^+) \vee \mathsf{M}_2(z_2, z_2^+) \tag{2.59}$$

where only one of the relational models must be `true` simultaneously. However (2.59) will not work as asynchronous composition. If, for example, M_1 is `true` then the variables z_2 and z_2^+ are free, which means that any values of z_2 and z_2^+ are valid in the model.

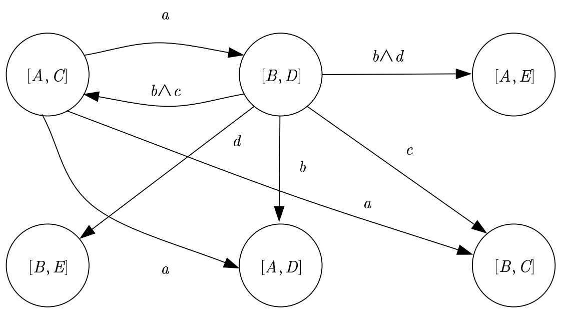
Instead, asynchronous composition is defined as follows.

Definition 2.16 Asynchronous Product
The *asynchronous product* ($\|_{\mathsf{A}}$) of $\mathsf{M}_1(z_1, z_1^+)$ and $\mathsf{M}_2(z_2, z_2^+)$ is

$$\begin{aligned}
\mathsf{M}_1(z_1, z_1^+) \|_{\mathsf{A}} \mathsf{M}_2(z_2, z_2^+) = {} & \mathsf{M}_1(z_1, z_1^+) \wedge \mathsf{M}_2(z_2, z_2^+) \vee && (2.60)\\
& \mathsf{M}_1(z_1, z_1^+) \wedge (z_2 \equiv z_2^+) \vee && (2.61)\\
& \mathsf{M}_2(z_2, z_2^+) \wedge (z_1 \equiv z_1^+) && (2.62)
\end{aligned}$$

Remark: Equation (2.60) is the relation for simultaneous transition. Equation (2.61) for transitions only in M_1, and (2.62) for transitions only in M_2. □

Example 2.9 Asynchronous Product
The asynchronous composition of M_1 and M_2 from Example 2.8 is illustrated by the automaton



where only transitions going from $[A, C]$ and $[B, D]$ are shown for clarity.

The resulting model is non deterministic which is natural since we have two systems connected but with unknown transition order.

Interleaved Product

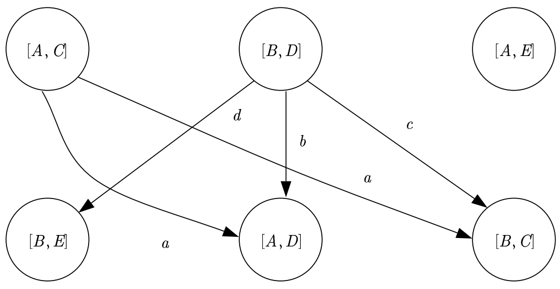
Interleaved composition means that a transition occurs either in $\mathsf{M}_1(z_1, z_1^+)$ or $\mathsf{M}_2(z_2, z_2^+)$ and *not* simultaneously.

Definition 2.17 Interleaved Product
The *interleaved product* ($\|_I$) of $M_1(z_1, z_1^+)$ and $M_2(z_2, z_2^+)$ is

$$M_1(z_1, z_1^+) \|_I M_2(z_2, z_2^+) = M_1(z_1, z_1^+) \wedge (z_2 {=\!=} z_2^+) \vee \tag{2.63}$$
$$M_2(z_2, z_2^+) \wedge (z_1 {=\!=} z_1^+) \tag{2.64}$$

□

Example 2.10 Interleaved Product
The interleaved composition of M_1 and M_2 from Example 2.8 is illustrated by the automaton



where only transitions going from $[A, C]$ and $[B, D]$ are shown for clarity.

The resulting model is non deterministic which natural since we have two systems connected but with unknown transition order. The interleaved model has the transitions of the asynchronous model except those changing state in both M_1 and M_2.

Our interpretation of the three composition methods synchronous, asynchronous and interleaved, is similar but not the same as the corresponding terms in the theory based on Ramadge-Wonham. For this event based framework, [33, 63], the interpretation of different synchronizations only concerns mutual events, i.e., events that are shared between the systems. This means that the the system M_1 can have transitions independently of M_2 for events that are not mutual to events of M_2 regardless of the type of composition used. Only mutual event can be synchronous, asynchronous or interleaved for a system composition in an event based framework.

2.2.5 Interconnecting Models

Suppose we have two systems M_1 and M_2 where the output y_1 of M_1 and the input u_2 of M_2 are connected, we compose an interconnected model by performing a product on M_1 and M_2 and then adding the constraint $y_1 = u_2$.

For the systems in Figure 2.4 we get the interconnected system $M(z, z^+)$, where $z = z_1 \cup z_2$ as

$$M(z, z^+) = (M_1(z_1, z_1^+) \| M_2(z_2, z_2^+)) \wedge (y_1 {=\!=} u_2) \tag{2.65}$$

By adding such a *connection relation* for all variables involved in the interconnection, we get the model for the total system. In the case of systems connected

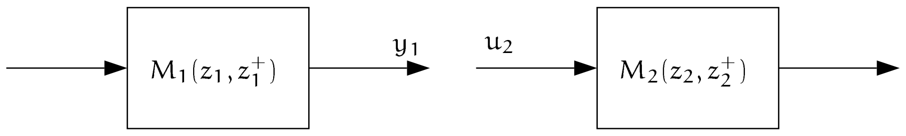


Figure 2.4: Two DEDS connected in series where $y_1 \in z_1$ and $u_2 \in z_2$.

into a loop, this method still works but in some cases we connect loops in a different way.

Example 2.11 Loop Connection
Assume that the systems M_1 and M_2 contain the explicit state space forms

$$\begin{aligned} x_1^+ &= f_1(x_1, u_1) \\ y_1 &= g_1(x_1, u_1) \end{aligned} \tag{2.66}$$

and

$$\begin{aligned} x_2^+ &= f_2(x_2, u_2) \\ y_2 &= g_2(x_2, u_2) \end{aligned} \tag{2.67}$$

respectively. Assume that these systems are connected synchronously into a loop:

$$M(z, z^+) = (M_1(z_1, z_1^+) \parallel M_2(z_2, z_2^+)) \wedge (y_1 == u_2) \wedge (y_2 == u_1) \tag{2.68}$$

Then the relation $M(z, z^+)$ will contain the expression

$$(y_1 == g_1(x_1, u_1)) \wedge (y_2 == g_2(x_2, u_2)) \wedge (y_1 == u_2) \wedge (y_2 == u_1) \tag{2.69}$$

By some substitution and elimination of output variables we get the relation

$$u_2 == g_1(x_1, g_2(x_2, u_2)) \tag{2.70}$$

Thus, we get an algebraic loop which imposes a new relation on the connected variables.

From the example above we see that a loop connection may create new restrictions in terms of new relations. This relation might be empty ($\equiv$ `false`) which will directly falsify the closed system $M(z, z^+)$. The reason for this is of course a consequence of the fact that these two systems cannot be connected in a loop this way. But in a more pragmatic perspective, when a controller and a plant is interconnected, we might have this situation even though we have done everything "right".

The problem can be solved by adding a delay somewhere in the loop. This approach is reasonable for practical applications, where transitions and information transportation take some amount of time.

For the systems in Example 2.11 above, we replace the last connection relation $y_2 = u_1$ by $y_2 = u_1^+$, i.e., the present y_2 will be the next u_1. By this we have a relation between u_1 and u_1^+, i.e., the input variable u_1 has become a state variable.

Example 2.12 Delayed Loop Connection
The systems from Example 2.11 are connected into a loop using a delay:

$$M(z, z^+) = (M_1(z_1, z_1^+) \parallel M_2(z_2, z_2^+)) \wedge (y_1 \equiv u_2) \wedge (y_2 \equiv u_1^+) \tag{2.71}$$

Some of the relations in $M(z, z^+)$ are

$$(y_1 \equiv g_1(x_1, u_1)) \wedge (y_2 \equiv g_2(x_2, u_2)) \wedge (y_1 \equiv u_2) \wedge (y_2 \equiv u_1^+) \tag{2.72}$$

By some substitution and elimination similar to the previous example we get

$$u_1^+ \equiv g_2(x_1, g_1(x_1, u_1)) \tag{2.73}$$

which shows that we have a mapping between u_1 and u_1^+. Therefore the number of states has increased because of the loop.

2.2.6 Restricting Models

It is often useful to be able to constrain the behavior of a system $M(z, z^+)$ in certain ways. A common operation performed on models is to *restrict* the system, which means that a relation corresponding so some constraint is added symmetrically to a relational model.

We make the following definition.

Definition 2.18 Restriction of Models
Let $M(z, z^+)$ be a relational model and $R(z)$ a relation representing some property. We define the function $\mathsf{Restrict}(M(z, z^+), R(z))$ to return a relational model with restricted behavior as follows

$$\mathsf{Restrict}(M(z, z^+), R(z)) = R(z) \wedge M(z, z^+) \wedge R(z^+) \tag{2.74}$$

$\square$

To restrict a relational model such that the system cannot reach forbidden states is common for synthesis methods for DEDS, see Section 2.4.

2.3 Analysis

Having a model of the DEDS behavior we can make different types of analysis. We will divide the analysis into static and dynamic analysis. By static analysis we mean analysis performed on the model in one single iteration, and by dynamic analysis we mean analysis performed on a systems dynamical behavior in any number of steps.

2.3.1 Static Analysis

Static analysis can be used to test if certain relations are included in the model or not. Normally, the model does not only contain dynamic relations, but also relations between the system variables in the model, e.g., the static analysis performed on the landing gear application in Section 7.4.2 verifies if two different outputs can be given simultaneously.

We will present an example of what we mean by static analysis.

Example 2.13 Static Analysis
Let $M(z, z^+)$ be a relational model and $Q(z, z^+)$ a relation corresponding to a property or constraint on the variables z and z^+. Then $Q(z, z^+)$ holds for the model $M(z, z^+)$ if

$$M(z, z^+) \to Q(z, z^+) = \mathtt{true} \tag{2.75}$$

but $Q(z, z^+)$ cannot hold if

$$M(z, z^+) \wedge Q(z, z^+) = \mathtt{false}. \tag{2.76}$$

A variant of (2.75) is to test if the model $M(z, z^+)$ and a specification $S(z, z^+)$ together will make the condition $Q(z, z^+)$ hold. This is true if

$$(M(z, z^+) \wedge S(z, z^+)) \to Q(z, z^+) = \mathtt{true} \tag{2.77}$$

Suppose we consider the special case where $Q(x)$ represents a subset of the state space and $x \subset z$. Then the model $M(z, z^+)$ is defined for all states in this set if

$$Q(x) \to \exists z^+, (z \setminus x).\, M(z, z^+) = \mathtt{true} \tag{2.78}$$

where we eliminate all but the state variables by an existential quantification before we test the implication.

We can also detect the existence of *dead locks* in a system by using static analysis. Let $M(x, u, x^+)$ be a model with state variable x and input variable u. The system M does not contain any dead lock states iff

$$\forall u.\, \big(\exists x^+.\, M(x, u, x^+) \to M(x, u, x)\big) = \mathtt{false}. \tag{2.79}$$

If the quantification above is not `false` then we have as a result a relation $R(x)$ which represents the dead lock states.

2.3.2 Dynamic Analysis

By dynamic analysis we mean analysis which takes any number of iterations of the model into account. One of the most important dynamic analyses is the computation of the set including all reachable states from a given initial state. In dynamic analysis the initial state is important information which might affect the behavior of the model drastically. We can for instance verify that two outputs cannot ever be given simultaneously for a given initial state by using dynamic analysis, even though the static analysis has indicated that possibility.

Dynamic (or multiple step) analysis gives more information about the behavior of the system than static analysis. Dynamic analysis answers questions over arbitrarily many time steps. The results from dynamic analysis may be a simple `true` or `false` or a set of states, e.g., the set of states that are reachable in zero or arbitrarily many steps from some initial state.

Reachable States

Given a model $M(z, z^+)$ we can compute the set of states $R_k(z)$ reachable in k steps or less from some initial set of states $I(z)$ as:

$$\begin{aligned} R_0(z) &= I(z) \\ R_{k+1}(z) &= R_k(z) \vee (\exists \tilde{z}.\ (R_k(\tilde{z}) \wedge M(\tilde{z}, z))) \end{aligned} \tag{2.80}$$

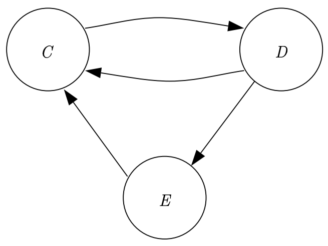
In each iteration we compute a relation representing all states that are reachable from the states represented by $R_k(z)$. This set is added by union (relation operation $\vee$) to $R_k(z)$.

Since we are dealing with finite state systems this iteration will eventually reach a fixed point, i.e., $R_d(z) = R_{d+1}(z)$ for some finite d. The number of steps d is the *depth* of the system which in most engineering applications is far below its maximal possible depth which is 2^n for an n variable binary system. The depth of the model from the case study of a Landing Gear System (LGS) in Section 7.4.1 is 5, which is far less than the maximal, which is 2^{26}.

Alternative methods such as testing or simulation are infeasible for complex systems, e.g., in the LGS we have in the order of 10 000 reachable states out of 2^{26} potentially reachable states.

Example 2.14 Reachable States

Reconsider the simple system from Example 2.8, i.e.,



where the inputs are omitted. The relational model is

$$\begin{aligned} M(x, x^+) = &(x = C) \wedge (x^+ = D) \vee \\ &(x = D) \wedge (x^+ = C) \vee \\ &(x = D) \wedge (x^+ = E) \vee \\ &(x = E) \wedge (x^+ = C). \end{aligned} \tag{2.81}$$

We can now compute the set of reachable states from state *C*, using (2.80) iteratively

$$\begin{aligned}
R_0(x) &= I(x) = (x == C) \\
R_1(x) &= (x == C) \lor (x == D) \\
R_2(x) &= (x == C) \lor (x == D) \lor (x == E) \\
R_3(x) &= (x == C) \lor (x == D) \lor (x == E).
\end{aligned}$$

Hence we reach a fixed point in $k = 2$ steps, i.e., we can reach any reachable state in two steps.

In this example we could not have found out that *E* is a reachable state just by static 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 *E*.

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

We will define four generic reachable state operators which will be denoted by a more complete notation where $M(z, z^+)$ is the relational model and $I(z)$ is the initial state set. For clarity, the arguments z and z^+ are not always given for the model and the initial state set.

Definition 2.19 Reachable States Operators
We define $\gamma_k^+[M, I](z)$, $\gamma_k^-[M, I](z)$, $\Gamma_k^+[M, I](z)$ and $\Gamma_k^-[M, I](z)$ as follows:

Forward: Computes the set of forward reachable states $\gamma_k^+[M, I](z)$ from the set $I(z)$ in *exactly* k steps:

$$\begin{aligned}
\gamma_0^+[M, I](z) &= I(z) \\
\gamma_{k+1}^+[M, I](z) &= \exists \tilde{z}.\ (\gamma_k^+[M, I](\tilde{z}) \land M(\tilde{z}, z))
\end{aligned} \tag{2.82}$$

Backward: Computes the set of backward reachable states $\gamma_k^-[M, I](z)$ from the set $I(z)$ in *exactly* k steps:

$$\begin{aligned}
\gamma_0^-[M, I](z) &= I(z) \\
\gamma_{k+1}^-[M, I](z) &= \exists \tilde{z}.\ (\gamma_k^-[M, I](\tilde{z}) \land M(z, \tilde{z}))
\end{aligned} \tag{2.83}$$

Accumulated Forward: Computes the set of forward reachable states $\Gamma_k^+[M, I](z)$ from the state set $I(z)$ in k *or less* steps. This is exactly the same formulation as in (2.80):

$$\begin{aligned}
\Gamma_0^+[M, I](z) &= I(z) \\
\Gamma_{k+1}^+[M, I](z) &= \Gamma_k^+[M, I](z) \lor (\exists \tilde{z}.\ (\Gamma_k^+[M, I](\tilde{z}) \land M(\tilde{z}, z)))
\end{aligned} \tag{2.84}$$

Accumulated Backward: Computes the set of backward reachable states $\Gamma_k^-[M, I](z)$ from the state set $I(z)$ in k *or less* steps:

$$\begin{aligned}\Gamma_0^-[M, I](z) &= I(z)\\ \Gamma_{k+1}^-[M, I](z) &= \Gamma_k^-[M, I](z) \vee (\exists \tilde{z}.\ (\Gamma_k^-[M, I](\tilde{z}) \wedge M(z, \tilde{z})))\end{aligned} \tag{2.85}$$

Remark: To compute the set of reachable states in any number of steps, $\Gamma_\infty^+[M, I](z)$ we perform a fixed point computation, i.e.,

$$\Gamma_\infty^+[M, I](z) = \lim_{k \to \infty} \Gamma_k^+[M, I](z) \tag{2.86}$$

$\Gamma_\infty^-[M, I](z)$ is computed similarly but for backward reachable states. □

Verification by Temporal Logic

Dynamic analysis of a system requires not only a model of the system, but also a specification of the desired behavior or statements describing error trajectories that the dynamic analysis is supposed to falsify. These statements, or specifications, are often not written formally. Instead, natural languages are used to document information of the specification.

In our framework we have used *temporal logic* to formally represent a specification given in a natural language. Temporal logic allows us to impose temporal conditions on the behavior, which means that we can have words like: *always, never, in future* and *after* in our behavioral specifications. We will also use temporal logic in the modeling of the landing gear process (in Chapter 7), where we build a discrete model of a continuous system. By using temporal logic for modeling we gain more expressiveness in the modeling process which in this case clearly reduces the complexity of the problem. Section 7.4.3 will show this and also give some insights into the dynamic analysis performed in the application of the landing gear system in Chapter 7. In Table 2.1 some of the most common temporal logic constructs are given.

Our interpretation of temporal logic is an extension of the specification language *computation tree logic* (CTL) [25], which has been adapted for the relational framework over finite domains in [40].

To verify if a relational model $M(z, z^+)$ fulfills a specification represented by a temporal logic expression $f(z)$ we compute the set of states from which the temporal logic statement becomes true. If this set is the complete state space, then we know that the temporal logic expression will hold unconditionally.

Definition 2.20 Implementation of Temporal Logic Verification
Assume that we have a relational model $M(z, z^+)$ and a temporal logic expression f with the temporal logic operators from Table 2.1. The verification, denoted $\mathsf{Verify}(M, f)(z)$, will return a relation in the system variables z and is performed as follows:

Temporal Language	Natural Language
$\mathsf{P}(z)$	$\mathsf{P}(z)$ holds in the initial state, z.
$\mathsf{EX}[\mathsf{P}(z)]$	$\mathsf{P}(z)$ can hold in the next time step.
$\mathsf{EU}[\mathsf{P}_1(z), \mathsf{P}_2(z)]$	$\mathsf{P}_1(z)$ will hold for finitely many steps and then $\mathsf{P}_2(z)$ can hold in the next step.
$\mathsf{EF}[\mathsf{P}(z)]$	$\mathsf{P}(z)$ can hold at some future time.
$\mathsf{EG}[\mathsf{P}(z)]$	$\mathsf{P}(z)$ can hold at all future times, i.e., from this point and onwards.
$\mathsf{AX}[\mathsf{P}(z)]$	$\mathsf{P}(z)$ must hold in the next time step.
$\mathsf{AU}[\mathsf{P}_1(z), \mathsf{P}_2(z)]$	$\mathsf{P}_1(z)$ will hold for finitely many steps and then $\mathsf{P}_2(z)$ must hold.
$\mathsf{AF}[\mathsf{P}(z)]$	$\mathsf{P}(z)$ must hold at some future time.
$\mathsf{AG}[\mathsf{P}(z)]$	$\mathsf{P}(z)$ must hold at all future times, i.e., from this point and onwards.

Table 2.1: Temporal logic operators. The term *can hold* should be interpret: *there must exist at least one trajectory such that it holds.*

Atomic expressions: Let f be an atomic expression. Here we mean atomic in the context of temporal logic expressions, i.e., relational expressions without temporal logic operators.

$$\mathsf{Verify}(M, f) = f$$

Combined expressions:

$$\begin{aligned}
\mathsf{Verify}(M, \neg f) &= \neg\mathsf{Verify}(M, f)\\
\mathsf{Verify}(M, f \wedge g) &= \mathsf{Verify}(M, f) \wedge \mathsf{Verify}(M, g)\\
\mathsf{Verify}(M, f \vee g) &= \mathsf{Verify}(M, f) \vee \mathsf{Verify}(M, g)\\
\mathsf{Verify}(M, f \to g) &= \mathsf{Verify}(M, f) \to \mathsf{Verify}(M, g)\\
\mathsf{Verify}(M, f \leftrightarrow g) &= \mathsf{Verify}(M, f) \leftrightarrow \mathsf{Verify}(M, g)
\end{aligned}$$

Next state expressions:

$$\begin{aligned}
\mathsf{Verify}(M, \mathsf{EX}[f]) &= \exists \tilde{z}.\ \big(M(z, \tilde{z}) \wedge \mathsf{Verify}(M, f)(\tilde{z})\big)\\
\mathsf{Verify}(M, \mathsf{AX}[f]) &= \forall \tilde{z}.\ \big(M(z, \tilde{z}) \to \mathsf{Verify}(M, f)(\tilde{z})\big)
\end{aligned}$$

where $a \to b = \neg a \vee b$.

Future state expressions: Future state expressions use a fixed point computation denoted by $\lim_{k\to\infty}$.

$$\begin{aligned}
\mathsf{Verify}(M, \mathsf{EU}[f,g]) &= \lim_{k\to\infty} \mathsf{Verify}(M, \mathrm{EUL}_k[f,g]) \\
\mathsf{Verify}(M, \mathsf{AU}[f,g]) &= \lim_{k\to\infty} \mathsf{Verify}(M, \mathrm{AUL}_k[f,g]) \\
\mathsf{Verify}(M, \mathsf{EF}[f]) &= \mathsf{Verify}(M, \mathsf{EU}[\mathtt{true}, f]) \\
\mathsf{Verify}(M, \mathsf{EG}[f]) &= \mathsf{Verify}(M, \neg\mathsf{AU}[\mathtt{true}, \neg f]) \\
\mathsf{Verify}(M, \mathsf{AF}[f]) &= \mathsf{Verify}(M, \mathsf{AU}[\mathtt{true}, f]) \\
\mathsf{Verify}(M, \mathsf{AG}[f]) &= \mathsf{Verify}(M, \neg\mathsf{EU}[\mathtt{true}, \neg f])
\end{aligned}$$

where

$$\begin{aligned}
\mathrm{EUL}_{k+1}[f,g] &:= g \lor f \land \mathsf{EX}[\mathrm{EUL}_k[f,g]] \\
\mathrm{EUL}_0[f,g] &:= g.
\end{aligned}$$

and

$$\begin{aligned}
\mathrm{AUL}_{k+1}[f,g] &:= g \lor f \land \mathsf{AX}[\mathrm{AUL}_k[f,g]] \\
\mathrm{AUL}_0[f,g] &:= g.
\end{aligned}$$

□

Example 2.15 Temporal Logic Verification

Consider the process from Example 2.14. We wish to verify the specification:

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

In terms of temporal logics this becomes:

$$\mathsf{EX}[x == D]$$

The actual verification is then performing the computations:

$$\begin{aligned}
&\mathsf{Verify}(M(x,x^+), \mathsf{EX}[x == D]) = \\
&\qquad = \exists x^+.\ \big(M(x,x^+) \land (x^+ == D)\big) \\
&\qquad = (x == C)
\end{aligned}$$

As expected this returns the state C, since this is the only state from which we can reach D in one step. Now, suppose that we have the model *and* an initial state specified, then the above temporal logic formula would be verified iff the returned set of states was a superset of the reachable states, i.e., we could reach D from every reachable state. In the case above this is clearly not the case, since the set of reachable states is $\{C, D, E\}$. Generally this extra level of reasoning is of course built into our verifier.

The verification of temporal logic expressions requires the same type of fixed point computations as was used in the reachability analysis above. For more details regarding temporal algebra (or temporal logic), see [25].

To conclude this section we will present an example which shows how to check if a system is *nonblocking* [104].

Example 2.16 Nonblocking
Let $\mathsf{m}(z)$ be a relation for the *marked states* which can be regarded as final states after that some task is completed. A nonblocking system $\mathsf{M}(z, z^+)$ is then a system where the marked state set $\mathsf{m}(z)$ is always reachable. This property can be expressed by using temporal logic.

A system $\mathsf{M}(z, z^+)$ with marked states $\mathsf{m}(z)$ is *nonblocking* iff

$$\mathsf{Verify}(\mathsf{M}(z, z^+), \mathsf{EF}[\mathsf{m}(z)]) = \mathtt{true} \tag{2.87}$$

2.4 Synthesis

By synthesis we mean to synthesize a controller, denoted $\mathsf{C}(z, z^+)$, which is a relational model, such that $\mathsf{C}(z, z^+)$ synchronously composed with the *open loop model* $\mathsf{M}(z, z^+)$ gives the *closed loop model* $\mathsf{G}(z, z^+)$ as

$$\mathsf{G}(z, z^+) = \mathsf{M}(z, z^+) \wedge \mathsf{C}(z, z^+) \tag{2.88}$$

Depending on the construction of $\mathsf{C}(z, z^+)$ the expression (2.88) holds for controllers mapping the output of the open loop model to the input, but also for other types of interconnections of controllers and systems, e.g., a controller performing prefiltering on the inputs.

This section will describe the relational framework solutions of two generic synthesis problems of DEDS: the *forbidden state problem* and the opposite *goal state problem*. The latter is also known as *planning* in the context artificial intelligence. See Allen et al. [3] and Shapiro [105] for an overview.

Synthesizing a controller $\mathsf{C}(z, z^+)$ in our relational framework returns a relational model including the complete feedback controller, i.e., the controller contains control information for all states of the system. Feedback control is in artificial intelligence known as *reactive planning*. See Lyons and Hendriks [85] for an overview.

Approaches like feedback control and reactive planning are in contrast with planning methods which produce a single trajectory which solves the problem if initial and final states are specifically given. These methods are efficient for a restricted class of problems. Even though the relational framework is less efficient in some cases the result covers all trajectories and the computation does not have to be repeated for every new initial state.

The performance of methods for complete controller and methods for partial plans is highly dependent on the structure of the problem and the purpose of the synthesis task. Further research and realistic test cases are needed to be able to produce guidelines in this matter.

2.4.1 The Forbidden State Problem

The *forbidden state problem* is the problem of finding a controller that guarantees that the forbidden states are avoided by the closed loop system.

Definition 2.21 Forbidden State Problem
Let $M(x, u, x^+)$ be a relational model, $I(x)$ the set of initial states and $S_f(x)$ a set of forbidden states.

The problem of finding a relational model $C(x, u)$ such that the reachable states of the closed loop system

$$G(x, u, x^+) = M(x, u, x^+) \wedge C(x, u) \tag{2.89}$$

do not intersect with $S_f(x)$, provided that

$$I(x) \wedge S_f(x) = \texttt{false}, \tag{2.90}$$

is called *the forbidden state problem.* □

Germundsson [40] has formulated the solution in the relational framework with the following algorithm.

Algorithm 2.1 Forbidden State Problem

Input: A model $M(x, u, x^+)$ and forbidden states $S_f(x)$.

Output: A controller $C(x, u)$.

Let $SA_k(x)$ denote the set of allowed states in k steps and $C_k(x, u)$ the controller for the states in $SA_k(x)$.

Initialization:

$$SA_0(x) = \neg S_f(x) \tag{2.91}$$

$$C_0(x, u) = \exists x^+. \big(M(x, u, x^+) \wedge SA_0(x)\big) \tag{2.92}$$

Main loop: In each iteration we remove transitions from the allowed to the unallowed states.

$$\begin{aligned} C_{k+1}(x, u) = {} & C_k(x, u) \wedge \\ & \neg \exists x^+. \big(SA_k(x) \wedge M(x, u, x^+) \wedge \neg SA_k(x^+)\big) \end{aligned} \tag{2.93}$$

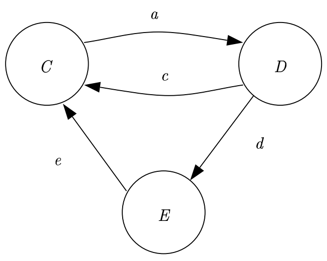
$$SA_{k+1}(x) = \exists u.\ C_{k+1}(x, u) \tag{2.94}$$

Exit condition: When the set of allowed states does not decrease any more.

$$SA_{k+1}(x) = SA_k(x) \tag{2.95}$$

□

Example 2.17 Forbidden State Problem
Reconsider the process from Example 2.14:



where we have the model

$$\begin{aligned}M(x,u,x^+) = (x{=}{=}\,C)\wedge(x^+{=}{=}\,D)\wedge a\ \vee\\ (x{=}{=}\,D)\wedge(x^+{=}{=}\,C)\wedge c\ \vee\\ (x{=}{=}\,D)\wedge(x^+{=}{=}\,E)\wedge d\ \vee\\ (x{=}{=}\,E)\wedge(x^+{=}{=}\,C)\wedge e\end{aligned} \tag{2.96}$$

where $u = \{a, c, d, e\}$.

Let $I(x) = (x{=}{=}\,C)$ be the initial state and $S_f(x) = (x{=}{=}\,E)$ the forbidden state.

The iterations of Algorithm 2.1 will then be

$$SA_0(x) = \neg(x{=}{=}\,E) = (x{=}{=}\,C)\ \vee\ (x{=}{=}\,D) \tag{2.97}$$

$$C_0(x,u) = (x{=}{=}\,C)\wedge a\ \vee\ (x{=}{=}\,D)\wedge c\ \vee\ (x{=}{=}\,D)\wedge d \tag{2.98}$$

$$C_1(x,u) = C_0(x,u)\wedge\neg\big((x{=}{=}\,D)\wedge d\big) = (x{=}{=}\,C)\wedge a\ \vee\ (x{=}{=}\,D)\wedge c \tag{2.99}$$

$$SA_1(x) = (x{=}{=}\,C)\ \vee\ (x{=}{=}\,D) \tag{2.100}$$

We have $SA_0(x) = SA_1(x)$ which means we have reached the fixed point. The controller is

$$C(x,u) = C_1(x,u) \tag{2.101}$$

The resulting controller removes transitions d and e from the process which will guarantee that the forbidden state E will never be reached.

Extensions and Variations

The forbidden state problem can be extended to other but similar types of problems.

One simple extension is to divide the inputs u of the system into *controllable inputs* u_c and *uncontrollable inputs* u_u. The uncontrollable inputs are those disturbances of the system that cannot be prevented by a controller. Algorithm 2.1 is modified so that the resulting controller will guarantee avoidance of forbidden states independently of the uncontrollable inputs. The controllable state space for the closed loop system will of course be smaller compared to the case when all inputs are controllable. See [40] for further details.

In addition to the modifications for controllable and uncontrollable inputs we may also replace the forbidden state space $S_f(x)$ by a specification model $M_s(x, u, x^+)$ which specifies both the allowed state space and allowed transitions. This is the problem setup of the classical Ramadge-Wonham approach [104], which in the relational framework is close to the forbidden state problem.

In the Ramadge-Wonham approach the problem of unobservable events is considered. In this problem class we let the model have an output mapping which specifies how the outputs from the system react upon inputs and states. This means that the state of the open loop system is not observable in general. The problem is to find a controller that only observes the system outputs but still guarantees that the closed loop behavior remains inside the behavior of the specification model. To make the problem non trivial we want the controller to be as *permissive* as possible, i.e., the closed loop behavior should be restricted as little as possible.

The solution to this problem in the relational framework is obtained by Germundsson [40] by reformulating the problem into an *output tracking problem*, where we find the smallest[2] system which behavior mimics the outputs from another system. The output tracking problem can in turn be reformulated into the forbidden state problem above.

The conclusion of this section is that the forbidden state problem can be used in several other types of problem setups, e.g, the Ramadge-Wonham approach. In Chapter 11 we make a comparison of how to apply the Ramadge-Wonham approach for the tank system application in Chapter 8.

2.4.2 The Goal State Problem (Planning)

The *goal state problem* is the problem of finding a controller that ensures that the closed loop system reaches a set of goal states. The difference between this problem setup and the forbidden state problem is that we want to reach a set of states in a *minimal* number of steps. Moreover we want the controller to work for all possible initial states of the system.

Definition 2.22 Goal State Problem
Let $M(x, u, x^+)$ be a relational model and let $S_g(x)$ represent the set of goal states. The *goal state problem* is to find a controller $C(x, u)$ such that the closed loop system

$$G(z, z^+) = M(x, u, x^+) \wedge C(x, u) \tag{2.102}$$

will always reach the state set $S_g(x)$ in a minimal number of steps.

Remark: Note that the initial state is not specified. □

Germundsson [40] has formulated the solution in the relational framework by the following algorithm.

[2]In the number of states.

Algorithm 2.2 Goal State Problem

Syntax: $\mathsf{Planner}(M(x, u, x^+), S_g(x))$

Input: A model $M(x, u, x^+)$ and goal states $S_g(x)$.

Output: A controller $C(x, u)$.

Let $SU_k(x)$ denote the set of states from which the goal is backward reachable in exactly k steps, $C_k(x, u)$ the set of controllers that move the system to the goal in k steps, and let $SA_k(x)$ contain all states backward, reachable from the goal states in k steps or less.

Initialization:

$$SA_0(x) = S_g(x) \tag{2.103}$$

$$SU_0(x) = S_g(x) \tag{2.104}$$

$$C_0(x, u) = \texttt{false} \tag{2.105}$$

Main loop: The key idea is to add the transitions going from the states $SU_k(x)$ to $SU_{k+1}(x)$ to the set of controllers $C_k(x, u)$, which gives $C_{k+1}(x, u)$:

$$\begin{aligned} SU_{k+1}(x) &= \exists x^+, u.\ \big(M(x, u, x^+) \wedge SU_k(x^+) \wedge \neg SA_k(x)\big) \\ C_{k+1}(x, u) &= C_k(x, u)\ \vee\ \exists x^+.\ \big(SU_{k+1}(x) \wedge M(x, u, x^+) \wedge SU_k(x^+)\big) \\ SA_{k+1}(x) &= SA_k(x)\ \vee\ SU_{k+1}(x) \end{aligned}$$

Exit condition:

$$SA_{k+1}(x) = SA_k(x) \tag{2.106}$$

□

Example 2.18 Goal State Problem

Consider the system from Example 2.17. Let $S_g(x) = (x{=\!=}E)$ be the goal state. The iterations of Algorithm 2.2 will then be

$$SU_0(x) = (x{=\!=}E) \tag{2.107}$$

$$C_0(x, u) = \texttt{false} \tag{2.108}$$

$$SA_0(x) = (x{=\!=}E) \tag{2.109}$$

$$SU_1(x) = (x{=\!=}D) \tag{2.110}$$

$$C_1(x, u) = (x{=\!=}D) \wedge d \tag{2.111}$$

$$SA_1(x) = (x{=\!=}E)\ \vee\ (x{=\!=}D) \tag{2.112}$$

$$SU_2(x) = (x{=\!=}C) \tag{2.113}$$

$$C_2(x, u) = (x{=\!=}D) \wedge d\ \vee\ (x{=\!=}C) \wedge a \tag{2.114}$$

$$SA_2(x) = (x{=\!=}E)\ \vee\ (x{=\!=}D)\ \vee\ (x{=\!=}C) \tag{2.115}$$

$$SU_3(x) = \texttt{false} \tag{2.116}$$

$$C_3(x, u) = (x{=\!=}D) \wedge d\ \vee\ (x{=\!=}C) \wedge a \tag{2.117}$$

$$SA_3(x) = (x{=\!=}E)\ \vee\ (x{=\!=}D)\ \vee\ (x{=\!=}C) \tag{2.118}$$

We have $SA_2(x) = SA_3(x)$ which means we have reached the fixed point. (As we see from above we may also use $SU_k(x) = \texttt{false}$ as exit condition.) The controller is

$$C(x, u) = C_2(x, u) \tag{2.119}$$

As expected we get a controller including the transitions *a* and *d*.

Note that the controller may be non deterministic if there exist more than one trajectory from a state to the goal with equal minimal length. The number of iterations of Algorithm 2.2 is equal to the longest possible path not passing a state more than once. If the system has n state variables each taking q different values the maximum number of iterations is q^n.

The most common theoretic area for planning is the research area of artificial intelligence. Sequential Action Structures (SAS) [74] is an alternative notation for finite state machines that are used to model systems like manufacturing plants. A planning algorithm is developed which returns a plan of actions taking the system from an initial state to the final state in polynomial time, provided that the system fulfills some explicit conditions. The cutback of having an efficient planning algorithm is that these methods only work for a restricted class of systems.

2.5 Summing Up

This chapter has introduced the entities *Relations* and *Relation sets* that we use as mathematical objects in finite domains.

DEDS are modeled by *relational models* which are models interacting with signals instead of events. The modeled system is assumed to be autonomous, i.e., the transitions of the system are not controlled by its environment.

Relational models can be composed as three different products: *synchronous*, *asynchronous* and *interleaved*. Interconnection of models is performed by restricting a model composition by relations representing the connected signals. For feedback interconnection a delay is added to avoid algebraic loops. The delay imposed an extra state variable to the system.

Analysis performed on relational models is divided into *static* and *dynamic* analysis where the latter is more powerful but not that common in commercial tools today. Dynamic analysis means different types of fixed point iterations, such as computing the reachable states. The specification language *temporal logic* can be used to specify verification tasks. The verification of temporal logic is also a fixed point iteration, which returns the states for which the temporal logic expression holds.

Appendix

2A Generating Trajectories

This appendix contains preliminary results of how to find counter examples that exemplify the results from the dynamic verification.

In Section 2.3 we described the temporal logic language which could be used to specify verification tasks. The result from this analysis can either be the constants `true` or `false` or a relation $\mathsf{R}(z)$ representing a set for which the temporal logic formula holds. From an engineering point of view the relation $\mathsf{R}(z)$ is not a well formed answer in general. The relation $\mathsf{R}(z)$ represents a state space area from where there exists trajectories fulfilling the criteria in the temporal logic verification. If one of these trajectories can be generated as a result of the verification process we will have a more convenient tool for the user.

There are several ways to generate trajectories. There are tools available that produces counter examples in som way, e.g., SMV [26]. Here we will present how to incorporate the planning algorithm from Section 2.4.2 in this process.

Definition 2A.1 Trajectory
Let $\mathsf{M}(z, z^+)$ be a relational model, then a *trajectory of* $\mathsf{M}(z, z^+)$, denoted

$$\ll z(0), z(1), \ldots, z(n) \gg \tag{2A.1}$$

is a sequence of system variable instances corresponding to the dynamics of $\mathsf{M}(z, z^+)$, i.e.,

$$\mathsf{M}(z(k), z(k+1)) = \texttt{true} \quad 0 \leq k < n \tag{2A.2}$$

Remark:

- The length of a trajectory may be infinite even though the system is a finite DEDS. Infinite trajectories are denoted

$$\ll z(0), z(1), \ldots \gg \tag{2A.3}$$

- For a model $\mathsf{M}(z, z^+)$ with initial relation $\mathrm{I}(z)$ we have

$$\Gamma^+_\infty[\mathsf{M}, \mathrm{I}](z(k)) = \texttt{true} \quad \forall\, k \geq 0 \tag{2A.4}$$

  i.e., a trajectory must be inside the set of reachable states.

- We use the convention to denote the instance of initial state by $z(0)$ and that $\mathrm{I}(z(0)) = \texttt{true}$ must hold.

- Trajectories correspond to *strings of events* in the domain of finite automata theory.

$\square$

We will now try to make a formal definition of the problem of generating trajectories that illustrate the result from a temporal logic verification.

First we will define what we mean by the inner most atomic expression in a temporal logic expression.

Definition 2A.2 Inner Most Atomic Expression
Let $\mathsf{TL}(f)$ be a function returning `true` iff expression f contains temporal logic operators at any level. Then the *inner most atomic expression*, denoted $\mathsf{IMAE}(f)$, is defined recursively as

$$\mathsf{IMAE}(f) = f \quad \text{if } \mathsf{TL}(f) = \texttt{true} \tag{2A.5}$$
$$\mathsf{IMAE}(\mathsf{EX}[f]) = \mathsf{IMAE}(f) \tag{2A.6}$$
$$\mathsf{IMAE}(\mathsf{AX}[f]) = \mathsf{IMAE}(f) \tag{2A.7}$$
$$\mathsf{IMAE}(\mathsf{EG}[f]) = \mathsf{IMAE}(f) \tag{2A.8}$$
$$\mathsf{IMAE}(\mathsf{AG}[f]) = \mathsf{IMAE}(f) \tag{2A.9}$$
$$\mathsf{IMAE}(\mathsf{EU}[f, g]) = \mathsf{IMAE}(g) \tag{2A.10}$$
$$\mathsf{IMAE}(\mathsf{AU}[f, g]) = \mathsf{IMAE}(g) \tag{2A.11}$$

Remark: When f is composed of Boolean operators like $f = g_1 \vee g_2$ the IMAE-function must chose either g_1 or g_2. Different strategies can be chosen. One way is to choose the subexpression which has the greatest depth of nested temporal operators to stress that the result should be the inner most expression. Another suggestion is to let the user decide which branch to choose. □

Definition 2A.3 Illustrating Trajectory
From a temporal logic verification process

$$\mathsf{R}(z) = \mathsf{Verify}(M, f) \tag{2A.12}$$

where M is a model, $\mathsf{R}(z)$ is a resulting relation, Verify is the verification and f is a temporal logic expression. The *illustrating trajectory* is any trajectory

$$\ll z(0), \ldots, z(n) \gg \tag{2A.13}$$

starting from any point in the verification result ($\mathsf{R}(z(0)) = \texttt{true}$) and where $z(n)$ satisfies the inner most atomic expression in f, i.e.,

$$\mathsf{IMAE}(f)(z(n)) = \texttt{true}. \tag{2A.14}$$

□

Before we describe how to generate the illustrating trajectories for general temporal expressions we will study each temporal operator separately.

