Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Regelungstechnik
Prof. Dr.-Ing. G. Roppenecker Prof. Dr.-Ing. Th. Moor

Bachelor Thesis

BDD based Controller Synthesis

A Software Implementation

vorgelegt als
Bachelor Thesis von

Rainer Hartmann

Betreuer: Prof. Dr.-Ing. Th. Moor

Ausgabedatum: 15.11.09

Abgabedatum: 01.04.10

Rainer Hartmann

BDD basierte Reglersynthese: eine Software Implementierung.

Aufgabenstellung:

Ein Hindernis, welches dem modellbasierten Reglerentwurf für Anwendungen realistischer Komplexität entgegensteht, ist der mit der Komponentenanzahl exponentiell anwachsende Zustandsraum. Laut jüngster Literatur sei die Darstellung von Automaten als Binary Decision Diagram (BDD) geeignet, dieses Hindernis zu umgehen.
Aufgabe ist es, aktuelle Forschungsergebnisse der Regelungstheorie anhand einfacher Beispiele zu erschließen und eine Softwareumgebung zu schaffen, die diese umsetzt und weiterführende Experimente ermöglicht.

Der im Rahmen der Bachelor-Arbeit erstellte Programmcode soll dabei die folgende Anforderungen erfüllen:

- nahtlose Integration in die libFAUDES;
- objektorientierte Programmierung in C++;
- Validierung anhand ausgewählter Beispiele.

Es wird ausdrücklich auf die „Richtlinien zur Anfertigung von Studien- und Diplomarbeiten“ hingewiesen.

Ausgabedatum: 15.11.2009

(Prof. Dr.-Ing. Th. Moor)

BDD based Controller Synthesis: A Software Implementation

Task of the Bachelor Thesis:

One of today‘s challenges for model-based controller-synthesis is the exponential growing of the state base. This is the main reason that prevents the usage on realistic systems. In recent reports using Binary Decision Diagrams (BDDs) for automata representation is said to break this barrier.

The task of this Bachelor Thesis is to create a software environment that implements the theoretic approaches and offers the possibility to make experiments with examples of higher complexity.

The source code has to be in accordance with the following rules:

- smoothly integration with the libFAUDES;
- object oriented C++ code
- validation with significant examples

In addition it has been explicitly refered to the „Richtlinien zur Anfertigung von Studien- und Diplomarbeiten“.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 01. April 2010

Rainer Hartmann

Contents

Chapter 1

Introduction

A large variety of technical processes like production lines or communication systems can be modeled as discrete event systems (DES). When discrete events appear, they release a transition in a space of finite states. For modeling large systems, they are often divided into small parts and put together by the synchronous product. By using this method one disadvantage is the exponential growing state space. In spite of increasing computational performance and cheaper memory, limits are reached very fast.

After the final presentation of a seminar at the chair of automatic control in summer 2009 I was really impressed about the possibilities of BDD-based controller synthesis. For simulating the performance results I searched for open-source code examples. Although many universities published reports about this topic, their libraries are not available under a public license. This was the initial spark for me to work on this bachelor thesis.

In this thesis I will consider a software implementation of a plug-in for the libFAUDES based on transition relations with Binary Decision Diagrams (BDDs). Symbolic methods using BDDs are used to represent relations for modeling, analysis and synthesis of DES. The advantage is that operations on a set can be calculated directly on BDDs without an explicit state traversal.

The Friedrich-Alexander University Discrete Event Systems library (libFAUDES) is a C++ library that implements data structures and algorithm for finite automata and regular languages. The library takes a control theoretic perspective as introduced by P.J. Ramadge an W.M. Wonham in the 1980's. Since then, many researchers have contributed to supervisory control theory, including extensions for hierarchical, modular and decentralized controller synthesis. The main purpose of libFAUDES is to provide a software environment for the implementation of recent approaches to the control of discrete event systems.

At the beginning as less theory as needed for understanding the basic concepts and algorithm is introduced by a short summary. If you have never heard of BDDs or discrete event systems before, please have a detailed look at the introductions from Andersen [And97] or Wonham and Ramadge [WR97]. The middle part was used for presenting how to implement these procedures for achieving C++ source code and barriers that had to be resolved. The performance and correctness had been verified and the results can be seen on the figures of chapter 4.

My special thanks go to my mentoring professor Thomas Moor for the great support during my work loke testing the code, supplying test scenarios and endless change requests. Besides this, I'm much obliged about any sort of distraction by my roommate at the "DES-Labor" Thomas Wittmann seconds before going mad while searching code bugs and his nice 'elevator' example. I also want to thank Fabian Kampfmann for proofreading the written part of this project.

Chapter 2

Synthesis of Discrete Event Systems

A discrete event is a dynamic system which reacts on specific discrete events. Ramadge and Wonham have established the modeling of discrete event systems (DES) by finite state machines (FSM) [RW87].

The task of controller synthesis is to modify the behavior of a given discrete event system in order to achieve a given set of constraints dictated by the specification. The resulting supervisor avoids strings that cause illegal conditions like blocking via dead or live locks on the automaton-behavior or physical problems like buffer overflowing in an automated manufacturing system on the other side.

The following chapter of the bachelor thesis describes the basic concepts and algorithm of the standard synthesis. After this, BDD based variants are introduced for achieving better performance results.

2.1 Introduction to ROBDDs

Before getting into detail of the algorithm, a short introduction to Binary Decision Diagrams (BDD) is needed. Bryant et al described in [Bry86] in detail the basics and operations. Please keep in mind that it is only possible to give a short introduction.

BDDs are rooted, directed, acyclic graphs for the representation of Boolean functions. They always start with a root node and have two outgoing edges for each node. One for the true and the other for the false evaluation of a variable. So each node level represents one variable. (This means that the maximum depth of levels is smaller or equal than the variable size.)

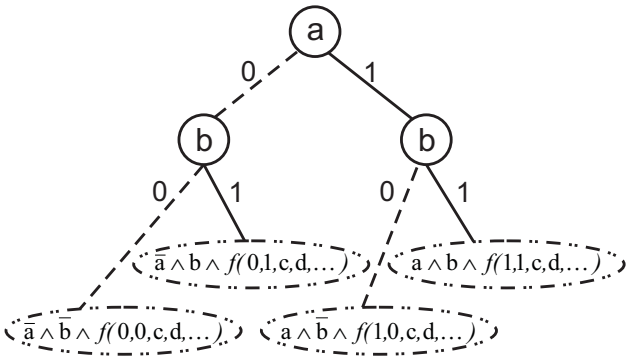

Figure 2.1: graphical representation of the Shannon expansion

Besides the decision nodes two terminal nodes called 0-terminal and 1-terminal are existing. These terminals represent the return value of a function. If the ordering of the variables is fixed the diagram is called Ordered Binary Decision Diagram (OBDD).

One of the most common ways building a BDD out of a Boolean Function is the classical Shannon expansion:

$$\begin{aligned} f(a,b,c,\ldots)) &= (a \wedge f(1,b,c,\ldots))) \vee (\overline{a} \wedge f(0,b,c,\ldots))) = \\ &= \big(a \wedge \big((b \wedge f(1,1,c,d,\ldots))) \vee \big(\overline{b} \wedge f(1,0,c,d,\ldots))\big)\big) \\ &\qquad \vee \big(\overline{a} \wedge \big((b \wedge f(0,1,c,d,\ldots))) \vee \big(\overline{b} \wedge f(0,0,c,d,\ldots))\big)\big)\big) \end{aligned} \quad (2.1)$$

The Shannon expansion has to be recursively applied until every variable is eliminated. The graphical representation can be seen in figure 2.1.

Another possibility is to set up a truth table for the Boolean function. These tables show the return value of the function for each possible variable evaluation.

For the function $f(a,b,c) = b \vee c$ (see figure 2.2 for the truth-table and BDD) 2^3 input vectors are existing. The BDD can be constructed by inserting a path for each row of the table and finally connecting them with the corresponding terminal node of the return value.

For improving the memory size, applying the following rules gives a Reduced Ordered Binary Decision Diagram (ROBDD):

1. Merge any isomorphic sub graphs.

2. Eliminate any node whose two children are isomorphic.

a	b	c	$b \vee c$
0	0	0	**0**
0	0	1	**1**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

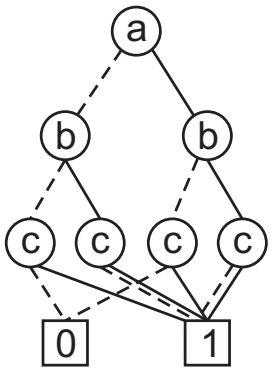

Figure 2.2: truth-table and binary decision diagram for $f(a,b,c) = b \vee c$

After applying these rules on the example function $f(a,b,c) = b \vee c$ the resulting ROBDD can be seen in figure 2.3.