2A.1 Single Temporal Logic Operators

Next State Operators

The next state temporal logic operators $\mathsf{EX}[f]$ and $\mathsf{AX}[f]$ will have illustrating trajectories of length 2 as

$$\ll z(0), z(1) \gg \tag{2A.15}$$

where $z(0)$ and $z(1)$ are chosen as

$$\begin{aligned} z(0) &= \mathsf{Pick}(R(z), z) \\ z(1) &= \mathsf{Pick}((z{=\!=}z(0)) \wedge M(z, z^+) \wedge f(z^+), z^+) \end{aligned} \tag{2A.16}$$

We use (2A.15) for illustrating both $\mathsf{EX}[f]$ and $\mathsf{AX}[f]$.

Finite Future Operators

The finite future temporal logic operators $\mathsf{EF}[f]$, $\mathsf{AF}[f]$, $\mathsf{EU}[f, g]$ and $\mathsf{AU}[f, g]$ have an important property.

Lemma 2A.1 Invariance of Illustrating Trajectory
Let $R(z) = \mathsf{Verify}(M, f)$, where f is one of EF, AF, EU, AU, EG or AG. Assume that $\ll z(0), z(1), \ldots, z(n) \gg$ is an illustrating trajectory for the verification $\mathsf{Verify}(M, f)$, then $z(k) \in R(z)$, $0 \leq k \leq n$. □

Proof We will make the proof for EF and AF. The operators EU, AU, (and EG, AG)[1] are proved similarly.

EF: Consider the verification $R(z) = \mathsf{Verify}(M, \mathsf{EF}[f])$. Let $z(k) \notin R(z)$ for some $k < n$ from where there exists a trajectory $\ll z(k), \ldots, z(n) \gg$, where $z(n) \in f(z)$. This means that $z(k) \in \mathsf{EF}[f(z)] \Rightarrow z(k) \in R(z) \Rightarrow$ contradiction.

AF: Consider the verification $R(z) = \mathsf{Verify}(M, \mathsf{AF}[f])$. Let $z(k) \notin R(z)$ for some $k < n$ from where all trajectories $\ll z(k), \ldots, z(n) \gg$ will reach f, i.e., $z(n) \in f(z)$. This means that $z(k) \in \mathsf{AF}[f(z)] \Rightarrow z(k) \in R(z) \Rightarrow$ contradiction. ■

Lemma 2A.1 states that the verification result $R(z)$ must hold for every step along an illustrating trajectory.

The procedure of generating an illustrating trajectory starts with computing a goal state controller with the intersection of $R(z)$ and f as the goal state set. Note that $f(z) \subseteq R(z)$ will not hold in general.

The plan is computed based on the model restricted by $R(z)$ to fulfill Lemma 2A.1, using Algorithm 2.2.

$$C(z) = \mathsf{Planner}(\mathsf{Restrict}(M, R(z)), R(z) \wedge f(z)) \tag{2A.17}$$

Then we pick a trajectory with the iteration

$$\begin{aligned} z(0) &= \mathsf{Pick}(R(z), z) \\ z(k+1) &= \mathsf{Pick}((z{=\!=}z(k)) \wedge C(z) \wedge M(z, z^+), z^+) \end{aligned} \tag{2A.18}$$

until $f(z(k+1)) = \mathtt{true}$.

[1]The lemma holds for these temporal operators also.

Infinite Future Operators

Until now we have generated illustrating trajectories for temporal operators representing conditions which will be true in a finite number of iterations.

For the operations EG and AG we have a different situation. These operators represent conditions that hold forever. The problem is how to illustrate such a condition.

One suggestion is to find a trajectory within $\mathsf{R}(z)$ which forms a loop, i.e., $z(0) = z(n)$. A loop is perhaps the ultimate way of illustrating a property that will hold forever.

To improve the search for a loop we need a good starting point of the trajectory. This point will be taken from a relation representing all loops and intermediate paths between the loops.

Definition 2A.4 Upper Approximation of Loops
The *upper approximation of loops*, denoted $\mathsf{ULoops}[M](z)$, is a relation which contains all states belonging to a loop or an intermediate path between two loops of the system M. □

The following lemma shows how $\mathsf{ULoops}[M](z)$ is computed.

Lemma 2A.2 Upper Approximation of Loops
For a system $M(z, z^+)$, the *upper approximation of loops* ($\mathsf{ULoops}[M](z)$) is computed as

$$\mathsf{ULoops}[M](z) = \gamma^+_\infty[M, \mathtt{true}](z) \wedge \gamma^-_\infty[M, \mathtt{true}](z) \tag{2A.19}$$

where

$$\gamma^+_\infty[M, I](z) = \lim_{k \to \infty} \gamma^+_k[M, I](z) \tag{2A.20}$$

$$\gamma^-_\infty[M, I](z) = \lim_{k \to \infty} \gamma^-_k[M, I](z) \tag{2A.21}$$

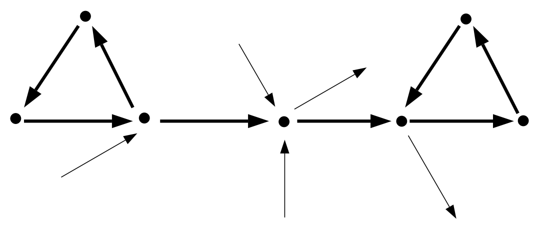
contains loops and intermediate paths between these loops and nothing more. □

Proof $\gamma^+_\infty[M, \mathtt{true}](z)$ contains all states that are reachable from all states by infinitely long paths. Since the state space is finite the only way to have infinite paths is by a loop. Therefore $\gamma^+_\infty[M, \mathtt{true}](z)$ will contain all loops and states forward reachable from these loops. All states not forward reachable from a loop are excluded.

Similar reasoning gives that $\gamma^-_\infty[M, I](z)$ contains all loops and states backward reachable from these loops.

The intersection of these sets will contain all loops and states that are both forward and backward reachable from or within a loop. ■

Example 2A.1 Upper Approximation of Loops
The following figure illustrates an example where we have two paths forming a loop and a path connecting these loops.



Only the bold transitions are included in $\mathsf{ULoops}[M](z)$.

The procedure for finding a loop starts by picking a starting point of the restricted model

$$z(0) = \mathsf{Pick}(\mathsf{ULoops}(M'(z,z^+)),z) \tag{2A.22}$$

where $M'(z,z^+) = \mathsf{Restrict}(M(z,z^+),R(z))$ and $R(z)$ is the verification result.

Then we find a loop forward from $z(0)$ by a variant of the reachable state computation performed on the restricted model:

$$M''(z,z^+) = \mathsf{Restrict}(M'(z,z^+),\mathsf{ULoops}(M'(z,z^+))). \tag{2A.23}$$

Let $k > 0$ be the minimal number of steps such that $P(z)$, computed as

$$P(z) = \Gamma^+_{k-1}[M'',(z{=\!=}z(0))](z) \wedge \gamma^+_k[M'',(z{=\!=}z(0))](z) \tag{2A.24}$$

is not `false`. Then let

$$z(k) = \mathsf{Pick}(P(z),z) \tag{2A.25}$$

and compute the two plans

$$C^1(z) = \mathsf{Planner}(M''(z,z^+),(z{=\!=}z(k))) \tag{2A.26}$$

which we use to generate a trajectory for the loop and

$$C^2(z) = \mathsf{Planner}(M'(z,z^+),(z{=\!=}z(k))) \tag{2A.27}$$

which we use to guide the system from any point in $R(z)$ to $z(k)$.

2A.2 Combined Temporal Logic Expressions

In the previous sections we suggested how to generate illustrating trajectories for atomic temporal expressions. These methods can be used iteratively when we have temporal logic expressions which consist of nested temporal expressions.

The basic idea to solve the problem for combined expressions is to let the verification process $\mathsf{Verify}(M,f)$ store the intermediate results from the verification of each temporal logic operator.

Example 2A.2 Intermediate Verification Results
Verification of the temporal expression

$$f(z) = \mathsf{EX}\underbrace{[\underbrace{\mathsf{EU}[g_1, \underbrace{\mathsf{EG}[g_2]}_{R_1(z)}]}_{R_2(z)}]}_{R_3(z)} \tag{2A.28}$$

would generate three intermediate results $R_1(z)$, $R_2(z)$ and $R_3(z)$ representing the state space of each temporal subexpression.

From the stored intermediate results we can generate an illustrating trajectory starting from the outermost result and continuing until it ends where the inner most atomic expression holds. Each part of the trajectory is computed by the procedure for the temporal operator representing that part.

Example 2A.3 Concatenation of Illustrating Trajectories
The illustrating trajectory for the temporal expression (2A.28) is generated in three steps. The first step starts in $R_3(z)$ and ends in $R_2(z)$. By using the procedure of (2A.16) for the operator EX we get the trajectory

$$\ll z(0), z(1) \gg \tag{2A.29}$$

where $z(0) \in R_3(z)$ and $z(1) \in R_2(z)$.

The next step continues from $z(1)$ to a point in $R_1(z)$. By using the procedure of (2A.18) for the operator EU we get the trajectory

$$\ll z(1), \ldots, z(k) \gg \tag{2A.30}$$

where $z(k) \in R_3(z)$.

The last step continues from $z(k)$ to a point in $g_2(z)$ which is the inner most atomic expression. Operator for this step is EG which means that the last step of the illustrating trajectory will end with a loop. We use the procedure (2A.22)-(2A.27) to generate such a loop

$$\ll z(k), \ldots, z(l), \ldots, z(l) \gg \tag{2A.31}$$

The final illustrating trajectory will be the concatenation.

$$\ll z(0), z(1), \ldots, z(k), \ldots, z(l), \ldots, z(l) \gg \tag{2A.32}$$

The methods presented in this appendix are suggested in order to present verification results in a more user friendly way. The complexity of these methods has not been discussed here. The performance depends on the representations we choose for the relations. Moreover, the methods presented above can, most likely, be optimized and integrated. For example, if we instead of letting the verification process return a state space relation, we could produce one illustrating trajectory directly. Further research in this area is needed.

Part I

Relational Representations

3

Relational Representations - an Introduction

Until now we have used relational models like $\mathsf{M}(z, z^+)$ to represent behaviors of systems. In these we have only been interested in the relations between the system variables z and z^+. We have also used logic operations such as $\wedge$, $\vee$ and $\neg$ on these relations to build a system theory for DEDS. Having a relational description of a system we know, from Chapter 2, the operations which must be implemented to get the tools of the relational system theory to work. But we have not yet presented how to implement these relations and operations. We will deal with this problem by first studying how to represent relations, and then find out how to implement the operations we need.

Already at this point, we will give some examples that show the main ideas of the different types of representations for relations and functions that we will present in this part of the thesis.

Functions and relations are in this part of the thesis almost the same. The representations that will be presented are representations of functions which are a mapping between two finite domains. Relations are here regarded as a special function mapping some finite domain to the Boolean domain, $\mathbb{B}$.

Since we only consider finite DEDS, and therefore only need to represent relations and functions in finite domains, the most intuitive and simple way to write down a function is to use a table.

Example 3.1 A Finite Function
Let a function $f(x,u)$, where $f : \mathbb{Z}_5^2 \to \mathbb{Z}_5$, be represented (and defined) by the table

$f(x,u)$		u				
		0	1	2	3	4
x	0	1	3	1	0	0
	1	2	1	1	1	0
	2	3	4	1	2	0
	3	4	2	1	3	0
	4	0	0	1	4	0

For each value of x and u we can look up the value of $f(x,u)$ directly from the table.

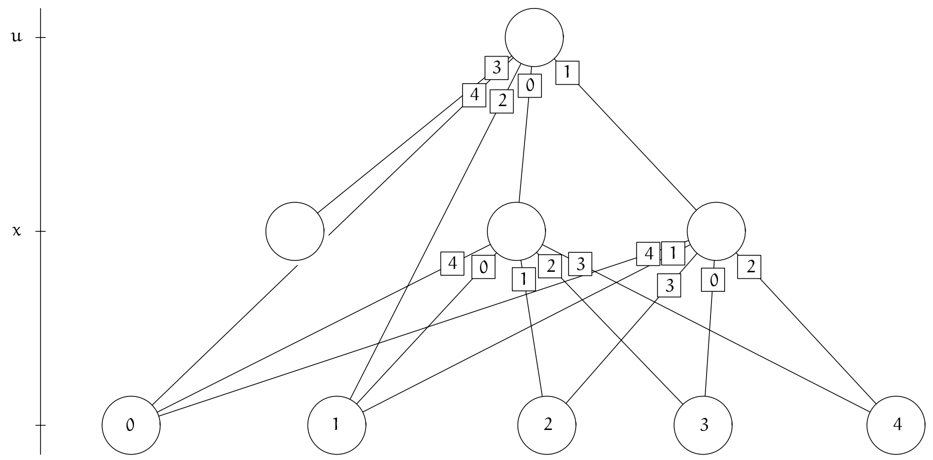
Tables like the one above can represent, e.g., a 5-state deterministic finite automaton with the transition relation $x^+ = f(x,u)$.

The table representation is rather straight-forward to implement in a computer but since relations and functions often get complex, for cases where a DEDS framework will be interesting to use, we must find something more intelligent. The table representation will not be elaborated any more in terms of representing relations, even though one can think of relational databases as tools especially designed for large tables of data. Nevertheless, tables are good to have as a less abstract mind map of relations.

3.1 Decision Diagrams

Instead of using a flat table we can use a tree representation where the variables in a relation is ordered and each vertex corresponds to one variable. Each vertex branches to vertices of variables with lower order until the terminals representing constant values at the bottom of the tree are reached. What we get is a DAG (Directed Acyclic Graph) with one root vertex.

Example 3.2 Integer Decision Diagram
The function $f(x,u)$ in Example 3.1 can be represented by the tree



The tree has three horizontal levels of vertices where the top level has one vertex (root) corresponding to the variable u. The next level has three vertices corresponding to the variable x and at the bottom level we have 5 vertices corresponding to constant values. We denote the variable order in this tree as $u > x$. Each vertex (except the constant vertices) has 5 branches since in this case $x, u \in \mathbb{Z}_5$. To look up the value for $f(1, 0)$ we start at the top vertex going downward following the branch labeled (square) 0 (for $u = 0$), then from the middle x-vertex we follow the branch labeled 1 (for $x = 1$) to the constant vertex 2, i.e., $f(1, 0) = 2$.

The left-most x-vertex corresponds to the function x, and is therefore not connected to the constants for clarity in the figure. For this vertex we get $f(x, 3) = x$.

The branch passing "behind" the left-most x-vertex connects the top vertex to the constant vertex for 0. This means that $f(x, 4) = 0$.

The tree representation shown in the example above is what we call the *integer decision diagram* (IDD) for the function $f(x, u)$ with variable order $u > x$. We have developed IDDs and implemented a software package in C with interface to the computer algebra system *Mathematica*. In Chapter 4 we define and develop algorithms for the IDD structure.

Compared to the table in Example 3.1 we have a more compact representation of $f(x, u)$ even though the difference is not dramatic in this case. The size of the table is $5^2 = 25$, whereas the IDD-tree in Example 3.2 above can be represented by the number of branches $5 \times 4 = 20$. The difference in size is more obvious in the next example, where we use an IDD to represent a relation rather than a function.

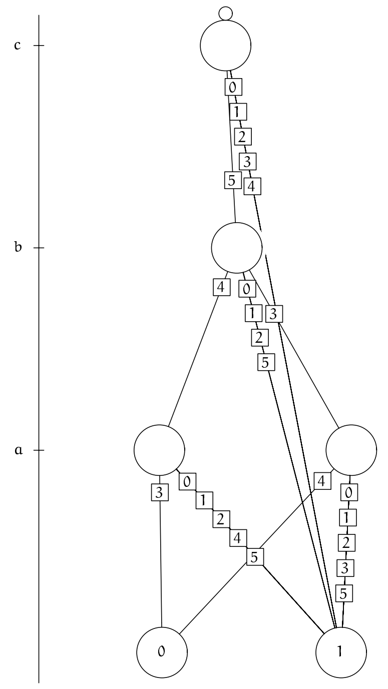
Example 3.3 An IDD Representation of a Relation
Let us represent the equation

$$a^2 + b^2 == c^2 \tag{3.1}$$

where $a, b, c \in \mathbb{Z}_6$ and $a, b, c > 0$. The relation can be written as

$$(a^2 + b^2 == c^2) \wedge (a > 0) \wedge (b > 0) \wedge (c > 0) \tag{3.2}$$

and translated to the IDD with variable order $c > b > a$ as



Since the IDD above represents a relation we will only have two constant vertices 0 and 1 representing `false` and `true`, respectively. Values of a, b, c fulfilling the relation in (3.2) will have a path from the top vertex down to the `true`-vertex. In this case, though, we have a *complement marker*[1] above the top vertex (the ring) which says that the solutions to the relation in (3.2) are those values of the variables having a path to the 0-vertex.

In this case we have two solutions $\{[a, b, c] : [3, 4, 5], [4, 3, 5]\}$ which in this case are better known as the Egyptian triangle relation.

From the example above we see that a relation for which a complete table representation would have been of size $6^3 = 216$ can be represented by 4 vertices with total size of $6 \times 4 = 24$. The key to success is that IDD-vertices are shared by many branches. In fact, the IDDs shown so far are *reduced*, i.e., they cannot be smaller by altering some branches but still represent the same function or relation with the same variable ordering. The size of the IDDs depends heavily on which variable order we chose. Moreover, the IDD is a canonical representation of a function or a relation, which is a feature that we will find very useful. As operations on IDDs we can use both logical operators for relations and arithmetic operators for integer functions.

For the special case of Boolean relations where all variables are Boolean we can use *Binary Decision Diagrams* (BDD) [15]. BDDs are the “original” decision

[1] Complement markers will be further explained in Chapter 4.

diagrams from which the IDD is developed. We will therefore in Chapter 4 start by presenting BDDs and then in Chapter 6 extend to IDDs.

3.2 Polynomials and Algebra

The representation of finite functions and relations can also be done using mathematical objects from commutative algebra, i.e., polynomials and polynomial operations. This is in contrast with the approaches of using decision diagrams which can be thought of as compact representations of the values[2] of function and relations. Instead, the algebraic approach tries to find efficient representations for polynomial formulas.

The idea of representing functions by polynomials, is best introduced by an example.

Example 3.4 Polynomial Representation
Again we use the function from Example 3.1 which was defined by the table

$f(x,u)$		u				
		0	1	2	3	4
x	0	1	3	1	0	0
	1	2	1	1	1	0
	2	3	4	1	2	0
	3	4	2	1	3	0
	4	0	0	1	4	0

This function is represented algebraically by the polynomial

$$f(x,u) = 1 + 4u + 3u^2 + x + 3u^2x + 4u^3x. \tag{3.3}$$

To be convinced that the polynomial in (3.3) really is a representation of the table above, evaluate the polynomial modulo 5 for all values of x and u.

As opposed to the decision diagrams we see that the polynomial representation does not store the values of the function, but instead a polynomial from which the values of the function can be computed. The polynomial representation is suited for some operations and relations, whereas the decision diagram representation is better for other. For instance an addition of two functions f_1+f_2 is easily performed in the polynomial case; just add the terms together. However, to find solutions to the equation $f = 0$ is much simpler if we represent f by a decision diagram. This will be further discussed in Section 6.3.

Two important issues of the polynomial representation are worth stressing:

[2]Values of the corresponding tables.

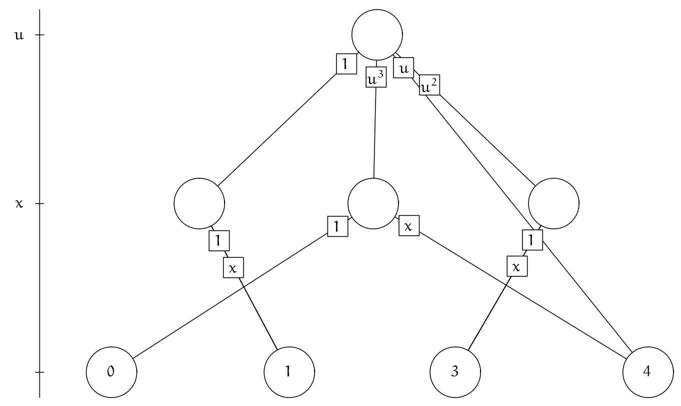
- First, the size of a polynomial representation varies depending on the represented function from small up to as many terms as there are slots in the corresponding table. As with decision diagrams the key to success is that the represented function has some structure that can easily be captured by polynomials, and that common subexpressions can be reused as often as possible in correspondence with the reuse of vertices in decision diagrams.
- Secondly, to have a canonical form for the polynomial representation is as important as for the decision diagrams.

In Chapter 5 we give an introduction to commutative algebra and the polynomial representation of relations and functions.

3.3 Polynomial Graph Representations

To meet the demand of a canonical form and the ability to reuse subexpressions we can use a polynomial form which in turn can be represented by a directed acyclic graph. The following example will introduce this idea.

Example 3.5 Reduced Recursive Polynomial Expression
The reduced recursive form of the polynomial in (3.3) is represented by the DAG



Each vertex in the tree above represents a polynomial in its corresponding variable. This means that the top vertex represents the polynomial

$$c_0 \cdot 1 + c_1 u + c_2 u^2 + c_3 u^3 \tag{3.4}$$

where the coefficients $\{c_0, \dots, c_3\}$ are polynomials (in x) corresponding to each subgraph the top vertex labeled with the corresponding power of u. Analogously, we get the polynomials for the x-vertices (listed from left to right):

$$1 \cdot 1 + 1 \cdot x, \quad 0 \cdot 1 + 4 \cdot x, \quad 3 \cdot 1 + 3 \cdot x \tag{3.5}$$

The coefficients for top vertex polynomial are therefore:

$$c_0 = 1 + x, \quad c_1 = 4, \quad c_2 = 3 + 3x, \quad c_3 = 4x \tag{3.6}$$

The complete polynomial for the tree above is:

$$(1 + x)1 + (4)u + (3 + 3x)u^2 + (4x)u^3 = \tag{3.7}$$
$$1 + 4u + 3u^2 + x + 3u^2x + 4u^3x \tag{3.8}$$

which apparently is the same polynomial as $f(x, u)$ from Equation (3.3) above.

The example above illustrates the *reduced recursive polynomial expression* (RRPE), introduced by Germundsson [40], where coefficients of polynomials are recursively represented as DAGs. For a given variable ordering (in this case $x < u$) the RRPE is a canonical form of the function $f(x, u)$.

3.4 Gröbner Bases

Another way of representing a relation using polynomials, is to use polynomial equations. By computing a *Gröbner basis*, [30], we can generate a set of polynomials that represents a relation in a canonical way.

Example 3.6 Gröbner Bases
Let us reconsider Example 3.3, where we had the equation

$$a^2 + b^2 = c^2 \tag{3.9}$$

We want to canonically represent, using equations of polynomials, the solutions of this equation for nonzero values of the variables. For reasons that will become clear later, the variables have to be defined over a *finite field*, in this case $\mathbb{F}_7$, which essentially means that the variables take the values $\{0, \ldots, 6\}$ and that all arithmetic operations on the variables are modulo 7.

The Equation (3.9) and the nonzero criteria for the variables, can be translated into a set of polynomial equations as

$$\begin{aligned} c^2 - b^2 - a^2 &= 0 \\ c^6 - 1 &= 0 \\ b^6 - 1 &= 0 \\ a^6 - 1 &= 0 \end{aligned} \tag{3.10}$$

The left-hand side of these equations can be rewritten into a canonical set of polynomials, using the Gröbner basis GB_7 with variable ordering $c > b > a$.

$$GB_7\left(\left\{\begin{array}{r} c^2 - b^2 - a^2 \\ c^6 - 1 \\ b^6 - 1 \\ a^6 - 1 \end{array}\right\}\right) = \left\{\begin{array}{c} c^2 - 2c^2 \\ b^2 - a^2 \\ a^6 - 1 \end{array}\right\} \tag{3.11}$$

The resulting set of polynomials is a canonical form for the solution set, i.e., Gröbner bases have one-to-one correspondence with solution sets.

Finding a solution to a set of equations can often be simpler when having the Gröbner basis as the one above. The last equation gives the solutions for the variable a which can be substituted into the other equations in the set. Compare with Gaussian elimination.

From the Gröbner basis we have the solutions $\{[a, b, c] : [3, 4, 5], [4, 3, 5]\}$ as expected but $[2, 5, 1]$ is also a solution, which might come as a surprise to us. In fact there are 24 solutions to this set of equations. The reason is that the polynomials in (3.10) belong to the *polynomial ring* $R_7[a, b, c]$, where all operations are modulo 7. Therefore we have in Equation (3.9) more solutions compared to what we had in Example 3.3. In Chapter 5 we give insight in this matter more thoroughly.

3.5 Road Map for Part I

As an introduction to decision diagrams we will describe function graphs and especially the *binary decision diagrams* (BDDs) in Chapter 4. This chapter serves as an introduction to decision diagrams, operations and algorithms that make decision diagram structures good for developing efficient tools for relational representations. Using Chapter 4 we will, in Chapter 6, present *integer decision diagrams* (IDDs) which are an extension to BDDs which we have developed and which is one of the contributions of this thesis.

To get familiar with the polynomial approach and the Gröbner basis tool, we need to go through some theory of commutative algebra in Chapter 5. This will help us formally present the *integrated monomial diagrams* (IMD) in Chapter 6, which we have developed as a reduced canonical graph representation of polynomials.

The main contributions of this part of the thesis is collected in Chapter 6.

4 Decision Diagrams

Binary Decision Diagrams (BDD) were introduced by Akers [2]. Then Bryant [15] introduced operations and algorithms that utilize the ordering of the variables in BDDs. Bryant showed that an “interesting” subset of Boolean functions could be represented by function graphs in size polynomial to the number of variables. After that many different diagrams similar to BDDs have emerged. See [17] for an overview of this research area.

The material in this chapter is based on the article by Bryant [15] in which Boolean functions are represented by Boolean function graphs, i.e., directed acyclic graphs where each vertex has only two children (or subgraphs). Bryant proved that by reducing the function graphs we get a canonical form which, in most interesting cases[1], represents Boolean functions efficiently. Efficient algorithms for Boolean operators were developed by Brace et al. [11]. This concept of representation and computation of Boolean expressions, was called Binary Decision Diagram (BDD). The term *ordered binary decision diagram* (OBDD) is also frequently used even though BDD and OBDD stand for the same thing nowadays.

There has been a successful use of BDDs in applications [16] which in turn has led to a massive research activity on how to improve the concept of decision diagrams to a wider range of applications. The first impression of this research is that it has led to an “alphabet soup”[2] of acronyms [17]. The following is only a partial list: OBDD, FBDD, FDD, OKFDD, EVBDD, MTBDD, BMD, *BMD, ZBDD, ABDD, HDD, TDD, and OPDD. The most important objective for these different types of decision diagrams is to derive constructions that will reduce the complexity of verifying hardware implementing arithmetic (integer) functions.

[1]Not all of the interesting cases, though.

[2]To which we will contribute with the acronyms IDD and IMD.

To extend the Boolean representation to general finite functions has been an important task. Srinivasan et al. [107] have developed the *multi-valued decision diagram* (MDD) for representing integer functions and relations. These MDDs serve as a a layer above the BDDs. Therefore the MDDs provide a mapping to BDDs, i.e., the BDDs are still used as the underlying representation of these relations.

In this chapter we will review the fundamental results of reduced function graphs which is the structure from which the BDDs and all other decision diagrams are built. We will call this class of function graph implementations for decision diagrams.

In Chapter 6 we present our results of how to represent finite functions by using *integer decision diagrams* (IDDs). We will also use the structure of the function graph to represent polynomials, in Chapter 6, suggesting a representation which we call *integrated monomial diagrams* (IMD). In contrast with the variants of decision diagrams developed so far, the IDDs and IMDs do not depend on any Boolean structures to represent functions and relations of finite but non Boolean domains.

4.1 Function Graphs

We will now formally define what type of graph we use for a decision diagram. This type of graph will be denoted *function graph.*

Definition 4.1 Function Graph

i) A *function graph* is a rooted, directed graph with vertex set V containing two types of vertices, *nonterminal* and *terminal.* A *nonterminal* vertex v has as attributes an argument index $\mathsf{index}(v) \in \{1, \dots, n\}$ and children $\mathsf{child}(v, i) \in V$, $0 \leq i < \mathsf{range}(v)$, where $n, \mathsf{range}(v) \in \mathbb{Z}_+$. A *terminal* vertex v has as attribute a value $\mathsf{value}(v) \in 0, \dots, M-1$, where $M \in \mathbb{Z}_+$.

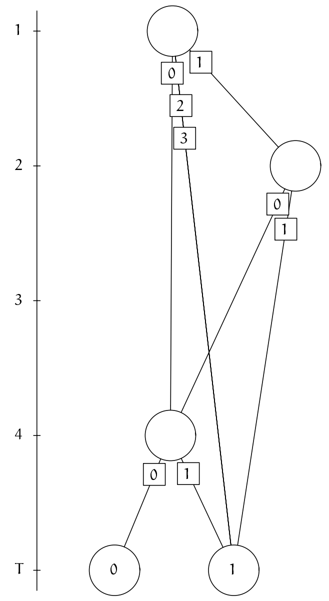
ii) For any nonterminal vertex v, if $\mathsf{child}(v, i)$ is also nonterminal, then $\mathsf{index}(v) < \mathsf{index}(\mathsf{child}(v, i))$.

iii) The range $\mathsf{range}(v)$ of a vertex v is equal for all vertices having the same index.

Remark: Note that the restriction on indexes makes the function graphs acyclic since any path from the root vertex to one of the terminal vertices must have strictly increasing index values. □

Most of the graphs depicted in this chapter are function graphs. Nevertheless, we will present a simple example to clarify how these depicted graphs are related to Definition 4.1.

Example 4.1 Function Graph
We give a simple function graph.



The vertical axis to the left shows the indexes for the vertices. We denote a vertex by $v_{i,j}$ where i is the index and j is the order in the picture starting from left with $j = 1$. The edge labels (squares) in the picture identifies each child of a vertex, such that the center of the label is positioned exactly on the edge. If there are more than one label on the same line[3], then there are several edges (upon each other) connecting the same pair of vertices. The terminals are placed at the bottom of the graph, and indicated by a T on the vertical axis. The values of the terminals are written inside the vertices.

We have the following for the function graph above:

$$\begin{aligned}
\mathsf{index}(v_{1,1}) &= 1 \quad \text{the root}\\
\mathsf{child}(v_{1,1}, 0) &= v_{4,1}\\
\mathsf{child}(v_{1,1}, 1) &= v_{2,1}\\
\mathsf{child}(v_{1,1}, 2) &= v_{T,2}\\
\mathsf{child}(v_{1,1}, 3) &= v_{T,2}\\
\mathsf{child}(v_{2,1}, 0) &= v_{4,1}\\
\mathsf{child}(v_{2,1}, 1) &= v_{T,2}\\
\mathsf{child}(v_{4,1}, 0) &= v_{T,1}\\
\mathsf{child}(v_{4,1}, 1) &= v_{T,2}\\
\mathsf{value}(v_{T,1}) &= 0\\
\mathsf{value}(v_{T,2}) &= 1\\
\mathsf{range}(v_{1,1}) &= 4\\
\mathsf{range}(v_{2,1}) &= 2\\
\mathsf{range}(v_{4,1}) &= 2
\end{aligned} \tag{4.1}$$

[3]Line illustrating an edge in graph figures.

T is not an index but is used as one for notational reasons. Note that there are no vertices with index 3 for this function graph. The indexes often correspond to one variable of the function which is represented by the function graph.

Having defined the structure of function graphs we will define the correspondence between function graphs and functions.

Definition 4.2 Function Graph to Function Connection
A function graph G having a root vertex v denotes a function f_v defined recursively as

1. If v is a terminal vertex, then $f_v = \mathsf{value}(v)$.
2. If v is a nonterminal vertex with $\mathsf{index}(v) = i$, then f_v is the function

$$f_v(x_i, x_{i+1}, \dots, x_n) = f_{\mathsf{child}(v,x_i)}(x_{i+1}, \dots, x_n) \tag{4.2}$$

where the function variable $x_i \in \{0, 1, \dots, \mathsf{range}(v) - 1\}$ and $1 \leq i \leq n$.

Remark: Here we assume that the maximum index of vertices in G is n. □

The definition above defines f_v as an evaluation following one path from the root vertex down to a terminal vertex. In this general form we have no symbolic representation of the final f_v for the complete graph G. Symbolic representations will be formulated later on in Section 4.2, where expressions from Boolean algebra are used as a symbolic representation, and then in Chapter 5 where polynomials over finite fields are used in the same way.

To be able to form function graphs with canonical properties we have to formulate some definitions before we can state the theorem of canonical function graphs.

Definition 4.3 Isomorphic Function Graphs
The function graphs G and G' are *isomorphic* if there exists a one-to-one function σ from vertices of G onto the vertices of G' such that for any vertex v if $\sigma(v) = v'$, then either both v and v' are terminal vertices with

$$\mathsf{value}(v) = \mathsf{value}(v') \tag{4.3}$$

or both v and v' are nonterminal vertices with

$$\mathsf{index}(v) = \mathsf{index}(v') \tag{4.4}$$

and

$$\sigma(\mathsf{child}(v, i)) = \mathsf{child}(v', i) \quad 0 \leq i < \mathsf{range}(v). \tag{4.5}$$

Remark: Note that $\mathsf{range}(v) = \mathsf{range}(v')$. □

The mapping σ is in fact quite constrained since a root vertex must be mapped to another root, and the order of the children must be preserved. The only freedom of the mapping is that a graph G can be either a tree where all vertices have only one parent, or vertices can be connected more than once to several parents if that is possible according to the rules in Definition 4.1.

Definition 4.4 Subgraph
For any vertex ν in a function graph G, the *subgraph rooted by* ν is defined as the graph consisting of ν and all of its descendants. □

The following lemma is rather immediate.

Lemma 4.1 Isomorphic Subgraphs
If G is isomorphic to G' by the mapping σ, then for any vertex ν in G, the subgraph rooted by ν is isomorphic to the subgraph rooted by $\sigma(\nu)$. □

The following definition gives us the *reduced function graph*, which is what we will use for the decision diagrams as well as for the graph representations for the polynomials.

Definition 4.5 Reduced Function Graph
A function graph G is *reduced* if it contains no vertex ν with

$$\mathsf{child}(\nu, 0) = \mathsf{child}(\nu, 1) = \cdots = \mathsf{child}(\nu, \mathsf{range}(\nu) - 1), \tag{4.6}$$

nor does it contain distinct vertices ν and ν' such that the subgraphs rooted by ν and ν' are isomorphic. □

Lemma 4.2 Reduced Subgraphs
For every vertex ν in a reduced function graph, the subgraph rooted by ν is itself a reduced function graph. □

We are now ready for the main message of this section. The following theorem shows that a reduced function graph is a canonical form for the corresponding function.

Theorem 4.1 Reduced Function Graph is a Canonical Form
For any function f over a finite domain, there exists a unique (up to isomorphism) reduced function graph denoting f, and any other function graph denoting f contains more vertices. □

Proof The theorem is proved by Bryant [15] for the Boolean case only, but the outline of that proof can easily be used in this case as well, which has been done by Srinivasan et al. [107]. ■

4.2 Binary Decision Diagrams

For the case of Boolean functions and variables we use a special form of the reduced function graph called binary decision diagram (BDD) [15]. In this section, we will take a closer look at the BDD structure and algorithms for operations, as a preparation for the IDDs in Chapter 6. The material presented in this section can be found in [15] and in the implemented code of the BDD-CMU package [11].

The basic idea used in binary decision diagrams is to rewrite Boolean expressions in a recursive form and reuse common subexpressions. 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. [34]):

$$\begin{aligned} f(x_1, \ldots, x_n) = &((\neg x_1) \wedge f(0, x_2, \ldots, x_n)) \\ &\vee (x_1 \wedge f(1, x_2, \ldots, x_n)). \end{aligned} \tag{4.7}$$

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

$$\begin{aligned} f(x_1, \ldots, x_n) = &\underbrace{\neg x_1 \wedge (\cdots ((\underbrace{\neg x_n \wedge \alpha)}_{g^{n-1}_{0,\ldots,0}(x_n)} \vee \underbrace{(x_n \wedge \beta)}_{g^{n-1}_{0,\ldots,1}(x_n)}))}_{g^1_0(x_2,\ldots,x_n)} \vee \\ &\underbrace{x_1 \wedge (\cdots ((\underbrace{\neg x_n \wedge \gamma)}_{g^{n-1}_{1,\ldots,0}(x_n)} \vee \underbrace{(x_n \wedge \delta)}_{g^{n-1}_{1,\ldots,1}(x_n)}))}_{g^1_1(x_2,\ldots,x_n)} \end{aligned} \tag{4.8}$$

where $\alpha, \beta, \gamma, \delta \in \{0, 1\}$. Compare with Definition 4.2.

We see that we obtain several subexpressions with progressively fewer variables. In fact, all expressions g^i_j 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.

This is achieved by constructing a reduced function graph G for the function f, where each nonterminal vertex v with $\mathsf{index}(v) = i$ corresponds to the variable x_i and the functions corresponding to the subgraphs of v are each equal to one of the subexpressions $g^i_j(x_{i+1}, \ldots, x_n)$.

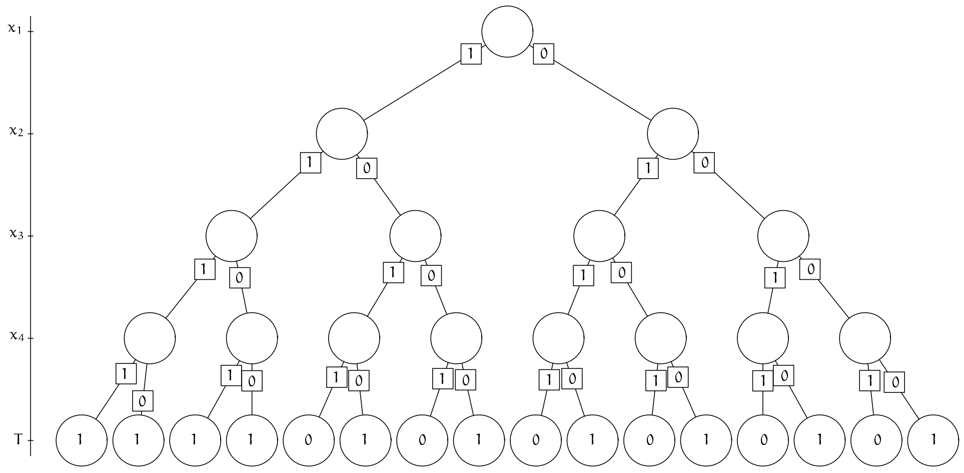
The recursive Boolean expression from above can be visualized as a binary tree, where each vertex corresponds to the $\vee$ operator, and where the g^i_j expressions in principle are subtrees. This is the basis for the name BDD. From Theorem 4.1 we know that a BDD is a canonical representation of a Boolean function for a given variable ordering.

According to Definition 4.1 we have for BDDs that $\mathsf{range}(v) = 2$ for all vertices and there are only two terminals, i.e., $M = 2$.

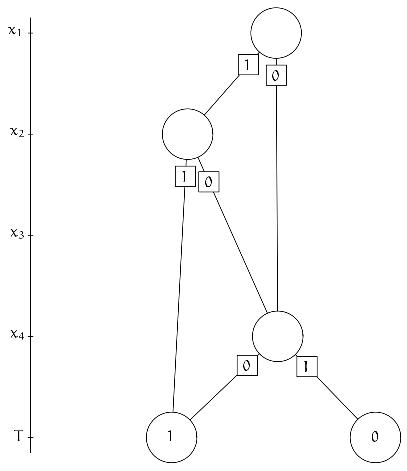
The number of remaining vertices (or equivalently the number of *different* subexpressions) after reduction, is a measure of the complexity of the given expression. This will be referred to as the number of vertices further on in this thesis. By changing the order in which we expand w.r.t. the variables we usually get large differences in the number of vertices. The ordering is called *variable ordering* and plays a significant role in lowering the representational complexity of the landing gear system in Section 7.3.

Example 4.2 Unreduced vs. Reduced BDD

The Boolean function $f = x_1 \wedge x_2 \vee \neg x_4$ is represented by the following ordered but unreduced binary tree if we use the variable ordering $x_1 < x_2 < x_3 < x_4$.



After reduction we get the BDD



The fact that we have Boolean expressions in this case makes it possible to reformulate Definition 4.2 to BDDs which is essentially the reverted Shannon expansion.

Definition 4.6 BDD to Function Connection
A BDD G having root vertex ν denotes a function f_ν defined recursively as

1. If ν is a terminal vertex, then $f_\nu = \mathsf{value}(\nu) \in \{0, 1\}$.

2. If ν is a nonterminal vertex with $\mathsf{index}(\nu) = i$, then f_ν is the function

$$f_\nu(x_i, x_{i+1}, \ldots, x_n) = \neg x_i \wedge f_{\mathsf{low}(\nu)}(x_{i+1}, \ldots, x_n) \vee x_i \wedge f_{\mathsf{high}(\nu)}(x_{i+1}, \ldots, x_n) \tag{4.9}$$

where $\mathsf{low}(\nu) = \mathsf{child}(\nu, 0)$ and $\mathsf{high}(\nu) = \mathsf{child}(\nu, 1)$.

Remark: Here we assume that the maximum index of vertices in G is n. □

4.2.1 Operations and Algorithms for BDDs

Having defined the structure for representing Boolean functions and relations we now need to specify how to implement operations on this structure.

The ITE-operator

The most common Boolean operators $\wedge$, $\vee$ and $\neg$ are enough to realize all binary Boolean functions. In fact, the NAND-operator $(\neg(f \wedge g))$ is also enough if we really want to reduce the set of operators to a minimum. In the case of BDDs we will go the opposite way by choosing an operator with more expressive power and which mimics the BDD structure closely.

If we associate a BDD vertex ν with the tuple $(\mathsf{v}, \mathsf{F}, \mathsf{G})$ where v is the corresponding variable and F and G are functions corresponding to the two subgraphs of the vertex ν, then the function Z corresponding to the vertex ν can be written as

$$\mathsf{Z} = \mathsf{v} \wedge \mathsf{F} \vee \neg \mathsf{v} \wedge \mathsf{G} \tag{4.10}$$

using Shannon's expansion (or Definition 4.6). This special form can be generalized to the three input function *ite*, If-Then-Else:

$$\textit{ite}(\mathsf{F}, \mathsf{G}, \mathsf{H}) = \mathsf{F} \wedge \mathsf{G} \vee \neg \mathsf{F} \wedge \mathsf{H} \tag{4.11}$$

which is the function we choose as the foundation from which all binary Boolean operators can be formulated.

Example 4.3 The ITE-operator
All (16 in this case, [11]) binary Boolean operators can be parameterized using the *ite*-formula, [11].

Four of these are:

$$\begin{aligned} \texttt{AND}(F, G) &= ite(F, G, 0) \\ \texttt{OR}(F, G) &= ite(F, 1, G) \\ \texttt{NAND}(F, G) &= ite(F, \neg G, 1) \\ \texttt{XOR}(F, G) &= ite(F, \neg G, G) \end{aligned}$$

By implementing the *ite*-operator we compute all Boolean operators in one iteration. Otherwise, only having the $\wedge$, $\vee$, $\neg$ operators we would need 5 iterations to compute an XOR-operation. The reason for this is that the *ite*-operator effectively exploits the function graph structure of the BDD.

Recursive Formulation of ITE

The BDD $Z = ite(F, G, H)$ where F, G, and H are BDDs can be computed recursively using a depth-first recursion.

Lemma 4.3 ITE Recursion
Assume that the top variable of F, G, H is x, i.e., x has the lowest index of all variables in F, G, H, then

$$ite(F, G, H) = ite(x, ite(F_x, G_x, H_x), ite(F_{\neg x}, G_{\neg x}, H_{\neg x})) \tag{4.12}$$

where F_x and $F_{\neg x}$ denotes

$$F_x = F_{\mathsf{high}(x)} \tag{4.13}$$
$$F_{\neg x} = F_{\mathsf{low}(x)} \tag{4.14}$$

and similarly for G and H. □

Proof ([11])

$$\begin{aligned} Z &= ite(x, Z_x, Z_{\neg x}) \\ &= x \wedge Z_x \vee \neg x \wedge Z_{\neg x} \\ &= x \wedge (F \wedge G \vee \neg F \wedge H)_x \vee \neg x \wedge (F \wedge G \vee \neg F \wedge H)_{\neg x} \\ &= x \wedge (F_x \wedge G_x \vee \neg F_x \wedge H_x) \vee \neg x \wedge (F_{\neg x} \wedge G_{\neg x} \vee \neg F_{\neg x} \wedge H_{\neg x}) \\ &= x \wedge ite(F_x, G_x, H_x) \vee \neg x \wedge ite(F_{\neg x}, G_{\neg x}, H_{\neg x}) \\ &= ite(x, ite(F_x, G_x, H_x), ite(F_{\neg x}, G_{\neg x}, H_{\neg x})) \end{aligned}$$

■

If one or more of F, G, and H are independent of x we will have, e.g., $G_x = G_{\neg x} = G$.

This recursion stops when all three arguments to the *ite*-operator are terminals. In fact, by using some simplification rules the recursion will normally stop much earlier. See the section describing the simplification rules below.

Unique Table

All vertices in a BDD are stored in the *unique table.* This table stores all BDD vertices and maintains an index that uniquely maps the triple $(\mathsf{F}, \mathsf{G}, \mathsf{H})$ to a corresponding BDD vertex if there exist one. If not, a new vertex is created and put into the table. In this way the table ensures that there is no duplicate vertices corresponding to the same pair of subtrees. By using hashing we get a constant look up time in the unique table. The hash function is denoted $\mathsf{key}(\mathsf{F}, \mathsf{G}, \mathsf{H})$ and is also used for cache table described later in this section.

The fact that the recursion of the *ite*-operator performs a depth-first computation means that the resulting BDD is created from the bottom and upwards while the unique table ensures that no duplicate vertices are created. Instead vertices are re-referenced if possible which guarantees that the resulting BDD will be reduced and canonical with respect to the variable order given.

Before we present the *ite* algorithm we will present some ways of increasing the performance of the algorithm.

Simplification

Simplification of the expression $ite(\mathsf{F}, \mathsf{G}, \mathsf{H})$ is used for those cases when we can immediately compute the result to a terminal or to F,G, or H. This will improve the performance in the recursion (4.12).

The simplification is done in two steps. First the rules

$$\begin{aligned} ite(\mathsf{F}, \mathsf{F}, \mathsf{H}) &\longrightarrow ite(\mathsf{F}, 1, \mathsf{H}) \\ ite(\mathsf{F}, \neg\mathsf{F}, \mathsf{H}) &\longrightarrow ite(\mathsf{F}, 0, \mathsf{H}) \\ ite(\mathsf{F}, \mathsf{G}, \mathsf{F}) &\longrightarrow ite(\mathsf{F}, \mathsf{G}, 0) \\ ite(\mathsf{F}, \mathsf{G}, \neg\mathsf{F}) &\longrightarrow ite(\mathsf{F}, \mathsf{G}, 1) \end{aligned} \tag{4.15}$$

are applied, and then the following rules are applied for the *terminal cases*:

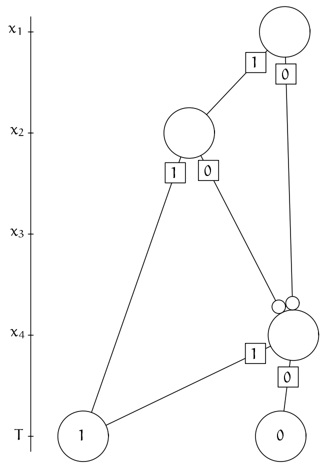
$$\begin{aligned} ite(1, \mathsf{G}, \mathsf{H}) &\longrightarrow \mathsf{G} \\ ite(0, \mathsf{G}, \mathsf{H}) &\longrightarrow \mathsf{H} \\ ite(\mathsf{F}, \mathsf{G}, \mathsf{G}) &\longrightarrow \mathsf{G} \\ ite(\mathsf{F}, 1, 0) &\longrightarrow \mathsf{F} \\ ite(\mathsf{F}, 0, 1) &\longrightarrow \neg\mathsf{F} \end{aligned} \tag{4.16}$$

Complement Edges

To reduce the size of the BDD we can use the same BDD for representing both f and its complement $\neg\mathsf{f}$. This method has been developed by Akers [2], Karplus [73] and Madre [86], and implemented by Brace et al. [11].

Having complement edges we have a mark for each edge in the BDD specifying if the corresponding function of the edge should be complemented or not.

Example 4.4 Complement Edges
For the Boolean function $f = x_1x_2 + \neg x_4$ from Example 4.2 we get the following BDD



where the circles, attached to the vertex corresponding to the function x_4, illustrates the complement edges.

To preserve the canonical form we take a look at the following four equalities:

$$\begin{aligned} ite(F, G, H) &= ite(\neg F, \neg G, \neg H) \\ ite(\neg F, G, H) &= ite(F, \neg G, \neg H) \\ ite(F, G, \neg H) &= ite(\neg F, \neg G, H) \\ ite(\neg F, G, \neg H) &= ite(F, \neg G, H) \end{aligned} \tag{4.17}$$

By choosing the expressions at the left-hand side of the equalities above as the only allowed assignments of the complement edges in BDDs, we preserve the canonical form. The left-hand side of the expressions have the "high" subgraph (G) uncomplemented. The BDD in Example 4.4 follows this rule.

Note that this means that the computation of operations and tests like $\neg F$ and $F = \neg G$ can be performed in constant time[4]. Good performance of these operations is important for the efficiency of the transformation rules presented earlier in this section.

Caching

Using a cache table we can increase the performance of the recursive computation in Equation (4.12) even further. The cache table maps[5] *ite*-expressions represented

[4]In the case of the CMU BDD package [11] the $\neg$-operation will toggle the least significant bit of a pointer, and the $=$-test will compare the equality of two pointers.

[5]By using the hash function $\mathsf{key}(F, G, H)$ that maps F, G, H to a scalar.

by the tuple $(\mathsf{F},\mathsf{G},\mathsf{H})$ to the result Z, provided that the *ite*-expressions has been computed before. By this the recursion will not need to evaluate subgraphs already computed, which means that the algorithm only needs to visit every combination of vertices once in a computation.

To further improve the hit rate of the look up operations in the cache table we can perform some transformations of the expression $ite(\mathsf{F},\mathsf{G},\mathsf{H})$ to reduce the number of different combinations of F, G, and H having the same result. The basic idea is to transform the tuple $(\mathsf{F},\mathsf{G},\mathsf{H})$ to $(\mathsf{F}',\mathsf{G}',\mathsf{H}')$ such that $\mathsf{index}(\mathsf{F}') \leq \mathsf{index}(\mathsf{G}') \leq \mathsf{index}(\mathsf{H}')$ holds whenever possible.

If $\mathsf{index}(\mathsf{F}) > \mathsf{index}(\mathsf{G})$ and $\mathsf{index}(\mathsf{F}) > \mathsf{index}(\mathsf{H})$, then we use the following rules for changing the order of the arguments for the *ite*-operator.

$$\begin{aligned}
ite(\mathsf{F},\mathsf{0},\mathsf{H}) &\longrightarrow ite(\neg\mathsf{H},\mathsf{0},\neg\mathsf{F})\\
ite(\mathsf{F},\mathsf{1},\mathsf{H}) &\longrightarrow ite(\mathsf{H},\mathsf{1},\mathsf{F})\\
ite(\mathsf{F},\mathsf{G},\mathsf{1}) &\longrightarrow ite(\neg\mathsf{G},\neg\mathsf{F},\mathsf{1})\\
ite(\mathsf{F},\mathsf{G},\mathsf{0}) &\longrightarrow ite(\mathsf{G},\mathsf{F},\mathsf{0})\\
ite(\mathsf{F},\mathsf{G},\neg\mathsf{G}) &\longrightarrow ite(\mathsf{G},\mathsf{F},\neg\mathsf{F})
\end{aligned} \tag{4.18}$$

The final transformation before looking up in the cache table is to move the complement edges according to the left-hand side in (4.17).

The ITE-Algorithm

We are now ready to formulate the ITE-algorithm.

Algorithm 4.1 Recursive ITE-Algorithm

Syntax: $ite\_apply(\mathsf{F},\mathsf{G},\mathsf{H})$.

Input: BDD references F, G, H.

Output: A BDD reference to the resulting BDD, Z.

1. Simplify and check for terminal cases according to (4.15) and (4.16).

2. Sort the arguments F, G, H with top variable first according to (4.18).

3. Transform the complement edges of F, G, H to standard form according to (4.17). If F is complemented set the flag $\mathtt{c} = \mathtt{true}$ and remove the complement edge from F.

4. Look up in cache table for a precomputed result Z. If found then if $\mathtt{c} = \mathtt{true}$ return $\neg\mathsf{Z}$, else return Z.

5. Let t be the top index: $\mathsf{t} = \mathsf{min}(\mathsf{index}(F), \mathsf{index}(G), \mathsf{index}(H))$

$$\{l_F, h_F\} := \begin{cases} \{\mathsf{low}(F), \mathsf{high}(F)\} & \text{if } \mathsf{index}(F) = t \\ \{F, F\} & \text{otherwise} \end{cases}$$

$$\{l_G, h_G\} := \begin{cases} \{\mathsf{low}(G), \mathsf{high}(G)\} & \text{if } \mathsf{index}(G) = t \\ \{G, G\} & \text{otherwise} \end{cases}$$

$$\{l_H, h_H\} := \begin{cases} \{\mathsf{low}(H), \mathsf{high}(H)\} & \text{if } \mathsf{index}(H) = t \\ \{H, H\} & \text{otherwise} \end{cases}$$

6. Start the recursion

$$Z_{\mathsf{low}} := \mathit{ite\_apply}(l_F, l_G, l_H)$$
$$Z_{\mathsf{high}} := \mathit{ite\_apply}(h_F, h_G, h_H)$$

7. Look up an entry in the *unique table* that corresponds to a vertex with index t and subgraphs Z_{low} and Z_{high}. If found, set Z to this vertex. If not found a new vertex Z is created and stored in the unique table.

8. Store an entry in the cache table for vertex Z and the tuple (F, G, H) as its key.

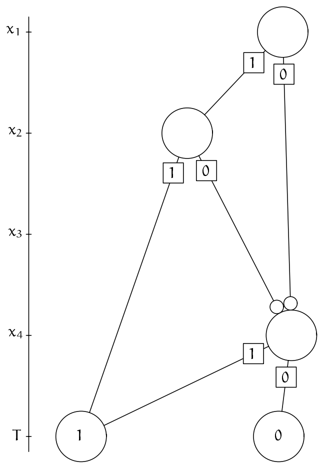
9. If $c = \texttt{true}$ return $\neg Z$ else return Z.

□

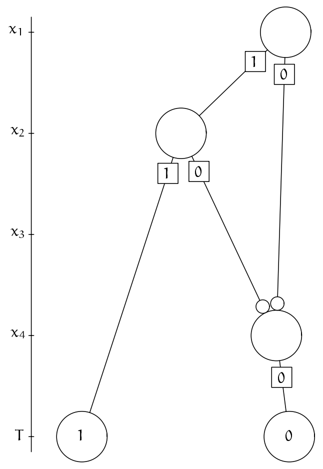
Restriction

To restrict a function means in this case to set one of the arguments (or variables) to a specific value. If we have the BDD F with top variable x we can easily compute the BDD corresponding to $F|_{x=0}$ or $F|_{x=1}$ by taking the subgraphs $\mathsf{low}(F)$ or $\mathsf{high}(F)$, respectively. Unfortunately it is not that easy if the variable is not the top variable. In this case we have to build the resulting BDD to preserve the canonical form.

Example 4.5 Restriction of BDD
Let us again consider the BDD from Example 4.4.



If we let $x_4 =$ `false`, then we can exclude the edge going from the vertex corresponding to x_4 to the terminal 1. The resulting graph is then:



The graph above seems to have both 0 and 1 as terminals but since the vertex corresponding to x_4 is connected upwards by complement edges we have, in fact, that all paths from the top vertex down to a terminal will have the value 1. This means that the graph above represents nothing but the constant 1 for which the canonical form is the terminal vertex 1.

The basic idea of computing $F|_{x=b}$ is to use depth-first recursion on the BDD until we reach a vertex v, where at least one of the children, say $\mathsf{child}(v, i)$, corresponds to the variable x. This edge is then changed in such a way that it refers directly to the "grand child" of v corresponding to $x = b$, i.e.,

$$v_i := \mathsf{child}(\mathsf{child}(v, i), b) \tag{4.19}$$

After this is done for all children of v, we look up in the unique table to ensure the canonical form. The result is stored in a temporary cache table with key equal to v. Later when the recursion reaches vertex v again, we look up in the hash table to prevents the result of this vertex to be recomputed.

The restrict algorithm can be implemented as follows.

Algorithm 4.2 Restrict

Syntax: $\textit{restrict}(F, x, b)$.

Inputs: F is a BDD, x is a variable of F, b is a constant 0 or 1.

Output: Returns a BDD Z corresponding to $F|_{x=b}$.

1. Check for terminal cases
   (a) F is constant $\Rightarrow$ return F
   (b) $\mathsf{index}(F) > \mathsf{index}(x) \Rightarrow$ return F
2. If top vertex of F is visited, denoted as $\textit{mark}(F) = \texttt{true}$, then look up the precomputed result Z from the hash table and return it.
3. If $\mathsf{index}(F) < \mathsf{index}(x)$ then
   (a) Make the recursive calls
$$l_F := \textit{restrict}(\mathsf{low}(F), x, b)$$
$$h_F := \textit{restrict}(\mathsf{high}(F), x, b)$$
   (b) Find or create a vertex Z in the unique table having index $\mathsf{index}(F)$ and l_F and h_F as subgraphs.
   (c) Mark the top vertex of F as visited ($\textit{mark}(F) := \texttt{true}$) and store Z in the temporary hash table with the top vertex as a key.
   (d) Return Z.
4. If $\mathsf{index}(F) = \mathsf{index}(x)$ then
   (a) Set $Z = \mathsf{child}(F, b)$.
   (b) Mark the top vertex of F as visited ($\textit{mark}(F) := \texttt{true}$) and store Z in the temporary hash table with the top vertex as a key.
   (c) Return Z.

□

Compose

The compose operation substitutes a variable x in a function f_1 with a function f_2, i.e., we get $f_1|_{x=f_2}$.

This operation is implemented with inspiration from the equation

$$f_1|_{x=f_2} = f_2 \wedge f_1|_{x=1} \vee (\neg f_2) \wedge f_1|_{x=0} \tag{4.20}$$

which can be reformulated as

$$f_1|_{x=f_2} = \mathit{ite}(f_2, f_1|_{x=1}, f_1|_{x=0}) \tag{4.21}$$

This shows that to compute $f_1|_{x=f_2}$ we first compute two restrict operations of f_1 and then we do an *ite*-operation. Since the implementation of the compose-operation turns out to be trivial the presentation of the algorithm will be omitted.

Quantifiers

The quantifiers $\exists$ and $\forall$ are essential in the verification process where variables in relations are eliminated to project a solution set to fewer variables. See, e.g., Section 2.3.

Both types of quantifiers use the same implementation, which is a generalization of the restrict-operation previously described. The operation *quantify*(F, X, e) will return the BDDs corresponding to

$$\begin{array}{ll} \exists X.\, F & \text{if } e = 0 \\ \forall X.\, F & \text{if } e = 1 \end{array}$$

where F is the BDD and X is a list of variables that will be quantified from F. Here we will assume that X is included in the variable set of F.

The basic idea behind the algorithm is to sort the list X with least index first and the highest index last, and then make a depth-first recursion where the subgraphs of vertices, corresponding to a variable in X, are composed using the $\vee$- or $\wedge$-operators for $\exists$ and $\forall$, respectively.

Algorithm 4.3 Quantification

Syntax: *quantify*(F, X, e).

Inputs: F is a BDD and X is a sorted list of variables. e is a Boolean flag indicating whether $\exists$ or $\forall$ is to be performed.

Output: Return a BDD Z where the variables in X have been excluded.

1. If F is a constant, then return F.
2. If F has been visited before, then return the BDD Z stored in the temporary hash table.
3. Drop variables in X with index less than $\mathsf{index}(F)$. By this we have that the first variable in X will have the index closest to $\mathsf{index}(F)$.

4. Make two recursive calls

$$\begin{aligned} l_F &= \textit{quantify\_step}(\mathsf{low}(F), X, e) \\ h_F &= \textit{quantify\_step}(\mathsf{high}(F), X, e) \end{aligned}$$

5. If $\mathsf{index}(F)$ is strictly less than the first variable in X then find or create a vertex Z in the unique table having index $\mathsf{index}(F)$ and l_F and h_F as subgraphs.

6. If $\mathsf{index}(F)$ is equal to the index of the first variable in X then call *ite_apply* to compute

$$\begin{aligned} Z &= l_F \lor h_F \quad \text{if } e = 0 \\ Z &= l_F \land h_F \quad \text{if } e = 1 \end{aligned}$$

7. Mark the top vertex of F as visited and store Z in the temporary hash table.

8. Return Z.

The marking of visited vertices is made analogously as in Algorithm 4.2. □

Solutions

The structure of the BDD turns out to be very suitable for the problem of finding solutions, i.e., having a Boolean function $f(x_1, \ldots, x_n)$ we want to find values $\alpha_1, \ldots, \alpha_n$ such that $f(\alpha_1, \ldots, \alpha_n) = 1$. The reason for this is presented by the following lemma.

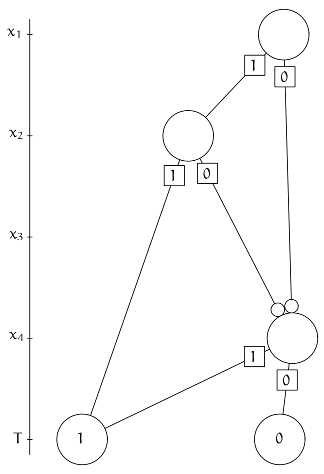
Lemma 4.4 Existence of Solutions in BDD Subgraphs
Every nonterminal vertex in a reduced function graph has a terminal vertex with value 1 as a descendant. □

This can be easily proved by considering a vertex in a reduced function graph with all its descendants equal to 0, which must be the zero terminal itself.

Therefore we can find a solution to a function represented by a BDD, by finding a path from the root vertex to the terminal 1. Starting from above we can follow any edge from the root that is not connected to the zero terminal and repeat this procedure for the next vertex and so on until we reach the terminal 1. For cases where the solution path does not pass vertices for some variables in the BDD we may chose solution values for these variables freely. Having n variables in the function we will find a solution in at most n number of steps.

This is illustrated by an example.

Example 4.6 Finding Solutions in a BDD
Let us again consider the BDD from Example 4.4 which represents the Boolean function $f = x_1 \land x_2 \lor \neg x_4$.



Starting from the top vertex corresponding to x_1 following (for example) the edge for $x_1 = 0$ we pass a complement marking and must therefore choose $x_4 = 0$ to reach the terminal vertex 1. This path corresponds to the solution

$$[x_1, x_4] = [0, 0] \tag{4.22}$$

but since the set of variables also, by definition, includes x_2 and x_3 the “real” solution set for this path is

$$[x_1, x_2, x_3, x_4] = \{[0,0,0,0], [0,0,1,0], [0,1,0,0], [0,1,1,0]\} \tag{4.23}$$

The number of steps to find a solution is in this case 2. The longest path takes 3 steps which is equal to the number of variables included in this BDD.

The example above illustrates one possible implementation of the Pick-function in Definition 2.8.

To compute the set of all solutions we make depth-first recursion where each completed path from the root down to the terminal one is printed out as a result.

We can also compute the number of solutions by traversing the BDD using depth-first recursion where each vertex ν is given the number of solutions α_ν for the subgraph of ν according to the formulas

$$\alpha_\nu = \mathsf{value}(\nu) \quad \text{if } \nu \text{ is terminal}$$
$$\alpha_\nu = \alpha_{\mathsf{low}(\nu)} \cdot 2^{\mathsf{index}(\mathsf{low}(\nu)) - \mathsf{index}(\nu) - 1} + \alpha_{\mathsf{high}(\nu)} \cdot 2^{\mathsf{index}(\mathsf{high}(\nu)) - \mathsf{index}(\nu) - 1}$$

The number of solutions of a BDD F is then $\alpha_{\mathsf{top}(F)}$.

Complexity

The worst case complexity of computing $ite(F, G, H)$ is limited by the number of combinations of vertices, i.e., $\mathcal{O}(|F||G||H|)$ where $|\cdot|$ denotes the number of vertices. This is true provided that every vertex combination is evaluated only once, by the construction of the cache table and that look-ups in the unique table and cache table take constant time. In practice [11], the typical performance is closer to the size of the resulting BDD. Moreover, the most commonly used[6] Boolean operations like AND, OR and NOT are computed by an *ite*-operator where at least one of the arguments is a terminal which gives better performance.

The complexity of the restrict operation is $\mathcal{O}(|\mathsf{F}| \log |\mathsf{F}|)$ according to [15] for a BDD F. In that case the restrict operation is implemented in two steps. The first step is a recursive function that alters the edges connected to vertices corresponding to the restricted variable, which gives an unreduced function graph. This step has the complexity $\mathcal{O}(|\mathsf{F}|)$. The second step reduces the function graph by a nonrecursive procedure which uses sorting which has the complexity $\mathcal{O}(|\mathsf{F}| \log |\mathsf{F}|)$.

The recursive Algorithm 4.2 only visits the vertices for the level of the restricted variable and above, and only once for each vertex. The worst case complexity for this algorithm, assuming a constant look up time in tables, is therefore $\mathcal{O}(|\mathsf{F}|)$.

The compose operation performs two restrict operations and then an *ite*-operation resulting in the worst case complexity of $\mathcal{O}(|\mathsf{F}_1|^2|\mathsf{F}_2|)$.

For the quantification operator we will only present a rough upper limit of the worst case complexity. If we quantify all variables in the BDD F, then we have to compute an *ite*-operation once in each one of the $|\mathsf{F}|$ vertices. The *ite*-operation only used for $\wedge$ and $\vee$ operations has complexity $\mathcal{O}(|\mathsf{F}|^2)$, which for the complete quantifier operation gives $\mathcal{O}(|\mathsf{F}|^3)$. Better estimates of performance can probably be found, though.

As a last comment on the complexity issues, it might be worth mentioning that all BDD operations presented in this section have at least polynomial complexity in terms of number of vertices in the BDD. Therefore the size of the BDD trees is a more critical parameter to keep as low as possible if BDDs should be usable in practice. By experience the author and others [16, 89] have found many cases where BDDs is a tractable structure for representing Boolean functions. The choice of a good variable ordering is a very important task since the BDD size is very much dependent on variable orderings.

However, there are functions where the number of vertices for the corresponding BDD will grow exponentially regardless of variable order, see [15]. One such function is multiplication of integers represented by words of Boolean bits.

[6]At least in the perspective of this thesis.

5 Polynomial Representation

In the previous chapter we presented decision diagrams to represent relations and functions over finite domains. The only algebra we have come across so far is the Boolean algebra with Boolean variables and operators like $\wedge$, $\vee$ and $\neg$.

In this chapter we will present a polynomial framework that gives us an algebra for general relations and functions over finite domains, provided that some criteria are fulfilled.

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 [30].

After the introduction we will present the special set of commutative algebra result for finite domains. These results are developed by Germundsson [40]. Similar results can also be found in Le Borgne et al. [80].

A computational tool for manipulating sets of polynomial equations, *Gröbner bases*, is presented as well as an interpretation of how to map logic operations into the polynomial framework.

See also [40] and [35, 36, 37, 38, 39] for more details on the polynomial representation.

5.1 Basic Commutative Algebra

In this section we 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 5.1 Abelian Group

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

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

□

Example 5.1 Abelian Group

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

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

Definition 5.2 Commutative Ring
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 $a, b \in \mathsf{R}$, the product ab is in R.

(iii) *Multiplicative identity.* There is an identity element $e \in \mathsf{R}$ that satisfies

$$ae = ea = a$$

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

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

□

Example 5.2 Commutative Ring
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 5.3 Field
A *field* k is a commutative ring where every element except 0 has a multiplicative inverse, i.e.,

$$\forall a \in \mathsf{R}\backslash\{0\},\ \exists \tilde{a} \in \mathsf{R} \text{ such that } a * \tilde{a} = e. \tag{5.1}$$

□

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., [88].

Theorem 5.1 Finite Field
If $q = p^m$ where p is a 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 5.4 Polynomial Ring
A *polynomial ring* (denoted $k[Z]$ or $k[z_1, \dots, z_n]$) is the set of all polynomials in the variables $z_1, z_2, \dots, z_n \in Z$ 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 5.5 Ideal
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 $hf \in I$.

□

Since ideals are a special subset of polynomial rings we will from now on use the notation

$$I \trianglelefteq k[z_1, \dots, z_n] \tag{5.2}$$

for an ideal I belonging to a polynomial ring $k[z_1, \dots, z_n]$.

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

Lemma 5.1 Generating Polynomials ([30])
Let $f_1, \dots, f_s$ be polynomials in $k[z_1, \dots, z_n]$ and set

$$\langle f_1, \dots, f_s \rangle = \left\{ \sum_{i=1}^{s} h_i f_i \mid h_1, \dots, h_s \in k[z_1, \dots, z_n] \right\}. \tag{5.3}$$

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

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, $V(I)$, denotes the set of common zeros for all polynomials in I, i.e.,

$$V(I) = \{z \in k^n : f(z) = 0 \text{ for all } f \in I\}. \tag{5.4}$$

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

Ideals correspond to varieties as relations correspond to relation sets, according to the informal definitions in Chapter 2.

Example 5.3
Let $I = \langle z_1^2 - 1,\ 5z_2,\ (z_1 - 1)(z_1 + 2)\rangle \trianglelefteq \mathbb{F}_7[z_1, z_2]$. The variety $V(I)$ is the set

$$V(I) = \{(z_1, z_2) : (1, 0)\}. \tag{5.5}$$

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 [39]. 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.

5.2 The Quotient Polynomial Ring, $\mathbb{R}_q[Z]$

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

Definition 5.6 Congruent Modulus ([30])
Let $I \trianglelefteq k[z_1, \ldots, z_n]$ be an ideal, and let $f, g \in k[z_1, \ldots, z_n]$. We say that f and g are *congruent modulo I*, written

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

if $f - g \in I$. □

It can be proved (see [30]) that if the polynomials $f, g \in k[Z]$ are such that $f \equiv g \mod I$ for the ideal $I \trianglelefteq k[Z]$, then f and g define the same function on the elements of the variety $V(I)$ of the ideal I.

Example 5.4 Congruent Modulus
Let $I = \langle z^3 - z\rangle \trianglelefteq \mathbb{F}_3[z]$, $f(z) = z^4$ and $g(z) = z^2$, then

$$f(z) \equiv g(z) \bmod I. \tag{5.6}$$

The variety of the ideal is

$$V(I) = \{z : 0, 1, 2\}. \tag{5.7}$$

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

$$f(2) = 2^4 \bmod 3 = 16 \bmod 3 = 1. \tag{5.8}$$

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 5.7 Quotient ([30])
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

$$k[z_1, \ldots, z_n]/I = \{h \mid h = [f]_I,\ f, h \in k[z_1, \ldots, z_n]\} \tag{5.9}$$

where the *equivalence class* $[f]_I$ is

$$[f]_I = \{g \in k[z_1, \ldots, z_n] \mid g \equiv f \bmod I\}. \tag{5.10}$$

□

From this we proceed to define the main object of the polynomial approach to our relational framework for DEDS, the quotient polynomial ring [40].

Definition 5.8 Quotient Polynomial Ring ([40])
The *quotient polynomial ring* $\mathbb{R}_q[Z]$ is defined as

$$\mathbb{R}_q[Z] = \mathbb{F}_q[Z]/\langle z_1^q - z_1, \ldots, z_n^q - z_n\rangle. \tag{5.11}$$

□

We note that $\mathbb{R}_q[Z]$ contains partitions of all polynomials in $\mathbb{F}_q[Z]$. The polynomials are partitioned into equivalence classes by the ideal

$$\langle z_1^q - z_1, \ldots, z_n^q - z_n\rangle. \tag{5.12}$$

It is shown in [40] 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 $\mathbb{R}_q[Z]$, i.e., there is a one-to-one correspondence between the relations and the equivalence classes in $\mathbb{R}_q[Z]$. Each equivalence class of polynomials in $\mathbb{R}_q[Z]$ can be represented by one of its members in all operations on relations. For every equivalence class there is a unique polynomial with degree less than q. Therefore the polynomial representing an equivalence class in $\mathbb{R}_q[Z]$ can be chosen such that the degree and the "length" of the polynomial has an upper limit. This gives some complexity advantages.

Polynomials in $\mathbb{R}_q[Z]$ will be one of the "languages" we use to represent relational models of DEDS.

5.3 Representing Functions with Polynomials

To illustrate how to represent a function with polynomials, the theorem of *functional completeness* from [39] will be presented.

If a polynomial ring is functionally complete, then there exists a corresponding polynomial in that ring for all functions over the domain.

Theorem 5.2 Functional Completeness
The polynomial ring $\mathbb{F}_q[z_1, \dots, z_n]$ is *functionally complete.*

□

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

$$f_p(Z) = \sum_{\xi \in \mathbb{F}_q^n} L_\xi(Z) f(\xi) \tag{5.13}$$

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

$$L_{\xi_i}(z_i) = \frac{\prod_{\nu \in \mathbb{F}_q \setminus \{\xi_i\}} (z_i - \nu)}{\prod_{\nu \in \mathbb{F}_q \setminus \{\xi_i\}} (\xi_i - \nu)} = \begin{cases} 1 & , z_i = \xi_i \\ 0 & , z_i \neq \xi_i \end{cases} \tag{5.14}$$

is the *Lagrange interpolating polynomial*, see, e.g., [108]. We then have $f(\xi) = f_p(\xi)$ for all $\xi \in \mathbb{F}_q^n$. ■

Example 5.5 Polynomial Representation of Functions over Finite Domains
Let the function $J(x)$ be defined by the table ($q = 7$):

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

In this case we can write (5.13) as

$$J_p(x) = \sum_{\xi \in \mathbb{F}_7} L_\xi(x) J(\xi) \tag{5.15}$$

where $J(\xi)$ is given by the table above and $L_\xi(x)$ is given by (5.14). The polynomial

$$L_0(x) = \frac{(x-1)(x-2)(x-3)(x-4)(x-5)(x-6)}{(0-1)(0-2)(0-3)(0-4)(0-5)(0-6)} = 1 + 6x^6 \tag{5.16}$$

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

$$J_p(x) = 3 + 3x + 6x^2 + x^3 + 2x^4 + 6x^5 + 2x^6 \tag{5.17}$$

where $J_p(x) \in \mathbb{R}_7[x]$. 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.

5.4 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 $\mathbb{R}_q[Z]$. 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

$$\begin{aligned} b(Z) = 0 &\Rightarrow \texttt{true} \\ b(Z) \neq 0 &\Rightarrow \texttt{false} \end{aligned} \tag{5.18}$$

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

$$a(Z) \wedge b(Z) = 1 - (1 - a(Z)^{q-1})(1 - b(Z)^{q-1}) \tag{5.19}$$

$$a(Z) \vee b(Z) = a(Z)b(Z) \tag{5.20}$$

$$\neg a(Z) = 1 - a(Z)^{q-1} \tag{5.21}$$

where the NOT-operation maps zero on one. This gives a natural extension of the logical operations to ideals in $R_q[Z]$. See [37] 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. See Chapter 2.

We will now show the close connection between $\mathbb{F}_2[Z]$ and Boolean algebra, by an example.

Example 5.6 Boolean Algebra Connection to $\mathbb{F}_2[Z]$
The Boolean expression $x_1 \wedge x_2$ is the same as the polynomial

$$1 - (1 - x_1)(1 - x_2) = x_1 + x_2 - x_1 x_2 \in \mathbb{F}_2[x_1, x_2]. \tag{5.22}$$

For $x_1 \vee x_2$ we get the polynomial $x_1 x_2 \in \mathbb{F}_2[x_1, x_2]$. Remember that $0 = \texttt{true}$ here.

The following example demonstrates a feature of the $\neg$-operation.

Example 5.7 Double Negations
Given the polynomial $a(x)$, let $b(x)$ be defined as

$$b(x) = \begin{cases} 1 & a(x) \neq 0 \\ 0 & a(x) = 0 \end{cases} \tag{5.23}$$

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

$$b(x) = \neg\neg a(x) \tag{5.24}$$

since the expression for NOT in (5.21) maps all zeros to ones.

For simplicity and clarity it is sometimes convenient to express polynomials as logical expressions. We can, e.g., express that a polynomial $p \in \mathbb{R}_3[Z]$ by

$$p(Z) = z_1 \wedge z_2. \tag{5.25}$$

Using the logical operation in (5.19), we interpret this as

$$p(Z) = z_1^2 + z_2^2 - z_1^2 z_2^2. \tag{5.26}$$

5.5 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, i.e., they provide a canonical form representing ideals. The Gröbner bases are in a sense a nonlinear generalization of Gaussian elimination.

5.5.1 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[2] term ordering gives an order to all monomials in the polynomial ring, and a way to define the degree, $\mathsf{deg}(f(Z))$, of a polynomial $f(Z)$. For example by *lexicographic order* $y > x$ we mean a term ordering where, e.g.,

$$\mathsf{deg}(y^2x) > \mathsf{deg}(yx^5) > \mathsf{deg}(yx^4) > \mathsf{deg}(x^9). \tag{5.27}$$

The *leading term*, $\mathrm{lt}(f(Z))$, is the term in $f(Z)$ with highest degree. The degree of a polynomial $f(Z)$ is

$$\mathsf{deg}(f(Z)) = \mathsf{deg}(\mathrm{lt}(f(Z))). \tag{5.28}$$

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

$$r = p - \sum_{f \in F} \alpha_f f \tag{5.29}$$

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

$$p \xrightarrow[F]{} r. \tag{5.30}$$

[1]Only a brief introduction is given in this thesis. Further details about Gröbner bases for ideals in a general polynomial ring k[Z] can be found in [30], and for $\mathbb{R}_q[Z]$ in [40].

[2]A monomial is a term without coefficient.

If $p \xrightarrow[F]{} p$ 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

$$S(p_1, p_2) = h_1 p_1 - h_2 p_2 \tag{5.31}$$

where h_1 and h_2 are terms of lowest degree such that $\mathrm{lt}(h_1 p_1) = \mathrm{lt}(h_2 p_2)$.

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

$$S(g_i, g_j) \in I \quad \forall\, i \neq j \tag{5.32}$$

i.e., $S(g_i, g_j) \xrightarrow[G]{} 0$. 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 if and only if

$$p \xrightarrow[G]{} 0 \tag{5.33}$$

where G is a Gröbner basis of I.

To compute a Gröbner basis we can use Buchberger's algorithm (see [30]):

Algorithm 5.1 Buchberger's Algorithm

Syntax: $\mathsf{GB}(F, O)$.

Inputs: $F = \{f_1, f_2, \dots, f_n\}$ is a set of generators for the ideal $I = \langle F \rangle$, and O is a variable ordering.

Output: A Gröbner basis $G = \langle g_1, g_2, \dots, g_m \rangle$.

1. Take the generators of I as candidates for a Gröbner basis. Denote this set G.
2. If all s-polynomials $S(g_i, g_j) \xrightarrow[G]{} 0$, $i \neq j$ according to O, then G is a Gröbner basis.
3. Otherwise add to G the remainder r computed as
$$S(g_i, g_j) \xrightarrow[G]{} r \neq 0$$
4. Make G autoreduced and go to 2.

$\square$

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, then the ideals are equal.

When there is no need to explicitly specify the term ordering we use the notation $\mathsf{GB}(\{f_1, f_2, \dots, f_n\})$ for clarity.

5.5.2 Gröbner Bases in $\mathbb{R}_q[Z]$

Gröbner bases are used and defined in $\mathbb{R}_q[Z]$ in analogy with the general polynomial ring $k[Z]$, except that in every arithmetic operation on the polynomials in $\mathbb{R}_q[Z]$,

- degrees higher than or equal to q are reduced as $z^q \to z$.
- coefficients only take values in $\mathbb{F}_q$.

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

Example 5.8 Term Ordering in $\mathbb{R}_q[Z]$
Let $f = y^4$, $g = y^3 \in \mathbb{R}_7[y]$. Then $fg = y$ and therefore $\mathsf{deg}(fg) \neq 7$.

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 5.8) included in the set of generators for the ideal *I*.

Definition 5.9 Gröbner Basis in $\mathbb{R}_q[Z]$
The Gröbner basis of the ideal $I = \langle f_1, f_2, \ldots, f_n \rangle \trianglelefteq \mathbb{R}_q[Z]$ for some given term ordering is denoted

$$\mathsf{GB}_q(\{f_1, f_2, \ldots, f_n\}). \tag{5.34}$$

□

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[3].

5.6 Functional Dependence

When computing a Gröbner basis 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 5.9 Functional Dependence
Let the ideals $I_1, I_2 \trianglelefteq \mathbb{R}_3[y, x_1, x_2]$ have the varieties

$$\begin{aligned} V(I_1) &= \{(y, x_1, x_2) : (1,2,3), (2,3,1), (1,3,0)\} \\ V(I_2) &= \{(y, x_1, x_2) : (1,2,3), (2,3,1), (1,3,1)\} \end{aligned} \tag{5.35}$$

[3]All ideals in $\mathbb{R}_q[Z]$ can be generated by a single polynomial, i.e., $\mathbb{R}_q[Z]$ is a *principal* polynomial ring. See [39].

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).

In this section, we present our results that show that if a variable can be computed as a function, then there exists a polynomial in the ideal representing this function. Moreover, we can compute this polynomial using a Gröbner basis computation. These properties are both necessary and sufficient. Similar results have been presented by Le Borgne et al. in [79] in the case of $\mathbb{R}_3[Z]$, but without proof.

We will first make a formal definition of *functional dependence*.

Definition 5.10 Functional Dependence
Given an ideal $I \trianglelefteq \mathbb{R}_q[y, x_1, \ldots, x_n]$ and the variety $V(I)$, which is a set of elements of the form $(y, x_1, \ldots, x_n) \in \mathbb{F}_q^{n+1}$, we say that the variable y is *functionally dependent* w.r.t. I, if for each value of the variables $x_1, \ldots, x_n$ in $V(I)$ there exists only one value of y. □

Lemma 5.2 Polynomials Representing Functional Dependence
Given an ideal $I \trianglelefteq \mathbb{R}_q[y_1, \ldots, y_n, x_1, \ldots, x_m]$ there exist polynomials $p_1, \ldots, p_n \in \mathbb{R}_q[x_1, \ldots, x_m]$ such that

$$\{y_1 - p_1, \ldots, y_n - p_n\} \subseteq I \tag{5.36}$$

iff $y_1, \ldots, y_n$ are functionally dependent w.r.t. I. □

Proof The values of the functionally dependent variables $y_1, \ldots, y_n$ in $V(I)$ are given as a function $f : \mathbb{F}_q^m \to \mathbb{F}_q^n$, which can be represented by polynomials, using (5.13), as

$$y_1 = p_1, \; \ldots, y_n = p_n. \tag{5.37}$$

Since (5.37) gives no constraints on $x_1, \ldots, x_n$ we have

$$\begin{aligned} V(\langle y_1 - p_1, \ldots, y_n - p_n \rangle) \supseteq V(I) \Leftrightarrow \\ \langle y_1 - p_1, \ldots, y_n - p_n \rangle \subseteq I \Leftrightarrow \\ \{y_1 - p_1, \ldots, y_n - p_n\} \subseteq I. \end{aligned} \tag{5.38}$$

■

Theorem 5.3 Gröbner Basis and Functional Dependence
Let the ideal $I = \langle f_1, \ldots, f_l \rangle \trianglelefteq \mathbb{R}_q[y_1, \ldots, y_n, x_1, \ldots, x_m]$. The variables $y_1, \ldots, y_n$ are functionally dependent w.r.t I iff the autoreduced Gröbner basis G of I with lex-ordering $y > x$ has the form

$$G = \{y_1 - h_1, \ldots, y_n - h_n, v_1, v_2, \ldots\} \tag{5.39}$$

where the polynomials $h_i, v_j \in \mathbb{R}_q[x_1, \ldots, x_m]$. □

Proof If the Gröbner basis G has the form (5.39) the claim follows directly from Definition 5.10.