The evaluation of a function is even simpler. All variables are replaced by the evaluation values. Incoming edges are connected with the target of the true-edge if the assignment of the variable is positive and otherwise. The remaining terminal node represents the return-value of the function is reached.

The existential quantification of a variable x_i for a function $f(\vec{x})$ where $1 \leq i \leq |\vec{x}|$ is defined as

$$f|_{x_i=1} \vee f|_{x_i=0}$$

Structure and size of a BDD depend extremely on the variable ordering. With a well chosen ordering, it is possible to achieve linear complexity. For the worst case the graph size grows exponential in the number of variables.

For example the output of a n-bit adder over the input variables $a_1, a_2, a_3, \ldots, a_n$ and $b_1, b_1, b_2, \ldots, b_n$ for any output-bit c_i out of $c_1, c_1, c_2, \ldots, c_n$ the BDD-representation has linear complexity for the ordering $a_1 < b_1 < a_2 < b_2 < \ldots < a_n < b_n$ and exponential growing for $a_1 < a_2 < \ldots < a_n < b_1 < b_2 < \ldots < b_n$.

On the other hand, the representation of a n-bit integer-multiplication is for every case exponential independent of the variable ordering. The problem is that an algorithm for finding the ideal variable ordering is NP-complete [GJ79].

All the binary Boolean operators on ROBDDs are implemented by the same general algorithm APPLY (op, u_1, u_2) that computes for two ROBDDs u_1 and u_2 the ROBDD of the Boolean operation u_1 op u_2.

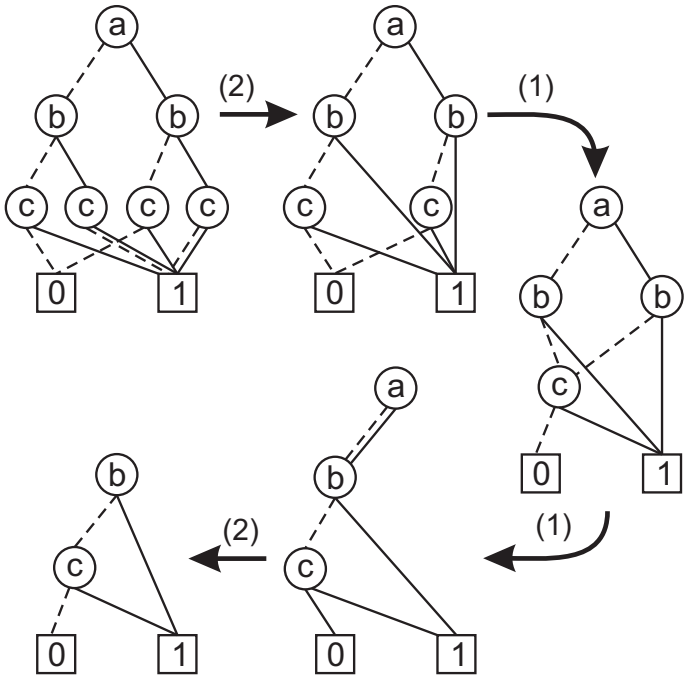

Figure 2.3: Example for the reduction rules of ROBDDs

The construction of APPLY is based on the Shannon expansion:

$$u_1(a,b,\ldots)\ \mathrm{op}\ u_2(a,b,\ldots) = (a \wedge u_1(a,b,\ldots)\ \mathrm{op}\ u_2(1,b,\ldots)) \vee (\overline{a} \wedge u_1(a,b,\ldots)\ \mathrm{op}\ u_2(0,b,\ldots))$$

The Shannon expansion has to be recursively repeated until u_1 and u_2 have reached terminal nodes (see figure 2.4). For calculating $N\&M$ 'op' has to be replaced by '&'. The new terminal nodes are 0 ($N1\&M1$),0 ($N2\&M1$),0 ($N1\&M2$) and 1 ($N2\&M2$).

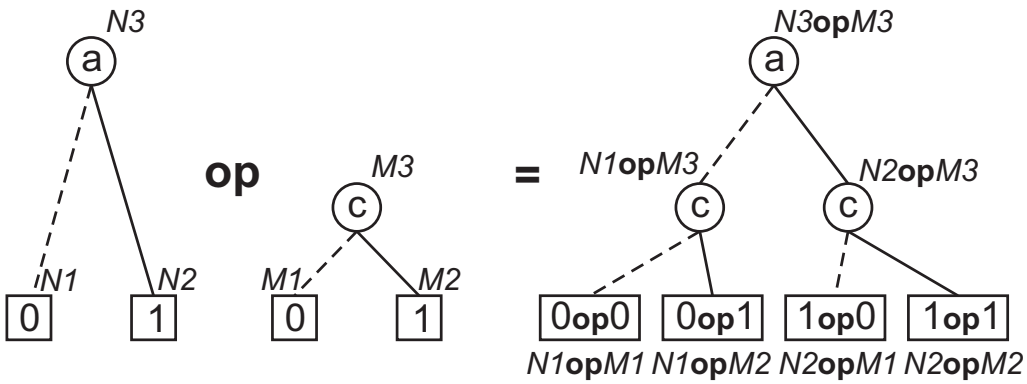

Figure 2.4: the apply operator

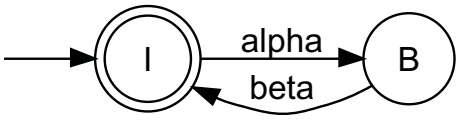

Figure 2.5: The VerySimpleMachine

2.2 Synthesis of Finite State Machines

In supervisory control theory (SCT), discrete event systems are modeled by controlled automata and their behaviors are described by formal languages [RW87].

The task of the supervisor is to disable as less transitions of the plant as possible until the system is restricted to a specified language. This means that only input strings which are part of the language are allowed.

This language consists of a subset of all possible events. (It is possible that our plant is only part of a large system and reacts only on some events.) All those events can either be selectively disabled (then they are called controllable, for instance pressing a button by the user) or uncontrollable (e.g. the appearance of an error) because the system has no influence on them.

The target of a good synthesizing algorithm is to allow the largest possible, the so called supremal controllable, sublanguage.

A FSM is defined as a quintuple:

$$G = \langle Q, \Sigma, \delta, q_i, Q_m \rangle$$

where Q is the set of possible states, Σ the set of all controllable and uncontrollable events, the transition function $\delta : Q \times \Sigma \rightarrow Q$, the initial state $q_i \in Q$ and the set of all marked states $Q_m \in Q$.

The transition function $\delta(q, \sigma)$ gives us for the actual state $q \in Q$ and an event $\sigma \in \Sigma$ the next state. If the event is not possible, an empty string is returned.

To guarantee a compact notation a supervisor G_{name} is always associated with its subsets Q_{name},Σ_{name},δ_{name},$q_{i,\text{ name}}$ and $Q_{m,\text{ name}}$.

As example scenario the VerySimpleMachine (figure 2.5) has been created. The machine can process one work piece at a time. The two events alpha and beta indicate the beginning and the end of the machine processing a work piece.

The transition function of the VerySimpleMachine returns for the state q_0 and the event $alpha$ the next state q_2.

For the alphabet Σ we have the partition

$$\Sigma = \Sigma_c \cup \Sigma_{uc}$$

where Σ_c is the subset of controllable and Σ_{uc} the subset of uncontrollable events.

The event 'alpha' of the VerySimpleMachine is controllable and 'beta' is uncontrollable.

$$\Sigma_c = \{\text{alpha}\}$$

$$\Sigma_{uc} = \{\text{beta}\}$$

2.2.1 Linguistic Preliminaries

The next step is to guarantee that the supervisor is reduced on reachable states, controllable and for some cases additionally nonblocking. Therefore the language $L(G)$ of an automaton G is introduced.

A sequence of events taken out of the alphabet forms a 'word' or 'string'. A string consisting of no events is called empty string ε. The length $|s|$ of a string s is the number of events contained in it. Multiple occurrences of one event are of course possible and are counted by the number of occurrences.

A language L defined over an event set Σ is a set of finite-length strings formed from events in Σ.

The connection between languages and automata is easily made by inspecting the finite state machine of an automaton.

$$L = \mathcal{L}(G) = \{s \in \Sigma^* : \delta(q_0, s) \text{ is defined } \}$$

Σ^* is the set of all strings formed by the event set Σ.

The marked language L_m is the set of all strings leading into a marked state, if the starting point in the initial state q_0 .

$$L_m = \mathcal{L}_m(G) = \{s \in \Sigma^* : \delta(q_0, s) \in Q_M\}$$

For event set $\Sigma = \{\alpha, \beta\}$ the VerySimpleMachine describes the language

$$L_{VerySimpleMachine} = \{\varepsilon, \alpha, \alpha\beta, \alpha\beta\alpha, \alpha\beta\alpha\beta, \ldots\}$$

$$L_{M,VerySimpleMachine} = \{\varepsilon, \alpha\beta, \alpha\beta\alpha\beta, \ldots\}$$

For a string $s \in \Sigma^*$ a string $t \in \Sigma^*$ is a prefix if $\exists o \in \Sigma^* : to = s$ and $t \leq s$.

The closure $\overline{L}$ of a language L is the set of all prefixes of L.

$$\overline{L} = \{t \in \Sigma^* | \exists s \in L : t \leq s\}$$

2.2.2 Synchronous Product

It is very popular to split up the plant (describes the behavior of the machine) and the specification (the forced behavior of the machine) in small parts. In general a big machine can often be regarded as a summation of indepentently working small machines.

The generator of the whole plant is calculated by the synchronous product:

$$G_{plant} = G^1_{plant} \parallel G^2_{plant} \parallel \ldots \parallel G^N_{plant}$$

and

$$G_{specification} = G^1_{specification} \parallel G^2_{specification} \parallel \ldots \parallel G^N_{specification}$$

The Synchronous Product for $G_1 = \langle Q_1, \Sigma_1, \delta_1, q_{0,1}, Q_{m,1} \rangle$ and $G_2 = \langle Q_2, \Sigma_2, \delta_2, q_{0,2}, Q_{m,2} \rangle$ is defined as:

$$G_1 \parallel G_2 = \left\langle Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{parallel}, (q_{0,1}, q_{0,2}), Q_{m,1} \times Q_{m,2} \right\rangle \tag{2.2}$$

$$\delta_{parallel}((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \\ (\delta_1(q_1, \sigma), q_2) & \text{if } \sigma \in \Sigma_1 \wedge \sigma \notin \Sigma_2 \\ (q_1, \delta_2(q_2, \sigma)) & \text{if } \sigma \notin \Sigma_1 \wedge \sigma \in \Sigma_2 \end{cases}$$

In the scenario both VerySimpleMachines are arranged in a row (figure 2.6). The plant can be calculated by the parallel composition of two automata (figure 2.7).

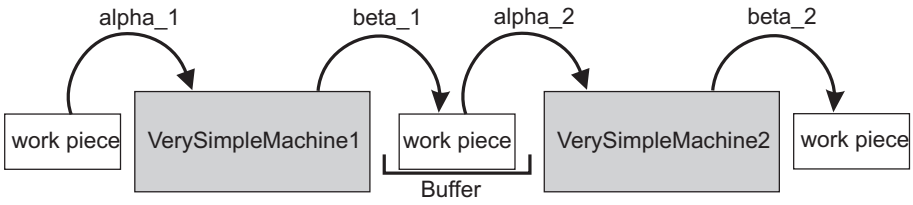

Figure 2.6: Two VerySimpleMachines arranged in a row

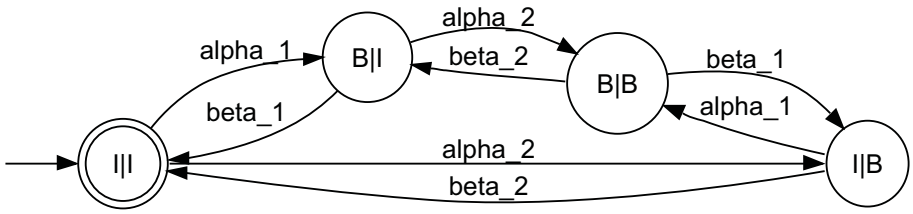

Figure 2.7: The Synchronous Product of two VerySimpleMachines

The two machines shall be arranged with a buffer to implement a two-stage production process. Each work piece must first be processed by SimpleMaschine1, then placed in the buffer, and finally processed by SimpleMaschine2. The buffer capacity is restricted to one work piece. Both plant models act independently. The buffer imposes a condition regarding the interaction of both machines and is modeled as specification, where the event beta_1 fills the buffer and alpha_2 empties the buffer (figure 2.8).

For the calculation of the supervisor the first step is to calculate the synchronous product (figure 2.9) of the plant and the specification as described in equation 2.2.

2.2.3 Reachability Analysis

The standard reachability algorithm is based on a breath-first traversal [CBM90] of finite-state machines. It considers all states and transitions reachable from the initial state q_0. The execution process picks up for each step all states the can be reached from set of reachable states in one step and pushes them on the set of reachable states. The end is reached when the set of new reachable states is empty.

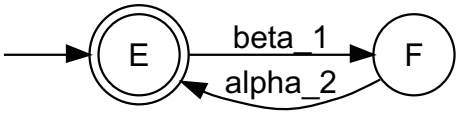

Figure 2.8: The specification for the VerySimpleMachine

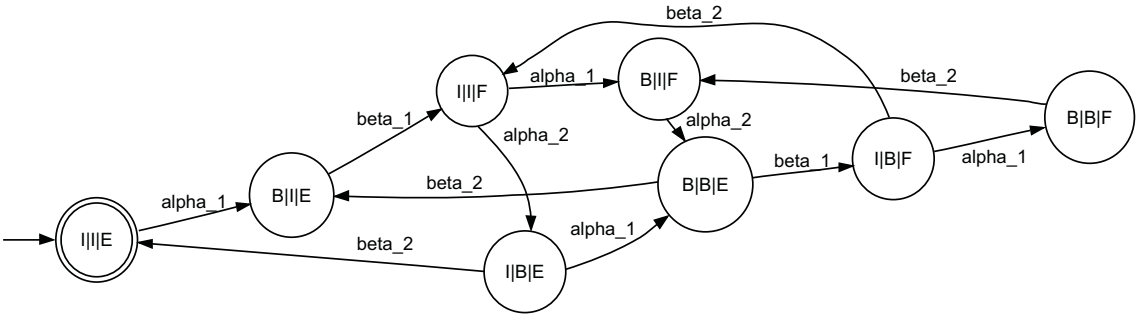

Figure 2.9: The Synchronous Product of Plant and Specification

```
1  ReachabilityAnalysis(G)
2  {
3   k = 0
4   Q_Rk = q_0
5   repeat
6   {
7    k++;
8    Q_Rk = Q_Rk-1 ∪ {q' ∈ Q : (∃q ∈ Q_Rk-1 ∧ ∃σ ∈ Σ : q' = δ(q,σ))}
9   }
10  until( Q_Rk == Q_Rk-1)
11  Q = Q_Rk
12 }
```

2.2.4 Supremal controllable Sublanguage

Supremal controllable sublanguages have been shown to play an important role in supervisor synthesis. Hence, the formal synthesis problem for a supervisor is posed as synthesizing the largest possible controllable and observable sublanguage of a specified language.

Supremal controllable sublanguages were discussed in [WR87] and a recursive algorithm was developed to compute a supervisor out of a given language (e.g. modeled as a specification-generator).

We call our supervisor S controllable if for every state uncontrollable events are not forbidden.

$$\mathcal{L}(K)\Sigma_u \cap \mathcal{L}(G) \subseteq \mathcal{L}(K)$$

where K is the implementation of a supervisor and $\mathcal{L}(K)$ its language.

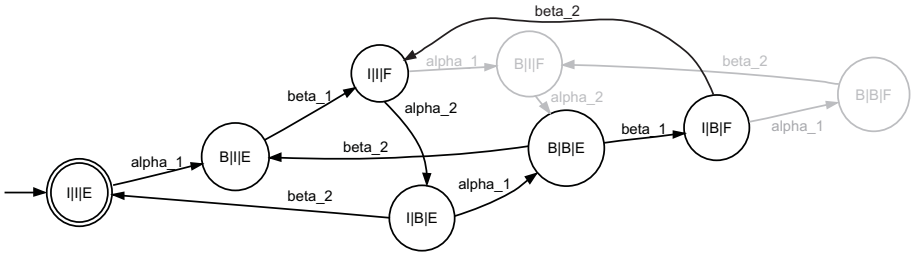

Figure 2.10: The Supervisor

```
1  Controllability(G_plant, G_plant||spec)
2  {
3    Q_problem = {(q,a) ∈ Q_plant||spec : (∃σ ∈ Σ_uc : δ_plant(q,σ) ≠ ∅ ∧ δ_plant||spec((q,a),σ) = ∅)}
4
5    while(Q_problem != ∅)
6    {
7      remove all transitions δ_plant||spec,i with δ_plant||spec((q̃,ã),σ̃) ∈ Q_problem
8
9      Q_problem = {(q,a) ∈ Q_plant||spec : (∃σ ∈ Σ_uc : δ_plant(q,σ) ≠ ∅ ∧ δ_plant||spec((q,a),σ) = ∅)}
10   }
11 }
```

2.2.5 Nonblocking

A Discrete Event System is nonblocking if

$$\mathcal{L}(G) = \overline{\mathcal{L}_m(G)}$$

The following algorithm makes a given generator nonblocking:

```
1  Nonblocking(G)
2  {
3   k = 0
4   Q_NB k = Q_M
5   do
6   {
7    k++;
8    Q_NB k = Q_NB k−1 ∪ {q ∈ Q : (∃q′ ∈ Q_NB k−1 ∧ ∃σ ∈ Σ : q′ = δ(q,σ))}
9   }
10  while( Q_NB k == Q_NB k−1)
11  Q = Q_NB k
12 }
```

2.3 Representation of Discrete-Event-Systems as Binary Decision Diagrams

A major reason preventing Supervisory Control Theory from achieving acceptance in industry is the problem of exponential state explosion. A BDD is an efficient data structure which can reach logarithmic compression of the state space as soon as memory and runtime performance are improved by efficient reachability searches.