Conversely, if $y_1, \ldots, y_n$ are functionally dependent w.r.t. I, then by Lemma 5.2 we have

$$\{y_1 - p_1, \ldots, y_n - p_n\} \subseteq I \tag{5.40}$$

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

$$A = \{y_1 - \tilde{p}_1, \ldots, y_n - \tilde{p}_n, \tilde{f}_1, \ldots\} \tag{5.41}$$

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

$$S(a_i, a_j) \underset{A}{\longrightarrow} r \in \mathbb{R}_q[x_1, \ldots, x_m]. \tag{5.42}$$

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 $G = \mathsf{GB}_q(\{f_1, \ldots, f_l\}) = \mathsf{GB}_q(A)$, G has the form stated in Equation (5.39). ■

5.7 Variable Domains

Consider a polynomial $p(Z) \in \mathbb{R}_q[Z]$ 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[4] 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". Compare with the notion of variable range, $\mathsf{range}(x)$, used for decision diagrams in Chapter 4.

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

$$\lambda_q^{r_i}(z_i) = \begin{cases} \texttt{true} & , z_i = 0, \ldots, r_i - 1 \\ \texttt{false} & , z_i = r_i, \ldots, q - 1 \end{cases} \tag{5.43}$$

□

As mentioned in Section 5.4 `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.

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

Example 5.10 Simplification using Lambda Polynomials
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

One corresponding polynomial, $f(d,u) \in \mathbb{R}_3[d,u]$, is

$$f(d,u) = d + 2d^2 + 2d^2u + 2du^2. \tag{5.44}$$

Since u is constrained to $\{0, 1\}$ we have from (5.43)

$$\lambda_3^2(u) = u^2 - u \tag{5.45}$$

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

$$f(d,u) \underset{\mathsf{GB}_\lambda}{\longrightarrow} d + 2d^2 + 2d^2u + 2du \tag{5.46}$$

where $\mathsf{GB}_\lambda = \mathsf{GB}_3(\{\lambda_3^2(u)\}) = \{u^2 - u\}$

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

Definition 5.12 Lambda Polynomial Set
Let Λ_q denote the set containing all $\lambda_q^{r_i}(z_i)$ corresponding to $z_i \in \{0, \ldots, r_i - 1\}$ where $r_i < q$. □

We can now state the main simplification rule:

To compute the simplified polynomial of $p(Z)$, do the following

$$\text{reduce } p(Z) \text{ w.r.t. } \mathsf{GB}_q(\{\Lambda_q\}). \tag{5.47}$$

A polynomial defined as in Definition 5.12 can also be used to eliminate false solutions outside the valid domain. Given some relation $p(Z) \subset \mathbb{R}_q[Z]$ 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 5.11 Eliminate False Solutions
Consider the function from Example 5.10. By solving the equation

$$d + 2d^2 + 2d^2u + 2du^2 = 0 \tag{5.48}$$

we obtain the following set of solutions:

$$\{(d,u) : (0,0),(0,1),(0,2),(1,0),(1,2),(2,2)\}. \tag{5.49}$$

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

$$I = \langle d + 2d^2 + 2d^2u + 2du^2, u^2 - u\rangle \tag{5.50}$$

we will get the variety

$$V(I) = \{(d,u) : (0,0),(0,1),(1,0)\} \tag{5.51}$$

which contains only correct solutions.

Instead of computing $\mathsf{GB}_q(\{\lambda_q^{r_i}(z_i)\})$ we can directly construct a $\lambda-$polynomial that is a Gröbner basis.

Lemma 5.3 Irreducible Lambda Polynomials
For the $\lambda-$polynomial

$$\lambda_q^{r_i}(z_i) = \prod_{0\le k<r_i}(z_i - k) \tag{5.52}$$

we have that $\mathsf{GB}_q(\{\lambda_q^{r_i}(z_i)\}) = \{\lambda_q^{r_i}(z_i)\}$, i.e., Equation (5.52) gives a Gröbner basis. □

Proof Since $\mathsf{GB}_q(\{\lambda_q^{r_i}(z_i)\})$ corresponds to a Gröbner basis of two univariate polynomials $\lambda_q^{r_i}(z_i)$ and $z_i^q - z_i$ in the same variable we actually perform a greatest common divisor computation.

We compute $\mathsf{GB}_q(\{\lambda_q^{r_i}(z_i)\})$ from the general Gröbner basis

$$\mathsf{GB}(\{\lambda_q^{r_i}(z_i), z_i^q - z_i\}). \tag{5.53}$$

We know that

$$z_i^q - z_i = \prod_{0\le k\le q}(z_i - k). \tag{5.54}$$

This means that $z_i^q - z_i$ can be written as $f(z_i)\lambda_q^{r_i}(z_i)$, i.e., $\langle\lambda_q^{r_i}(z_i), z_i^q - z_i\rangle = \langle\lambda_q^{r_i}(z_i)\rangle$. Then a polynomial with r_i roots must have the degree r_i, and by construction the leading coefficient will be 1 corresponding to the autoreduced Gröbner basis. ■

Example 5.12
All Gröbner basis of λ-polynomials in $\mathbb{R}_7[z]$ are listed below:

r	$\lambda_q^r(z)$
1	z
2	$6z + z^2$
3	$2z + 4z^2 + z^3$
4	$z + 4z^2 + z^3 + z^4$
5	$3z + 6z^2 + 4z^4 + z^5$
6	$6z + z^2 + 6z^3 + z^4 + 6z^5 + z^6$
7	$6z + z^7$

5.8 Gröbner Basis Tools

The Gröbner Basis algorithm defined in Section 5.5.1 can be used as a computational engine in which we implement the operations needed for our relational framework.

Arithmetic operations on functions represented by a single polynomial are trivially implemented. Relational operations like $<$ cannot be represented by operations on these polynomials since there is no ordering for elements in $\mathbb{F}_q$. Equality of two polynomials $f(x) = g(x)$ can be given a canonical form by computing the Gröbner basis

$$\mathsf{GB}_q(\{f(x) - g(x)\}) \tag{5.55}$$

5.8.1 Logic operations

Here we will present how to implement the $\wedge$, $\vee$, and $\neg$ operations as well as the quantifiers $\exists$ and $\forall$. As stated in Section 5.4 we regard $= 0$ as `true` and $\neq 0$ as `false`.

The inputs to the operations are ideals $\langle A \rangle$ and $\langle B \rangle$ represented by the set of generators $A = \{a_1, \dots, a_s\}$ and $B = \{b_1, \dots, b_t\}$, respectively. Ideals and polynomials belong to the quotient ring $\mathbb{R}_q[Z]$.

For proofs of the following statements see [40].

And-operation

Since the $\wedge$-operation corresponds to intersecting the solution sets, i.e., the common roots, we have

$$\langle A \rangle \wedge \langle B \rangle = \mathsf{GB}_q(A \cup B). \tag{5.56}$$

Or-operation

This operation corresponds to the union of solution sets. The $\vee$-operation is computed as

$$\langle A \rangle \vee \langle B \rangle = GB_q(A \times B) \tag{5.57}$$

where $A \times B$ is the cross product of the polynomial sets which produces the set of pairwise products of polynomials from A and B.

Not-operation

To compute the $\neg$-operation on a single polynomial we have from Section 5.4:

$$\neg a_i = 1 - a_i^{q-1} \tag{5.58}$$

followed by reducing the result by a Gröbner basis computation. For an ideal we do the following:

$$\neg \langle A \rangle = GB_q(\{(1 - a_1^{q-1}) \cdots (1 - a_s^{q-1})\}) \tag{5.59}$$

since the complement of an intersection is the union of complements.

Quantification

The existential quantification like $\exists y.\, A(x, y)$ performs a mapping of the two dimensional solution set of $A(x, y) = 0$ into one dimension, in this case the x-axis.

By choosing a special lexical ordering of the variables when computing a Gröbner basis we will have the polynomials representing the projection included in the resulting polynomial set.

Having the variable sets $X = \{x_1, \ldots, x_s\}$ and $Y = \{y_1, \ldots, y_t\}$, and the ideal $\langle A(X, Y) \rangle \trianglelefteq \mathbb{R}_q[X, Y]$, the existential quantifier is computed as

$$\exists Y.\, \langle A(X, Y) \rangle = \langle GB_q(A(X, Y), Y > X) \cap \mathbb{F}_q[X] \rangle \tag{5.60}$$

which will return an ideal in $\mathbb{R}_q[X]$ where the variety corresponds to the projection of $A(X, Y)$ on X.

Unfortunately there is no similar way to compute $\forall Y. A(X, Y)$. Therefore we formulate the universal quantifier from the existential one and by using the $\neg$-operator.

$$\forall Y.\, \langle A(X, Y) \rangle = \langle \neg(\exists Y.\, (\neg A(X, Y))) \rangle \tag{5.61}$$

It is not necessary to reduce the inner $\neg$-operation by a Gröbner basis computation. Therefore the universal quantifier only needs two Gröbner basis computations.

Example 5.13 Logic Operations using Gröbner Bases
This example is presented as a *Mathematica* notebook (ver 3.0). The total time of the computations in this notebook is 15 seconds on a Sun Sparc 20. % refers to previous output.

We will define GB to be the lexical Gröbner basis ($x_1 > x_2$) for the variables x_1 and x_2 in the quotient polynomial ring $R_5[x_1, x_2]$. GB takes a list of generator polynomials as input.

```
In[1]:=   GB[poly_] :=
              GroebnerBasis[poly ∪ {x1^5 − x1, x2^5 − x2}, {x1, x2}, Modulus → 5];
```

The $\wedge$ -operation is computed as a Gröbner basis of the union of the generators.

```
In[2]:=   gbAnd[A_, B_] := GB[A ∪ B];
```

The $\vee$ -operation is computed as the Gröbner basis of all pairwise multiplications of A and B. `Outer` performs a matrix of two cross multiplied vectors. This matrix is then flattened to a list.

```
In[3]:=   gbOr[A_, B_] := GB[Flatten[Outer[Times, A, B]]];
```

For the $\neg$ -operation we apply the function $1 - p^4$ on the generators and then take the product of them together.

```
In[4]:=   gbNot[A_] := GB[{Apply[Times, Map[(1 − #1^4 &), A]]}];
```

Compute the relations $(x_1 {=\!=} 0) \wedge (x_2 {=\!=} 4) \vee (x_1 {=\!=} 1) \wedge (x_2 {=\!=} 3)$

```
In[5]:=   R1 = gbOr[gbAnd[{x1}, {x2 − 4}], gbAnd[{x1 − 1}, {x2 − 3}]]

Out[5]=                   {2 + 3x2 + x2^2, 1 + x1 + x2}
```

and $(x_1 {=\!=} 1) \wedge (x_2 {=\!=} 3) \vee (x_1 {=\!=} 2) \wedge (x_2 {=\!=} 3)$.

```
In[6]:=   R2 = gbOr[gbAnd[{x1 − 1}, {x2 − 3}], gbAnd[{x1 − 2}, {x2 − 3}]]

Out[6]=                   {3 + 4x2, 2 + 2x1 + x1^2}
```

Check if these relations has the correct set of solutions:

```
In[7]:=   Solutions[R1]

Out[7]=                        {{0, 4}, {1, 3}}


In[8]:=   Solutions[R2]

Out[8]=                        {{1, 3}, {2, 3}}
```

The Gröbner basis of $R_1 \wedge R_2$:

```
In[9]:=   gbAnd[R1, R2]

Out[9]=                     {3 + 4x2, 1 + 4x1}


In[10]:=   Solutions[%]
```

```
Out[10]=                    {{1, 3}}
```

The Gröbner basis of $R_1 \vee R_2$:

```
In[11]:=    gbOr[R1, R2]

Out[11]=        {2 + 3x2 + x2^2, 3x1 + 4x1x2, 4 + x1 + 3x1^2 + 4x2}


In[12]:=    Solutions[%]

Out[12]=                {{0, 4}, {1, 3}, {2, 3}}
```

The Gröbner basis of $\neg R_1$:

```
In[13]:=    gbNot[R1]

Out[13]=    {4x2 + x2^5, 4x2 + 2x1x2 + 2x1^2x2 + 2x1^3x2 +
                3x1^4x2 + x2^2 + 4x1x2^2 + 4x1^2x2^2 + 4x1^3x2^2 + 3x1^4x2^2 + 4x2^3 +
                3x1x2^3 + 3x1^2x2^3 + 3x1^3x2^3 + 4x1^4x2^3 + x2^4 + x1x2^4 + x1^2x2^4 + x1^3x2^4,
              4x1 + x1^5}


In[14]:=    Solutions[%]

Out[14]=        {{0, 0}, {0, 1}, {0, 2}, {0, 3}, {1, 0}, {1, 1}, {1, 2}, {1, 4},
                  {2, 0}, {2, 1}, {2, 2}, {2, 3}, {2, 4}, {3, 0}, {3, 1},
                  {3, 2}, {3, 3}, {3, 4}, {4, 0}, {4, 1}, {4, 2}, {4, 3}, {4, 4}}
```

The universe of discourse in this case has 5^2 elements. The relation R_1 has 2 solutions. Therefore the number of solutions of $\neg R_1$ is:

```
In[15]:=    Length[%]

Out[15]=                    23
```

We will make the following implementation of the $\exists$-operator, where `poly` is the generator set, q is the quantified variable and p is all the other variables.

```
In[16]:=    gbExist[poly_, q_, p__] := Select[
                GroebnerBasis[poly ∪ {x1^5 − x1, x2^5 − x2}, {q, p}, Modulus → 5],
                Variables[#1] == {p}&];
```

And for the universal quantifier:

```
In[17]:=    gbForAll[poly_, q_, p__] := gbNot[gbExist[gbNot[poly], q, p]];
```

The Gröbner basis of $\exists x_2 . R_1$ is then computed to:

```
In[18]:=    gbExist[R1, x2, x1]
```

```
Out[18]=                    {4x_1 + x_1^2}

In[19]:=   Solutions[%]

Out[19]=                    {{0},{1}}
```

But for $\forall x_2.R_1$ we get:

```
In[20]:=   gbForAll[R_1, x_2, x_1]

Out[20]=                    {1}
```

The ideal $\langle 1 \rangle$ has no solutions and represents the relation identical to `false`. Finally we will reveal the implementation of `Solutions` :

```
In[21]:=   Solutions[pset_List] :=
             Module[{iter, vars = Union@@Variables/@pset},
           iter = Sequence@@({#, 0, 4}&/@vars);
           Position[
                   Table[max @@(Mod[#1, 5]&)/@pset, Evaluate[iter]], 0] − 1
           ];
```

5.8.2 Complexity

The complexity of Gröbner basis computations is large in general. As always it is important to reduce the number of variables as much as possible. The number of variables is heuristically more important for the complexity than the size of the variable domains. Therefore Gröbner bases might be a good choice if we have large variable domains.

Gröbner basis computations for polynomial sets of $\mathbb{R}_q[Z]$ show much better complexity figures than general polynomials, due to the fact that there is an upper limit on the degree of the polynomials in $\mathbb{R}_q[Z]$. Normally q and Λ_q are known a priori for a specific problem. Then the Gröbner basis algorithm can be optimized so that the result of every step in the algorithm is reduced according to the zero-polynomials and Λ_q.

More information on Gröbner bases and complexity for finite field problems can be found in [40].

6 IDD & IMD

In Chapter 4, the use of reduced function graphs was discussed mainly in the case of Boolean functions and relations, which could efficiently be represented by BDDs. The Boolean representations are very important in many cases, especially for verification of hardware, where the function of digital implementations are verified.

However, in the case of having conceptual models for general finite domains we need to search for better representations. Often, we need to represent integer relations directly without having to bother about Boolean bit representations of integers. Moreover, the complexity of the representations often depends on the number of variables included in the function. Therefore we will gain complexity advantages if integer entities are represented by single variables. We also hope that by using the internal structure of entities in models and by tailored representation for every entity, we keep the structure of the model unchanged which in turn will reduce complexity contributions from improper representations.

This chapter contains the main results of this thesis regarding representational issues. First we will introduce the *integer decision diagram* (IDD), which is an extension to the BDD structure. The IDD can represent any finite domain relation or function, in particular the arithmetic functions like addition and multiplication. Upper limits of the complexity for addition and multiplication are also proved, e.g., the number of vertices grows polynomially with the number of variables for IDDs representing multiplication. Moreover, the IDD structure supports the same features as BDDs for efficient computation of relations, i.e., the Boolean operators are extended to relations including non-Boolean entities. The IDDs are efficiently implemented in a C code package which is linked to *Mathematica* and integrated into our tools for modeling and analysis of DEDS. The Landing Gear project, presented in Chapter 7, is used as a benchmark for the IDDs.

In this chapter, we will also present how to efficiently represent integer relations and functions by using a reduced graph representation of polynomials. This means that relations and functions are first represented mathematically by polynomials in $\mathbb{R}_q[Z]$, see Chapter 5, then the polynomials are represented by reduced function graphs. The first form of this type of representations is the *reduced recursive polynomial expression* (RRPE) [40]. We will introduce a new form which we will call *integrated monomial diagram* (IMD) for reasons that will become clear later on in this chapter. The purpose of representing polynomials by reduced function graphs is to represent unique polynomial subexpressions only once in a graph. This means for instance, that the size of an IMD is less than ordinary fully expanded polynomials. The IMD will utilize the reuse of subexpressions even further by separating integer factorials. It will also allow different value ranges of the variables which means that the polynomials can be tailored to fit specific variable domains. (Compare with the λ-polynomials in Section 5.7.)

These tools of graph represented polynomials have not yet been implemented and tested in real applications. However, we know that they are efficient for arithmetic operations like, addition and multiplication, where we have polynomial growth with respect to the number of variables.

6.1 Integer Decision Diagrams

In this section we will consider functions $f : A \rightarrow B$ as represented by reduced function graphs called *integer decision diagrams* (IDDs). Here A is a vector of finite sets and B is a finite scalar.

Example 6.1 Functions over Finite Domains
For the function $f(x_1, x_2, x_3), f : A \rightarrow B$ we have the domains

$$A = \{\mathbb{Z}_2, \mathbb{Z}_3, \mathbb{Z}_4\} \tag{6.1}$$

and

$$B = \mathbb{Z}_2. \tag{6.2}$$

This means that the values of the arguments are

$$x_1 \in \{0, 1\}, \quad x_2 \in \{0, 1, 2\}, \quad x_3 = \{0, 1, 2, 3\} \tag{6.3}$$

and that $f(x_1, x_2, x_3) \in \{0, 1\}$.

The motivation of the BDD structure was the Shannon expansion which was recursively applied for the terms of the Boolean expression according to some variable order. In the case of integer functions over finite domains we do not have any algebraic formulation in general. Therefore we cannot extend the Boolean algebra directly.

Instead we may just consider the integer function as represented by a table from which we extract some structure according to a variable ordering. To do this we need to define the *Lagrange function*.

Definition 6.1 Lagrange Function
The *Lagrange function*[1] denoted $L_k(x)$ is defined as

$$L_k(x) = \begin{cases} 1 & k = x \\ 0 & k \neq x \end{cases} \tag{6.4}$$

□

Instead of using the Shannon expansion we may write:

$$\begin{aligned} f(x_1, \dots, x_n) &= L_0(x_1)\, \underbrace{f(0, x_2, \dots, x_n)}_{f_0^1(x_2, \dots, x_n)} + \cdots \\ &\quad + L_{d_1}(x_1)\, \underbrace{f(d_1, x_2, \dots, x_n)}_{f_{d_1}^1(x_2, \dots, x_n)} \\ f_0^1(x_2, \dots, x_n) &= L_0(x_2)\, f_{0,0}^2(x_3, \dots, x_n) + \cdots \\ &\quad + L_{d_2}(x_1)\, f_{0,d_2}^2(x_3, \dots, x_n) \\ &\vdots \\ f_{d_1, \dots, d_{n-1}, 0}^n &= c_0 \\ &\vdots \\ f_{d_1, \dots, d_{n-1}, d_n}^n &= c_{m-1} \end{aligned}$$

where $d_i = \mathsf{range}(x_i) - 1$ and the number of constant terminals c_j is

$$m = \prod_{0<i\leq n} \mathsf{range}(x_i) \tag{6.5}$$

The function $f(x_1, \dots, x_n)$ is divided into subexpressions with progressively fewer variables. Each subexpression can be regarded as a subgraph (in this case a subtree) in the fully expanded function tree of $f(x_1, \dots, x_n)$.

Analogously to the BDDs we can represent $f(x_1, \dots, x_n)$ with a reduced function graph where identical subexpressions are represented by a single subgraph.

We will call this reduced function graph for functions over finite domains, integer decision diagram and define it as follows.

Definition 6.2 Integer Decision Diagrams
An *integer decision diagram* is a reduced function graph (definitions 4.1 and 4.5) where the number of terminals M is arbitrary and the variable domain $\mathsf{range}(x_i)$ may be chosen arbitrarily for each variable x_i.

[1]The name Lagrange function is natural considering the *Lagrange interpolation polynomial* in Chapter 5.3.

Remark:

- A BDD is a special case of an IDD.
- From Theorem 4.1 we know that an IDD is a canonical form.

□

The connection between IDDs and functions follows immediately from Definition 4.2 but may be restated using the Lagrange function in order to obtain something more similar to Equation (4.7).

Definition 6.3 IDD to Function Connection
An IDD G having root vertex v denotes a function f_v defined recursively as

1. If v is a terminal vertex, then $f_v = \mathsf{value}(v) \in \{0, \ldots, M-1\}$.
2. If v is a nonterminal vertex with $\mathsf{index}(v) = i$, then f_v is the function

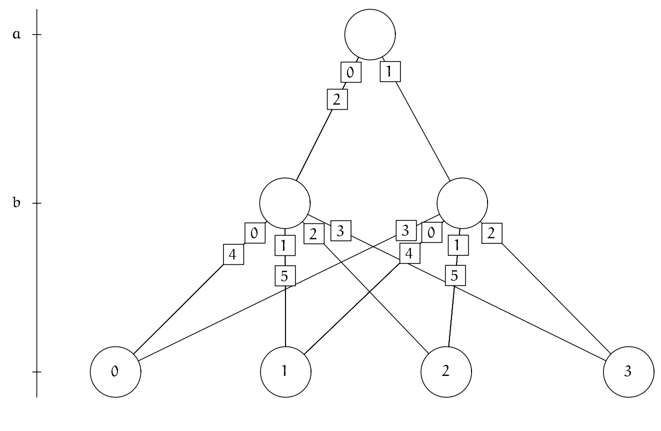
$$\begin{aligned} f_v(x_1, \ldots, x_n) = {} & L_0(x_i)\, f_{\mathsf{child}(v,0)}(x_1, \ldots, x_n) + \\ & L_1(x_i)\, f_{\mathsf{child}(v,1)}(x_1, \ldots, x_n) + \cdots + \\ & L_{\mathsf{range}(x_i)-1}\, f_{\mathsf{child}(v,\mathsf{range}(x_i)-1)}(x_1, \ldots, x_n) \end{aligned}$$

Here we assume that the maximum index of vertices in G is n. □

Example 6.2 IDD
Consider the two variables $a \in \mathbb{Z}_3$ and $b \in \mathbb{Z}_6$, and the function

$$f(a, b) = (a^2 + b) \bmod 4 \tag{6.6}$$

which is represented by the following IDD:



Each path from the root vertex down to a terminal corresponds to specific values of the arguments a and b. There is only one copy of each vertex corresponding to a unique expression, e.g., for both $a = 0$ and $a = 2$ we have, due to the modulo operation, the same expression in b.

Nr	Name	Interpretation	Simplification	Com.
1	*Plus*(a, b)	$\min(a+b, M-1)$	$a = 0 \Rightarrow b$	yes
2	*PlusMod*(a, b, c)	$a + b \bmod c,\ c \le M$	$a = 0 \Rightarrow b \bmod c$	yes
3	*Minus*(a, b)	$\max(a-b, 0)$	$b = 0 \Rightarrow a$	no
4	*Times*(a, b)	$\min(a * b, M-1)$	$a = 0 \Rightarrow 0$ $a = 1 \Rightarrow b$	yes
5	*TimesMod*(a, b, c)	$a * b \bmod c,\ c \le M$	$a = 0 \Rightarrow 0$ $a = 1 \Rightarrow b \bmod c$	yes
6	*Mod*(a, c)	$a \bmod c,\ c \le M$		no
7	*Equal*(a, b)	if $a = b$ then 1 else 0		yes
8	*LessThan*(a, b)	if $a < b$ then 1 else 0	$b = 0 \Rightarrow 0$	no
9	*LessThanEQ*(a, b)	if $a \le b$ then 1 else 0	$b = M - 1 \Rightarrow 1$	no

Table 6.1: Arithmetic and relational operations in IDDs. The column "Simplification" shows rules that give faster recursion termination. If the operation is commutative ('Com.') only the simplification rules for a is presented.

6.1.1 Operations and Algorithms

The set of operations that can be used for IDDs includes all operations from BDDs such as the *ite*-operator. In addition, we will also implement arithmetic operations like *plus* and *times* as well as relational operations like $>$ and $=$ (equality).

Arithmetic and Relational Operations

Unfortunately we cannot find a unifying three input operator for the arithmetic operations as we did for the BDDs by defining the *ite*-operator. We can of course define a *caseof*-operation that follows closely the IDD structure, see Equation (6.15), but we will not gain anything in doing that for the arithmetic operations. However, the implementations of the arithmetic operations are very similar and we will present all implementations together in one algorithm. The implemented arithmetic and relational operations are presented in Table 6.1. Note that we interpret $0 =$ `false` and $1 =$ `true` as for the BDDs.

Since we have a finite domain we have to be careful how to interpret the behavior of the operators at the limits of the integer range specified. Note that we, at this point, have not used any of the algebraic properties from Chapter 5 such as representing the finite domain as a ring or as a field. The arithmetic operations of the IDD are regarded as in $\mathbb{Z}$, which means that we use the normal interpretation except for the endpoints. Table 6.1 describes this interpretation in detail.

The main algorithm for implementing the arithmetic operations applicable to IDDs follow the same sketch as Algorithm 4.1 for the *ite*-operation. A depth-first recursion is used which utilizes the *unique table* and *cache table* in the same way as for the BDDs.

First we try to make some simplification and checks for terminal cases. The recursion stops when all arguments are terminals but most often the recursion is terminated before this happens due to the simplification rules. See Table 6.1.

Then the arguments are sorted according to the indexes of the arguments. This is possible only for the commutative operations (see Table 6.1). For generating unique hash values for the operators we need to add the operator identity to generate good cache keys. Compare this with the caching of $ite(F, G, H)$-operations where the three arguments uniquely specified the operation.

For a general arithmetic operation $op(a, b, c)$ in IDDs we use the BDD hash key function $\mathsf{key}(F, G, H)$ with the following values of F, G and H:

$$F = op + 16 * c \tag{6.7}$$

$$G = a \tag{6.8}$$

$$H = b \tag{6.9}$$

where c is the modulus for the operations *PlusMod*, *TimesMod* and *Mod*. The number *op* uniquely identifies the arithmetic and relational operation, see table 6.1. In this way we have a key function for operation identity and for two arguments. But we have to separate evaluations of the same operations and arguments using different modulus. This is done by (6.7) which allocates the 4 least significant bits to the operation identity *op* and the remaining 28 bits to the modulus. Since there are only 9 operations, 4 bits are enough. The largest modulus permitted is then $28^2 - 1 = 268435455$.

Relations and Complement Edges

To reduce the number of vertices in a reduced function graph the technique of using the same graph for f and $\neg f$ was introduced for BDDs in Section 4.2. To distinguish between f and $\neg f$ a complement marking of edges was used.

In the case of IDDs we will also use this method to represent relations. A relation in the form of an equation

$$f(x_1, \dots, x_n) = 0 \tag{6.10}$$

can be regarded as a function returning either 0 or 1, which in turn can be regarded as `false` and `true`, respectively. The variables $x_1, \dots, x_n$ may belong to any finite domain but the "value" of the relation $f = 0$ is always 0 and 1. The complement of Equation (6.10) differs only in that when the value of (6.10) is 0 then the complement is 1 and vice versa. This means that we can represent both $f = 0$ and $\neg(f = 0)$ by a single graph for the IDDs as well.

The method to implement this is to add the attribute $bool(v)$ to the IDD vertex v. This attribute is used as follows:

- When $bool(v) = $ `true`, the function f_v of v only evaluates to 0 or 1.

- When $bool(v) = \texttt{false}$, there exists a choice of the arguments of f_v such that the value of f_v is neither 0 nor 1.

Definition 6.4 Boolean Subgraph of IDD
The IDD subgraph rooted by v is called *Boolean* if the attribute $bool(v) = \texttt{true}$, where *bool* is recursively defined as

1. If v is a terminal then

$$bool(v) = \begin{cases} \texttt{true} & \mathsf{value}(v) \in \{0,1\} \\ \texttt{false} & \mathsf{value}(v) \notin \{0,1\} \end{cases} \tag{6.11}$$

2. If v is a non-terminal then

$$bool(v) = \bigwedge_{0 \le i < \mathsf{range}(v)} bool(\mathsf{child}(v,i)) \tag{6.12}$$

Remark:

- A representation of a relation always has $bool(v) = \texttt{true}$ for all vertices in the graph.
- All function graphs having only 0 and 1 as terminals, will have $bool(v) = \texttt{true}$ for all vertices.

□

The last remark may seem as a disadvantage. For example, the function defined as

$$f(x) = g(x) \bmod 2 \tag{6.13}$$

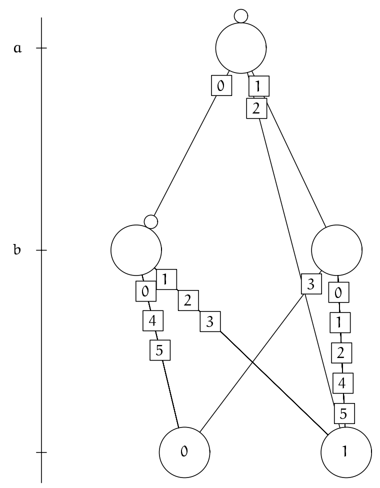
where $g(x)$ is an arbitrary function over some finite domain, will only evaluate to the values 0 or 1 and therefore be a Boolean function, which is probably not the intention. However, we must preserve a canonical representation and from the view of the IDDs there is no difference between (6.13) and the relation $f(x) = 1$. On the other hand, since Boolean relations are represented by 0 and 1 it is rather immediate what will happen if a Boolean function is an input to an arithmetic operation even though the result of course is not Boolean in general.

The relational operations *Equal*, *LessThan* and *LessThanEQ* in Table 6.1 will always return Boolean IDDs as results.

Example 6.3 Complement Edges in IDD
Consider the two variables $a \in \mathbb{Z}_3$ and $b \in \mathbb{Z}_6$, and the relation

$$(b > 2a) \wedge (b < 4) \tag{6.14}$$

which is represented by the following IDD:



The complement edges are showed as edges terminated by a ring. The root vertex is also complemented which means, e.g, that the edge corresponding to $a = 2$ should be interpreted as having the value 0 (`false`).

Boolean operators can easily be extended to work with relations of variables of any domain. The only difference is that the recursion of the *ite*-algorithm must follow all $\mathsf{range}(v)$ edges going from v.

The extended *ite*-algorithm can be reformulated as follows.

Algorithm 6.1 Extended ITE Algorithm

Syntax: *ite_apply*(F, G, H)

Input: IDD references F,G,H.

Output: A IDD reference to the resulting IDD, Z.

1. Simplify and check for terminal cases according to (4.15) and (4.16).

2. Sort the arguments F,G,H with top variable first according to (4.18).

3. Transform the complement edges of F,G,H to standard form according to (4.17). If F is complemented set the flag $c =$ `true` and remove the complement edge from F.

4. Look up in the cache table for a precomputed result Z. If found then if $c =$ `true` return $\neg Z$, else return Z.

5. Let t be the top index: $t = \mathit{Min}(\mathsf{index}(F), \mathsf{index}(G), \mathsf{index}(H))$ and start the recursion

```
for i := 0 to range(t) − 1
    begin
        $f_i := \begin{cases} \mathsf{child}(F, i) & t = \mathsf{index}(F) \\ F & \text{else} \end{cases}$
        $g_i := \begin{cases} \mathsf{child}(G, i) & t = \mathsf{index}(G) \\ G & \text{else} \end{cases}$
        $h_i := \begin{cases} \mathsf{child}(H, i) & t = \mathsf{index}(H) \\ H & \text{else} \end{cases}$
        $z_i := \mathit{ite\_apply}(f_i, g_i, h_i)$
    end
```

6. Let $Z = \mathit{FindOrMake}(t, z_0, \ldots, z_{\mathsf{range}(t)-1})$.
7. Store an entry in the cache table for vertex Z and the tuple (F, G, H) as the key.
8. If c return $\neg Z$, else return Z.

□

The function *FindOrMake* handles the update of the unique table, the complement edges and the *bool*-attribute.

Algorithm 6.2 Find or Make IDD Vertex

Syntax: $\mathit{FindOrMake}(\mathit{index}, z_0, \ldots, z_{n-1})$.

Input: The index of the root variable *index*, and n IDD subgraphs referenced by z_i.

Output: An IDD reference to the resulting IDD, Z.

1. If all subgraphs are identical $z_0 = \cdots = z_{n-1}$, then return $Z = z_0$.
2. Create a new vertex v' with index *index*.
3. Let $\mathit{bool}(v') = \mathtt{true}$ iff $\mathit{bool}(z_i) = \mathtt{true}$ for all i, according to Definition 6.4.
4. If z_1 is a complemented edge, let $c = \mathtt{true}$ and invert the complements on all edges $z_0, \ldots, z_{n-1}$.
5. Let $\mathsf{child}(v', i) = z_i$, $\forall\, i \in \{0, \ldots, n-1\}$.

6. Check if v' is in the unique table. If found, remove v' and let Z refer to the vertex found. If not found, put v' in the unique table and let $Z = v'$.
7. If c return $\neg Z$, else Z

□

The BDD operations *restrict*, *compose* and the quantification operators are extended in the same way to IDDs. From the *compose*- and the *restrict* operations we can formulate a *caseof* operation

$$Z = \mathit{caseof}(F, G_0, G_1, \dots, G_{\mathsf{range}(F)-1}) \tag{6.15}$$

which computes the function $Z = G_i | F = i$, i.e., the value of F chooses the corresponding G_i. The *caseof*-operation for IDDs is what the *ite*-operation is for BDDs, since the *caseof*-operation mimics the structure of IDD vertices in the same way as the *ite*-operation does for BDDs.

The Arithmetic Algorithm

We can now formulate the recursive algorithm for the arithmetic operations in Table 6.1.

Algorithm 6.3 Recursive Arithmetic IDD Operators
Head: *arith_apply*(op, A, B).

Input: Operation identity *op* and IDD references A, B.

Output: A reference to the resulting IDD, Z.

1. Simplify and check for terminal cases according to Table 6.1.
2. Sort the arguments A, B with top variable first if *op* is commutative according to Table 6.1.
3. Look up in cache table for a precomputed result Z. If found, return Z.
4. Let $t = \min(\mathsf{index}(A), \mathsf{index}(B))$ and do the recursion:

```
for $i := 0$ to $\mathsf{range}(t) - 1$
    begin
        $a_i := \begin{cases} \neg_A \mathsf{child}(A, i) & t = \mathsf{index}(A) \\ A & \text{otherwise} \end{cases}$
        $b_i := \begin{cases} \neg_B \mathsf{child}(B, i) & t = \mathsf{index}(B) \\ B & \text{otherwise} \end{cases}$
        $z_i := \mathit{arith\_apply}(op, a_i, b_i)$
    end
```

where the operator $\neg_v$ inverts the complement marking if v is complemented.

5. Let $Z = \mathit{FindOrMake}(t, z_0, \ldots, z_{\mathsf{range}(t)-1})$.

6. Store an entry in the cache table for vertex Z and the tuple (op, A, B) as key.

7. Return Z.

□

6.1.2 Complexity

The complexity of arithmetic and Boolean operations follows the worst case estimates presented in Section 4.2.1 for BDDs. The reason for this is the use of *unique* and *cache* tables that prevent the algorithms from recomputing old results and revisiting vertices in the recursions. Therefore we know that the operations for IDDs have a polynomial complexity with respect to the number of IDD vertices.

Since the complexity depends strongly on the number of vertices we will discuss how addition and multiplication of integer variables increases the size of the resulting IDD. For addition we can easily derive an explicite formula describing how variable ranges and the number of variables affect the IDD size. For multiplication we will only give an estimate.

If we first consider addition we have the following result.

Lemma 6.1 Complexity of IDD Additions
The number of vertices $S_{add}(n, D)$ (not including terminals) of an IDD representing an addition of n variables with the same variable domain D is given by

$$S_{add}(n, D) = \frac{n}{2}(n(|D| - 1) - |D| + 3) \tag{6.16}$$

□

Proof The idea behind the proof is to build the IDD with n variables from the IDD with $n-1$ by replacing the terminals at the bottom with new vertices for the n:th variable. Induction then gives the result. Let $d = |D|$.

For $n = 1$ we get the variable itself and therefore $S_{add}(1, D) = 1$. The number of terminals T_1 is d.

Now let $n = m$, where $S_{add}(m, D) = S_m$ and the number of terminals is T_m. Since addition does not cancel terms we must add a new nonterminal with index $m+1$ for every terminal when adding the $(m+1)$:th variable to the IDD. We get $S_{m+1} = S_m + T_m$. The number of new terminals T_{m+1} will increase by $d-1$ since we can add the value $d-1$ to the greatest of the previous terminals. Therefore $T_{m+1} = T_m + d - 1 \Rightarrow T_m = (m-1)(d-1) + d$ for $m > 1$. This means that $S_m = S_{m-1} + (m-2)(d-1) + d$ for $m > 1$ and $S_1 = 1$, which gives

$$S_m = \frac{m}{2}(md - m - d + 3) \tag{6.17}$$

■

From Lemma 6.1 we see that the size of addition depends quadratically on the number of variables and linearly on the domain size.

We can compare this result with the integer addition performed by using BDDs. In this case integers are represented by words of n bits, and the addition is represented by a BDD z_i for each bit i in the resulting word. The complexity for z_i is in the best case linear, and in the worst case exponential with respect to n, see [16]. The total number of vertices will in the worst case be $\mathcal{O}(ne^n)$.

If we use an IDD instead, we need 2 variables with domain 2^n. We will have $2^n + 1$ vertices in the resulting IDD, i.e., we get the complexity $\mathcal{O}(e^n)$. This indicates that the worst case complexity is similar for IDDs and BDDs if we have two variables with domains of any power of 2. But if we increase the number of variables we will probably get a worse increase of complexity for BDDs than for IDDs. Moreover, IDDs can adjust to any variable domain compared to BDDs where the domain always is 2^n.

A complexity result for multiplication is harder to find in a closed form. Instead we present an upper limit.

Lemma 6.2 Complexity of IDD Multiplication
An upper limit of the number of vertices $S_{mul}(n, D)$ (not including terminals) of an IDD representing a multiplication of n variables with the same variable domain D is given by

$$S_{mul}(n, D) \leq \binom{n + |D| - 2}{n - 1} \tag{6.18}$$

□

Proof Let S_n denote the number of non-terminal vertices for the IDD representing multiplication of n variables. The number of terminals T_n is equal to the number of distinct values M of the product

$$M = \prod_{1 \leq i \leq n} x_i \tag{6.19}$$

where $x_i \in \{0, \ldots, d-1\}$ and $d = |D|$.

With concrete methods of discrete mathematics [47] and some major insight from Gunnar Farnebäck, Linköping University, we will present an upper limit on T_n.

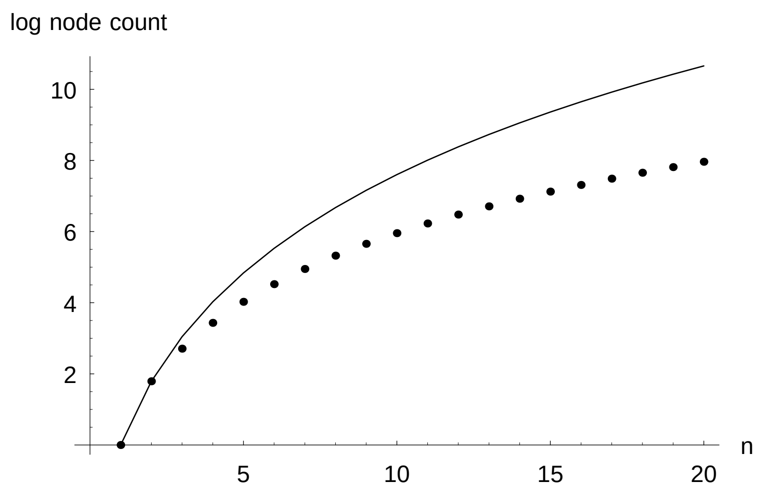
We can derive an upper approximation of T_n from the product

$$M' = \prod_{1 \leq i \leq n} p_i \tag{6.20}$$

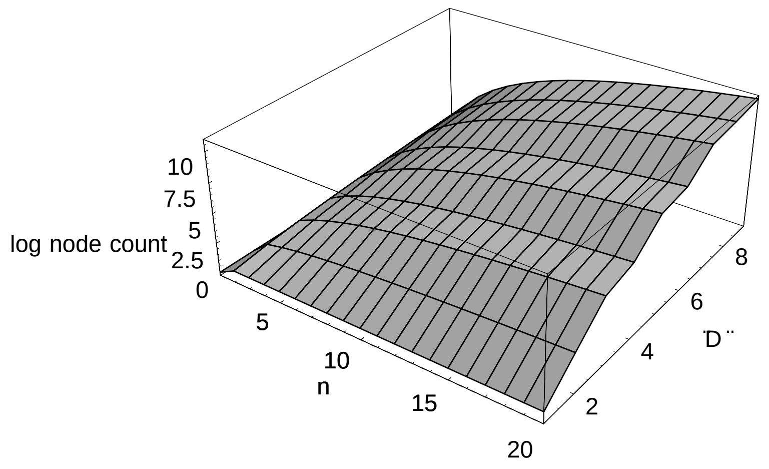
where p_i is chosen from 0, 1 and the $d-2$ first primes, i.e., the d first *semi primes*

$$p_i \in \{0, 1\} \cup \{P_j \mid 1 \leq j \leq d-2\}. \tag{6.21}$$

where P_j is the i:th prime number starting with $P_1 = 2$.



a)



b)

Figure 6.1: Multiplication complexity: a) Shows the logarithm of upper approximation (solid) and the actual number of vertices (dotted) for $|D| = 5$. b) Shows the logarithm of the actual number of vertices for $2 \leq |D| \leq 10$ and $1 \leq n \leq 20$.

Exclude for now the $M' = 0$ case, and let $p_i > 0$ and note that multiplication of elements from the set of semi primes (6.21) uniquely specifies one value of M' if the order of the p_i is not considered. But M is not a product of semi primes and therefore we will not have a unique M for every choice of x_i. Therefore we have more distinct values of M' than of M.

We can pick n values from the semi primes (6.21) in $\binom{n+d-2}{n}$ ways. By adding the zero case we get

$$T_n \leq \binom{n+d-2}{n} + 1 \tag{6.22}$$

The number of vertices for $n+1$ variables is $S_{n+1} = S_n + T_n - 1$ since the zero terminal vertices will not contribute. Thus

$$S_n \leq S_1 + \sum_{1<k\leq n} \binom{n+d-3}{n-1} = \binom{n+d-2}{n-1} \tag{6.23}$$

■

For a large n the upper limit grows asymptotically as $\mathcal{O}(n^{d-1})$. At the same time we have the growth $\mathcal{O}(d^{n+1})$ for large d. Hence, from this upper approximation we know that the complexity of multiplication is polynomial in both the number of variables and size of domains.

The fact that Lemma 6.2 gives a rather rough upper limit is illustrated by the following example.

Example 6.4 Complexity of IDD Multiplication
Figure 6.1 a) shows the natural logarithm for the number of vertices for both the upper approximation (solid line) and the actual number of vertices (dotted curve) for $|D| = 5$ and $1 \leq n \leq 20$.

We can let a computer plot the multiplication complexity for all combinations of $2 \leq |D| \leq 10$ and $1 \leq n \leq 20$. The result is showed in Figure 6.1 b).

Complexity results are often too conservative compared to the measures of applications where decision diagrams have been used. One such example is the modeling of the landing gear controller where both BDDs and IDDs were used. The time for compilation of the controller code written in Pascal was reduced by 50% using IDDs, and the number of vertices was in some cases reduced by 75%. See Section 7.5 for all details of this BDD/IDD benchmark.

6.2 Integrated Monomial Diagrams

This section deals with polynomials as the basic representation of functions, which might be a nice and efficient representation for functions where BDDs and IDDs are not suitable. We will use reduced function graphs to represent the polynomials, which in turn represent the functions. We will only use the structure of function graphs, not their interpretation. To define something called *polynomial graphs* would introduce a more appropriate terminology but would not give anything more than that.

We have developed a reduced function representation for polynomials which we call an *integrated monomial diagram* (IMD). IMDs follow the basic ideas from the IDDs, but we will develop the IMDs from the *reduced recursive polynomial expressions* (RRPE) [40]. During this presentation we will also mention the *binary momentum diagrams* (BMDs) which were introduced by Bryant et al. [18]. BMDs are a special case of IMDs in the same way as BDDs is a special case of IDDs.

6.2.1 Reduced Recursive Polynomial Expression

We will start the development of the reduced graphs for polynomials by considering *reduced recursive polynomial expressions* [40].

Definition 6.5 Recursive Polynomial Expression
Let $f \in \mathbb{R}_q[x_1, \ldots, x_n]$ be a polynomial expression with variables $x_i \in \mathbb{F}_q$. Then $f(x_1, \ldots, x_n)$ is in *recursive form* with respect to the variable ordering $x_1 > x_2 > \ldots > x_n$ iff

$$\begin{aligned} f(x_1, \ldots, x_n) &= \sum_{0 \le i < q} f^1_i(x_2, \ldots, x_n) x_1^i \\ f^1_i(x_2, \ldots, x_n) &= \sum_{0 \le j < q} f^2_{i,j}(x_3, \ldots, x_n) x_2^j \\ &\vdots \end{aligned}$$

□

A recursive polynomial expression with variable ordering $x_1 > x_2 > \ldots > x_n$ is a polynomial in x_1, where the other variables are collected as coefficients of this “univariate” polynomial in x_1. Then the same is done for x_2 locally in these coefficients, and so on for all variables in the polynomial.

Example 6.5 Recursive Polynomial Expression
The following polynomial in distributed form

$$f = 3 + 4\,x_3^3 + 2\,x_1{}^3 + x_1{}^3\,x_2 + x_1\,x_2{}^2 + 5\,x_1{}^3\,x_2{}^2\,x_3 \tag{6.24}$$

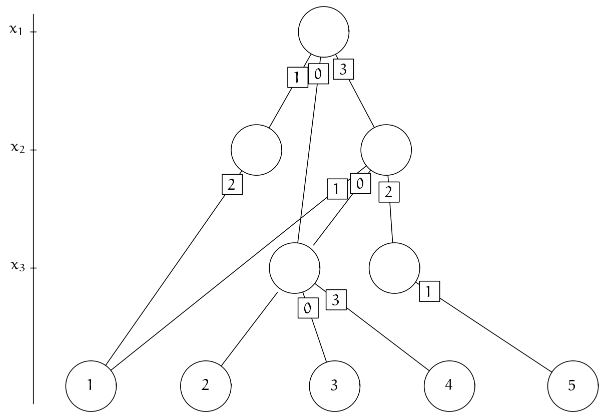
can be rewritten in recursive form as

$$f = (3 + 4x_3^3)1 + (x_2^2)x_1 + (2 + x_2 + (5x_3)x_2^2)x_1^3 \tag{6.25}$$

with the variable ordering $x_1 > x_2 > x_3$.

Recursive polynomial expressions can be represented by a tree where the edges are marked by a monomial in one variable. The subgraph of each edge correspond to the coefficient expression for a certain power of that variable.

Example 6.6 Recursive Polynomial Expression Graph
A function graph representation for the recursive polynomial expression in (6.25) is illustrated by the following figure:



Remember that the edge labels correspond to powers for the corresponding variables. The root vertex has three edges corresponding to x_1^1, x_1^0 and x_1^3, from left to right. All edges connecting to the terminal 0 have been omitted for clarity.

When a part of an expression is present in several places in a recursive polynomial expression we can refer to the corresponding subgraph of that expression several times. This corresponds to the reduced function graph defined in Section 4.1. For the polynomial representation we get the *reduced recursive polynomial expression* (RRPE) [40]. The structure of the RRPE is an ordinary reduced function graph. This means that the abbreviation RRPE stands for the graph representation from now on. The difference will be more obvious when we define how the RRPE-graphs connect to polynomials.

Definition 6.6 Reduced Recursive Polynomial Expression
A *Reduced Recursive Polynomial Expression* for the polynomial $f \in \mathbb{R}_q[Z]$ is a reduced function graph where

- The number of terminals is $M = q - 1$.
- Each variable $z_i \in Z$ corresponds to vertices with index i, assuming the variable ordering $z_1 > z_2 > \ldots > z_n$.
- The range $\mathsf{range}(v) = q$ for all vertices.

Remark:

- The RRPE is a canonical form.
- The root vertex of an RRPE corresponds to the highest variable in the ordering $z_1 > z_2 > \ldots > z_n$ even though the index of the root vertex is the lowest.

□

Since we have defined RRPEs to be reduced function graphs we have to define how RRPEs connect to polynomials.

Definition 6.7 The RRPE to Polynomial Connection
An RRPE G having root vertex v denotes a polynomial $p_v \in \mathbb{R}_q[Z]$, which is recursively defined as follows

1. If v is a terminal vertex, then $p_v = \mathit{value}(v) \in \{0, \ldots, q-1\}$.
2. If v is a nonterminal vertex with $\mathit{index}(v) = i$, then p_v is the polynomial

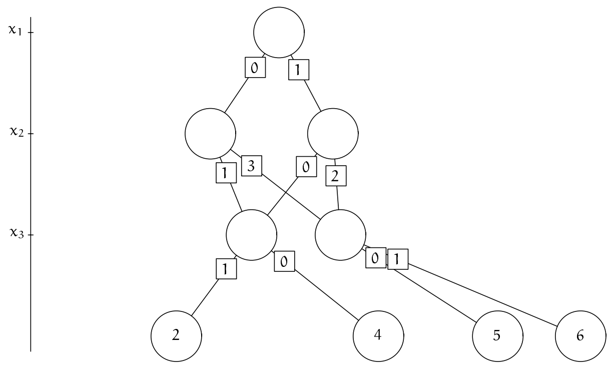
$$p_v = \sum_{0 \leq k < q} p_{\mathsf{child}(v,k)}\, z_i^k \tag{6.26}$$

□

Example 6.7 Reduced Recursive Polynomial Expression
For the following polynomial

$$4x_1 + 4x_2 + 5x_1 x_2{}^2 + 5x_2{}^3 + 2x_1 x_3 + 2x_2 x_3 + 6x_1 x_2{}^2 x_3 + 6x_2{}^3 x_3 \tag{6.27}$$

in $\mathbb{R}_7[x_1, x_2, x_3]$ we get the RRPE



for the variable ordering $x_1 > x_2 > x_3$.

Notice that both vertices corresponding to x_3 is referred two times each. In this simple case we have 8 edges in the RRPE compared to 10 terms in the polynomial.

The main arithmetic operations for RRPEs are addition and multiplication. Boolean operations are reformulated as addition and multiplication by the rules in Section 5.4.

Consider two RRPEs $f = \sum_{0 \le i < q} f_i x^i$ and $g = \sum_{0 \le i < q} g_i x^i$ having the same top variable x, then:

$$f + g = \mathit{RRPE\_Reduce}\Big(\sum_{0 \le i < q} (f_i + g_i) x^i \Big) \tag{6.28}$$

$$f \cdot g = \mathit{RRPE\_Reduce}\Bigg(\sum_{0 \le k < q} \Big(\sum_{\substack{i+j=k \bmod q \\ i,j \in \mathbb{Z}_q}} f_i \cdot g_i \Big) x^i \Bigg) \tag{6.29}$$

where *RRPE_Reduce* makes the graph reduced.

We see that multiplication is not local to each variable as for IDDs. This makes multiplication a very complex operation to perform. Therefore we will tune the representation even further to gain performance for these operations.

6.2.2 Integrated Monomial Diagrams

It is desirable to have a graph based representation for polynomials that mimics the power and flexibility of IDDs. The key features of IDDs that we will address here are:

- The domain can be specified for each variable independently.
- IDDs represent both relations and functions, where relations are marked and treated specially.
- Complements are handled by special markings for improved space performance.
- IDDs implement operations recursively and such that reduction is performed in every step. No reductions necessary afterwards.
- IDDs use caching of intermediate results for improved termination of recursions.

All these aspects of the IDD structure can be implemented for polynomials as well. The development of this structure starts from the RRPE to something that we might regard as a close relative to IDDs, which we will call *integrated monomial diagrams* (IMDs). Since this is a work in progress all details of IMDs cannot be expressed here.

As often these ideas were first developed for the Boolean case. *Binary moment diagrams* (BMDs) [18] are basically the same as the IMDs restricted to Boolean function.

Variable domain

For a given and fixed quotient polynomial ring $\mathbb{R}_q[Z]$ we want to manage variables with smaller value ranges than $\mathbb{Z}_q$.

In Section 5.7 we constructed lambda polynomials $\lambda_q^r(z) \in \mathbb{R}_q[Z]$, which are true only for valid values of z. The lambda polynomials for all "subranged" variables were collected in the polynomial set Λ_q and were reduced by a Gröbner basis calculation if necessary. Lemma 5.3 showed that the degree of $\lambda_q^r(z)$ is r which in turn means that all for polynomials in $\mathbb{R}_q[Z]/\Lambda_q$ the degree of the variable z will be less than r.

This is good news for graph represented polynomials. By using Λ_q for reduction in every step we will never need more than r edges of any vertex for the variable z. This makes the IMD a good choice if we have variables with different domains. Moreover, since the implementation of operations will use recursive algorithms extensively we gain performance by reducing the number of edges as much as possible.

Relational polynomials and Complements

We will mimic the IDD method to mark functions only evaluating to 0 and 1 with a special attribute *bool*(v), see Section 6.1.

For IMDs we regard the roots to a polynomial as `true` and the non zero values as `false`. This ambiguity for `false` has the consequence that two different polynomials may, if they have the same roots, represent the same relation. (Gröbner bases solved this problem syntactically by algebra.)

For the IMD we may use a more semantic approach. We say that the values of a relational polynomial have to be either 0 or 1, i.e., $0 =$ `true` and $1 =$ `false`. In this way we have a one-to-one correspondence between relations and polynomials. Moreover, we get the following features:

- To reduce a polynomial f to a relational polynomial with the same roots we compute f^{q-1}.
- We use a special attribute *bool*(v) on the vertices to indicate that the polynomial corresponding to the subgraph of v is a relational polynomial.
- The constant term in a relational polynomial is either 0 or 1. This can easily be verified by evaluating the polynomial with all variables set to zero.

The last remark above is enough to define a proper way to impose complement markings. If f is a relation polynomial corresponding to the relation R, then $1 - f$ is the polynomial corresponding to the complement relation $\neg R$.

This interpretation of relations in IMDs gives an efficient implementation of the $\neg$-operation since we only have to compute $1 - f$ to get the complement. By experience though, it turns out that relation representations with polynomials which are equal to 1 for values not belonging to the relation, give much more complex polynomials compared to the Gröbner basis representation. The Gröbner bases always return the best polynomial representation that preserves the roots, but let the non-root values be freely chosen. In some cases the polynomial representations, where `false`is forced to be equal to 1, will have a complexity of magnitudes larger than the corresponding polynomial for Gröbner bases. See the equations (6.32) and (6.34) for a convincing example.

Further research on the IMD representation aims to incorporate the IMDs with Gröbner bases for $\mathbb{R}[Z]$ in such a way that we get a balance between efficiency of size or computation.

To represent the relation R canonically we will choose one of the polynomials f or $\neg f$ with no constant term (zero), say f, and then add an edge marking for the complement if appropriate. Since the relational polynomials do not have constant terms we gain better complexity measures, both concerning memory space and computational performance.

When computing Boolean operators on relational polynomials we know that the result also will be a relational polynomial. But for other operations there is no such simple rule. We cannot derive the Boolean marking directly from the children of a vertex, and it is not generally true that all vertices in a subgraph to a Boolean vertex are Boolean. The attribute $\mathit{bool}(v)$ is sufficient but not necessary for this.

Extracting Constant Factors

For polynomial expressions like $1 + 2x$ and $3 + 6x$ we see immediately that they differ only by the constant factor 3. We want to exploit this for IMDs by allowing edge attributes for constant factors. This means that the expression $1 + 2x$ and $3 + 6x$ will be represented by the same IMD graph, but where the edges connecting it will have 1 and 3 as constant factors, respectively.

For polynomials in $\mathbb{R}_q[Z]$ we may choose to factor out an integer such that the term with highest degree always will have 1 as its coefficient.

To factor out integers from relational polynomials will not change the roots of the relation, but the complement values will be equal to 1 in general. To utilize integer factorization for relations we cannot have that `false` $= 1$ which in turn speaks for the Gröbner basis interpretation of relations as discussed above.

We will define the structure of IMD as follows.

Definition 6.8 Integrated Monomial Diagrams

An *integrated monomial diagram* is a reduced function graph with two terminal vertices 0 and 1, where each vertex v has the attributes $\mathit{factor}(v, i) \in \mathbb{Z}_+$, $\mathit{bool}(v, i) \in \mathbb{B}$ and $\mathit{comp}(v, i) \in \mathbb{B}$ for all the children $\mathsf{child}(v, i)$ connected to v.

The index of the root vertex of an IMD must correspond to the variable of highest ordering which is included in the corresponding polynomial expression of the IMD.

Remark:

- The last sentence removes the ambiguity of representing single constants. These are instead represented by vertices with an index set to the highest possible, connecting to the terminal vertex 1 with the appropriate value for $\mathit{factor}(v, i)$.

- The zero terminal can be removed and replaced by null edges not connecting any vertex.

□

The interpretation of IMDs is defined by the polynomial connection.

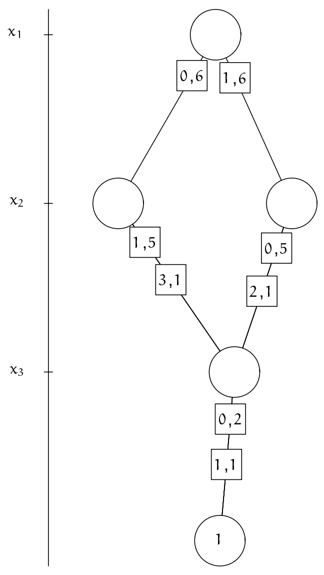
Definition 6.9 IMD to Polynomial Connection
An IMD G having root vertex v denotes a polynomial $p_v \in \mathbb{R}_q[Z]$ defined recursively as follows.

1. If v is a terminal vertex, then $p_v = \mathit{value}(v) \in \{0, 1\}$.
2. If v is a nonterminal vertex with $\mathit{index}(v) = i$, then p_v is the polynomial

$$p_v = \sum_{0 \leq k < q} p_{\mathsf{child}(v,k)}\, z_i^k\, \mathit{factor}(v, k) \tag{6.30}$$

□

Example 6.8 IMD
For the same polynomial as in Example 6.7 we get the IMD



The edge labels for the IMD consists of two integers. The first one specifies the power of the variable according to the RRPE. The second specifies the integer factor that the polynomial corresponding to a vertex should be multiplied with.

This will be exemplified by taking a closer look at two paths from the root vertex down to the terminal. These paths corresponds to the leading term and the term with minimum degree.

Leading term: Starting from the root vertex we follow the edge with the maximum power, i.e., the edge labeled $\boxed{1,6}$ with the factor 6. This edge corresponds to the monomial $6x_1$. From the right-most x_2-vertex we again follow the edge with the maximum power which lead to the monomial x_2^2. Then the last part of this path gives x_3. If we multiply these monomial contributions we get the term $6x_1x_2^2x_3$ which is equal to the leading term of Equation (6.27).

Minimum term: In the same way as for the leading term except that we always follow the minimal power, we get the term $6 \cdot 5x_2 \cdot 2$, which is $4x_2$ modulo 7. This term is also present in Equation (6.27).

Each distinct path from the root vertex to the terminal corresponds to one term of the polynomial in Equation (6.27). Notice also that since we can factor integers the implementation of IMDs always only need one terminal vertex.

The IMD has a leading coefficient equal to 1 for all vertices except for the root vertex in this case. We see from the IMD in this example that we factor the IMD by 6 from the root vertex.

From the example above it is easy to realize that the IMD has low complexity in representing arithmetic operations like addition and multiplication. The sum as well as the product of n variables will have linear growth in the IMD. Both of these operations will result in an IMD with n vertices connected into a chain with only one path. The same will hold for all linear polynomials with constant coefficients and for many other special polynomial structures.

Results on algorithms implementing operations for IMDs and complexity results for these cannot yet be presented.

6.3 Comparisons: Decision Diagrams vs. Polynomials

The BDD structure of representing Boolean formulas has been a standard solution in several engineering tools for formal verification. The reason is that the BDD has turned out to be capable of representing the class of Boolean expressions, which are frequent in practice, in an efficient way. Moreover, all the highly sophisticated software tools for BDDs available nowadays have of course increased the popularity of BDDs.

By the same reason we hope that the IDD concept will be useful for more general problem domains where integer functions and relations turn out to be less appropriate for a BDD representation. By construction the IDD is an extension to BDD, which means that implementations of IDD will have the same performance measures as BDD, but with better capability to represent relations of different domains. In Section 7.5 we have compared the use of IDDs and BDDs for a large application.

Then, is there a need for the polynomial representations? The only way to really answer this question is to try the polynomial approach in practice, i.e., to test if a tuned implementation of IMDs (or Gröbner bases) will be able to compete with representations based on decision diagrams for some large applications. This type of evaluations has not yet been performed though.

Instead, we reason about theoretical aspects of polynomial representations from the fact that there will always exist examples of relations for which any representation will explode. Therefore we know theoretically that no representation is better than another, if no specific class of problem is considered. The important issue is which types of problem are most common in practice. In other words; what is the most common structure of the problems that we have to deal with when using formal tools? What is clear from experience is that there are many problems with a structure suited for decision diagrams, but for polynomial representations we cannot say yet.

One observation for decision diagrams is that they provide very efficient algorithms for finding solutions (or roots) to relations. For BDDs and IDDs we can generate solutions in time proportional to the number of variables in the relation. One might interprete this observation in such a way that decision diagrams are only good for the class of problems where solutions are easy to find. In other words; problems with inherent solving complexity must be represented by complex decision diagrams. For these types of problems the polynomial representation might be a better choice.

6.3.1 Final Example

To conclude this discussion of the comparison between decision diagrams and polynomial representations we will use the Egyptian triangle relation ($\mathsf{a}^2 + \mathsf{b}^2 == \mathsf{c}^2$) and represent it by an IDD, a Gröbner basis, a principal polynomial and by an IMD.

The Egyptian triangle relation ETR is simply the relation between the two elements $\{[3,4,5],[4,3,5]\}$. The relation is constructed as

$$\text{ETR} = (\mathsf{a}==3)\wedge(\mathsf{b}==4)\wedge(\mathsf{c}==5) \vee (\mathsf{a}==4)\wedge(\mathsf{b}==3)\wedge(\mathsf{c}==5) \tag{6.31}$$

The universe of discourse will be set to $\mathbb{Z}_7 \times \mathbb{Z}_7 \times \mathbb{Z}_7$ which will correspond to the quotient polynomial ring $\mathbb{R}_7[\mathsf{a},\mathsf{b},\mathsf{c}]$.

IDD

The IDD representation with ordering $\mathsf{a} > \mathsf{b} > \mathsf{c}$ for the ETR is shown in Figure 6.2, where the two paths corresponding to the solution set are clearly visible.

Gröbner basis

The Gröbner basis representation of the ETR is the following set of polynomials:

$$\{\ \mathsf{a}+\mathsf{b},\ 5+\mathsf{b}^2,\ 5+6\mathsf{c}\ \} \tag{6.32}$$

where we have used the variable ordering $\mathsf{a} > \mathsf{b} > \mathsf{c}$. The last polynomial gives one unique solution for c, the middle polynomial gives two solutions for b, and the first gives a unique solution to a given the value of b.

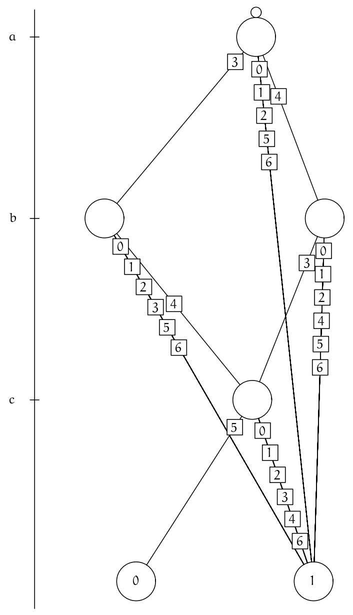


Figure 6.2: The ETR $(a^2 + b^2 == c^2)$ represented as an IDD.

Principal Polynomial

As we know, all relations can be represented by a single polynomial[2] by composing the polynomials from (6.32) together using the logical $\wedge$-operation, see Section 5.4.

$$1 - \left(1 - (a+b)^6\right)\left(1 - (5+b^2)^6\right)\left(1 - (5+6c)^6\right) \tag{6.33}$$

[2]$\mathbb{R}_q[Z]$ is a principal domain, see footnote in Section 5.5.2, which means that a relation can be represented by a single polynomial.

which will result in the following huge polynomial

$$\begin{aligned}
&1 + 4abc + 2a^3bc + a^5bc + 5a^2b^2c + 6a^4b^2c + 3a^6b^2c + \\
&2ab^3c + a^3b^3c + 4a^5b^3c + 6a^2b^4c + 3a^4b^4c + 5a^6b^4c + ab^5c + \\
&4a^3b^5c + 2a^5b^5c + 3a^2b^6c + 5a^4b^6c + 6a^6b^6c + 5abc^2 + 6a^3bc^2 + \\
&3a^5bc^2 + a^2b^2c^2 + 4a^4b^2c^2 + 2a^6b^2c^2 + 6ab^3c^2 + 3a^3b^3c^2 + \\
&5a^5b^3c^2 + 4a^2b^4c^2 + 2a^4b^4c^2 + a^6b^4c^2 + 3ab^5c^2 + 5a^3b^5c^2 + \\
&6a^5b^5c^2 + 2a^2b^6c^2 + a^4b^6c^2 + 4a^6b^6c^2 + abc^3 + 4a^3bc^3 + \\
&2a^5bc^3 + 3a^2b^2c^3 + 5a^4b^2c^3 + 6a^6b^2c^3 + 4ab^3c^3 + 2a^3b^3c^3 + \\
&a^5b^3c^3 + 5a^2b^4c^3 + 6a^4b^4c^3 + 3a^6b^4c^3 + 2ab^5c^3 + a^3b^5c^3 + \\
&4a^5b^5c^3 + 6a^2b^6c^3 + 3a^4b^6c^3 + 5a^6b^6c^3 + 3abc^4 + 5a^3bc^4 + \\
&6a^5bc^4 + 2a^2b^2c^4 + a^4b^2c^4 + 4a^6b^2c^4 + 5ab^3c^4 + 6a^3b^3c^4 + \\
&3a^5b^3c^4 + a^2b^4c^4 + 4a^4b^4c^4 + 2a^6b^4c^4 + 6ab^5c^4 + 3a^3b^5c^4 + \\
&5a^5b^5c^4 + 4a^2b^6c^4 + 2a^4b^6c^4 + a^6b^6c^4 + 2abc^5 + a^3bc^5 + \\
&4a^5bc^5 + 6a^2b^2c^5 + 3a^4b^2c^5 + 5a^6b^2c^5 + ab^3c^5 + 4a^3b^3c^5 + \\
&2a^5b^3c^5 + 3a^2b^4c^5 + 5a^4b^4c^5 + 6a^6b^4c^5 + 4ab^5c^5 + 2a^3b^5c^5 + \\
&a^5b^5c^5 + 5a^2b^6c^5 + 6a^4b^6c^5 + 3a^6b^6c^5 + 6abc^6 + 3a^3bc^6 + \\
&5a^5bc^6 + 4a^2b^2c^6 + 2a^4b^2c^6 + a^6b^2c^6 + 3ab^3c^6 + 5a^3b^3c^6 + \\
&6a^5b^3c^6 + 2a^2b^4c^6 + a^4b^4c^6 + 4a^6b^4c^6 + 5ab^5c^6 + 6a^3b^5c^6 + \\
&3a^5b^5c^6 + a^2b^6c^6 + 4a^4b^6c^6 + 2a^6b^6c^6
\end{aligned} \tag{6.34}$$

which has the property that all non zero values are equal to 1.

IMD

From the polynomial (6.34) above we can develop the IMD representation of the ETR. This is done by rewriting (6.34) recursively corresponding to the variable ordering $a > b > c$, identifying common subexpressions and factorizing the expression so that the leading term coefficient is 1.

The IMD for the ETR is shown in Figure 6.3. From the figure we see a significant reduction of representation size. In fact, the IMD has the same number of vertices (not counting terminals) as the corresponding IDD for this example.

6.4 Summing Up

This chapter has been devoted to present our results for relational representations. Integer decision diagrams have been defined as an extension of BDDs, for which algorithms both for logical and arithmetic operations have been defined. Complexity results for addition and multiplication have been derived, which shows that addition grows quadratically in the number of variables, whereas multiplication has an upper limit which is polynomial in the number of variables. The IDDs are implemented by extending the highly efficient BDD package [11] and IDDs have been used in the landing gear application, Section 7.5, for comparison with the BDD approach.

Suggestions to develop efficient tools for polynomial representations have also been given in this chapter. The structure of *integrated monomial diagrams* (IMDs) has been defined to be one such tool. It can easily be verified that IMD has good size complexity for arithmetic operations. Further research will show the details of operator algorithms for the IMD.

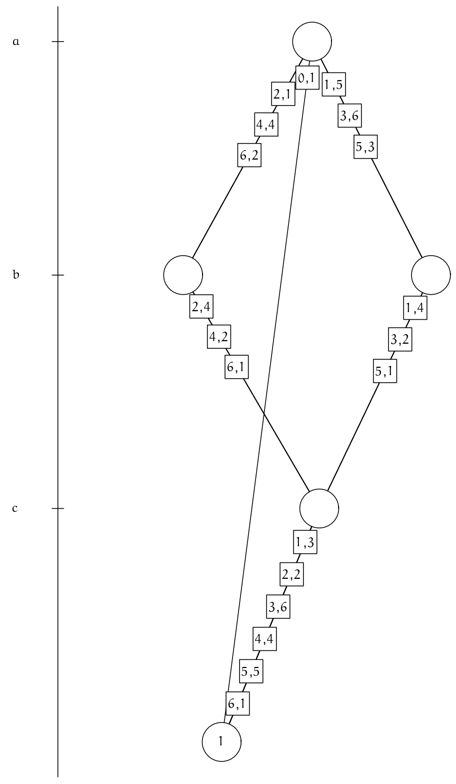


Figure 6.3: The ETR $(\mathsf{a}^2 + \mathsf{b}^2 == \mathsf{c}^2)$ represented as an IMD.

Part II

Applications

7 Verification of Landing Gear Controller

7.1 The Landing Gear Process

The case study in Sections 7.3 and 7.4 concerns the landing gear system on the Swedish fighter JAS 39 Gripen, depicted in Figure 7.1.

The landing gear system consists of a landing gear controller and three landing

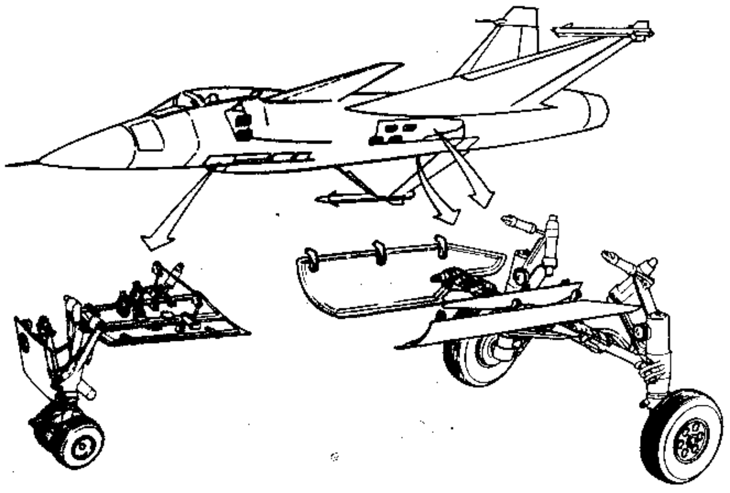


Figure 7.1: The fighter JAS 39 Gripen.

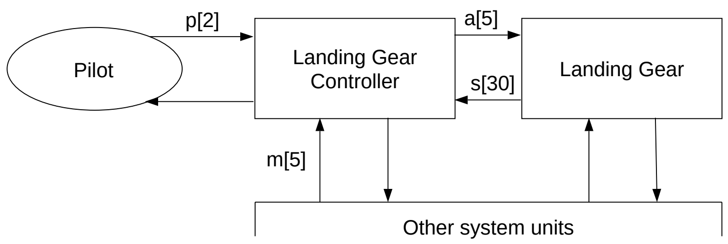


Figure 7.2: The landing gear system (number of signals in brackets).

gears with corresponding doors. A simplified block description of the complete system is shown in Figure 7.2, 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.

7.1.1 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 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

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 7.1: 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 7.2: Actuator signals: Commands to the hydraulic system.

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.

7.1.2 The Signal Interface

As shown in Figure 7.2 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 7.1, 7.3 and 7.4. 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 7.2. 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 7.3: 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 7.4: Landing gear feedback: Output from microswitches on gears and doors.

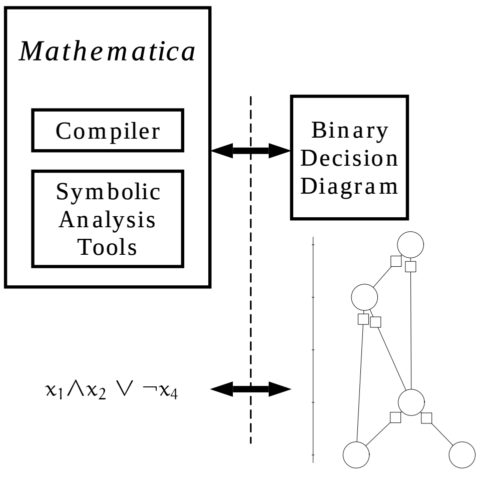


Figure 7.3: The software configuration based on *Mathematica*. The BDD-tool is kept in an external C-package which communicates with *Mathematica* thorough MathLink. From *Mathematica* the BDDs act like ordinary Boolean expressions.

7.2 Software Tools

Using an efficient symbolic algebraic computation engine is crucial if we are to be able to analyze realistically sized examples. In this project we use an experimental software system consisting of *Mathematica* [114, 115] code together with externally linked C code for critical operations through the MathLink structured communication protocol, see Figure 7.3.

The C code used in this package is an efficient implementation of *binary decision diagrams* (BDDs) and *integer decision diagrams* (IDDs) which we use as a computation engine for relations. Efficient tools for BDD exists [11] whereas the author has developed the code for the IDD as an extension of the BDD package. The IDD software was not available in the beginning of the landing gear project. Therefore BDDs were used throughout that project, even though a comparison between IDD and BDD has been performed after the completion of the landing gear project. This is presented in Section 7.5.

7.2.1 Modeling

In the modeling part of the landing gear project, see Section 7.3, the implemented Pascal code of the LGC is compiled to a relational model $M(z, z^+)$. The Pascal code is first parsed to an intermediate code called MPascal which essentially is the same Pascal code written as a *Mathematica* expression. This code is then processed by a compiler, also written in *Mathematica*. The result from the compiler is a relational model $M(z, z^+)$ represented as a BDD, where all relations between input

Commands	**Description**
`ReachableStates[`$M(z,z^{+})$`,`$I(z)$`]`	Returns the set of states reachable from $I(z)$ for the model $M(z,z^{+})$. See (2.80).
`BDDTLEvaluate[`$M(z,z^{+})$`, F]`	Returns the set of states from which the temporal algebra expression F is true. `BDDTLEvaluate` works as the operator Verify in Section 2.3.2.
`BDDSolve[`$R(z)$`]`	Returns all solutions of $R(z) =$ `true` for the variables z.
`BDDCountSolutions[`$R(z)$`]`	Counts all solutions of $R(z) =$ `true`. This is often more interesting than a huge set of solutions.
`BDDRandomSolve[`$R(z)$`]`	Returns one possible solution of $R(z) =$ `true` chose by random.

Table 7.5: Some of the most common commands for analysis.

variables and output variables are stored whereas temporary variables in the code are removed. See Section 7.3.2 for details.

7.2.2 Analysis

Tools for analysis of the polynomial models were developed in *Mathematica* as well. For these tools the efficiency of the underlying computation engine is even more important than for modeling. For multiple step analysis we have to do fix point computations, i.e., to iterate until the answer remains the same between two iterations, and it is essential to reduce data complexity in these iterations. In our case this is done by the BDD package that always represents expressions as simple as possible with respect to the variable order chosen.

To give a taste of how to use the analysis tools Table 7.5 shows the most important commands used in the project.

The commands `BDDSolve` and `BDDCountSolutions` searches recursively through the BDD (or IDD) graph starting at the top down to the constant 1. `BDDRandomSolve` chooses one possible path down to the constant 1.

7.3 Modeling Based on the Controller Implementation

This section 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 a relational model.

7.3.1 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 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 7.6. 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.

7.3.2 Translating Pascal to Relations

The translation from Pascal to a relational model 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 [100] and we will therefore begin with an introduction to this

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

Table 7.6: Allowed Pascal primitives. $\mathbb{I}$ and $\mathbb{B}$ stands for integer and Boolean respectively.

formalism. We will then describe the parsing and compilation of the landing gear controller code in some detail and comment on the resulting relational model.

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 [100]. A related formalism is *natural semantics* which is described in e.g., [71]. 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 relational model representing the code.

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

$$\sigma = \{x_i \mapsto \eta_i\}, \quad i = 0, \dots n \tag{7.1}$$

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 7.1 Binding Environment
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 7.1 Binding Environment
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 expressions, it will be useful to define the logical operations $\{\wedge, \vee, \neg\}$ on σ (see Section 2.1.3 for a relational interpretation of the logical connectives).

Definition 7.2 BE Operators
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 7.2 BE Operators
Let $\sigma_1 = \{x \mapsto a, y \mapsto b\}$, $\sigma_2 = \{x \mapsto c, y \mapsto d\}$ and $g \in \mathbb{B}$. Then

$$\begin{aligned} \sigma_1 \wedge \sigma_2 &= \{x \mapsto a \wedge c, y \mapsto b \wedge d\} \\ \sigma_1 \vee \sigma_2 &= \{x \mapsto a \vee c, y \mapsto b \vee d\} \\ \neg\sigma_1 &= \{x \mapsto \neg a, y \mapsto \neg b\} \\ \sigma_1 \wedge g &= \{x \mapsto a \wedge g, y \mapsto b \wedge g\} \end{aligned} \tag{7.2}$$

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

Definition 7.3 BE Intersection
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 7.3 BE Intersection
Let $\sigma_1 = \{x \mapsto a, y \mapsto b, z \mapsto c\}$ and $\sigma_2 = \{x \mapsto d, y \mapsto e\}$. Then

$$\cap_{\sigma_2}\sigma_1 = \{x \mapsto a, y \mapsto b\}. \tag{7.3}$$

Let a sequence of code primitives be denoted L. We restrict L to contain the primitives from Table 7.6 together with the empty string ϵ. With L and σ as basic elements we can now make the following definitions.

Definition 7.4 Configuration
A *configuration* θ is a tuple (L, σ). □

Definition 7.5 Transition
A *transition* $\vdash$ is a mapping from a configuration $\theta = (\mathsf{L}, \sigma)$ to a configuration θ' or to a binding environment σ',

$$\theta \vdash \theta' \quad \text{or} \quad \theta \vdash \sigma'. \tag{7.4}$$

□

Definition 7.6 Transition Relation
A *transition relation* τ is a rule of the form

$$\tau = \frac{\mathcal{C}_1 \ \dots \ \mathcal{C}_n}{\theta \vdash \theta'} \tag{7.5}$$

where the numerator is a set of conditions which are premises for the denominator. The interpretation of the notation in (7.5) is that θ' can be derived from θ if all conditions $\mathcal{C}_i$ $(1 \leq i \leq n)$ hold. □

Remark: Note that names *transition* and *transition relation* above are not to be confused by transitions of a DEDS.

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 7.4 Conditions

(i) The condition

$$\mathcal{C}_1 = \quad \theta \vdash \theta' \tag{7.6}$$

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

(ii) The condition

$$\mathcal{C}_2 = \quad N \in \mathbb{Z} \tag{7.7}$$

means that the value of N must be an integer.

(iii) The condition

$$\mathcal{C}_3 = \quad t = \texttt{MB} \tag{7.8}$$

means that t must equal the symbol `MB`.

Remark:

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

$$L \vdash L' \tag{7.9}$$

to simplify the notation.

(ii) Unconditional transition relations will be written as

$$\tau = \quad \theta \vdash \theta'. \tag{7.10}$$

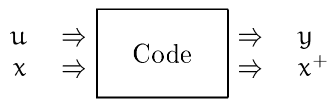


Figure 7.4: Signal interaction with the code.

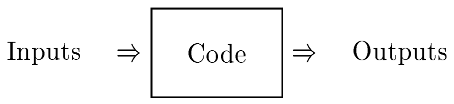


Figure 7.5: Input-output view of the code.

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 (2.37) we must determine which variables correspond to system state, next state, input and output. The equations in (2.37) can be represented by a block diagram as in Figure 7.4, 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 values to output variables depending on input variables, see Figure 7.5. If we compare the figures 7.4 and 7.5, we find that the state variables are equal to the variables in the set $\text{Inputs} \cap \text{Outputs}$. 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 7.7.

Temporary variables which are neither input nor output variables can be omitted in a model like the one in Figure 7.4, since we are only interested in the output behavior of the system. Still, the compiler must use the temporary variables in the code to compute the relations between input and output variables.

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 7.7: State variables in the code.

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 7.3.1.

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

Example 7.5 Parsing

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 (7.18), (7.22), (7.25) and (7.31), defined below.