Before working on efficient operations on BDDs, the FSM of our generator has to be transformed as described in [GVSSV07].

For the finite state machine $A := \langle Q, \Sigma, \delta, q_i, Q_m \rangle$ the characteristic function is defined as $T : Q \times \Sigma \times Q \rightarrow \{0,1\}$. For the coding of N elements at least $\lceil \lg N \rceil$ binary variables are used.

- $e_Q(q), q \in Q$ is the state coding, $e_q : Q \rightarrow \mathcal{B}^n \;\; n \geq \lceil log_2(|Q|) \rceil$
- $e_\Sigma(\sigma), \sigma \in \Sigma$ is the event coding $e_\sigma : \Sigma \rightarrow \mathcal{B}^n \;\; n \geq \lceil log_2(|\Sigma|) \rceil$
- $e_{Q'}(q'), q' \in Q$ is the next state coding $e_{q'} : Q \rightarrow \mathcal{B}^n \;\; n \geq \lceil log_2(|Q|) \rceil$

The state function χ_Q, event function χ_Σ, next state function $\chi_{Q'}$ and a binary transition relation function χ_T are defined as

- $\chi_Q(q) = \begin{cases} 1 \text{ if } q \in Q \\ 0; \end{cases}$
- $\chi_\Sigma(\sigma) = \begin{cases} 1 \text{ if } \sigma \in \Sigma \\ 0; \end{cases}$
- $\chi_{Q'}(q') = \begin{cases} 1 \text{ if } q' \in Q' \\ 0; \end{cases}$

As example, the transfer-function VerySimpleMachine of figure 2.5 has been converted to a binary function. At least 2 bits are used for the events because the first event 'alpha' has the index 1. In the implementation later on, the first 10 bits are reserved for event coding. These variables have the same global index.

The following functions define the coding of the events, states and nextstates:

- $e_Q('idle') = [0,1]$
- $e_Q('busy') = [1,0]$
- $e_\Sigma('alpha') = [0,1]$
- $e_\Sigma('beta') = [1,0]$
- $e_{Q'}('idle') = [0,1]$
- $e_{Q'}('busy') = [1,0]$

$$\begin{aligned} \chi_Q(q_0,q_1) &= \vee \quad \overline{q}_1 q_0 \vee \quad q_1 \overline{q}_0 \\ \chi_\Sigma &= \vee \quad \overline{\sigma}_1 \sigma_0 \vee \quad \sigma_1 \overline{\sigma}_0 \\ \chi_{Q'}(q_0,q_1) &= \vee \quad \overline{q'}_1 q'_0 \vee \quad q'_1 \overline{q'}_0 \end{aligned} \tag{2.3}$$

$$\chi_T(q_0,q_1,\sigma_0,\sigma_1,q'_0,q'_1) = \left(\overline{q}_1 q_0 \overline{\sigma}_1 \sigma_0 q'_1 \overline{q'}_0\right) \vee \left(q_1 \overline{q}_0 \sigma_1 \overline{\sigma}_0 \overline{q'}_1 q'_0\right) \tag{2.4}$$

Figure 2.11 represents χ_T (2.4) as BDD. For the following algorithm the transition function χ_T, χ_Q respectively χ_Σ are replaced by their shortcuts T Q and Σ are used. If you compare it with the representation of the transition function of the VerySimpleMachine as function-table (2.12) the high compression rate and compact representation is obvious.

2.4 BDD-based Synthesis of DES

The next step is to transfer the algorithm from the previous section to the BDD problem. It is very important, that all events have the same binary coding and variable indexes. Before calculating the synchronous product, it is important that the variables for the state coding of both automata are free of intersection.

The variables CurrentStates, NextStates and AllEvents of the following algorithm are sets with all variables used for the respective coding.

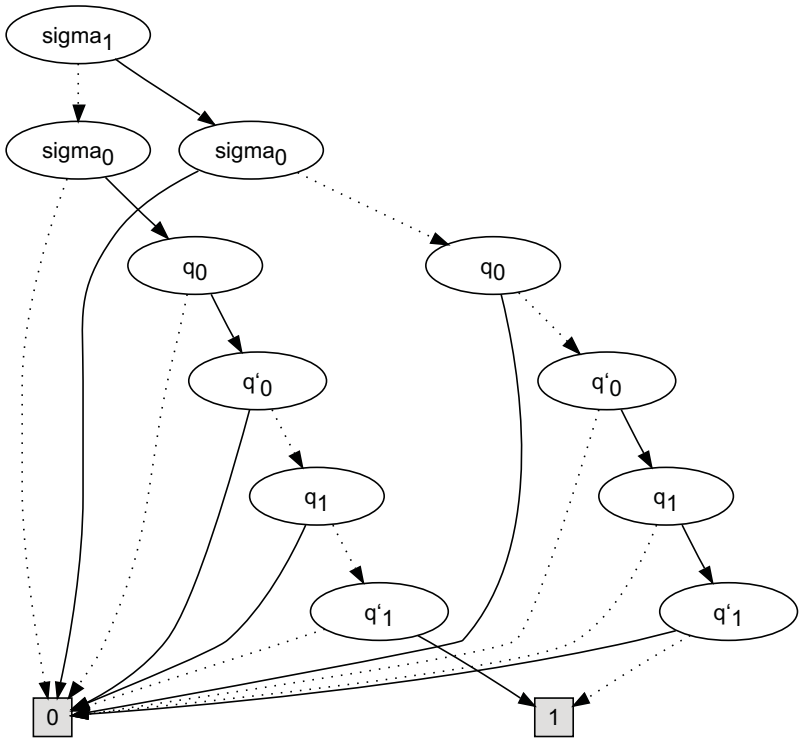

Figure 2.11: BDD of the transition relation of the VerySimpleMachine

```
parallel(G1,G2)
{
 Check if state codings are free of intersection
 If not correct by replacing variables
 T = T1 & T2
}
```

```
reachable(Q0, T)
{
 T̃ = ExistentialQuantification(T,AllEvents)

 Qreach = false
 Qreach,new = Q0

 do
 {
   Qreach = Qreach,new
   Qtemp = ExistentialQuantification(Qreach & T̃,CurrentStates)
   Qtemp = SwapVariables(Qtemp, NextStates, CurrentStates)
   Qreach,new = Qreach | Qtemp
 }
```

```
15  while(Q_reach = Q_reach,new)
16  Q = Q_reach
17 }
```

```
1  nonblocking(Q_M, T)
2  {
3  T~ = ExistentialQuantification(T,AllEvents)
4
5  Q_nb = false
6  Q_nb,new = Q_M
7
8  do
9  {
10   Q_nb = Q_nb,new
11   Q_temp = SwapVariables(Q_nb, NextStates, CurrentStates)
12
13   Q_temp = ExistentialQuantification(Q_temp & T~,NextStates)
14
15   Q_nb,new = Q_nb | Q_temp
16 }
17 while(Q_nb = Q_nb,new)
18 Q = Q_nb
19 }
```

```
1  controllable(T_plant, T_parallel, Σ_uc)
2  {
3
4  Q_problem = T_parallel - (T_plant & Σ_uc)
5  Q_problem = ExistentialQuantification(Q_problem, AllEvents & NextState)
6
7  while(Q_problem != false)
8  {
9    Q_problem = SwapVariables(Q_problem, NextStates, CurrentStates)
10   T -= Q_problem
11
12   Q_problem = T_parallel - (T_plant & Σ_uc)
13   Q_problem = ExistentialQuantification(Q_problem, AllEvents & NextStates
         )
14 }
15 do
16 }
```

2.5 Disjunctive partial transition relations

The main problem with composition of large sets of automata is that the total transition relation becomes extremely complex. [BCL91], [BLVA06] and [GV01] show that splitting the transition into a set of less complex sub relations, e.g. the disjunctive transfer relation, is a very useful method to get a significant smaller memory usage.

This can be traced back to the fact that BDDs do not grow linear by the number of satisfying variable assignments (see figure 2.13). The main reason behind the intermediate BDD size explosion is that new elements are added to the set Q_k in such pseudo-random order that its BDD cannot be reduced efficiently. For the monotonic BDDs of randomly growing set with 2^n elements the same behavior can be observed.

For a small size it is very unlike that reduction rules are applicable. The best efficiency can be reached if half of the variable assignments end in a terminal one and the rest in zero. In general the transition function is in the very beginning of the left part of the graph. A satisfaction of 100% would mean, that every state has a transition for every event with each state. Even for non-deterministic automata this is not possible.