We are now ready to define the parser for the keywords of Table 7.6. In the parser we will not need the binding environment, which means that all transitions will be of the form (7.9). The parser consists of the following transition rules:

Constants

$$\mathtt{TRUE} \vdash \mathtt{MTrue}[] \tag{7.11}$$

$$\mathtt{FALSE} \vdash \mathtt{MFalse}[] \tag{7.12}$$

$$\frac{\mathsf{N} \in \mathbb{Z}}{\mathsf{N} \vdash \mathsf{N}} \tag{7.13}$$

Arithmetic expressions

+

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 + e_2 \vdash \mathtt{MPlus}[e_1', e_2']} \tag{7.14}$$

-

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 - e_2 \vdash \mathtt{MMinus}[e_1', e_2']} \tag{7.15}$$

Relational expressions

>

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 > e_2 \vdash \mathtt{MGreaterThan}[e_1', e_2']} \tag{7.16}$$

<

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 < e_2 \vdash \mathtt{MLessThan}[e_1', e_2']} \tag{7.17}$$

=

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 = e_2 \vdash \mathtt{MEqual}[e_1', e_2']} \tag{7.18}$$

NOT

$$\frac{e_1 \vdash e_1'}{\mathtt{NOT}\, e_1 \vdash \mathtt{MNot}[e_1']} \tag{7.19}$$

AND

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1 \,\mathtt{AND}\, e_2 \vdash \mathtt{MAnd}[e_1', e_2']} \tag{7.20}$$

`OR`

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{e_1\ \texttt{OR}\ e_2\ \vdash\ \texttt{MOr}[e_1', e_2']} \tag{7.21}$$

Control primitives

`IF THEN ELSE`

$$\frac{c \vdash c' \quad s_1 \vdash s_1' \quad s_2 \vdash s_2'}{\texttt{IF}\ c\ \texttt{THEN}\ s_1 \texttt{ELSE}\ s_2\ \vdash\ \texttt{MIfThen}[c', s_1', s_2']} \tag{7.22}$$

$$\frac{c \vdash c' \quad s_1 \vdash s_1'}{\texttt{IF}\ c\ \texttt{THEN}\ s_1\ \vdash\ \texttt{MIfThen}[c', s_1', \epsilon]} \tag{7.23}$$

`CASE OF`

$$\frac{e \vdash e' \quad k_1 \vdash k_1' \ \ldots\ k_n \vdash k_n' \quad s_1 \vdash s_1' \ \ldots\ s_n \vdash s_n'}{\texttt{CASE}\ e\ \texttt{OF}\ k_1 : s_1\ ;\ \ldots\ ; k_n : s_n\ \texttt{END}\ \vdash\ L} \tag{7.24}$$

where

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

`BEGIN` s_1`;` s_2 `END`

$$\frac{s \vdash s'}{\texttt{BEGIN}\ s\ \texttt{END}\ \vdash\ \texttt{MBeginEnd}[s']} \tag{7.25}$$

$$\frac{s_1 \vdash s_1' \quad s_2 \vdash s_2'}{s_1\ ;\ s_2\ \vdash\ s_1'\ ,\ s_2'} \tag{7.26}$$

Procedures and functions

Declaration:

$$\frac{\mathit{args} \vdash \mathit{args}' \quad s \vdash s'}{\texttt{PROCEDURE}\ \mathit{id}(\mathit{args})\ s\ \vdash\ \texttt{MProcedure}[\mathit{id}, \{\mathit{args}'\}, s']} \tag{7.27}$$

$$\frac{\mathit{args} \vdash \mathit{args}' \quad s \vdash s' \quad e \vdash e'}{\texttt{FUNCTION}\ \mathit{id}(\mathit{args}):t\ \texttt{BEGIN}\ s;\ \mathit{id}\texttt{:=}e\ \texttt{END}\ \vdash\ L} \tag{7.28}$$

where

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

Arguments of procedures and functions:

$$\begin{aligned} &v_1, \ldots, v_k : t_1 ; v_{k+1}, \ldots, v_n : t_2 ; \ldots \vdash \\ &\qquad v_1, \ldots, v_k, v_{k+1}, \ldots, v_n \end{aligned} \tag{7.29}$$

Procedure and function call:

$$\frac{e_1 \vdash e_1' \ \dots \ e_n \vdash e_n'}{\mathtt{id}(e_1, \dots, e_n) \ \vdash \ \mathtt{id}[e_1', \dots, e_n']} \tag{7.30}$$

Miscellaneous code primitives

`:=`

$$\frac{e \vdash e'}{v \mathbin{:=} e \ \vdash \ \mathtt{MAssign}[v, e']} \tag{7.31}$$

`VAR`

$$\frac{t_1 \vdash t_1' \quad t_2 \vdash t_2'}{\mathtt{VAR}\, v_1, \dots, v_k : t_1 ; v_{k+1}, \dots, v_n : t_2 ; \dots \ \vdash \ L} \tag{7.32}$$

where

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

$$\mathtt{BOOLEAN} \ \vdash \ \mathtt{MB} \tag{7.33}$$

$$\mathtt{INTEGER} \ \vdash \ \mathtt{MI} \tag{7.34}$$

`PROGRAM`

$$\frac{s \vdash s'}{\mathtt{PROGRAM}\, s \ \vdash \mathtt{MProgram}[s']} \tag{7.35}$$

By these transition rules we have defined the parsing step of the compilation of the restricted Pascal language defined in Table 7.6. To separate input and output variables in the code (see Section 7.3.2), the keywords `MInput` and `MOutput` are added to the MPascal language. These keywords are used with the same syntax as for `MVar`.

Input variables:

$$\frac{t_1 \vdash t_1' \quad t_2 \vdash t_2'}{\mathtt{VAR}\, u_1, \dots, u_k : t_1 ; u_{k+1}, \dots, u_n : t_2 ; \dots \ \vdash \ L} \tag{7.36}$$

where

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

Output variables:

$$\frac{t_1 \vdash t_1' \quad t_2 \vdash t_2'}{\mathtt{VAR}\, y_1, \dots, y_k : t_1 ; y_{k+1}, \dots, y_n : t_2 ; \dots \ \vdash \ L} \tag{7.37}$$

where

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

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 7.3.2.

Compiling MPascal to a Relational Model

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

$$\mathcal{Y} = [\tilde{y}_1, \dots \tilde{y}_{p+n}] = [y_1, \dots, y_p, x_1^+, \dots, x_n^+] \tag{7.38}$$

are affected by the input variables[2]

$$\mathcal{U} = [\tilde{u}_1, \dots, \tilde{u}_{m+n}] = [u_1, \dots, u_m, x_1, \dots, x_n]. \tag{7.39}$$

This information, which corresponds to a function $f_{\tilde{y}_i} : \mathcal{U} \rightarrow \tilde{y}_i$, 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 expressions 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 7.7 Integer Bit Representation
The vector space of *bit-represented integers*, denoted $\bar{\mathbb{I}}_n$, is

$$\bar{\mathbb{I}}_n = \{\{e_{n-1}, \dots, e_0\} | e_i \in \mathbb{B}[Z], \quad 0 \le i < n-1\} \tag{7.40}$$

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 $2^n - 1$. □

Consider the following example

[1]This is essentially the same as traversing a corresponding control flow graph, see e.g., [6].

[2]This corresponds to a data flow analysis for each output variable, see e.g., [6].

Example 7.6 Integer Bit Vector
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 $\overline{\mathbb{I}}_4$:

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

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 $\overline{\mathbb{I}}_n$.

Definition 7.8 Bit Vector Operations

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

(ii) Let $\mathsf{Int} : \mathbb{B}^n \to \mathbb{Z}_{2^n}$ 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 $\mathsf{BAdd} : \overline{\mathbb{I}}_n \times \overline{\mathbb{I}}_n \to \overline{\mathbb{I}}_n$ be a function that computes the sum of two bit-represented integers using Boolean operations on the Boolean expressions in the vectors.

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

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

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

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

$$\{x_{n-1}, \dots, x_0\} \overset{s}{\leftrightarrow} \{y_{n-1}, \dots, y_0\} = \bigwedge_{i=0}^{n-1} x_i \leftrightarrow y_i \tag{7.41}$$

□

The Boolean relations for adding two bit-represented integers can be computed recursively using a *full adder* [34], see the example below.

Example 7.7 Addition on Bit Vectors
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

$$\begin{aligned}\mathsf{BAdd}(x,y) = \{&(y_0 \wedge ((y_1 \wedge ((x_0 \wedge x_1) \vee ((\neg x_0) \wedge (\neg x_1)))) \vee \\ &((\neg y_1) \wedge ((x_0 \wedge (\neg x_1)) \vee ((\neg x_0) \wedge x_1))))) \vee \\ &((\neg y_0) \wedge ((y_1 \wedge (\neg x_1)) \vee ((\neg y_1) \wedge x_1))), \\ &(y_0 \wedge (\neg x_0)) \vee ((\neg y_0) \wedge x_0)\}.\end{aligned} \tag{7.42}$$

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 $\mathsf{BAdd}(x,y) = \{\mathtt{true}, \mathtt{true}\}$.

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

As mentioned in Section 7.3.2 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 7.8 Compilation
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 (7.57), (7.58) and (7.60).

All variables are Boolean and $f^-_{\tilde{y}_2}$ defines the value of the variable `y2` before this piece of code is executed. If `y2` has not been assigned a value before, $f^-_{\tilde{y}_2}$ 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

$$R(z,z^+) = (y1 \leftrightarrow (c \wedge q \vee d \wedge \neg q)) \wedge (y2 \leftrightarrow (\bot \wedge q \vee e \wedge \neg q)). \tag{7.43}$$

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

Evaluation of expressions

Constants:

$$\texttt{MTrue[]} \vdash \texttt{true} \tag{7.44}$$

$$\texttt{MFalse[]} \vdash \texttt{false} \tag{7.45}$$

$$\frac{N \in \mathbb{Z}}{N \vdash \mathsf{Bit}(N)} \tag{7.46}$$

Arithmetic expressions:

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MPlus}[e_1, e_2] \vdash \mathsf{BAdd}(e_1', e_2')} \tag{7.47}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MMinus}[e_1, e_2] \vdash \mathsf{BSub}(e_1', e_2')} \tag{7.48}$$

Relational expressions:

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MGreaterThan}[e_1, e_2] \vdash \mathsf{BGreaterThan}(e_1', e_2')} \tag{7.49}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MLessThan}[e_1, e_2] \vdash \mathsf{BLessThan}(e_1', e_2')} \tag{7.50}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2' \quad (e_1', e_2' \in \bar{\mathbb{I}}_4)}{\texttt{MEqual}[e_1, e_2] \vdash \mathsf{BEqual}(e_1', e_2')} \tag{7.51}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2' \quad (e_1', e_2' \in \mathbb{B}[X])}{\texttt{MEqual}[e_1, e_2] \vdash e_1' \leftrightarrow e_2'} \tag{7.52}$$

$$\frac{e \vdash e'}{\texttt{MNot}[e] \vdash \neg e'} \tag{7.53}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MAnd}[e_1, e_2] \vdash e_1' \wedge e_2'} \tag{7.54}$$

$$\frac{e_1 \vdash e_1' \quad e_2 \vdash e_2'}{\texttt{MOr}[e_1, e_2] \vdash e_1' \vee e_2'} \tag{7.55}$$

Compiling statements

Sequences of statements:

$$\frac{(s_1, \sigma_1) \vdash \sigma_2}{(\ s_1, s_2\ , \sigma_1)\ \vdash\ (s_2, \sigma_2)} \tag{7.56}$$

Assignment:

$$\frac{(e, \sigma) \vdash (e', \sigma)}{(\texttt{MAssign}[v, e], \sigma)\ \vdash\ \sigma[v \mapsto e']} \tag{7.57}$$

Control statements:

$$\frac{(c, \sigma) \vdash (c', \sigma) \quad (s_1, \sigma) \vdash \sigma_1 \quad (s_2, \sigma) \vdash \sigma_2}{(\texttt{MIfThen}[c, s_1, s_2], \sigma)\ \vdash\ \sigma_1 \wedge c' \vee \sigma_2 \wedge \neg c'} \tag{7.58}$$

$$\frac{(e, \sigma) \vdash (e', \sigma) \quad k_1 \vdash k_1' \ \ldots\ k_n \vdash k_n' \quad (s_1, \sigma) \vdash \sigma_1\ \ldots\ (s_n, \sigma) \vdash \sigma_n}{(\texttt{MCaseOf}[e, k_1, s_1, \ldots, k_n, s_n], \sigma)\ \vdash\ \theta} \tag{7.59}$$

where

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

$$\frac{(s_1, \ldots, s_n, \sigma_1) \vdash \sigma_2}{(\texttt{MBeginEnd}[s_1, \ldots, s_n], \sigma_1)\ \vdash\ \sigma_2} \tag{7.60}$$

Variable declarations[3]

Temporary variables:

$$\frac{t = \texttt{MB}}{(\texttt{MVar}[\{t, v_1, \ldots, v_n\}, \mathit{vardecl}], \sigma)\ \vdash\ \theta} \tag{7.61}$$

where

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

$$\frac{t = \texttt{MI}}{(\texttt{MVar}[\{t, v_1, \ldots, v_n\}, \mathit{vardecl}], \sigma)\ \vdash\ \theta} \tag{7.62}$$

where

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

[3]Note that these rules are defined recursively.

Input variables:

$$\frac{t = \mathtt{MB}}{(\mathtt{MInput}[\{t, u_1, \dots, u_n\}, \mathit{vardecl}], \sigma) \ \vdash \ \theta} \tag{7.63}$$

where

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

$$\frac{t = \mathtt{MI}}{(\mathtt{MInput}[\{t, u_1, \dots, u_n\}, \mathit{vardecl}], \sigma) \ \vdash \ \theta} \tag{7.64}$$

where

$$\theta = (\mathtt{MInput}[\mathit{vardecl}], \sigma[u_1 \mapsto \{u_1[3], u_1[2], u_1[1], u_1[0]\}, \dots, \\ u_n \mapsto \{u_n[3], u_n[2], u_n[1], u_n[0]\}])$$

Output variables:
The transitions for the declaration of output variables (`MOutput`) are similar to the transitions for `MVar`, see (7.61) and (7.62).

Procedures and functions

Declaration:

$$(\mathtt{MProcedure}[\mathit{id}, \{\mathit{args}\}, s], \sigma) \ \vdash \\ \sigma[\mathsf{Arg}(\mathit{id}) \mapsto \mathit{args}, \mathsf{Code}(\mathit{id}) \mapsto s] \tag{7.65}$$

$$(\mathtt{MFunction}[\mathit{id}, \{\mathit{args}\}, s], \sigma) \ \vdash \\ \sigma[\mathsf{Arg}(\mathit{id}) \mapsto \mathit{args}, \mathsf{Code}(\mathit{id}) \mapsto s] \tag{7.66}$$

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

Execution:

$$\frac{E \quad ([\![\sigma_1(\mathsf{Code}(\mathit{id}))]\!], \sigma_1[[\![\sigma_1(\mathsf{Arg}(\mathit{id}))]\!] \mapsto \{e'_1, \dots, e'_n\}]) \vdash (e_r, \sigma_2)}{(\mathit{id}[e_1, \dots, e_n], \sigma_1) \ \vdash \ (e_r, \cap_{\sigma_1} \sigma_2)} \tag{7.67}$$

where

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

$$\frac{(e, \sigma) \vdash (e', \sigma)}{(\mathtt{MReturn}[e], \sigma) \ \vdash \ (e', \sigma)} \tag{7.68}$$

The `PROGRAM` keyword

$$\frac{s \vdash \epsilon}{(\mathtt{MProgram}[s], \emptyset) \ \vdash \ (\epsilon, \sigma_{\mathit{final}})} \tag{7.69}$$

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 relational model $C(z, z^+)$. For this application we let integer variables take values in the range $\{0, \dots, 15\}$, 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 5.7). Then we can compute the final BDD relation as

$$\tilde{C}(z, z^+) = C(z, z^+) \wedge \Lambda(u) \wedge \Lambda(u^+). \tag{7.70}$$

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

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 relation model whose behavior is in exact correspondence with the original code. In the next section we will use this relational model to analyze the landing gear controller.

7.4 Analysis of the Landing Gear Controller

In this section 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 static analysis and dynamic analysis of the system.

7.4.1 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 $\Gamma^+_\infty(x)$ denotes the reachable states, the reduced model[4] is computed as

$$C_{red}(z, z^+) = \tilde{C}(z, z^+) \wedge \Gamma^+_\infty(x) \wedge \Gamma^+_\infty(x^+). \tag{7.71}$$

In this case, the complexity of the model is reduced considerably, from 193223 to 136841 vertices, see Section 7.5.

Using the reduced model it is possible to draw conclusions about the behavior of the system.

7.4.2 Static Analysis

By static analysis (see Section 2.3.1) 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 static properties of the landing gear controller with the use of NPCircuit [84], 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 the tables 7.1, 7.2, 7.3 and 7.4 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.

$$\neg m_1$$

[4]The size will decrease in most cases.

3. All feedback switches on gears and doors have two contacts. These have the same value.
$$s_j^1 \leftrightarrow s_j^2, \quad j = 1, \ldots, 15$$
4. The switches on the doors do 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 do 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.
$$(s_3^i \rightarrow s_2^i) \wedge (s_8^i \rightarrow s_7^i) \wedge (s_{13}^i \rightarrow s_{12}^i), \quad i = 1, 2$$
7. Power supply is working.
$$\neg m_5$$
8. The variables m_3 and m_4 have values according to their definitions.
$$(m_3 = 1) \leftrightarrow (s_3^1 \wedge s_8^1 \wedge s_{13}^1)$$
$$(m_3 = 0) \rightarrow (\neg s_3^1 \wedge s_8^1 \wedge s_{13}^1)$$
$$(m_4 = 0) \rightarrow (m_3 = 1)$$

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 $P_1(u)$.

Static 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
$$(C_{red}(z, z^+) \wedge P_1(u) \wedge P_1(u^+) \wedge (s_4^1 \vee s_4^2) \wedge (s_1^1 \vee s_1^2) \wedge$$
$$(x_7 = 0 \vee x_7 = 1 \vee x_7 = 2)) \rightarrow (a_3 \rightarrow a_5). \quad (7.72)$$

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.

Static 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

$$(C_{red}(z, z^+) \wedge P_1(u) \wedge P_1(u^+) \wedge (s_4^1 \vee s_4^2) \wedge (s_2^1 \vee s_2^2) \wedge \\ \neg(x_8 = 4) \wedge \neg(x_8 = 5) \wedge \neg(x_8 = 7)) \rightarrow \neg a_2. \quad (7.73)$$

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.

7.4.3 Dynamic Analysis on Closed Loop Landing Gear System

With dynamic analysis (see Section 2.3.2) 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 static 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.

Up to now the analysis has been performed on the controller, i.e., the open loop system. We will now perform dynamic analysis on both the controller model and a model of the plant connected in a closed loop. Temporal logic will be used both for specification and modeling of the plant.

The physical plant of the landing gear system is the landing gear itself with inputs controlling the hydraulic actuators, and outputs connected to several switches.

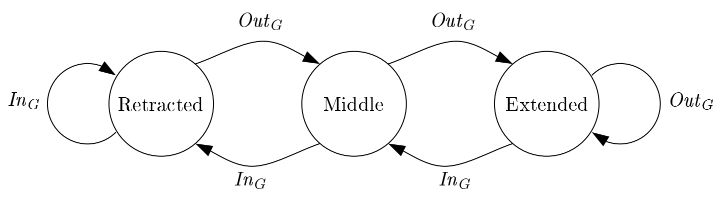


Figure 7.6: Plant model with fixed depth.

Gear state	SW_{GearR}	SW_{GearE}
Retracted	`true`	`false`
Middle	`false`	`false`
Extended	`false`	`true`

Table 7.8: Output mapping from gear plant model.

As mentioned in Section 7.1.2 the interface between the plant and the LGC is discrete, which means that we can make a discrete model with the same behavior as the continuous system seen from the LGC.

The LGC can only determine the state of the gears to be in three different regions: retracted, middle and extended. The same is also true for the doors which have the regions: closed, middle and open. All the gears and the doors are operated in parallel.

The simplest model with this behavior is a double 3-state automaton. The automaton for the gears will look like Figure 7.6 where the input symbol Out_G and In_G are place holders for the LGC actuator outputs. The output from the plant are signals from the switches indicating the positions of gears and doors. For the gear model the outputs are defined as in Table 7.8.

For the doors we get a similar model.

The plant model for both gears and doors are written in MPascal and then compiled into a relational model $P_2(z, z^+)$. The resulting model has 8 Boolean state variables. Each automaton has a single integer state variable, and integers are represented by 4 Boolean variables in this case. This makes it easy to enlarge the model later. There are 12 output variables and 5 input variables.

The structure of this model is

$$P_2(z_p, z_p^+) := (x_p^+ = f_p(x_p, u_p)) \wedge (y_p = g_p(x_p)).$$

The outputs y_p of the plant do not directly depend on the inputs u_p. This is important to ensure that we avoid an algebraic loop when connecting the plant and the LGC. The structure of the LGC model is

$$C_{red}(z_c, z_c^+) := (x_c^+ = f_c(x_c, u_c)) \wedge (y_c = g(x_c, u_c))$$

where y_c depend directly on u_c.

The input/output variables in the plant model $\mathsf{P}_2(z_p, z_p^+)$ are the same as the corresponding output/input variables in the LGC model. Therefore we can easily compute the closed loop system $G_c(z, z^+)$ as

$$\mathsf{G}_1(z, z^+) := \mathsf{C}_{red}(z_c, z_c^+) \wedge \mathsf{P}_2(z_p, z_p^+) \tag{7.74}$$

where $z = z_c \cup z_p$.

Before analyzing the closed loop model we need a specification or a test case from which we can formulate a temporal logic expression. For the landing gear system the most critical maneuver is extension, i.e., the landing gears should always reach the extended state when the pilot pushes the gear extension button. More formally we say:

The gears should always reach the extended state Gear(ext) in finite time, when the pilot command is extension Pilot(ext).

This specification can be directly translated to the temporal logic expression $\mathsf{AG}[\mathit{Pilot}(\mathit{ext}) \to \mathit{Gear}(\mathit{ext})]$. By verifying this expression we will obtain all states from which the specification is true. It is often more convenient to search for the states not fulfilling the specification. Therefore we define the first temporal expression in the opposite way as

$$\mathsf{F}_1(z) := \mathsf{EG}[\neg(\mathit{Pilot}(\mathit{ext}) \to \mathit{Gear}(\mathit{ext}))]. \tag{7.75}$$

We verify this statement by

$$\mathsf{S}_1(z) := \texttt{BDDTLEvaluate[}\mathsf{G}_1(z, z^+)\texttt{,}\mathsf{F}_1(z)\texttt{]}$$

where we use one of the commands in Table 7.5. We obtain a relation $\mathsf{S}_1(z)$ including 82 variables and with 5 different solutions. By analyzing these solutions further, using knowledge from the SAAB company, we found that for all 5 solutions a time out condition *TimeOut* was set. This means that the analysis had found the cases where extension time had exceeded its limit and the maneuver was stopped. Then the pilot has to restart the extension by choosing retraction first and then extension. To avoid this trap we reformulate the specification as follows

Having the pilot command retract Pilot(ret) and in the next state having Pilot(ext) and not TimeOut the gear should always reach the state Gear(ext) in finite time.

As before we formulate the temporal expression for finding the errors.

$$\begin{aligned} \mathsf{F}_2(z) := &\mathit{Pilot}(\mathit{ret}) \wedge \mathsf{EX}[\mathsf{EG}[\\ &\neg((\mathit{Pilot}(\mathit{ext}) \wedge \neg \mathit{TimeOut}) \to \\ &\mathit{Gear}(\mathit{ext}))]] \end{aligned} \tag{7.76}$$

This is verified by

$$\mathsf{S}_2(z) := \texttt{BDDTLEvaluate[}\mathsf{G}_1(z, z^+)\texttt{,}\mathsf{F}_2(z)\texttt{]}$$

which returns $\mathsf{S}_2(z) \equiv \texttt{false}$, i.e., we have proved that the gears will always reach the extended state provided the conditions above. Note that we do not specify an initial state for the analysis. This means that this verification holds for all behaviors of the system preceding this extension maneuver. By this we have proved a *liveness* property, i.e., the system cannot be trapped in a dead-lock situation. This proof is only valid with respect to the simple model of the plant.

For the plant used above we have distinguished between three different regions for the gears: retracted, middle and extended. We used a three state automaton as a model for the plant behavior. Doing this we have made an important assumption about the system. Since the plant model works synchronously to the LGC it might be relevant to consider how many iterations the LGC needs during a normal plant maneuver. For the plant model P_2 the LGC only needs 2 iterations to reach the end state of the plant. The real implemented LGC iterates several times per second, which means that a reasonable assumption is that it takes more than 20 iterations to fulfill a maneuver. To get a more realistic plant model we can build a model with several middle states. If we let the number of middle states be larger than the maximal depth of the LGC and if the dynamics of the LGC always reach a fixed point, we know that this fixed point will be reached during the middle states. In our case the maximal depth of the LGC is 5. By choosing a plant with 8 states, of which 6 are middle states we have an appropriate plant model.

However there are some disadvantages with this method. First of all the complexity increases since we add more states to the system. It turns out that by adding more states to the plant we get harder complexity problems doing verification compared to increasing the complexity of the temporal logic statements. Second, we can only verify the system for a fixed depth (fixed number of middle states) of the plant. We cannot in one verification test plants with several different depths, without discrepancies. The last drawback is that we at this stage do not know if the LGC reaches a fixed point. This feature can be examined by our analysis tools, but it is not necessary if we use the power of temporal logic instead.

The most general plant in the sense of capturing all possible depths would be the nondeterministic automaton which remains in the middle state an arbitrary number of iterations and then goes to the end state. But if the plant should be used for liveness verification we have to have a model that reaches an end state after a finite number of steps. We do this by introducing condition signals Man_G (*Man*euver) for the gears and Man_D for the doors. See Figure 7.7 for the gear model. The model for the doors is constructed in the same way. The gear and door models are combined into a relational model $\mathsf{P}_3(z_\mathrm{p}, z_\mathrm{p}^+)$ which is used for the closed loop model

$$\mathsf{G}_2(z, z^+) := \mathsf{C}_{red}(z_\mathrm{c}, z_\mathrm{c}^+) \wedge \mathsf{P}_3(z_\mathrm{p}, z_\mathrm{p}^+). \tag{7.77}$$

We adjust the definition of the temporal expression $\mathsf{F}_2(z)$ for the plant model $\mathsf{P}_3(z_\mathrm{p}, z_\mathrm{p}^+)$ such that the model will reach the extended state after a finite time, i.e., Man_G and Man_D are true only in a finite time. The result is:

$$\begin{aligned} \mathsf{F}_3(z) :=& \mathit{Pilot}(\mathit{ret}) \wedge \\ & \mathsf{EX}[\mathsf{EU}[\tilde{\mathsf{F}}_2(z), \mathsf{EG}[\neg Man_G \wedge \neg Man_D \wedge \tilde{\mathsf{F}}_2(z)]]] \end{aligned} \tag{7.78}$$

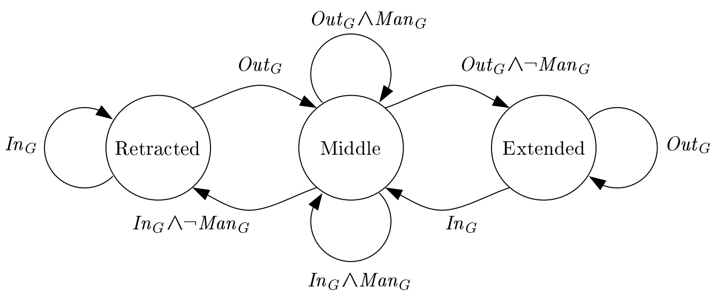


Figure 7.7: Plant for arbitrary finite depth. The Man_G-signal is `true` as long as the plant should be (maneuvering) in the middle state.

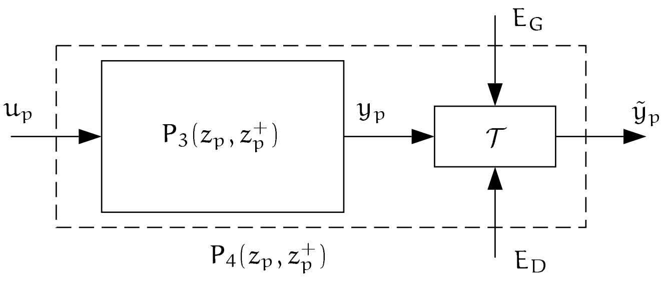


Figure 7.8: Plant model with sensor failures.

where $\tilde{F}_2(z) = \neg((\mathit{Pilot}(\mathit{ext}) \wedge \neg \mathit{TimeOut}) \rightarrow \mathit{Gear}(\mathit{ext}))$. In this way we have captured all plants with a finite depth

We verify this statement by

$$S_3(z) := \texttt{BDDTLEvaluate[}G_2(z, z^+)\texttt{,}F_3(z)\texttt{]}.$$

The relation $S_3(z)$ is identical to `false` which shows that we will always reach the extended state for all plants with finite depth, i.e., the specification F_3 holds independently of the number of iterations for the LGC during an extension maneuver.

This analysis was possible to perform without increasing the complexity of the model. Instead, by using temporal expressions to build more general models we can analyze more complex behavior.

The plant model $P_3(z_p, z_p^+)$ used above does not account for errors or disturbances on the signals. But there is a need to take some possible failures in the switches into account. Therefore another plant model is created from the first one, see Figure 7.8, where the outputs are filtered and disturbed by two signals E_G and E_D. The disturbance mapping $\mathcal{T}$ models the possibility of short circuits in the switches measuring the state of the plant. Therefore when $E_G =$ `true` all gear switches are `true`, i.e., the gears seams to be both retracted and extended simultaneously. The disturbance E_D works in the same way for the doors. Note that the reason for having double switches at each sensor, see Section 7.1, was to be able to handle these type of failures.

We will refer to this plant model as the relational model $P_4(z_p, z_p^+)$. The closed loop system is now

$$G_3(z, z^+) := C_{red}(z_c, z_c^+) \wedge P_4(z_p, z_p^+) \tag{7.79}$$

In spite of the disturbance on the plant outputs we want the LGC to fulfill its task as stated in $F_3(z)$, i.e., the extension maneuver should be completed in finite time for $P_4(z_p, z_p^+)$ also. By computing

$$S_4(z) := \texttt{BDDTLEvaluate[}G_3(z, z^+)\texttt{,}F_3(z)\texttt{]}$$

we get the result $S_4(z)$, which is equal to `false`. This shows that we will always reach the extended state for all plants with finite depth and sensor failures.

7.5 Performance Improvements using IDDs

As mentioned in Section 7.2 the software for this project was originally designed for BDDs. The experience of the BDDs in the project is that the representation of integers are rather cumbersome. This motivated us to implement the IDDs from Chapter 6 by rewriting the BDD package into a IDD package.

We have run all the computations of the landing gear project both with BDDs and IDDs to be able to compare performance. The main difference of course is that when we use the IDD we represent all integer variables in the Pascal code with only one IDD variable. Moreover, we do not need to compute any constraints (λ-polynomial) for every integer since the IDD lets us specify the range for each variable separately. The number of variables for the relations are presented in Table 7.9. The same variable ordering is used both for the BDD and IDD case.

The comparison is done by a script in *Mathematica* that performs all the steps from the compilation of the Pascal code to the dynamic verification at the end. During this process we have logged the time for some check points of the different computations. These checkpoints are presented in Table 7.10.

The times for these checkpoints are presented in Figure 7.9. The total computation time for BDD is 35 minutes and for IDD 16 minutes. As we se from Figure 7.9, this proportional difference is approximately the same for all computations performed. The most time consuming part, as expected, is the compilation of the LGC. But we have to remark that during this compilation the BDD/IDD package has to rebuild the structure several times, i.e., new memory must be allocated and the unique and cache tables are rebuilt. A second compilation of the LGC would take perhaps 10% less time to compute.

More surprisingly though, is that the composition of the LGC-model and the plant model takes more time than the dynamic analysis where two fixed point iterations are performed. But as a payback for the computation of the closed loop model we get a smaller BDD/IDD as result. See Table 7.9.

Description	**BDD**	**IDD**
Number of variables in LGC relation	105	70
Number of variables corresp. to integers in LGC relation	47	12
Size of LGC relation	193223	127784
Number of variables in reachable state rel.	26	12
Number of variables corresp. to integers in reach. state relation	20	5
Size of reachable state relation	346	93
Size of LGC relation restricted by the reachable state relation	136841	84556
Size of closed loop relation	117934	62825
Memory size of one vertex/node. (32 bits pointer arithmetics)	16 bytes	Booleans, 16 bytes Integers, 72 bytes
Size of LGC relation (in bytes)	3091552	3287968
Size of reachable state relation (in bytes)	5536	4960
Total computation time.	35 min	16 min

Table 7.9: Comparison of size and number of variables for the LGC and reachable state relations represented by BDD and IDD. The same variable ordering is used for both BDD and IDD

We will also present some figures comparing the size (number of vertices) of the LGC relation represented by BDD and IDD respectively. Figure 7.10 shows the *width* of the LGC relation model represented as a BDD. The width of a BDD/IDD graph is the number of vertices for each variable included in the graph. For the top variable (top vertex) we have the width 1, but variable number 61 has 9992 vertices as show in Figure 7.10. In the figure we have also marked the variables corresponding to integer bits by a dot.

Figure 7.10 shows that we have two peaks where the variables have many vertices. The variables for the first peak are integer bit variables whereas the second peak corresponds to Boolean variables in the LGC.

The LGC relation represented by the IDD, with similar variable ordering as for the BDD case above, gives the width presented in Figure 7.11. We notice that the first peak from Figure 7.10 is reduced whereas the second peak remains the same. This indicates that we have reduced the complexity of representing integer valued relations by the use of IDD. Still, as the figure shows, the first peak has not vanished, which indicates that the variable ordering may not be the best. How to find an optimized variable ordering is a hard problem and not within the scope of this thesis.

The width of the relation representing the reachable states of the LGC, is shown

Checkpoints	Description
1	Start of compilation of the LGC.
2	End of compilation of the LGC.
3	Start of computation of reachable states.
4	First step finished
5	End of computation of reachable states.
6	The number of reachable states computed.
7	Start of compilation of plant model.
8	End of compilation of plant model.
9	Closed loop model composed. Start of dynamic analysis of formula (7.78).
10	End of dynamic analysis.

Table 7.10: Checkpoints of the computations for comparing BDDs and IDDs.

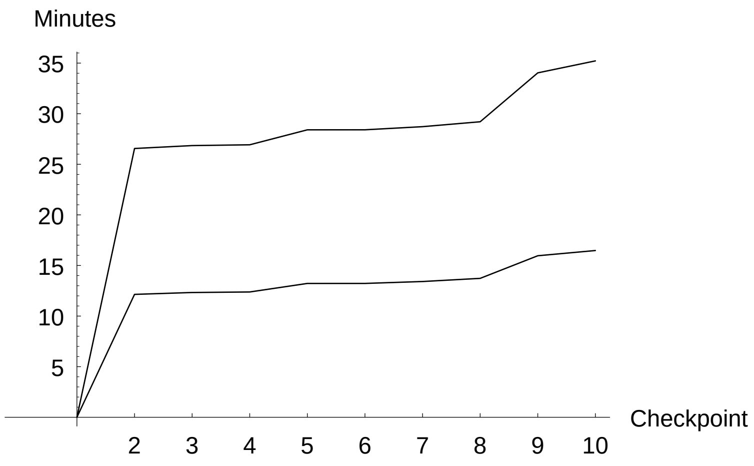


Figure 7.9: Times for the checkpoints in Table 7.10. The upper curve for BDD and the lower for IDD.

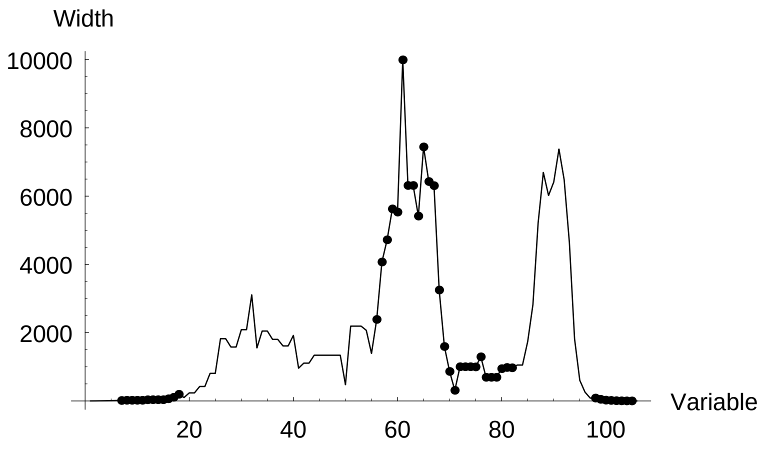


Figure 7.10: The width of the BDD representing the LGC relation. The marked points correspond to integer bit variables.

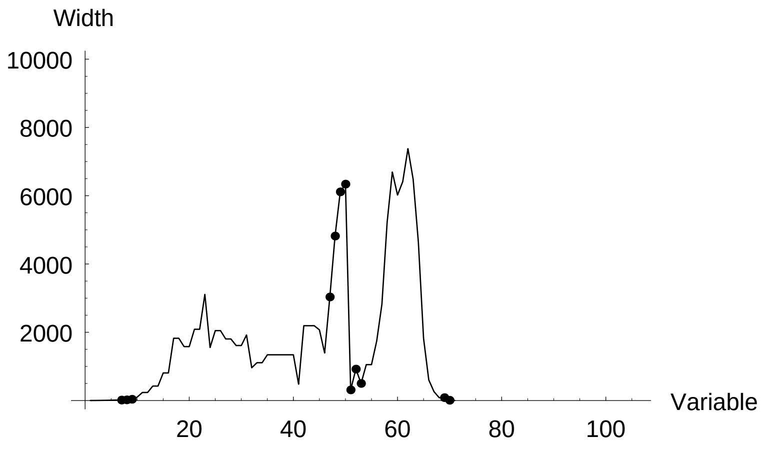
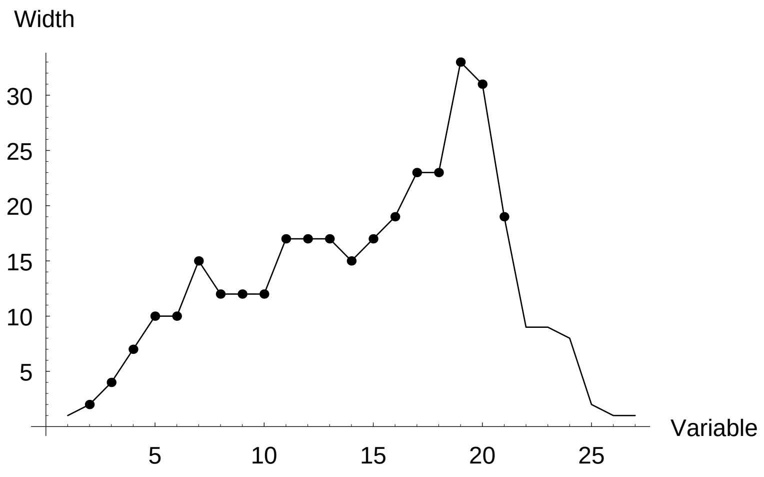


Figure 7.11: The width of the IDD representing the LGC relation. The marked points correspond to integer variables.



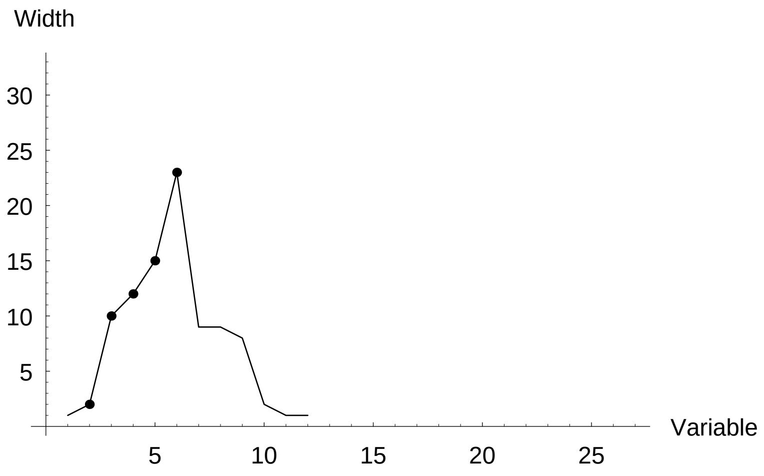


Figure 7.12: The width of the BDD (upper) and IDD (lower) representing the reachable state relation. The marked points correspond to integer bit variables.

in Figure 7.12. In this relation the differences between the BDD and IDD, are not as evident as for the LGC relation. Note though that the number of total vertices is magnitudes less than for the LGC relation in both cases. This means that the computation of the relation for the reachable states is not as complex in this case, and that the BDD/IDD structures take advantage of the sparse reachable state space. Moreover the fraction of variables corresponding to integers are bigger for this relation than for the relation of the LGC model. This gives a greater advantage for the IDD since the number of variables included in the relation is reduced by 50%, which in turn gives that the size of the relation is reduced by 73% according to Table 7.9.

Table 7.9 shows the sizes in bytes of the LGC relation and the reachable state relation. There is no major differences in sizes between the BDD and the IDD representation. The reason for this is that the implementations of BDDs and IDDs use 32 bit (4 bytes) to represent each edge in the diagrams. This is a disadvantage for the IDDs, since each integer vertex of an IDD has 16 children. This gives the total vertex size of 72 bytes. In this project though, we will not need to span 2^{32} bytes of memory and therefore the vertices could be made smaller. This leds to a non standard pointer arithmetics which is not within the scope of this thesis.

This comparison has shown that the use of IDD has increased performance and utilized the integer structure as expected. The fraction of integer variables was 17% in the IDD representation (45% in the BDD), which led to reduced computation time by 54% and the number of vertices reduced by 34% for the LGC relation and by 73% for the reachable state relation.

7.6 Conclusions and Future Work

The goal of this case study was to investigate if our relational framework 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.

7.6.1 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 closed loop process:

$$\mathsf{G_c}(z, z^+) := \mathsf{C}(z, z^+) \wedge \mathsf{P}(z, z^+) \tag{7.80}$$

where $\mathsf{C}(z, z^+)$ is the controller and $\mathsf{P}(z, z^+)$ is the plant.

We can also build a simple model of the function specification $\mathsf{S}(z)$ using temporal logic. Using the closed loop system model $\mathsf{G_c}(z, z^+)$ and the specification $\mathsf{S}(z)$ 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 relational 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 relational format, which is suitable for analysis. This procedure eliminates potential discrepancies between documents and the actual system. Pascal is not suited for descibing DEDS. The reasons for this will be discussed more thoroughly in Section 10.2.
- Analysis has been performed successfully on the different models. Static analysis has given results corresponding to prior knowledge from SAAB. The most promising result is that a dynamic analysis has been performed on the closed loop system by using temporal algebra as a representation of informal specifications.
- A comparison have been made on IDDs and BDDs. This has shown that the use of IDDs gains performance advantages, when integer variables are included in the problems.

7.6.2 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 BDD/IDD 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 dynamic 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.

8

Synthesis of a Tank System

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 [104]. Two good surveys are [111] for the automata-theoretic approach to supervisory control and [65] 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 Section 8.2 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 Section 5, the reader may benefit from beginning with this chapter.

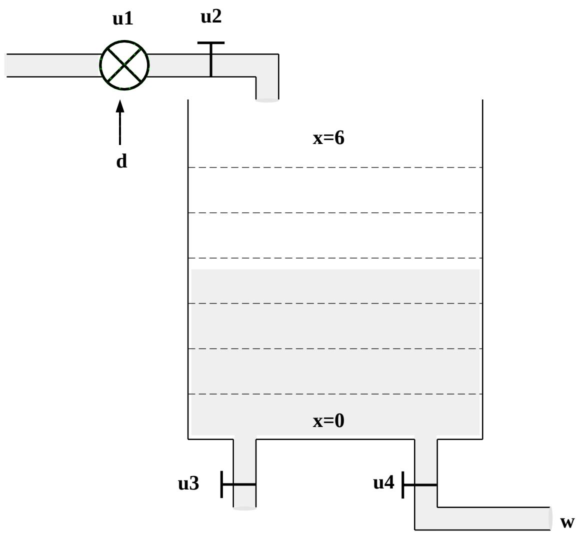


Figure 8.1: The water tank.

8.1 Modeling the Water Tank

The example system chosen for controller design is a water tank (see Figure 8.1) 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.

8.1.1 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 $\phi = 0$

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 8.1: 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 8.2: Variables with corresponding domain.

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, 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 8.1, 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 8.2.