For a partitioned representation of the transition relation BDDs are vital smaller and the sum of nodes of all partial transition functions is approximately as big as the monolithic transition function.

The reason that BDD reduction is so important relates to the time-complexity of certain BDD operations. Under normal conditions, BDD operations have a time complexity of $O(|x||y|)$, where $|x|$ and $|y|$ denotes the number of nodes in the BDDs x and y. Besides this finding a good variable ordering is still very important and can be applied to each sub-automata.

In addition this effect is amplified in several dimensions for a reachability search that utilizes breadth-first traversal.

A complex discrete event system can often be efficiently represented in modules. In this case, this means that the user models the system by small subsystems. Instead of calculating the monolithic transition relation, the splitting is used for setting up a partial transition relation. A transition relation is partitioned if it is represented as a conjunction or disjunction of terms.

The main reasons for resorting to problem decomposition when facing complex tasks are quite obvious: A 'divide-and-conquer' approach is expected to reduce memory requirements and in the best case it may also improve time efficiency. Another advantage of this

approach is that we get rid of memory limitations of traditional BDD packages.

This means that for a modular system of N finite automata

$A = A^1 \parallel A^2 \parallel \ldots \parallel A^N$

the monolithic transition relation is

$T = T^1 \wedge T^2 \wedge \ldots \wedge T^N$

The dependency set of an automaton is defined as the set of automata that shares at least one event excluding itself.

$D\left(A^i\right) = A^i \cup \left\{A^j : \Sigma^i \cap \Sigma^j \neq \emptyset\right\}$

Furthermore the dependency set of a partial transition relation is needed:

$D\left(T^i\right) = \left\{T^j : A^j \in D\left(A^i\right)\right\}$

Now a rule for calculating the i-th partial transition relation is needed and had been proven by [BLVA06]:

$$\widetilde{T}^i = \left\{\left\langle q^1 q^2 \ldots q^N, \sigma, q'^1 q'^2 \ldots q'^N \right\rangle : \sigma \in \Sigma^i \wedge \left\langle q^1 q^2 \ldots q^N, \sigma, q'^1 q'^2 \ldots q'^N \right\rangle \in T\right\}$$

The monolithic transition relation is the disjunction of all partial transition relations:

$T = \widetilde{T}^1 \vee \widetilde{T}^2 \vee \ldots \vee \widetilde{T}^N$

$$\text{Statetupels}\,(a,b) = \begin{cases} 1 \text{ if } \exists\, q \in Q : e_Q(q) = a \wedge e_{Q'}(q) = b \\ 0; \end{cases}$$

```
1  partial_parallel(G_1, G_2, ..., G_N)
2  {
3   Check if state coding are free of intersection
4   If not correct by replacing variables
5
6   for(i = 1:N)
7   {
8     T̃_i = T_i
9
10    foreach(T_j in D(T^i))
11    {
12      T_shared = T_j & Σ_i
13      T_single = Statetupels & (Σ_i - Σ_j)
14      T̃_i &= T_single | T_shared
15    }
```

```
    foreach(T_j not in D(T^i))
    {
      T̃_i &= Statetupels & Σ_i
    }
}
```

```
partial_reachable(Q_0, T[N])
{
 Q_reach = false
 Q_reach,new = Q_0

 do
 {
   Q_reach = Q_reach,new
   Q_temp = false
   for(i = 1:N)
   {
     Q_temp = ExistentialQuantification(Q_reach & T[i],CurrentStates &
         AllEvents)
   }
   Q_temp = SwapVariables(Q_temp, NextStates, CurrentStates)
   Q_reach,new = Q_reach | Q_temp
 }
 while(Q_reach = Q_reach,new)
 Q = Q_reach
}
```

```
partial_nonblocking(Q_M, T[N])
{
 Q_nb = false
 Q_nb,new = Q_M

 do
 {
   Q_nb = Q_nb,new
   Q_temp = SwapVariables(Q_nb, NextStates, CurrentStates)

   for(i = 1:N)
   {
     Q_temp = ExistentialQuantification(Q_temp & T[i],NextStates &
         AllEvents)

```

```
    Q_nb,new = Q_nb | Q_temp
  }
  while(Q_nb = Q_nb,new)
  Q = Q_nb
}
```

```
partial_controllable(Tplant[N], Tparallel[M], Σ_uc)
{
  T_crit = false
  for(i = 1: M)
  {
    T_crit |= ExistentialQuantification((T_plant[i] & Σ_uc), NextStates);
  }
  for(i = 1: N)
  {
    T_crit -= ExistentialQuantification((T_parallel[i] & Σ_uc), SpecStates &
        SpecNextStates);
  }
  Q_problem = ExistentialQuantification(T_crit, AllEvents & NextStates)

  while(Q_problem != false)
  {
    Q_problem = SwapVariables(Q_problem, NextStates, CurrentStates)

    for(i = 1: N)
    {
      T[i] -= Q_problem
    }

    T_crit = false
    for(i = 1: M)
    {
      T_crit |= ExistentialQuantification((T_plant[i] & Σ_uc), NextStates);
    }
    for(i = 1: N)
    {
      T_crit -= ExistentialQuantification((T_parallel[i] & Σ_uc), SpecStates &
          SpecNextStates);
    }
    Q_problem = ExistentialQuantification(T_crit, AllEvents & NextStates)
  }
}
```

$sigma_1$	$sigma_0$	q_0	q'_0	q_1	q'_1	f(x)
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	1	1	1	0
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	0	0
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	0
0	1	0	0	0	1	0
0	1	0	0	1	0	0
0	1	0	0	1	1	0
0	1	0	1	0	0	0
0	1	0	1	0	1	0
0	1	0	1	1	0	0
0	1	0	1	1	1	0
0	1	1	0	0	0	0
0	1	1	0	0	1	1
0	1	1	0	1	0	0
0	1	1	0	1	1	0
0	1	1	1	0	0	0
0	1	1	1	0	1	0
0	1	1	1	1	0	0
0	1	1	1	1	1	0
1	0	0	0	0	0	0
1	0	0	0	0	1	0
1	0	0	0	1	0	0
1	0	0	0	1	1	0
1	0	0	1	0	0	0
1	0	0	1	0	1	0
1	0	0	1	1	0	1
1	0	0	1	1	1	0
1	0	1	0	0	0	0
1	0	1	0	0	1	0
1	0	1	0	1	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	0
1	0	1	1	0	1	0
1	0	1	1	1	0	0
1	0	1	1	1	1	0
1	1	0	0	0	0	0
1	1	0	0	0	1	0
1	1	0	0	1	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	0	1	0	1	0
1	1	0	1	1	0	0
1	1	0	1	1	1	0
1	1	1	0	0	0	0
1	1	1	0	0	1	0
1	1	1	0	1	0	0
1	1	1	0	1	1	0
1	1	1	1	0	0	0
1	1	1	1	0	1	0
1	1	1	1	1	0	0
1	1	1	1	1	1	0

Figure 2.12: function table of the transition relation of the VerySimpleMachine

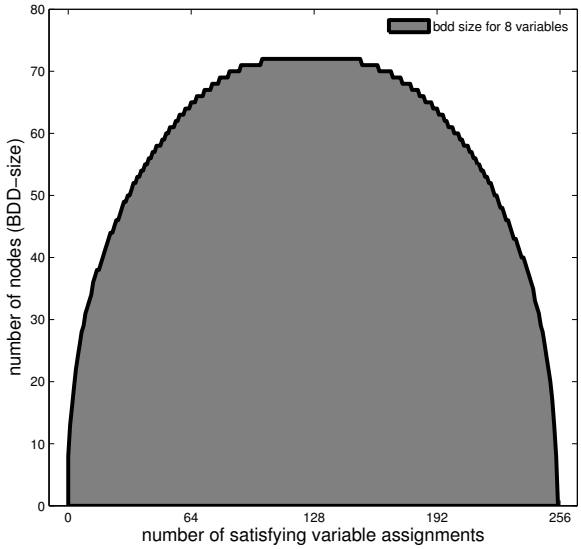

Figure 2.13: average size of a random BDD

Chapter 3

Implementation

The task of the Bachelor thesis was to create a software environment that implements the theoretic approaches and offers the possibility to make experiments with examples of higher complexity. 'libFAUDES' is a C++ library that implements data structures and algorithms for finite automata and regular languages and there was evidence to suggest creating a new plug-in for it. In this chapter the most considerable challenges and their solutions by implementing beforehand described algorithm are presented. If you are interested in a detailed description of the software have a closer look to the user reference of 'libFAUDES'. All required information can be found under 'C++ API/PlugIns/BDD Synthesis PlugIn'.

3.1 Wrapper class and BDD library

For the implementation of the basic BDD operations, standard and performance optimized libraries are used. A special wrapper class (bds_wrapper) provides the advantage to use a large area of available BDD libraries. Each of them has different advantages. So it was not evident which of them should be used. BuDDy provides a comfortable C++ interface, nice features and functions that were later on adapted on the wrapper class, examples for BDD based automaton traversing and finally a lowly constricting license. But perhaps one of the following listed (or perhaps your own implementation) is even better.

BuDDy [bud10] [AS07] [SW07] is a Binary Decision Diagram package that provides all of the mostly used functions for manipulating BDDs. The package also includes functions for integer arithmetic such as addition and relational operators. BuDDy started as a technology transfer project between the Technical University of Denmark and Bann Vi-

sualstate. The BDD package presented here was made as part of a Ph.D. project by Jorn Lind-Nielsen on model checking of finite state machines. The package has evolved from a simple introduction to BDDs to a full blown BDD package with all the standard BDD operations, reordering and a wealth of documentation. First of all a program needs to initialize the BDD package. Getting the most out of any BDD package is not always easy. It requires some knowledge about the optimal order of the BDD variables. If we allocate as much memory as possible from the very beginning, then BuDDy does not have to waste time trying to allocate more whenever it is needed. Included in the BDD package is a set of functions for manipulating values of finite domains. These functions are used to allocate blocks of BDD variables to represent integer values instead of only true and false. This concept was adapted and improved and is now generally usable for the wrapper class.

The ideal implementation for a specific application can only be found by trying out for different test cases. To make this possible, the plug-in is not restricted. All calculations are done with the wrapper-class that is based on templates and new BDD libraries can be connected with little work.

The functions of BuDDy that the wrapper class uses are listed below. More details about this package can be found in the package documentations, available at http://www.itu.dk/research/buddy.

- bdd_init
- bdd_done
- bdd_varnum
- bdd_setvarnum
- bdd_ithvar
- bdd_nithvar
- bdd_apply
- bdd_exist
- bdd_replace
- bdd_newpair
- bdd_freepair

- bdd_fprintdot
- bdd_file_hook
- bdd_nodecount

Here is a list of the most common BDD implementations. It would be a rewarding task to try out one or the other.

- *ABCD*: The ABCD package was implemented by Armin Biere at the Johannes Kepler Universität in Linz. ABCD is very small and fast.
- *CMU BDD*, BDD package by Carnegie Mellon University, Pittsburgh. Although Garbage collection, dynamic variable reordering and multi-variable quantification are supported.
- *CrocoPat*, BDD package and a high-level querying language, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
- *CUDD* [cud09]: BDD package, University of Colorado, Boulder. The package provides a large set of operations on BDDs, ADDs, and ZDDs, functions to convert BDDs into ADDs or ZDDs and vice versa, and a large assortment of variable reordering methods.
- *DDD*: A C++ library with support for integer valued and hierarchical decision diagrams.
- *JINC*: A C++ library developed at University of Bonn, Germany, supporting several BDD variants and multi-threading.
- *OBDD*: A Haskell package for OBDD

The wrapper-class itself provides the following operations:

- operator&=
- operator|=
- operator-=
- operator==

- operator!=
- operator=
- nodecount: returns the number of nodes of the BDD
- bddtrue: returns a BDD for the function f(x)=true.
- bddfalse: returns a BDD for the function f(x)=false.
- var: returns a BDD that is true for the variable 'index'.
- nvar: returns a BDD that is true for the negated variable 'index'.
- unique: unique quantification of variables.
- exist: existential quantification of variables
- replace: variable replacement
- writeGraph: make a gif out of the BDD.
- DotWrite: make a '.dot' file for the generator.
- useVar: allocate Variable

3.2 Domain

The problem was, that a transition relation consists of tuples like $\langle q_1, q_2, \ldots, \sigma, q_1', q_2', \ldots \rangle$ $q_1 \in Q_1, \ldots, \sigma \in \Sigma, q_1' \in Q_1', \ldots$.

This implies for the Boolean transition function that we need for every element of the tuple $|Q_i|$ respectively $|\Sigma|$ variables for representation.

For this case a domain class was introduced. It administrates for every sub domain the amount of variables (see figure 3.1) and provides a function to transfer a function from one domain to another. The variable size is extensible because the amount of sub domains (for each partial transition relation one sub domain is needed) and variables per subdomain (new states can be added step by step) is unknown at initialization time.

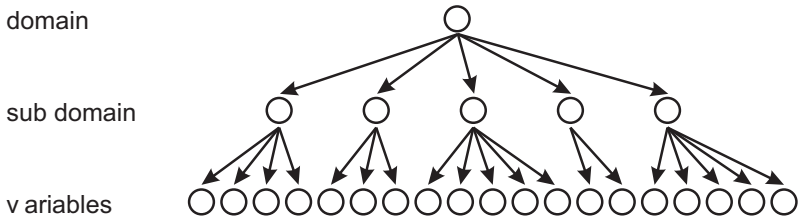

Figure 3.1: Construction of a domain

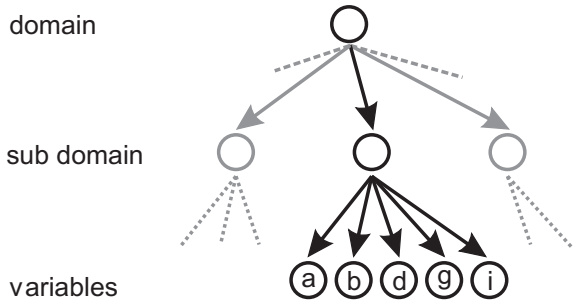

Figure 3.2: Exemplary construction of the domain

This means for instance: If we want to get the binary function that returns true for the value for a specific sub domain the member function

ithvar(uint sub domain, uint value)

at first looks up if there exist at least $\lceil log_2(value) \rceil$ variables for the sub domain. If not, variables are added. Finally the correct function is returned. For a domain like printed in figure 3.2 the value $14_{10} = 01110_2$ corresponds with the binary function $f(a,b,\ldots) = \bar{i}gdb\bar{a}$ and the BDD in figure 3.3.

3.3 bddGenerator

The bddGenerator is the most important class in the plug-in. It saves the names and indices of the states, the transition and many attributes like marked states, controllable events or initial state. It can be constructed on different ways. The most common is read a '.gen'-file. Of course there exists also an import-function to transfer a cGenerator or vGenerator to a bddGenerator.

The bddGenerator is internly build up by one or more bddPartGenerators. If a new bddGenerator is created it has always one BDDPartGenerator. It is responsible for saving the BDD for the transition-function and the state coding.

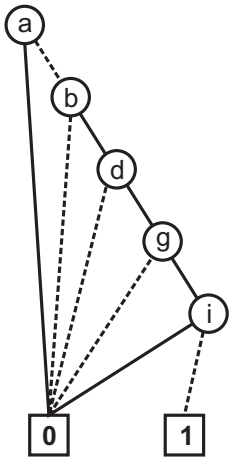

Figure 3.3: BDD of the binary function $f(a,b,\ldots) = \bar{i}gdb\bar{a}$

A bddPartGenerators is an extended partial transition relation. Beside the BDD it provides a couple of manipulative functions on the BDD. For instance replacing variables and transfer to other domains.

If you call the function parallel, the first step is to copy all bddPartGenerators from both passed bddGenerator to the resulting. Then the transition functions are transferred to the new domain. This means for example that generator1 uses variable 0 to 15 (this means 6 variables for state-coding, 3 for current and 3 for the next states and 10 variables are always reserved for the event-coding) and generator2 the variables 0 to 13 (it has at most 2^2 states).

If we would calculate the partial transition function simply by $T = T_1 \wedge T_2$ we get an intersection of the variables. Consequently, the first sub domain of the resulting generator needs 6, the second 4 variables and each variable of the old sub domain has to be transferred to the corresponding new variable. For the previous example the first variable of sub domain 1 in generator2 (index: 10) has to be replaced by the new variable with index 16. The old and new domains can be seen in figure 3.4.

If the variables have been updated in every BDD the next step is to calculate the partial transition relation. You can read this in the theory part.