8.1.2 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

$$\begin{aligned} x^+ &= f_1(x, \phi) \\ \phi &= f_2(u_1, u_2, u_3, u_4, w, d) \end{aligned} \tag{8.1}$$

where x^+ denotes the next value of the state. Note that this is an untimed DEDS, 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 8.2 we see that x is the variable with the largest domain ($\{0,\dots,6\}$). Therefore we have to represent all signals by polynomials in $\mathbb{R}_7[Z]$, 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 $\mathbb{R}_7[Z]$ using the following relations (see Section 5.7)

$$\begin{aligned}
\lambda_7^2(u_i) &= u_i^2 - u_i, \quad i = 1,\dots,4 \\
\lambda_7^2(w) &= w^2 - w \\
\lambda_7^3(d) &= d(d-1)(d-2) \\
\lambda_7^3(\phi) &= \phi(\phi-1)(\phi-2)
\end{aligned} \tag{8.2}$$

The set of these λ-polynomials will, according to Definition 5.12, be denoted Λ_7.

When deriving the polynomial f_1 in (8.1) 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 5.3 we can derive the polynomial description of the system. The two polynomials f_1 and f_2 in (8.1) become

$$\begin{aligned}
f_1(x,\phi) = x &+ 3\phi - 3\phi^2 - 3\phi x + 3\phi^2 x + 3\phi x^2 - 3\phi^2 x^2 - \\
&- 3\phi x^3 + 3\phi^2 x^3 + 3\phi x^4 - 3\phi^2 x^4 - 3\phi x^5 + 3\phi^2 x^5 - \\
&- x^6 + \phi x^6
\end{aligned} \tag{8.3}$$

and

$$\begin{aligned}
f_2(u_1,u_2,u_3,u_4,w,d) = 1 &+ 3du_2 - 3d^2u_2 + 2du_1u_2 - d^2u_1u_2 - \\
&- u_3 - u_4w + u_3u_4w - 3du_2u_3u_4w + \\
&+ 3d^2u_2u_3u_4w - 2du_1u_2u_3u_4w + \\
&+ d^2u_1u_2u_3u_4w.
\end{aligned} \tag{8.4}$$

where $f_1, f_2 \in \mathbb{R}_7[z, z^+]$ 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 8.1
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 (8.4) 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 (8.3) we see that $f_1 = 4$. Thus the next state will be $x^+ = 4$.

In (8.3) and (8.4) 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.

8.2 Controller Design

In this chapter we will describe the process of controller design for the water tank modeled in Section 8.1. We begin by formulating the design criteria in terms of polynomials in Section 8.2.1 and then we derive the control laws using successive Gröbner basis computations in Section 8.2.2. Finally we analyze the designed controller in Section 8.2.3.

8.2.1 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

$$p(x) = 1 + x + 6x^2 + x^3 + 6x^4 + x^5 + 5x^6. \tag{8.5}$$

This polynomial can be generated from e.g., a table, using the Lagrange interpolating polynomial described in Section 5.3. 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 5.5.

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

(8.6)

If we represent the weighting function as a polynomial, we have from (5.17):

$$J(x) = 3 + 3x + 6x^2 + x^3 + 2x^4 + 6x^5 + 2x^6 \tag{8.7}$$

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

$$J(x^+) = J(f_1(x, \phi)). \tag{8.8}$$

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

$$\phi = \arg \min_{\phi \in \mathbb{F}_3} J(f_1(x, \phi)). \tag{8.9}$$

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

$$p_1(x, \phi, m) = J(f_1(x, \phi)) + m - J(x). \tag{8.10}$$

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

$$p_1(x, \phi, m) = 0. \tag{8.11}$$

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

$$p_2(\phi, u_1, u_2, u_3, u_4, w, d) = 0 \tag{8.12}$$

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

$$p_2(\phi, u_1, u_2, u_3, u_4, w, d) = f_2(u_1, u_2, u_3, u_4, w, d) - \phi. \tag{8.13}$$

The explicit expressions for the polynomials p_1 and p_2 are

$$\begin{aligned} p_1(x, \phi, m) = {} & -3\phi + 3\phi^2 + m + x + \phi x - 2\phi^2 x - 2x^3 - 2\phi x^3 - \\ & - 3\phi^2 x^3 - 2\phi x^4 + 2\phi^2 x^4 + 2x^5 - 2\phi x^5 \end{aligned} \tag{8.14}$$

$$\begin{aligned} p_2(\phi, u_1, u_2, u_3, u_4, w, d) = {} & 1 - \phi + 3du_2 - 3d^2u_2 + 2du_1u_2 - \\ & - d^2u_1u_2 - u_3 - u_4w + u_3u_4w - \\ & - 3du_2u_3u_4w + 3d^2u_2u_3u_4w - \\ & - 2du_1u_2u_3u_4w + d^2u_1u_2u_3u_4w \end{aligned} \tag{8.15}$$

8.2.2 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

$$\begin{aligned} u_1 &= K_{u_1}(x, w, d) \\ u_2 &= K_{u_2}(x, w, d) \\ u_3 &= K_{u_3}(x, w, d) \\ u_4 &= K_{u_4}(x, w, d) \end{aligned} \tag{8.16}$$

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 5.10), since the choice of ϕ is unambiguous for every x.

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

$$\mathsf{GB}_a = GB_7\left(\{p_1(x, \phi, m), \lambda_7^3(\phi)\}\,|_{m=1}\right) \tag{8.17}$$

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

$$\mathsf{GB} = \{g_1(\phi, x),\ g_2(x), \ldots, g_i(x)\}. \tag{8.18}$$

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

$$\mathsf{GB}_a = \{g_1(\phi, x),\ g_2(x)\}. \tag{8.19}$$

By the construction of p_1 from (8.10) we have no ambiguity for $m = 1$, i.e., ϕ is a functionally dependent variable w.r.t. $\langle p_1, \lambda_7^3\rangle$ and the first polynomial in GB_a can be written (see Section 5.6)

$$\phi - k_{\phi_a}(x) \tag{8.20}$$

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

$$\nu_{\phi_a}(x) = g_2(x) \wedge g_3(x) \wedge \ldots \wedge g_i(x). \tag{8.21}$$

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

$$\nu_{\phi_a}(x) = g_2(x). \tag{8.22}$$

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 $\nu_{\phi_a} \neq 0$. Compute

$$\mathsf{GB}_b = GB_7\left(\{p_1(x, \phi, m), \lambda_7^3(\phi), \neg\nu_{\phi_a}(x)\}\,|_{m=0}\right) \tag{8.23}$$

with lexicographic order $\phi > x$ to get

$$\phi - k_{\phi_b}(x) \tag{8.24}$$

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

We can now compute ϕ as a function of x as

$$\phi = K_\phi(x) = k_{\phi_a}(x)\neg\nu_{\phi_a}(x) + k_{\phi_b}(x)\neg\nu_{\phi_b}(x) \tag{8.25}$$

with valid area

$$V_\phi = \nu_{\phi_a} \vee \nu_{\phi_b}. \tag{8.26}$$

It turns out that $V_\phi \equiv$ `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

$$K_\phi(x) = 2 - x - 3x^2 - 3x^3 + 2x^4 - 2x^5. \tag{8.27}$$

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 (8.16) by using equation (8.27).

Finding an Unambiguous Control Law

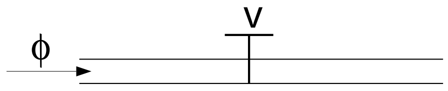
In order to compute the u_i as functions of ϕ, w and d we need the solution to equation (8.12) to be unambiguous. Obviously, this is not the case with the water tank. There are several control actions that, given values of w and d, give the same net flow ϕ. Therefore we need to make a priority among 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 $u_{1,\mathsf{def}} = 0$.

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 (8.12) in these cases, we need extra constraints in the Gröbner basis computations. This is illustrated by a simple example.

Example 8.2 Extra Constraints
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

$$c(\phi, v) = (\phi - 1)(v - 1). \tag{8.28}$$

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.

Computation of the Actuator Signals

First define default values for the actuators

$$\begin{aligned} u_{1,\mathrm{def}} &= 0 \\ u_{2,\mathrm{def}} &= 1 \\ u_{3,\mathrm{def}} &= 0 \\ u_{4,\mathrm{def}} &= 1 \end{aligned} \tag{8.29}$$

The extra constraints are given by

$$\begin{aligned} c_1(u_1, d) &= (d-1)(u_1 - u_{1,\mathsf{def}}) \\ c_2(u_2, d) &= (d-2)(u_2 - u_{2,\mathsf{def}}) \\ c_3(u_3, w) &= w(u_3 - u_{3,\mathsf{def}}) \\ c_4(u_4, w) &= (w-1)(u_4 - u_{4,\mathsf{def}}) \end{aligned} \tag{8.30}$$

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

Start by computing

$$\mathsf{GB}_1 = \mathsf{GB}_7\left(\{p_2, c_1, \Lambda_7\}|_{u_2=1, u_3=0, u_4=1}\right) \tag{8.31}$$

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

$$u_1 - k_{u_1,1}(\phi, d, w) \tag{8.32}$$

as the first polynomial in GB_1, and a valid area for this solution which we denote $\nu_1(\phi, d, w)$. In this valid area, we let the other actuators take their default values, that is

$$\begin{aligned} k_{u_2,1} &= 1 \\ k_{u_3,1} &= 0 \\ k_{u_4,1} &= 1 \end{aligned} \tag{8.33}$$

Next use both u_1 and u_2. Since this control action should not be allowed in ν_1 we add $\neg\nu_1$ in the Gröbner basis computation. That is, compute

$$\mathsf{GB}_2 = \mathsf{GB}_7\left(\{p_2, c_1, c_2, \neg\nu_1, \Lambda_7\}|_{u_3=0, u_4=1}\right) \tag{8.34}$$

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

$$\mathsf{GB}_3 = \mathsf{GB}_7\left(\{p_2, c_1, c_2, c_3, \neg\nu_1 \wedge \neg\nu_2, \Lambda_7\}|_{u_4=1}\right) \tag{8.35}$$

and

$$\mathsf{GB}_4 = \mathsf{GB}_7\left(\{p_2, c_1, c_2, c_3, c_4, \neg\nu_1 \wedge \neg\nu_2 \wedge \neg\nu_3, \Lambda_7\}\right) \tag{8.36}$$

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

$$K_{u_i}(\phi, w, d) = \sum_{j=1}^{4} k_{u_i,j} \neg\nu_j. \tag{8.37}$$

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

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

$$V_u(\phi, w, d) = 1 - 3\phi + d\phi + 2d^2\phi + 3\phi^2 - d\phi^2 - 2d^2\phi^2 \qquad (8.38)$$

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 (8.27) into (8.38) to get the equations in the desired form (8.16). 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.

8.2.3 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?

Controllability

The polynomial V_u in (8.38) 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 (8.38) for all values of ϕ, w and d, we find that V_u is `false` only for the case

$$\begin{cases} \phi = 2 \\ d = 0 \end{cases} \qquad (8.39)$$

(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 8.1 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.

The Closed Loop System

From (8.3), (8.4), (8.27) and (8.38) 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).

$$u_{1,\mathsf{def}} = 0,\ u_{2,\mathsf{def}} = 1,\ u_{3,\mathsf{def}} = 0,\ u_{4,\mathsf{def}} = 1 \tag{8.40}$$

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

$$u_{1,\mathsf{def}} = 0,\ u_{2,\mathsf{def}} = 1,\ u_{3,\mathsf{def}} = 0,\ u_{4,\mathsf{def}} = 0 \tag{8.41}$$

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

$$x^+ = f_1(x, \phi_0) \tag{8.42}$$
$$\phi_0 = f_2(u_1, u_2, u_3, u_4, w, d) \tag{8.43}$$
$$u_1 = K_{u_1}(\phi, w, d) \tag{8.44}$$
$$u_2 = K_{u_2}(\phi, w, d) \tag{8.45}$$
$$u_3 = K_{u_3}(\phi, w, d) \tag{8.46}$$
$$u_4 = K_{u_4}(\phi, w, d) \tag{8.47}$$
$$\phi = K_\phi(x, w, d) \tag{8.48}$$

where we substitute (8.48) into (8.47) 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

$$\begin{aligned}G_{c1}(x,d,w) = {} & -2d + 3d^2 - 2x + 3dx - d^2x - 2wx + 3dwx - \\ & - d^2wx - dx^2 - 2d^2x^2 + 3wx^2 - dwx^2 - 2d^2wx^2 - \\ & - x^3 - 3dx^3 + d^2x^3 + 2wx^3 - 3dwx^3 + d^2wx^3 + \\ & + 3dx^4 - d^2x^4 - 2wx^4 + 3dwx^4 - d^2wx^4 + x^5 - \\ & - dx^5 - 2d^2x^5 + 3wx^5 - dwx^5 - 2d^2wx^5 + 3x^6 - \\ & - dx^6 - 2d^2x^6 + 2wx^6 - 3dwx^6 + d^2wx^6\end{aligned} \tag{8.49}$$

$$\begin{aligned}G_{c2}(x,w,d) = {} & -2d + 3d^2 - 2x + 3dx - d^2x - dx^2 - 2d^2x^2 - \\ & - x^3 - 3dx^3 + d^2x^3 + 3dx^4 - d^2x^4 + x^5 - dx^5 - \\ & - 2d^2x^5 + 3x^6 - dx^6 - 2d^2x^6\end{aligned} \tag{8.50}$$

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.

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 (8.5). Now consider the ideal

$$A = \langle \neg p(x^+),\ x^+ - G_c(x,w,d),\ \Lambda_7 \rangle. \tag{8.51}$$

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)

$$\mathsf{GB}_7\left(\{\neg p(x^+), x^+ - G_{c1}(x,w,d), \Lambda_7(w,d)\}\right) = \\ \{\, d,\ -w+w^2,\ -x+wx,\ x^+,\ 3x+4x^2\} \quad (8.52)$$

(ii)

$$\mathsf{GB}_7\left(\{\neg p(x^+), x^+ - G_{c2}(x,w,d), \Lambda_7(w,d)\}\right) = \\ \{\, d,\ -w+w^2,\ 2x+x^+,\ x^2\} \quad (8.53)$$

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 Table 8.3.

8.3 Conclusions

The example has shown that the relational framework based on a polynomial representation 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. This topic is further discussed in Plantin [99].

8.3.1 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. 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.

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.

		x						
d	**w**	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

		x						
d	**w**	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

Table 8.3: Values for $x^+ = G_{c1}(x, d, w)$ and $x^+ = G_{c2}(x, d, w)$ respectively.

8.3.2 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 [30] in *Mathematica* [114]. 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.

When this thesis is printed the author has tried some Gröbner Basis computations on the new version (3.0) of *Mathematica* [115], see Section 6.3. This version offers much better performance for Gröbner Basis computations. The computations in this chapter have not been recomputed using this version, though, but the author's impression is that the Gröbner Basis in this chapter will be computed in a few minutes.

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 functionally dependent 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

Modeling of DEDS - General Aspects

9 Approaches to Describing Discrete Event Dynamic Systems

The area of *discrete event dynamic 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 this chapter we 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 detail in Chapter 11.

9.1 The Automata Theoretic Approach

The notion of *finite automata* was introduced by McCulloch and Pitts [87] in the mid-forties and since then the area of finite automata has been developed further by, e.g., Moore and Mealy [92, 90] in the fifties.

In the area of automatic control, Ramadge and Wonham [104] 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 [68] and finite automata, chosen for conceptual simplicity and generality rather than modeling efficiency or computational tractability. See [104] for a survey of supervisory control theory. Properties of DEDS such as controllability [116] and observability [83] have been investigated in the theory of supervisory control.

To reduce complexity of synthesizing a control law for DEDS, horizontal [101] and vertical (hierarchical) [118] decomposition of the overall control task has 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* [95] are used for formal verification. See [94] 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* [103]. 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 automata, *statecharts* were introduced by Harel [62] and now exist in a number of versions. 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 [13] have a suggestion of an algorithm solving the reachability problem of HSMs.

For a survey of the area discussed above, see [111] and [78].

9.2 The Petri Net Approach

Petri nets were introduced by Petri in 1962 [98] 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. [67]. For an introduction to Petri nets, see Peterson [97] and Murata [93].

Not until recently have Petri nets been used for analysis and synthesis of control laws for DEDS. Krogh and Holloway [75] and Ichikawa and Hiraishi [69] have introduced *controlled Petri nets* (CPN) which is a Petri net extension, from which they have formulated theories and methods for supervisory control.

The dynamics 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 formulated in a control context by Giua, DiCesare and Silva [45], and by Li and Wonham [81, 82].

The notation of Petri nets is often extended, [97], to supports new types of systems, e.g., *Timed-Event-Graphs* (TEG) [5] which 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 [29]. Max-plus is a dioid algebra [5] which is the mathematical framework of the system theory of TEG models. The TEG theory has many analogies of the theory of linear systems, but the applications of the theory are mainly concerned with performance and optimization issues rather than feedback control. For a tutorial of max-plus and a survey of the system theory of TEG, see [28] and [5].

9.3 The COCOLOG Approach

In 1990 Caines and Wang [20] 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* [10] called Blitzensturm [19].

For an introduction see [20] and the application of controlling an elevator [32].

9.4 Simulation and Perturbation Analysis

Perturbation analysis [46] 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 [21, 64]. For a research survey of the area see [22].

10

Aspects of Modeling using Algebraic Methods

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 between 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 10.3.

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.

10.1 Polynomial Representation of DEDS

In the previous parts of this thesis we have modeled systems using polynomials in $\mathbb{R}_7[Z]$. We have also used Boolean relations which can be regarded as polynomials in $\mathbb{R}_2[Z]$. Modeling DEDS with Boolean equations is a well known technique from Moore and Mealy [92, 90].

To be able to indicate some conceptual differences between modeling a system

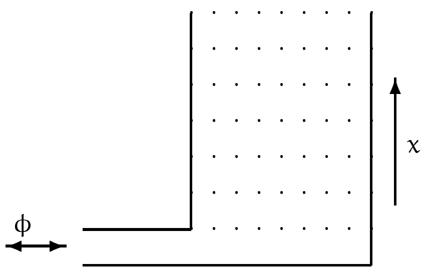


Figure 10.1: Simplified tank

in a smaller or larger field we remodel a simplified version of the tank from Section 8.1.2 twice. First we do a more thorough modeling of the tank using $\mathbb{R}_7[Z]$, compared to Section 8.1. Then we do the same using Boolean expressions.

By doing this remodeling manually we hope to see if there are some advantages using $\mathbb{R}_7[Z]$ 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 10.1.

Example 10.1 Polynomial Tank Model
For the tank in Figure 10.1 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 8.1.2.

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 express the level with only one state variable. Basically we now have the following state equation

$$x^+ == x + (\phi - 1) \tag{10.1}$$

where $(\phi - 1)$ takes values from $\{-1, 0, 1\}$. (ϕ is defined as in Section 8.1.2.)

Equation (10.1) 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

$$x^+ == x + (\phi - 1) * C(x, \phi) \tag{10.2}$$

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

$$C(x, \phi) = (x == 6) \wedge (\phi == 2) \vee (x == 0) \wedge (\phi == 0) \tag{10.3}$$

The polynomial representing $C(x, \phi)$ can now be computed using the rules described in

Section 5.4.

$$\begin{aligned} C(x,\phi) &= (x{=}{=}6)\wedge(\phi{=}{=}2) \vee (x{=}{=}0)\wedge(\phi{=}{=}0) \\ &= (x-6)\wedge(\phi-2) \vee (x)\wedge(\phi) \\ &= (1-(1-(x-6)^6)(1-(\phi-2)^6)) \vee (1-(1-x^6)(1-\phi^6)) \\ &= (1-(1-(x-6)^6)(1-(\phi-2)^6))(1-(1-x^6)(1-\phi^6)) \\ &= 5\,\phi+3\,\phi^2+4\,\phi\,x+3\,\phi^2\,x+3\,\phi\,x^2+4\,\phi^2\,x^2+4\,\phi\,x^3+3\,\phi^2\,x^3+ \\ &3\,\phi\,x^4+4\,\phi^2\,x^4+4\,\phi\,x^5+3\,\phi^2\,x^5+x^6+5\,\phi\,x^6+\phi^2\,x^6 \end{aligned} \tag{10.4}$$

If we plug (10.4) into (10.2) and expand the result we get

$$\begin{aligned} x^+ {=}{=}\, & x+3\,\phi-3\,\phi^2-3\,\phi\,x+3\,\phi^2\,x+3\,\phi\,x^2-3\,\phi^2\,x^2-3\,\phi\,x^3+3\,\phi^2\,x^3+ \\ & 3\,\phi\,x^4-3\,\phi^2\,x^4-3\,\phi\,x^5+3\,\phi^2\,x^5-x^6+\phi\,x^6 \end{aligned} \tag{10.5}$$

which is the same expression as (8.3) in Section 8.1.2.

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 (10.1) 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 10.2 Boolean Tank Model

For the tank in Figure 10.1 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 8.1.2.

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 represent 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 $\mathbb{I}_3$ (see Definition 7.7) 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 [`false`, `false`, `false`] for the bottom level ($x{=}{=}0$) up to [`true`, `true`, `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 $\phi =$ [`true`, `false`] corresponds to increasing the level whereas $\phi =$ [`false`, `true`] corresponds to decreasing the level of the tank. With $\phi =$ [`false`, `false`] the level is constant. $\phi =$ [`true`, `true`] is not allowed.

We want to find the functions f_1, f_2, f_3 in the state space equation below.

$$[x_3^+, x_2^+, x_1^+] {=}{=} [f_3(x_3, x_2, x_1, \phi_1, \phi_2), f_2(x_2, x_1, \phi_1, \phi_2), f_1(x_1, \phi_1, \phi_2)] \tag{10.6}$$

As in the previous example we will start by considering the increasing and decreasing behavior of the model. By adding two bit representations like $[x_3, x_2, x_1]+$[`false`, `false`, ϕ_1]

we will get the appropriate expressions for the increasing behavior, and since -1 is represented by [`true`, `true`, `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 and decreasing flow 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 7.8,

$$[\tilde{f}_3, \tilde{f}_2, \tilde{f}_1] = \mathsf{BAdd}([x_3, x_2, x_1], [\phi_2, \phi_2, \phi \vee \phi_2]) \tag{10.7}$$

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 (10.8) \\
& (\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 and 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

$$\rho = \left\{ \begin{array}{l} \phi_1 \rightarrow \phi_1 \wedge \neg(x_3 \wedge x_2 \wedge \neg x_1) \\ \phi_2 \rightarrow \phi_2 \wedge \neg(\neg x_3 \wedge \neg x_2, \wedge \neg x_1) \end{array} \right\} \tag{10.9}$$

and with ρ we can now compute the final state space expressions

$$[f_3, f_2, f_1] = [\tilde{f}_3, \tilde{f}_2, \tilde{f}_1]|_\rho \tag{10.10}$$

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 (10.8) without using a computer is not a tempting task.

It is not fair to draw any precise conclusions from the modeling of one system only. But we have seen that in this case the polynomial approach with $\mathbb{R}_7[Z]$ gives simpler computations and model. This indicates that there are cases where a larger field is more appropriate to use.

In Section 8.2.1 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 harder. It seems that the arithmetic operations following from the use of a larger field, are essential for simplifying both the modeling and design efforts.

Theoretically we know that there are always cases which have the worst case complexity independent 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 an appropriate polynomial approach with a good choice of the finite field $\mathbb{F}_q$.

10.2 Modeling Description Language

To model systems directly with polynomials is not realistic except for trivial cases, since the polynomial representation is hard to interpret manually. 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, the experiences achieved from the modeling of the landing gear controller and the tank in Part 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 than design or synthesis aspects although we have to take precaution to include these aspects as well.

10.2.1 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 [24], Simulink, Pascal and C are used as engineering tools for developing control systems and are widely used in the industries. Other MDLs like VHDL [70], ASA [112], SDL [23] 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 [62], SIGNAL [8], ESTRELLE [9], and LUSTRE [61]. Results on object-orientation for modeling can be found in Zeigler [117]. For a comparative survey of different MDLs applied to a design problem see [27].

MDLs are used in all stages in development, e.g., ASA for specification, Simulink for testing and analysis and Pascal for implementation. See [41] 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., all quantities of the model are of finite domains, then it is possible to translate the MDL model into the relational domain without introducing discrepancies [40]. The mapping from an MDL to a corresponding relational representation must capture the system behavior described by the MDL representation exactly.

10.2.2 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 Section 8.1 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 system 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 Section 7.3 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 7.6.1, 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 question 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.

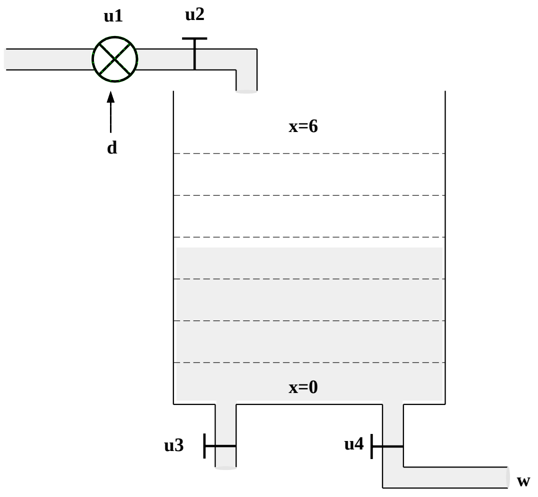
2. *Pascal is a language that is meant to be used for synthesis of systems.*
   Many computer languages like Pascal or C are used for system development and not as a modeling tool. There are very few, if any, languages for modeling.

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 contain 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 10.3 Pascal Model of the Tank



If we write a model of the tank from Section 8.1 using the subset of Pascal presented in Section 7.3.2, 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.

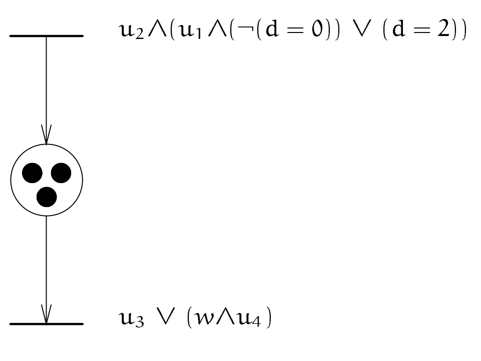
From Example 10.3 above we can conclude that it is very difficult to see any structural similarities between the Pascal code and the physical system. If we only had access to the Pascal code as a description of the system it would be hard to see that the actual system is a tank.

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 [97].

Example 10.4 Petri Net Model of the Tank
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.


$u_2 \wedge (u_1 \wedge (\neg(d = 0)) \vee (d = 2))$
$u_3 \vee (w \wedge u_4)$

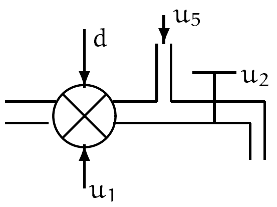


Figure 10.2: Modified inflow construction.

Further details of how to model the tank using Petri nets is presented in Section 11.2.

This example shows that the graphical language of Petri nets in this case can 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 discovered 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 at the bottom of 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 Section 8.1.

A problem with the Petri net in Example 10.4 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 10.2), it is not clear how to modify the Petri net model in Example 10.4.

Modifications in systems should be easy to apply in the MDL describing the system. In case of Figure 10.2 we added a device to the system which would correspond to adding two models together in the MDL. This feature must be included in a good MDL.

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* [96, 72] 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 relational domain.

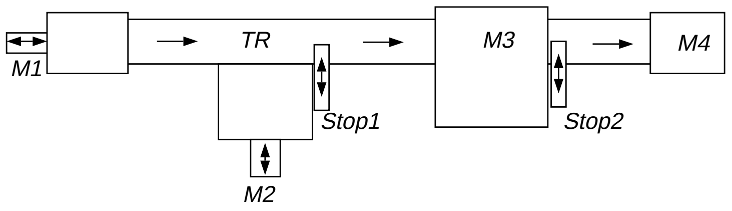
The bond graph language is composed of a small number of primitive elements that are connected in a graph. The functionality of the system is described by this graphical language at the same time as the overall structure and topology of the system can be preserved. In this sense the bond graphs gives inspiration to the features we would like to have in the perfect MDL.

Bond graphs have been used with great success for continuous systems [14]. Strömberg et al. [110] introduced switched bond graphs in order to handle discontinuities. This was done by adding a new primitive element *the switch* to the bond graph 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 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 to model 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 not decided until 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 10.5 LEGO Car Factory
At the Dept. of AC. in Linköping there is a LEGO car factory laboratory process [109]. 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.



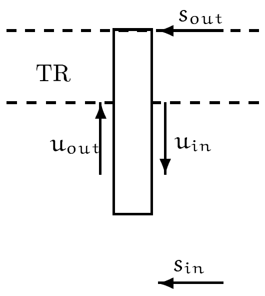
The factory consists of four machines, namely *M1*, *M2*, *M3* and *M4* working along an assembly line *TR*. There are two stoppers *Stop1*, *Stop2* positioned after *M2*, *M3* respectively. The production of a car follows the sequence:

1. The machine *M1* delivers a chassis to the conveyor *TR*.
2. When the chassis reaches the front of *M2*, *Stop1* blocks the conveyor and *M2* puts a roof on the chassis.

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 on 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 the interaction between the stopper and its environment decides how the interacting models should be mapped to a relational descriptions. Compare this with the acausal property of bond Graphs were the computational causality is first decided by global rules applied to the bond graph (see [96, 110]).

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 8.2.2 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 8.2.2 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 solution 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 an MDL to preserve the structure of the system. For MDLs that preserve the structure of the system, it will be easier to do analyze and synthesize modularly in order to reduce complexity.

To conclude this section we will repeat the desirable features[2] of an MDL.

[1]Compare this with computing local control laws from local specifications.
[2]This is a wish list.

(i) The MDL should be graphically represented.

(ii) The MDL should preserve the structure of the modeled system.

(iii) Sequential description of the behavior should not be necessary for the MDL.

(iv) Abstraction and modularity should be supported in the MDL.

(v) The MDL should support a smart context dependent interconnection of general devices.

(vi) 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 language that fulfills the specification above or if it has to be invented or if such a language cannot even exist.

10.3 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 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 10.1 Signals & Events

(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.

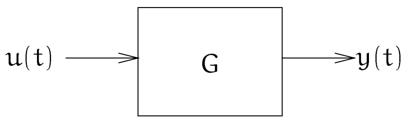


Figure 10.3: A continuous system.

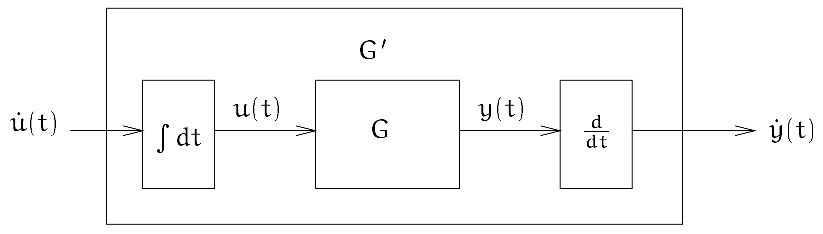


Figure 10.4: A system interacting with signal changes.

10.3.1 Going from Signals to Events

We will make some analogies with continuous systems. Consider the system G in Figure 10.3, 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 signal changes, we will have a the system in Figure 10.4.

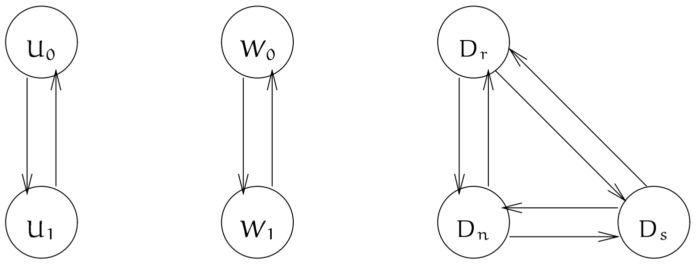
We see in Figure 10.4 that by adding an integrator to the inputs and a differentiator to the outputs we can derive G' from G. Note that the number of states will increase with the number of input signals. These states can be regarded as input observers for the system G.

For DEDS 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 a finite automaton for every input signal that can observe events corresponding to the signal changes.

Example 10.6 Automata as Input Observers
The tank in Section 8.1 has 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 automata will have the structure as shown in the figure below:



where the labels of the transitions are omitted but easily can be deduced from the table above.

We see that the number of states is increased by a factor 12 ($2 \times 2 \times 3$) for the event driven model.

Transforming model outputs from signals to events can 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 10.6 are the inflow and outflow events, the output states (level in the tank) are still necessary since the output events depends on the level.

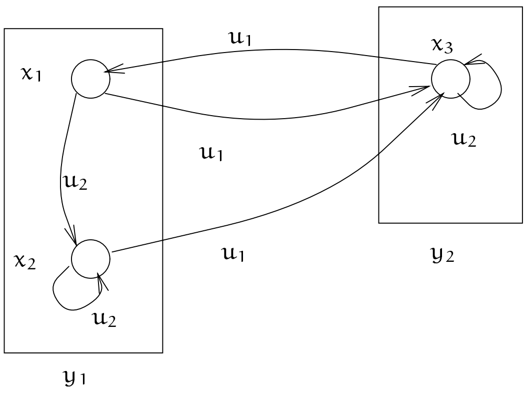
Example 10.6 indicates that the transformation from signal models to event driven models, increases the state dynamics for the inputs in the systems. The effect of this is also shown in Sections 11.1.2 and 11.2.2 where finite automata and Petri nets are used to model the tank system.

10.3.2 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 Chapter 2 where finite automata were modeled by relational models.

Consider the following example:

Example 10.7 More Machine
Let the Moore machine [92] 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

$$\begin{aligned} f(x^+, x, u) = x_1^+ &\leftrightarrow (x_3 \wedge u_1) \wedge \\ x_2^+ &\leftrightarrow (x_1 \wedge u_2 \vee x_2 \wedge u_2) \wedge \\ x_3^+ &\leftrightarrow (x_1 \wedge u_1 \vee x_2 \wedge u_1 \vee x_3 \wedge u_2) \end{aligned} \tag{10.11}$$

and the output relation is given by

$$\begin{aligned} g(y, x) = y_1 &\leftrightarrow (x_1 \vee x_2) \wedge \\ y_2 &\leftrightarrow x_3 \end{aligned} \tag{10.12}$$

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 variables 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 $\mathbb{R}_3[Z]$ instead of the Boolean representation for the system in Example 10.7, 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 take place.

To include symbols for “no events” is often recommendable in general. It is particularly important if the model is to be implemented in a simulator that evaluates the model periodically independent of event occurences.

11

Comparative Reviews of the Tank Model

As described in Chapter 9 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 types of problems.

In this chapter we will focus on three different DEDS approaches from the area of automatic control: Ramadge-Wonham [104], Petri nets [98] and COCOLOG [20]. By remodeling the tank from Section 8.1 using these approaches, we get a measure of how appropriate it is to use the relational framework for modeling of DEDS, compared to these three established approaches. The purpose is not to make the comparison complete in all details, but to concentrate on the main features of methods in these approaches. We will, for example, not deal with aspects of computational performance.

The similarity between the chosen approaches is that their main objectives are analysis and synthesis of supervisory control laws for DEDS, i.e., control laws that prevent the system from reaching 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 model from one framework to another often can be performed. 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 Section 8.1) 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.

11.1 The Ramadge-Wonham Approach

Ramadge and Wonham (RW) [104] 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 in control of DEDS.

The tank from Section 8.1 will be modeled as a *finite automaton* (FA) [68] following the principles of converting signals to events that we discussed in Section 10.3. We will also discuss the problem of modeling the behavior of a system that is influenced by real time, since we are only using theory for logical DEDS (DEDS with no time description).

Before we can go further into these details we give a short introduction to the theory of Ramadge-Wonham.

11.1.1 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.

Definition 11.1 Formal Language

(i) Let Σ denote the finite set of events (the alphabet) that labels the state transitions of the model.

(ii) Let Σ^* denote the set of all finite strings of elements of Σ including 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 string. 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 11.2 Finite Automata (FA)

(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 11.3 Control Mechanism

(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 11.4 Closed Loop System
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 11.5 Desired Language & Controllability

(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

$$\overline{K}\Sigma_u \cap L \subseteq \overline{K} \tag{11.1}$$

is satisfied.

□

When composing a desired closed loop behavior it is often useful to intersect languages of subspecifications.

Lemma 11.1 Prefixed Closed Languages ([104])
If the languages $K_1, K_2 \in \Sigma^*$, then

$$\overline{K_1 \cap K_2} \subseteq \overline{K_1} \cap \overline{K_2} \tag{11.2}$$

and if K_1, K_2 are *prefix closed* languages then

$$\overline{K_1 \cap K_2} = \overline{K_1} \cap \overline{K_2} \tag{11.3}$$

□

Ramadge and Wonham has found a necessary and sufficient condition to determine when it is possible to find a supervisory control law f.

Theorem 11.1 Existence of Supervisor ([104] and [102])
For nonempty $K \subseteq L$ there exists a supervisor f such that $L_f = K$ iff K is prefix closed and controllable. □

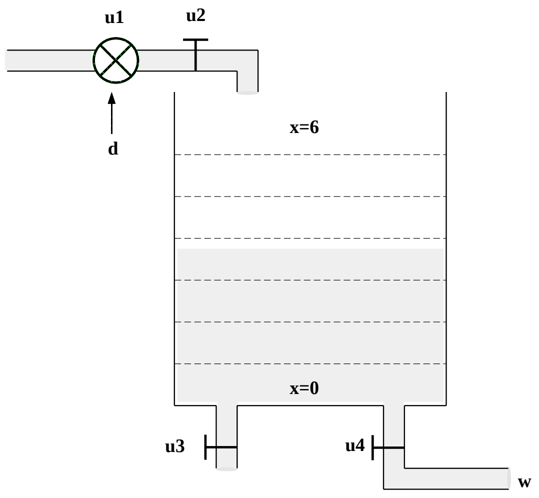


Figure 11.1: The tank

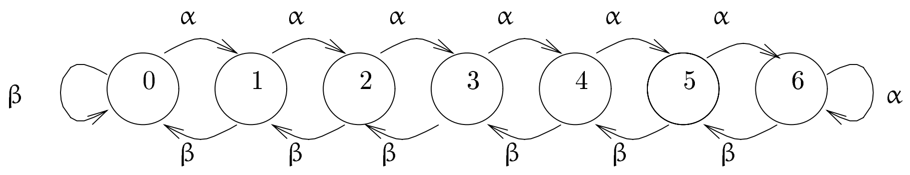


Figure 11.2: FA model of the tank.

11.1.2 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 11.1 using the RW approach.

When we did the control law synthesis of the tank using the polynomial approach (see Section 8.2.1) 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 a comparative view of synthesis between the approach of RW and the polynomial approach see the thesis of Plantin [99].

The notations and interpretation of FA models can vary to some extent, and it

Signal	**Events Σ_I**	**Event States**
u_1	$u_1(\textit{off}), u_1(\textit{on})$	U_0, U_1
w	$w(\textit{stop}), w(\textit{open})$	W_0, W_1
d	$d(\textit{stuck}), d(\textit{normal}), d(\textit{run})$	D_s, D_n, D_r

Table 11.1: Events and states of tank input signals.

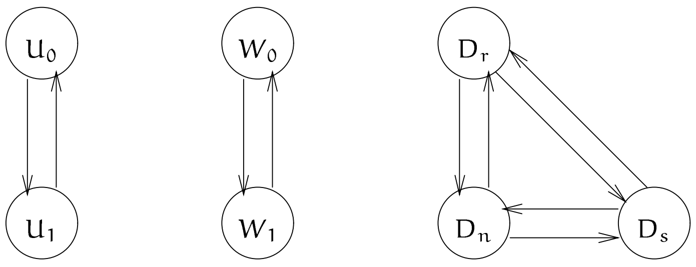


Figure 11.3: FAs modeling the tank input. The arcs correspond to the events given in Table 11.1.

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 effort of tank modeling (see Section 8.1.2) we considered the inputs to the system to be regarded as signals. Keeping the notion of signals we could make the automaton shown in Figure 11.2 where

$$\begin{aligned}\alpha &= u_2 \wedge ((d{=}{=}2) \vee u_1 \wedge \neg(d{=}{=}0)) \\ \beta &= u_3 \vee (u_4 \wedge w)\end{aligned} \tag{11.4}$$

are predicates that must be true to make a state transition possible. This model is not yet a pure event driven model since (11.4) are functions of signals and therefore α and β also are signals. (See Section 10.3.) We also need a more detailed description of how 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 10.3 we associate each value of each signal with a unique event that represents a signal change towards that value. For the tank we denote the set of these events as Σ_I (see Table 11.1).

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 an FA for each signal. For the three signals u_1, w, d we get the FAs in Figure 11.3.

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 11.2: Total state space for input FAs where α and β denotes the net flow in the tank. '-' means no event.

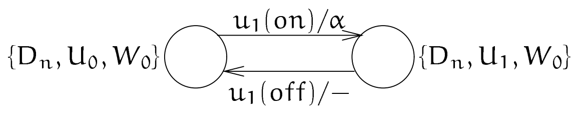


Figure 11.4: Mealy FA sending event α in one of the transitions. The labels of the transitions are *input/output*.

With the three FAs specified by Figure 11.3 and Table 11.1 we now have a model for the inputs to the tank. In order to proceed we must decide how to specify the behavior these FAs give the system.

For each combination of states of the FAs in Figure 11.3, 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 [68]) of the FAs in Figure 11.3. 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 11.2. 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 (11.4). Table 11.2 describes all 12 states of the total input FA, where no information of how the arcs are connected is presented, but the structure of the arcs is trivially deduced from Figure 11.3.

In the Table 11.2 we have indicated for which states the events α and β should occur. We must now decide how to represent these events properly by the input FA.

Let us assume that the events α, β are generated from the FA of Table 11.2 formed as a Mealy machine [68, 90] in the way shown in Figure 11.4, where we take a closer look at the pair of states $\{D_n, U_0, W_0\}$ and $\{D_n, U_1, W_0\}$.

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

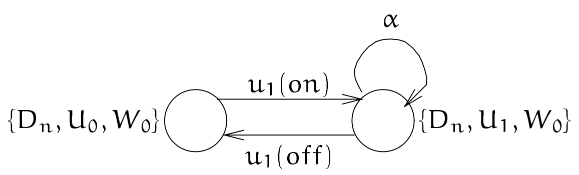


Figure 11.5: FA enabling event α in one of the states.

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 11.2, but for a β-state the tick-events will make transitions toward state 0.