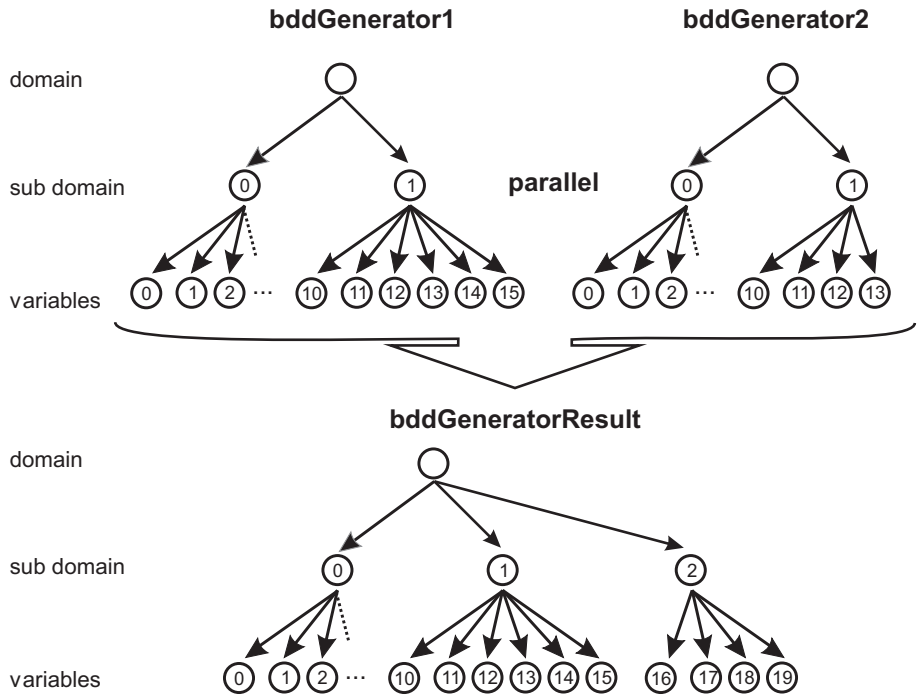

Figure 3.4: Domain before and after parallel-composition

Chapter 4

Evaluation

To put the preceding algorithm on the test, time measurements with significant examples followed the implementation. All experiments ran on a 2-GHz Intel-Core-Duo machine with a 2-Gbyte main memory.

I expected to get the best result for the parallel composition of systems with independent events. For this case, the so called 'testmachine*' (see figure 4.1) was introduced. It has three independent events 'alpha*", 'beta*' and 'gamma*'. Now it is possible to calculate synchronous products up to a magnitude of 10^{17} states in a few minutes. The synchronous product of 35 testmachines has $3^{35} \approx 5 * 10^{16}$ states and more than 10^{17} transitions. So it is obvious that a representation is only possible by symbolic methods and iterating over the state set would take hundreds of years.

As mentioned in the theory part, the calculation of the synchronous product itself has a polynomial complexity. Nevertheless the execution-time of the parallel() function is increasing exponential (see figure 4.2 for a shielded out reachability analysis we get the expected complexity. In this point, further improvements of the algorithm seem to be very promising.

The second graph (see figure 4.3) compares the 'Synthesis Plug-In' with the 'BDD-

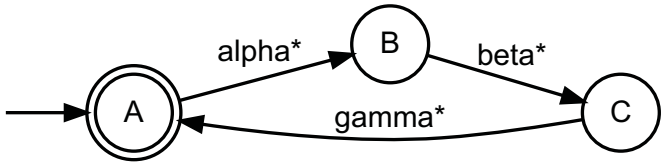

Figure 4.1: testmachine for performance measurements

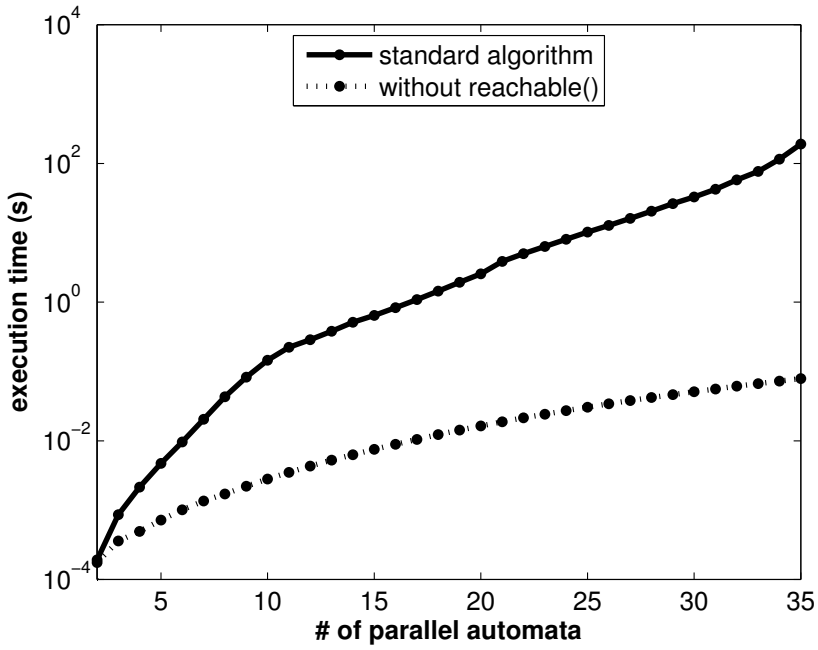

Figure 4.2: Execution time for computation of the parallel composition of up to 35 test-machines with and without reachability analysis

Synthesis Plug-In'. By now, calculations of the synchronous product of 13 systems took around 100 seconds. In the same time, the new BDD plug-in handles 33 automaton. One reason for this is the polynomial growing of the memory for the transition relation in comparison to exponential growing for the standard algorithm. The increasing node size of the BDDs is shown in figure 4.4.

In reality, it's very inconvenient to get completely independent systems. For this case 'gamma*' was changed to a global event 'gamma'. As we can see in figure 4.5 the computation is a nuance faster because of the smaller resulting transition function.

Above, I wrote about the performance advantage of partial transition relations. Unfortunately it is not possible to show the performance advantage. The problem is that there exists no monolithic implementation of the algorithm. In figure 4.6 the memory overlay of the partial transition relation is pointed out. On figure 4.7 and 4.8 the BDDs of the monolithic and partial transition function are showed. The additional memory is used for the multiple declaration of the event-variable branch and shared events. This means if an event is used in generator A and B, the partial transition functions of both automata have a true assignment for this event. For the monolithic way both branches can be reduced to one.

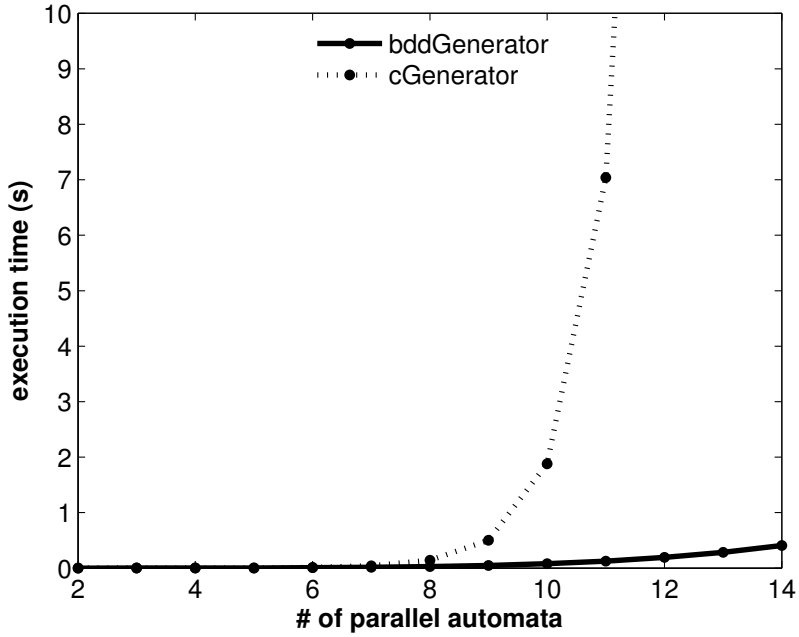

Figure 4.3: Execution time for computation of the parallel composition with the standard and BDD-based algorithm

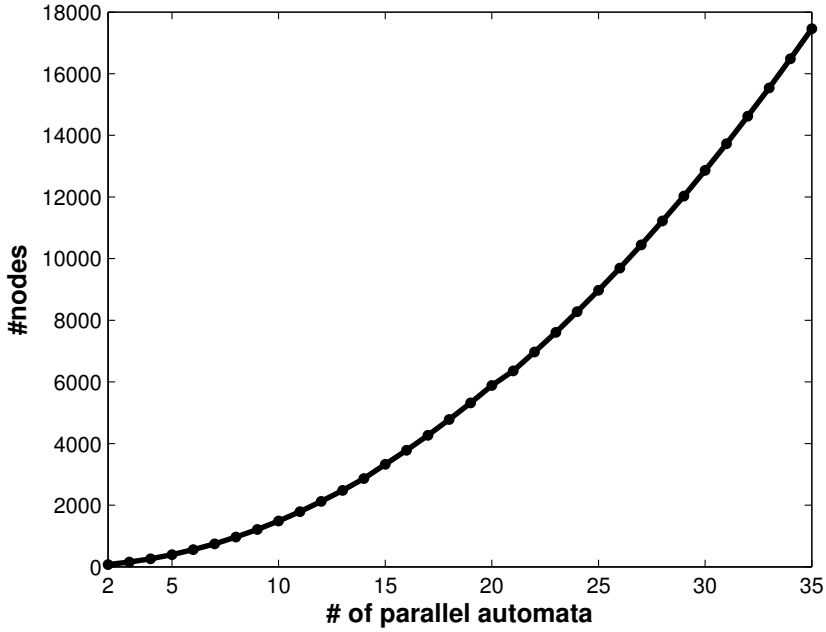

Figure 4.4: BDD-nodes of the partial transition relation for the parallel composition of up to 35 testmachines