We have until now tried to create an FA that generates the events α, β. This is the most intuitive approach since α and 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 11.4 we get the result shown in Figure 11.5, where only α is allowed to happen in the state $\{D_n, U_1, W_0\}$, and none of α, β are allowed to happen in state $\{D_n, U_0, W_0\}$.

By adding an arc as in Figure 11.5 for each state in Table 11.2 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 11.2 by adding all input events Σ_I as possible events for each state. This will not make the FA defined for all possible events. If this modification is not done this FA will prevent all occurences of events in Σ_I. The resulting FA is denoted $\mathcal{M}_T$ and is shown in Figure 11.6.

We can now consider the FA models $\{\mathcal{M}_I, \mathcal{M}_T\}$ as FA descriptions of the system in Figure 11.1. It would be possible to combine these FAs using a 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 8.1.2 that by

[1]The tick events are introduce in framework of "Timed Discrete Event Systems" [12].

[2]The cross product results in an FA where the number of states is equal to the product of the number of states for the input FAs.

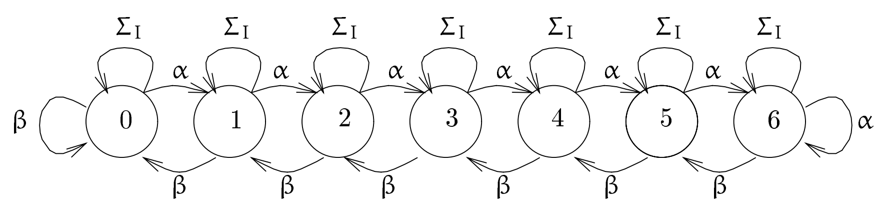


Figure 11.6: $\mathcal{M}_T$: Final FA model for the level dynamics. Σ_I denotes the set of all input events.

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 $\{\mathcal{M}_I, \mathcal{M}_T\}$.

11.1.3 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

$$\Sigma = \Sigma_I \cup \{\alpha, \beta\} \tag{11.5}$$

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

$$L(\mathcal{M}_T) = \Sigma^* \tag{11.6}$$

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

$$\Sigma_I^* \subset L(\mathcal{M}_I). \tag{11.7}$$

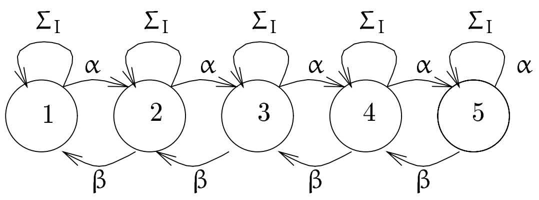


Figure 11.7: FA $\mathcal{M}_K$ specifying the non forbidden behavior.

Since $L(M_I) \subset L(M_T)$ we can specify the language L accepted by the FA $\mathcal{M}$ to be

$$L = L(\mathcal{M}) = L(\mathcal{M}_I) \tag{11.8}$$

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

$$\Sigma_c = \{u_1(\mathit{off}), u_1(\mathit{on})\} \tag{11.9}$$
$$\Sigma_u = \Sigma \setminus \Sigma_c \tag{11.10}$$

Deducing K

To express our control objectives of the tank from Section 8.2.3 as a language we use an 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 8.2.3). By simply removing the end states and the corresponding arcs from $\mathcal{M}_T$ we get the FA denoted M_K shown in Figure 11.7.

The FA $\mathcal{M}_K$ restricts the maximum length of string containing 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.

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 K-language should valid for any initial state of the input signals to the tank. Therefore (11.8) 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 $\mathsf{K} \subseteq \mathsf{L}$ accepted by the FA $\mathcal{M}_{\mathsf{K}}$ initialized with $q_0 = 3$, i.e.,

$$\mathsf{K} = \mathsf{L}(\mathcal{M}_{\mathsf{K}}|_{q_0=3}) \cap \mathsf{L} \tag{11.11}$$

Since both $\mathsf{L}(\mathcal{M}_{\mathsf{K}})$ and L are prefix closed languages we know from Lemma 11.1 that

$$\overline{\mathsf{K}} = \overline{\mathsf{L}(\mathcal{M}_{\mathsf{K}}|_{q_0=3})} \cap \overline{\mathsf{L}} \tag{11.12}$$

which gives $\overline{\mathsf{K}} = \mathsf{K}$, 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, [104].

To conclude the modeling of the tank we note that the quadruple

$$\{\mathsf{L}, \mathsf{K}, \Sigma_c, \Sigma_u\} \tag{11.13}$$

specifies the language of the tank, the objective of the desired behavior, controllable events and uncontrollable events. From (11.13) it is possible to make synthesis of a supervisory control law. Note that the first thing to do in the synthesis process is to check whether (11.13) is controllable or not. In fact, in this case we will find, by using the constraint (11.1) in Definition 11.5, 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.

11.1.4 Summing Up

By making a model intended for the Ramadge-Wonham approach we get some understanding of how this modeling correlates with the relational methods used in Section 8.1. 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 an FA as an event observer. This may of course increase the number of states rapidly. In the relational framework 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 relational approach.

If we instead use the relational approach to model a pure event driven system, we can always keep the number of states the same, i.e., the number of states in the relational 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 relational model corresponds to an occurrence of an event. In this case the transformation from the relational model to an 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 $\mathsf{L}(\mathcal{M})$ of an 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 by a new event symbol, for the assumption that all events can be ordered sequentially to hold. For the relational approach it is possible to let different input variables change values independently of each other. This may in some cases be an advantage in modeling non-sequential systems.

Neither Ramadge-Wonham nor the relational 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 relational modeling presented in Section 10.1.

11.2 Petri Nets

Petri nets were introduced by Petri in 1962 [98] 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. [67].

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 support new types of systems [97], e.g., *Timed-Event-Graphs* [5] 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 [75, 66] 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 9 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 Section 8.1. This modeling will be compared to the modeling performed in Section 10.1 where we used the relational approach.

11.2.1 Introduction to Petri Nets

We begin by a formal introduction to Petri nets, adapting the notation of Peterson [97].

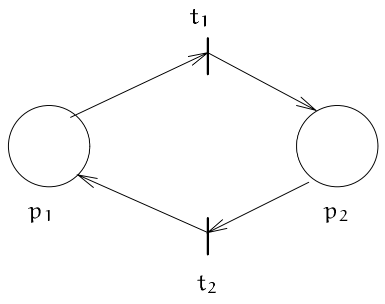
Definition 11.6 Petri Nets
A *Petri net* (PN) is a four-tuple $C = (P, T, I, O)$.
$P = \{p_1, \ldots, p_n\}$ is a finite set of *places*, $n \geq 0$.
$T = \{t_1, \ldots, t_m\}$ is a finite set of *transitions*, $m \geq 0$.
The set of places and the set of transitions are disjoint, $P \cap T = \emptyset$. $I : T \to P^\infty$ is the *input* function, a mapping from transitions to bags[3] of places. $O : T \to P^\infty$

[3]A bag is a generalization of sets, which allows multiple occurences of elements. The set of all bags of places is denoted P^∞.

is the *output* function, a mapping from transitions 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) [68].

Example 11.1 A Petri Net
A switch with two states (on,off) is represented by the Petri net $C_1 = (P, T, I, O)$ shown in the figure below.



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:

$$\begin{aligned} I(t_1) &= \{p_2\} \\ I(t_2) &= \{p_1\} \\ O(t_1) &= \{p_1\} \\ O(t_2) &= \{p_2\} \end{aligned} \tag{11.14}$$

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 • can be regarded as an object that jumps from one place to another (or to the same) by passing through one of the transitions in T.

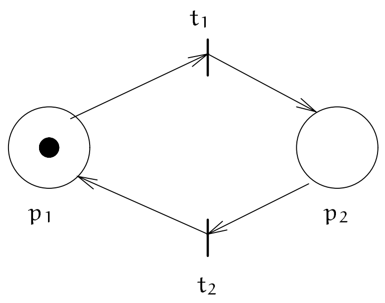
The marking of a PN gives the number of tokens for every place in the PN. Formally this is defined as

Definition 11.7 Place Marking
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 11.8 Petri Net Marking
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 11.2 A Marked Petri Net
Initiating the PN C_1 of the switch in Example 11.1 with a token like



means that the switch is "on".

The marking of the PN is

$$M(P) = [1, 0] \tag{11.15}$$

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

$$m(p) \geq \#(p, I(t)) \quad \text{for all } p \in I(t) \tag{11.16}$$

where the function $\#(p, I(t))$ gives the number of occurrences of 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

$$m'(p) = m(p) - \#(p, I(t)) \quad \text{for all distinct } p \in I(t) \tag{11.17}$$

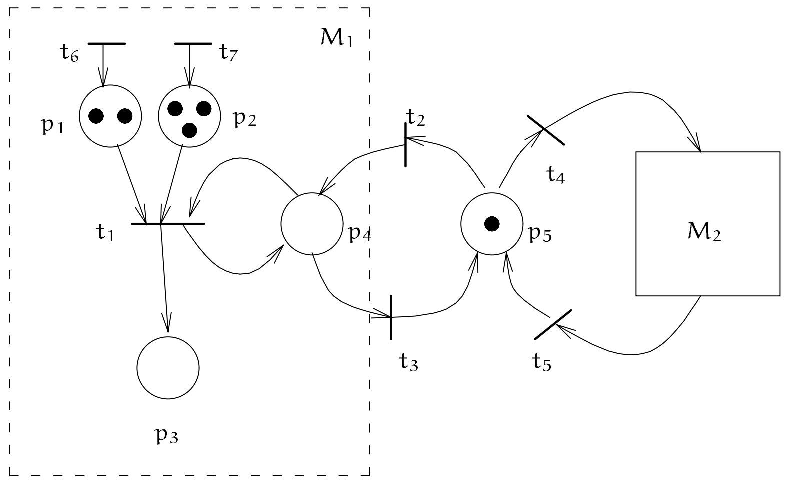
where $m'(p)$ denotes the new number of tokens in place p.

(iii) A firing of a transition $t \in T$ will increase the number of tokens in output places as

$$m'(p) = m(p) + \#(p, O(t)) \quad \text{for all distinct } p \in O(t) \tag{11.18}$$

To illustrate executions of Petri nets we use an example.

Example 11.3 Transmitters
Consider two equal transmitters M_1 and M_2 sharing 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 sent 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 at M_1, and the channel is initialized to idle.

A place can contain any number of tokens in a Petri net. As shown 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 need 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 relational framework.

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 relational models 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 [5].

In Example 11.3 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 in [97] will be used consequently in the modeling of the tank later in this section.

We make the following definition.

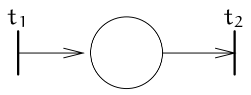
Definition 11.9 Source & Sink

(i) A *source transition* $t_u \in T$ is a transition such that $I(t_u) = \emptyset$, i.e., t_u does 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 does not have any output transitions.

□

Example 11.4 Source & Sink
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 11.8 using general PN.

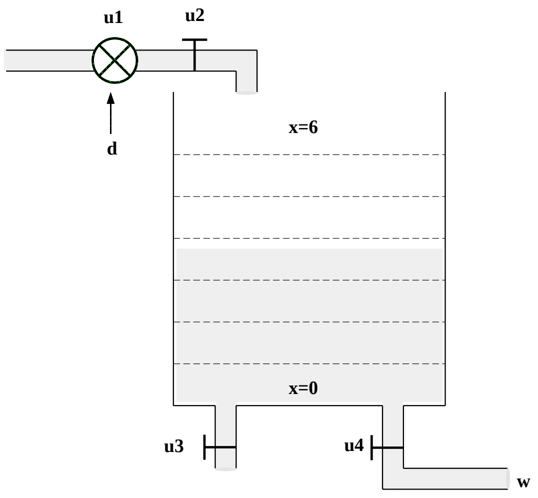


Figure 11.8: The tank

11.2.2 The Petri Net Model of the Tank

We will make a Petri net model of the tank from Section 8.1 shown in Figure 11.8.

As in the modeling presented in Section 11.1.2 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.

In analogy with previous modeling of the tank we will start by the modeling the increase and decrease of the water level.

Since a Petri net place can model accumulating buffers as described above, we will try to model the tank as in Figure 11.9 where α is the event of an inflow of one

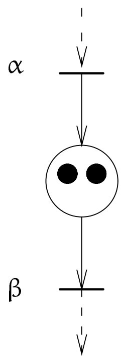


Figure 11.9: PN model of infinite tank.

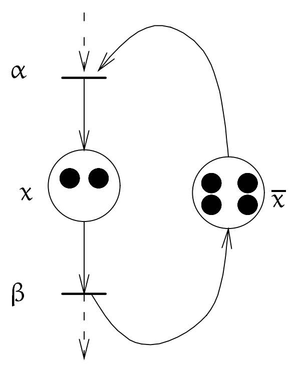


Figure 11.10: PN model of a finite tank.

unit (one token) and β is the event of an outflow of one unit. Figure 11.9 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 a 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 11.3.

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 volume in the tank. The PN is now as in Figure 11.10 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 11.8.

We have derived a model for the dynamics of the level in the tank. This model will now be connected to PN models of the pump and the uncontrollable outflow w.

Peterson [97] 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 should 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 11.2.2 initiated to the state "no outflow".

The PN model in Figure 11.2.2 corresponds to the switch discussed in Example 11.2. 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

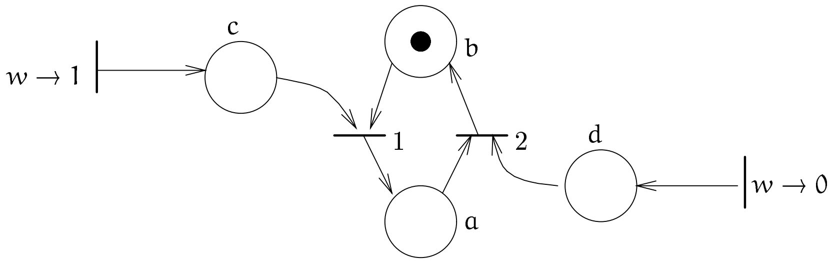


Figure 11.11: PN model for the outflow.

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 to the PN of the outflow w 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	$(l, h) \ \vee \ (l, i)$	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 11.2.2 shows the Petri net of the pump derived from the table above. As we can see in this figure the Petri net is divided into two separate nets. The reason

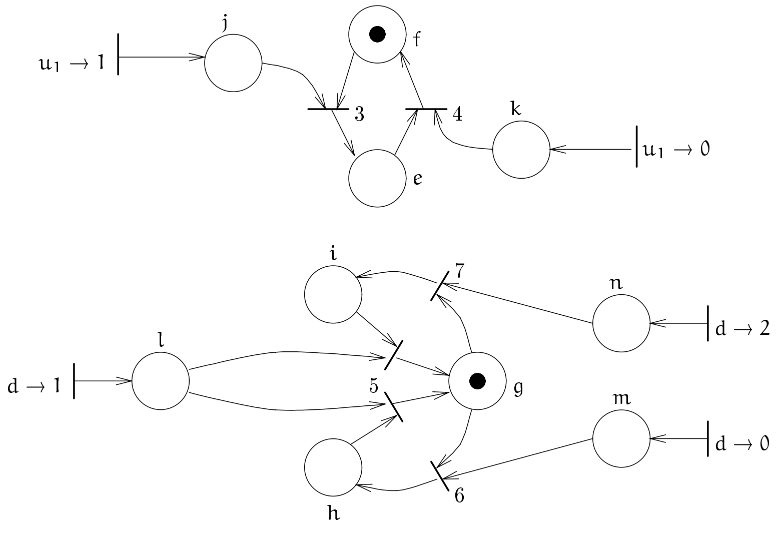


Figure 11.12: Two separate PN models. The first represent the pump behavior of events on the actuator u_1. The second represents failure behavior of the pump.

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, the outflow and the pump. It seems reasonable that new conditions and events will not be necessary. In fact, one major feature of Petri nets is that it is easy to combine Petri nets, 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 have previously been 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	$(e,g,\overline{x})_1 \lor (i,\overline{x})_2$	$(e,g,x)_1 \vert (i,x)_2$
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 in 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 is presented in Figure 11.13 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 automata (FA) in Section 11.1.2 where the resulting model included 84 states. The Petri net model in Figure 11.13 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 verify since the sum of possible markings of these 9 places is 84. This shows the power of Petri nets in terms of to capturing modularities in systems.

11.2.3 Petri Net Synthesis

Having a Petri net model for the tank in Figure 11.8 we can hopefully use this model to synthesize a control law following the specifications stated in Section 8.2.1. We will discuss the possibilities for our model to fit with some of the synthesis methods in the area of Petri nets. For a tutorial survey see Holloway and Krogh [65].

To be able to apply synthesis methods to Petri nets an extension of Petri nets has been defined called *Controlled Petri Nets* [69, 75]. For controlled Petri nets the primitive *control place* is introduced. Control places are 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

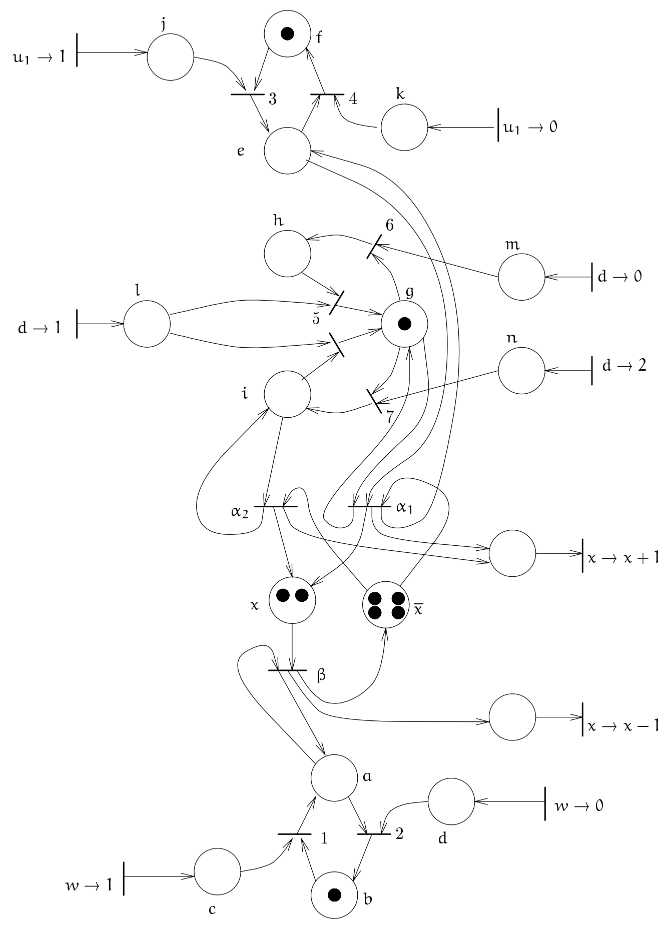


Figure 11.13: The final Petri net model of the tank.

prevent events from occuring. For this type of control to be possible, the control places must correspond to some actuators of the plant. In the case of the tank in Figure 11.8 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 11.13, 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 which acts on the control places, from these observations. 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 [43].

State feedback policies for controlled Petri nets have been investigated by a number of researchers, see the overview in [65]. 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 [77], this cannot be done in general. Alternative methods are *linear integer programming* approaches (Giua et al. [44, 45] and Li and Wonham [81, 82]) and *path based* algorithms (Krogh et al. [77, 76, 66]).

In [66], Krogh and Holloway present 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 11.13 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 has been shown that controllability is undecidable for the most general Petri net languages [106]. The Petri net model of the tank corresponds to the FA model in Section 11.1.2 from which it is possible to create a supervisory control law using the Ramadge-Wonham approach. Therefore we draw the conclusion that the control problem of the Petri net model in Figure 11.13 is decidable in spite that it will not fit into one of the Petri net subclasses specialized for synthesis.

11.2.4 Summing Up

We have used Petri nets to model the tank system shown in Figure 11.8, by adapting the modeling method in [97] 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 an extra modeling effort is needed to convert signals to events, as in in the case of Ramadge-Wonham approach in Section 11.1.2.

The resulting Petri net model cannot be used for synthesizing a supervisory control law with the approach introduced by Holloway and Krogh [66]. The reason

[4]A cycle in a controlled Petri net is a directed graph beginning and ending at the same node with all nodes in the path occuring only once.

for this is that the synthesis method is based on a restricted class of Petri nets, and the final Petri net model of the tank cannot be transformed to that class.

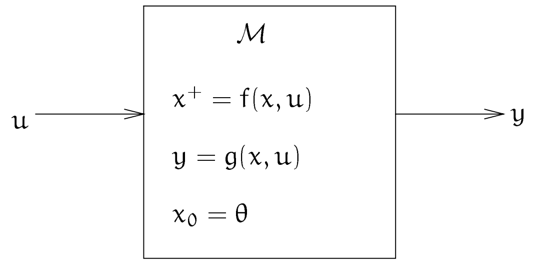
11.3 COCOLOG

In 1990 Caines and Wang [20] 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.

11.3.1 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., 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 in general have no metric that can be used to measure the distance between two quantities 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 states in linear systems.

Within the framework of COCOLOG, another approach has been proposed for feedback control based on the set of observed inputs and outputs denoted $\mathcal{U}, \mathcal{Y}$

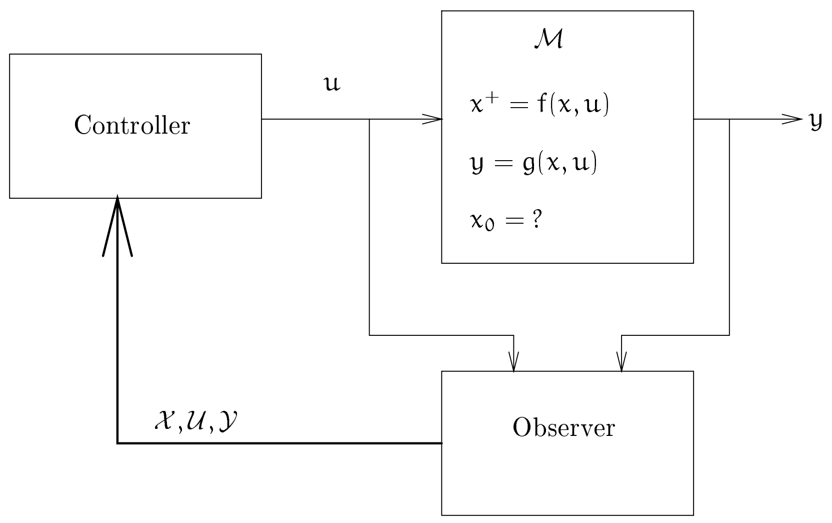


Figure 11.14: Structure of a COCOLOG controller.

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 the set of possible states $\mathcal{X}$ in which the present state of the system is included, from samples of the inputs $\mathcal{U}$, outputs $\mathcal{Y}$ and the model f, g. 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 11.14.

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 no conditions C_i 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 construction we will always get an unambiguous control action from the observed inputs and outputs.

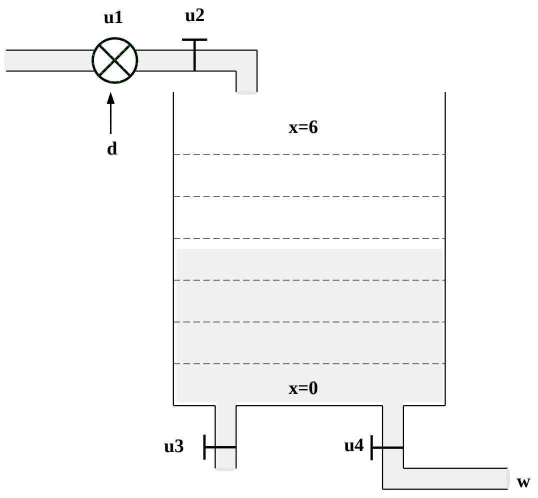


Figure 11.15: The Tank

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 in COCOLOG are 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 instances up to 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:

[5]This can be a delicate matter of synthesis by hand.

- Logic axioms, defining the standard set of axiom schemata for the first order logic.
- Equality axioms, defining equality and substitution axioms like $\mathtt{Eq}(t, t') \rightarrow \mathtt{Eq}(t', t)$ which defines the symmetric property of equality.
- Finite arithmetic axioms, defining arithmetical operations like $+$ and $-$ on a subset of symbols $\{0, \dots, N\}$ 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., axioms for computing the set of reachable states of a model.

The computations in COCOLOG are performed by the *automatic theorem prover* Blitzensturm [19] 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 [113]. 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 [113].

11.3.2 Modeling the Tank with COCOLOG

We will in this section model the tank in Figure 11.15 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 the previous sections, we can express the rules of the dynamics as

$$\mathtt{IF}\ \alpha \wedge \neg(x{=}6)\ \mathtt{THEN}\ x^+ = x + 1 \tag{11.19}$$

$$\mathtt{IF}\ \beta \wedge \neg(x{=}0)\ \mathtt{THEN}\ x^+ = x - 1 \tag{11.20}$$

$$\alpha = u_2 \wedge ((d = 2) \vee u_1 \wedge \neg(d{=}0)) \tag{11.21}$$

$$\beta = u_3 \vee (u_4 \wedge w) \tag{11.22}$$

From the description of the dynamics above we can define the state transition functions as

$$\begin{aligned} \Phi(x, \alpha) &= \alpha \wedge \neg \mathtt{Eq}(x, 6) \rightarrow x +_6 1 \\ \Phi(x, \beta) &= \beta \wedge \neg \mathtt{Eq}(x, 0) \rightarrow x -_6 1 \end{aligned} \tag{11.23}$$

where the operations $+_6, -_6$ will be defined below.

The set of state transition axioms are then defined as

$$\begin{aligned} AXM^{dyn} \equiv \{ & Eq(\Phi(x,\alpha),x^+), \\ & Eq(\Phi(x,\beta),x^+), \\ & \alpha \leftrightarrow u_2 \wedge (Eq(d,2) \vee u_1 \wedge \neg Eq(d,0)), \\ & \beta \leftrightarrow u_3 \vee (u_4 \wedge w)\} \end{aligned} \tag{11.24}$$

The axioms in (11.24) capture all information in the system and therefore (11.24) is the complete model in a sense. But, as mentioned above, more axioms must 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

$$I_6 = \{0, 1, 2, \ldots, 5, 6, 6+1\} \tag{11.25}$$

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 given by the following addition and subtraction operations:

$$a +_6 b = \begin{cases} a+b & \text{if } a+b \leq 6 \\ 6+1 & \text{if } a+b > 6 \end{cases} \tag{11.26}$$

$$a -_6 b = \begin{cases} a-b & \text{if } a-b \geq \text{ and } a \neq 6+1 \text{ and } b \neq 6+1 \\ 6+1 & \text{if } a-b < 0 \text{ or if } a = 6+1 \end{cases} \tag{11.27}$$

11.3.3 Summing up

COCOLOG is a framework based on theorem proving in the first order logic domain. The COCOLOG approach to control is based on a conditional observer and controller 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 relational approach. If we for example need the conditional relation $>$ defined for a set of integers, we will have to define this operation axiomatically in both COCOLOG and the relational approach.

The most important advantages of the relational approach are the ability of reducing complexity and that it can be supported by efficient software tools.

A Note on Notation

The notation used in this thesis follows the notation from [40], and the standard in commutative algebra and theory for DEDS.

Symbols

Algebra, Set Theory

k	General field.
$\mathbb{F}_q$	Finite field.
$\mathbb{B}$	The Boolean values $\{\mathtt{true}, \mathtt{false}\}$.
$\mathbb{I}, \mathbb{Z}$	Integers
$. \ \mathbb{Z}_+$	Positive integers, > 0.
$\mathbb{Z}_n$	The n first integers $0, \dots, n-1$.
R	Ring.
R	Relation.
$\mathcal{R}$	Relation set.
I	Ideal.
Z	Set of variables.
$k[\mathsf{Z}]$	General polynomial ring.
$\mathbb{F}_q[\mathsf{Z}]$	Polynomial over finite field $\mathbb{F}_q$.
$\mathbb{R}_q[\mathsf{Z}]$	Quotient polynomial ring.
$\mathcal{U}_n$	Universe of discourse.
$[e_1, e_2, \dots, e_n]$	Element in a set or relation.

The Relational Framework and Applications

x	General variable, State variable.
y	Output variable.
u	Input variable.
z	System variable.
z^+	Next state variable.
$M(z, z^+)$	Relational model.
$x^+ = f(x, u)$	Explicit transition function.
$R(x^+, x, u)$	Implicit transition relation.
$\ll, \ldots, \gg$	Trajectory.
$\mathsf{IMAE}(f)$	Inner most atomic expression.
$\mathsf{ULoops}[M](z)$	Upper approximation of loops.
$\langle g_1, \ldots, g_n \rangle$	The ideal generated by $g_1, \ldots, g_n$.
ϕ	Net flow in the tank system.
$J(x)$	Weighting function.

Ramadge-Wonham

Σ	Set of events (the alphabet).
L	Formal language.
$\overline{L}$	Prefix closure
δ	Transition function of finite automata.

Structured operational semantics, Compiler

σ	Binding environment.
Θ	Configuration.
$\mapsto$	Transition.
$\bar{\mathbb{I}}$	Vector space of *bit-represented integers*

Operators and Functions

Algebra, Set Theory

$\cup$	Union
$\cap$	Intersection
$\overline{S}$	Set complement.
$\setminus$	Set minus
$\times$	Cross product.
$\pi_S(\mathcal{R})$	Projection of $\mathcal{R}$ on the set S.
$\epsilon_S(\mathcal{R})$	Embedding, $\mathcal{R} \times S$.
$\wedge$, $\vee$, $\neg$, $\leftrightarrow$, $\rightarrow$	Relational and logic operators.
$\exists$, $\forall$	Existential and universal quantification.
$A \trianglelefteq k[Z]$	A is an ideal in the polynomial ring $k[Z]$.
$V(I)$	The variety of the ideal I.
deg	Degree of polynomial.
lt	Leading term.
$p \xrightarrow[F]{} r$	The polynomial p is reduced to r w.r.t. the polynomial set F.
$S(g_1, g_2)$	THe S-polynomial of g_1 and g_2.
$\lambda_q^r(z)$	The lambda polynomial.
Λ_q	The lambda polynomial set.
$\longrightarrow$	Rule.

The Relational Framework and Applications

$\mathsf{Pick}(R(x,y),x)$	Returns a random solution of $R(x,y)$ for variable x.
$f(x) == g(x)$	Returns the relation for the equation $f(x) = g(x)$.
$\parallel$	Synchronous product of relational models.
$\parallel_A$	Asynchronous product of relational models.
$\parallel_I$	Interleaved product of relational models.
$\mathsf{Restrict}(M(z,z^+), R(z))$	Restricts the models behavior by $R(z)$.
$\Gamma_k^+[M, I]$	Forward reachable states model M in k steps or less form initial set I.
$\Gamma_k^-[M, I]$	Backward reachable states model M in k steps or less form initial set I.

$\gamma_k^+[M, I]$	Forward reachable states model M in exactly k steps form initial set I.
$\gamma_k^-[M, I]$	Backward reachable states model M in exactly k steps form initial set I.
$\mathsf{Verify}(M, f)(z)$	Performs dynamic verification of temporal logic formula f, on the model M.
EX, AX, EU, AU, EF, AF, EG, AG	Temporal logic operators.
GB	General Gröbner basis.
GB_q	Gröbner basis over $\mathbb{R}_q[Z]$.
ite(F, G, H)	The *if-then-else* operator.
$\mathsf{key}(F, G, H)$	Hash function.
`BDDTLEvaluate[`$M(z, z^+)$`,f]`	*Mathematica* command implementing $\mathsf{Verify}(M(z, z^+), f)$.

Structured operational semantics

$[\![\sigma(x_i)]\!]$	Gives the value of x_i from the binding environment σ.
$\sigma[x_i \mapsto \eta_i]$	Sets the value of x_i to η_i.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] S. B. Akers. Functional testing with binary decision diagrams. In *Eighth Annual Conference on Fault-Tolerant Computing*, pages 75–82, 1978.

[3] J. Allen, J. Hendler, and A. Tate. *Readings in Planning.* Morgan Kaufmann, San Mateo, CA, 1990.

[4] D. Arnon. A bibliography of quantifier elimination for real closed fields. *J. Symbolic Comput.*, 5(1-2):267–274, 1988.

[5] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: an Algebra for Discrete Event Systems.* John Wiley & Sons, 1992.

[6] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, second edition, 1990.

[7] 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.

[8] 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.

[9] F. Boussinot and R. de Simone. The ESTRELLE language. another look at real time programming. *Proceedings of IEEE*, 79(9):1293–1304, 1991.

[10] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[11] 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.

[12] B. A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. Technical Report 9210, System Control group, Department of Electrical and Comp. Engineering., University of Toronto, August 1992.

[13] 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.

[14] 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.

[15] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[16] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, July 1992.

[17] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *International Conference on Computer-Aided Design (ICCAD)*, November 1995.

[18] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *32nd Design Automation Conference*, June 1995.

[19] 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.

[20] 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.

[21] C. G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis*. Irwin, 1993.

[22] 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. DES94; Ecole de Mines de Paris; Inria, Springer-Verlag, June 1994.

[23] CCITT. CCITT specification and description language (SDL), 1993. ITU-T Recommendation Z.100.

[24] CEI-IEC. Preparation of function charts for control systems. Standard 848, IEC, 1988. First edition.

[25] 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.

[26] 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.

[27] T. L. Claus Lewerentz. *Formal Development of Reactive Systems, Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[28] 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. DES94; Ecole de Mines de Paris; Inria, Springer-Verlag, June 1994.

[29] 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.

[30] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer-Verlag, 1992.

[31] J. Davenport, Y. Siret, and E. Tournier. *Computer Algebra. Systems and Algorithms for Algebraic Computation*. Academic Press, 1988.

[32] D. N. Dyck and P. E. Caines. The logical control of an elevator. *IEEE Transactions on Automatic Control*, 40(3):480–486, March 1995.

[33] M. Fabian. *On Object Oriented Nondeterministic Supervisory Control*. PhD thesis, Chalmers University of Technology, 1995.

[34] E. D. Fabricius. *Modern Digital Design and Switching Theory*. CRC Press, 1992.

[35] R. Germundsson. Basic results on ideals and varieties in finite fields. Technical Report LiTH-ISY-I-1259, Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, September 1991.

[36] R. Germundsson. Logic proofs $\equiv$ ideal inclusions. Technical Report LiTH-ISY-I-1286, Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, November 1991.

[37] R. Germundsson. A polynomial view of logics. Technical Report LiTH-ISY-I-1301, Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, November 1991.

[38] R. Germundsson. Analysis of polynomial dynamical systems over finite fields. Technical Report LiTH-ISY-I-1347, Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, March 1992.

[39] 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, S-581 83 Linköping, Sweden, May 1993. Lecture notes for a seminar at ITM, May 14, 1993.

[40] R. Germundsson. *Symbolic Systems - Theory, Computation and Applications.* PhD thesis, Linköping University, September 1995.

[41] 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, S-581 83 Linköping, Sweden, June 1994.

[42] 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. Available through ftp at ftp://ftp.control.isy.liu.se/pub/Reports/1995/1718.ps.Z.

[43] 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.

[44] A. Giua and F. DiCesare. Petri net structural analysis for supervisory control. *IEEE Transactions on Robotics and Automation*, April 1994.

[45] 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.

[46] P. Glasserman. *Gradient Estimation via Pertubation Analysis.* Kluwer, 1991.

[47] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, second edition, 1994.

[48] R. P. Grimaldi. *Discrete and Combinatorial Mathematics.* Addison Wesley, second edition, 1989.

[49] J. Gunnarsson. On modeling of discrete event dynamic systems, using symbolic algebraic methods. Licentiate Thesis LiU-TEK-LIC-1995:34, Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, June 1995.

[50] J. Gunnarsson. Algebraic methods for discrete event systems - a tutorial. In *Workshop on Discrete Event Systems*. IEE, August 1996.

[51] J. Gunnarsson. Algebraic methods for discrete event systems - a tutorial. Technical Report LiTH-ISY-R-1906, Department of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, October 1996.

[52] J. Gunnarsson. Symbolic algebraic discrete systems - applied to the JAS 39 fighter aircraft, part ii. Technical Report LiTH-ISY-R-1873, Department of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, August 1996. Available through ftp at ftp://ftp.control.isy.liu.se-/pub/Reports/1996/1873.ps.Z.

[53] J. Gunnarsson and R. Germundsson. Dynamic verification of a large discrete system. In *Proc. of 35th IEEE Conference on Decision and Control*, Kobe, Japan, December 1996. IEEE.

[54] J. Gunnarsson and R. Germundsson. Dynamic verification of a large discrete system. Technical Report LiTH-ISY-R-1907, Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, October 1996.

[55] J. Gunnarsson and J. Plantin. Control law synthesis for a discrete event system. Technical Report LiTH-ISY-R-1651, Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, August 1994. Available electronically at ftp://ftp.control.isy.liu.se/pub/Reports/1994/1651.ps.Z.

[56] J. Gunnarsson and J. Plantin. Automatic synthesis for simultaneous supervision and control – a first example. In *American Control Conference*, volume 5, pages 3157–3162. The American Automatic Control Council, IEEE, June 1995.

[57] J. Gunnarsson and J. Plantin. Synthesis of a discrete system using algebraic methods. In *Workshop on Discrete Event Systems*. IEE, August 1996.

[58] J. Gunnarsson and J. Plantin. Synthesis of a discrete system using algebraic methods. Technical Report LiTH-ISY-R-1905, Department of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, October 1996.

[59] J. Gunnarsson, J. Plantin, and R. Germundsson. Verification of a large discrete system using algebraic methods. In *Workshop on Discrete Event Systems*. IEE, August 1996.

[60] J. Gunnarsson, J. Plantin, and R. Germundsson. Verification of a large discrete system using algebraic methods. Technical Report LiTH-ISY-R-1904, Department of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, October 1996.

[61] 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 Issue on Real Time Programming.

[62] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[63] M. Heymann and G. Meyer. An algebra of discrete event processes. *NASA Technical Memorandum*, 1991.

[64] Y. Ho and X. R. Cao. *Pertubation Analysis of Discrete Event Dynamic Systems.* Kluwer, 1991.

[65] 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. DES94; Ecole des Mines de Paris; INRIA, Springer-Verlag, 1994.

[66] 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.

[67] 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.

[68] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[69] 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.

[70] IEEE. IEEE standard VHDL language reference manual, 1993. IEEE Std 1076-1993.

[71] G. Kahn. Natural semantics. In *Proc. STACS'87*, number 247 in Lecture Notes in Computer Science, pages 22–39. Springer-Verlag, 1987.

[72] D. Karnopp, R. Rosenberg, and D. Margolis. *System dynamics - A unified approach.* John Wiley & Sons, second edition, 1990.

[73] K. Karplus. Representing boolean functions with if-then-else dags. In *Computer Engineering UCSC-CRL-88-28, UC Santa Cruz*, Dec 1988.

[74] I. Klein. *Automatic Synthesis of Sequantial Control Schemes.* PhD thesis, Linköping University, 1993.

[75] B. H. Krogh. Controlled Petri nets and maximally permissive feedback logic. In *Proceedings of 25th Annual Allerton Conference.* University of Illinois, Urbana, September 1987.

[76] B. H. Krogh and L. E. Holloway. Synthesis of feedback control logic for discrete manufacturing systems. *Automatica*, 27(4):641–651, July 1991.

[77] 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.

[78] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems.* Kluwer Academic Publishers, 1995.

[79] M. Le Borgne, A. Benveniste, and P. Le Guernic. Polynomial ideal theoretic methods in discrete events and hybrid dynamical systems. In *Proceedings of the 28th IEEE Conference on Decision and Control*, pages 2695–2700, 1989.

[80] M. Le Borgne, A. Benveniste, and P. Le Guernic. Dynamical systems over Galois fields and DEDS control problems. In *Proceedings of the 30th IEEE Conference on Decision and Control*, pages 1505–1510, December 1991.

[81] 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.

[82] 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.

[83] F. Lin and W. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(2):173–198, 1988.

[84] Logikkonsult NP AB. Using NP-circuit to model and verify full-scale systems. Preprint, 1994.

[85] D. M. Lyons and A. J. Hendriks. Planning. In *Encyclopedia of Artificial Intelligence* [105], pages 1171–1181.

[86] J.-C. Madre and J.-P. Billon. Proving circuit correctness using formal comparison between expected and extracted behavior. In *Proc. 25th Design Automation Conference*, pages 205–210, 1988.

[87] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophysics*, 5:115–133, 1943.

[88] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1987.

[89] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[90] G. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[91] B. Mishra. *Algorithmic Algebra.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[92] E. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton University Press, 1956.

[93] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[94] 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.

[95] J. S. Ostroff and W. M. Wonham. A framework for real-time discrete-event control. *IEEE Transactions*, 35(4):386–397, 1990.

[96] H. M. Paynter. *Analysis and design of egineering systems.* MIT Press, Cambridge Mass., 1961.

[97] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[98] C. Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, 1962.

[99] 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.

[100] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, Aarhus, Denmark, September 1981.

[101] 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.

[102] 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.

[103] 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.

[104] P. J. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[105] S. C. Shapiro. *Encyclopedia of Artificial Intelligence*, volume 2. Wiley, New York, NY, 2nd edition, 1992.

[106] R. S. Sreenivas. A note on deciding the controllability of a language $\mathtt{k}$ with respect to a langauge $\mathtt{l}$. *IEEE Transactions on Automatic Control*, pages 658–662, April 1993.

[107] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *1990 IEEE International Conference on Computer-Aided Design*, November 1990.

[108] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag, 1980.

[109] J.-E. Strömberg. Styrning av lego-bilfabrik. Technical report, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 1991. Manual for control laboratory session.

[110] J.-E. Strömberg. *A Mode Switching Modelling Philosophy.* PhD thesis, Dept. of Electrical Engineering, Linköping University, 1994.

[111] 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. DES94; Ecole de Mines de Paris; Inria, Springer-Verlag, June 1994.

[112] VERILOG. *ASA, Automata and Structured Analysis, Reference Manual.* VERILOG SA, October 1991.

[113] 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.

[114] S. Wolfram. *Mathematica. A System for Doing Mathematics by Computer.* Addison-Wesley, second edition, 1991. ISBN 0-201-51502-4.

[115] S. Wolfram. *The Mathematica Book.* Wolfram Media, third edition, 1996. ISBN 0-9650532-02.

[116] 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.

[117] B. P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models.* Academic Press, Boston Mass., 1990.

[118] 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.

Subject Index