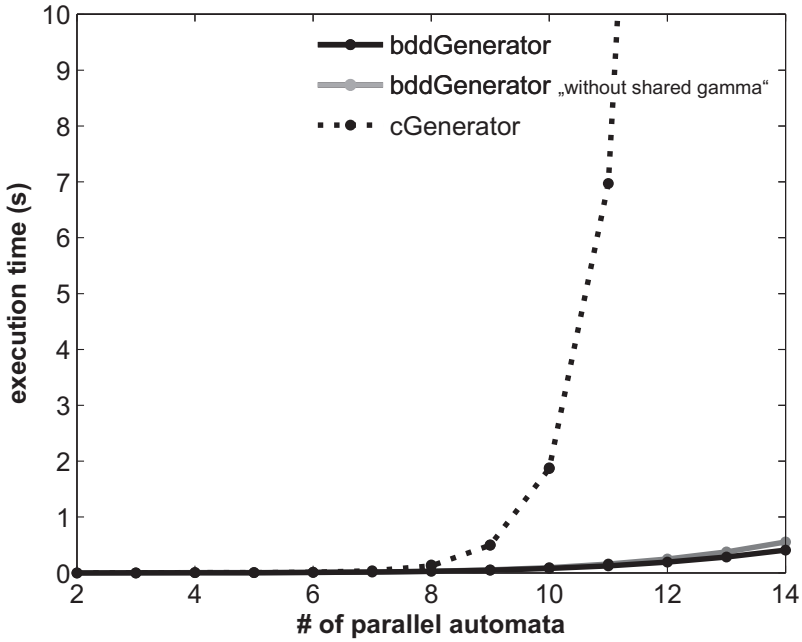

Figure 4.5: Execution time for computation of the parallel composition with the standard and BDD-based algorithm and a global ’gamma’

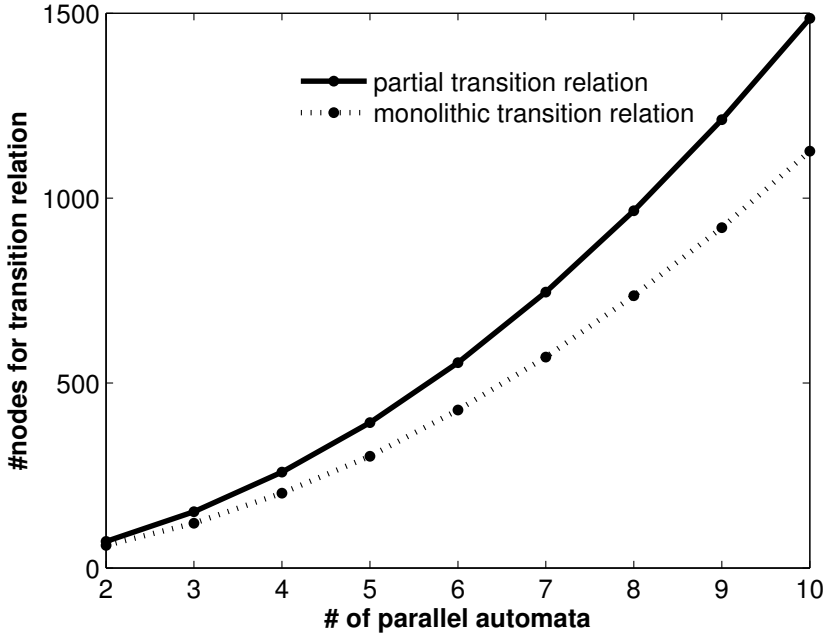

Figure 4.6: Comparison of the number of nodes for the partial and monolithic transition relation

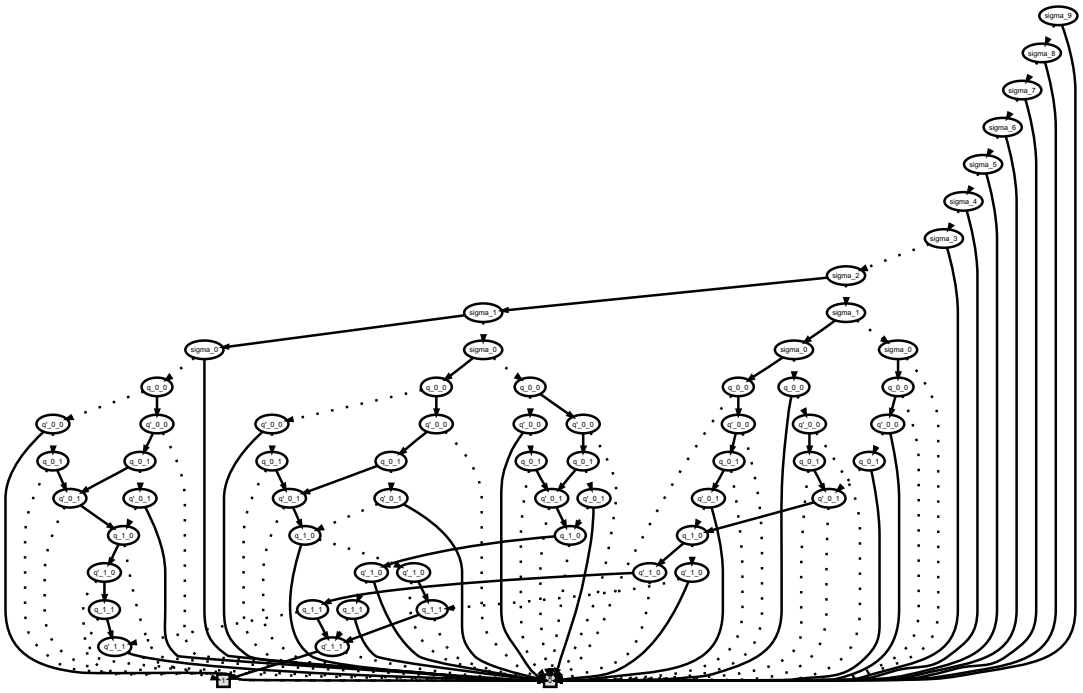

Figure 4.7: BDD of the monolithic transition relation for the parallel composition of 2 automata

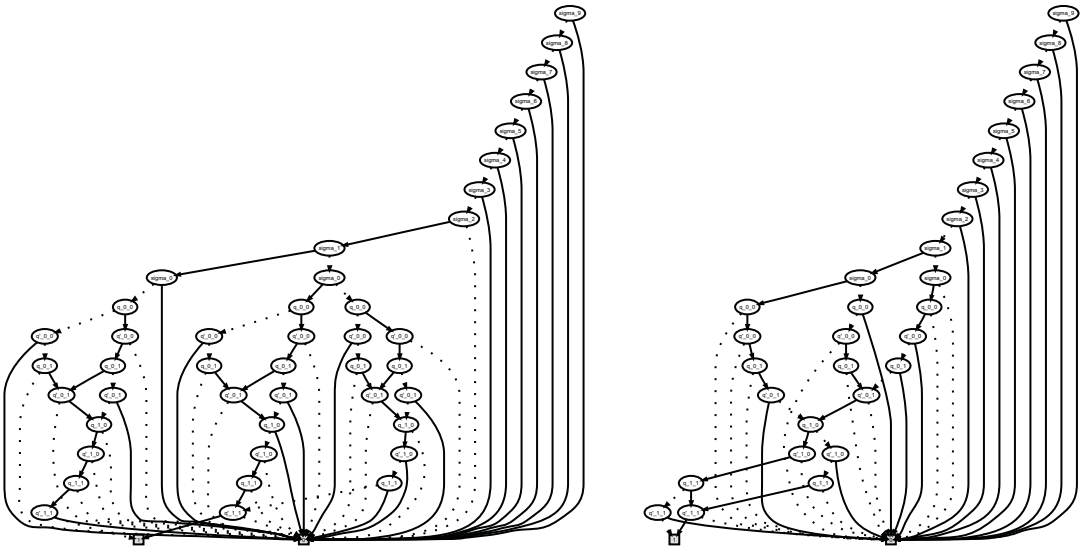

Figure 4.8: BDDs of the partial transition relation for the parallel composition of 2 automata

Chapter 5

Conclusion

After this Bachelor thesis it is possible to try out different application cases of discrete event synthesis with BDDs on an open source software platform. The plug-in for the libFAUDES has reached a state where experiments show fast performance results. Of course it was not possible to use the time frame of a Bachelor thesis to create a closed project of such complexity. Let it rather be an impulse for further improvements and enlargement. The performance results were very impressing and make hope that this plug-in has a chance for future student projects.

I think the next steps will be to provide functions such as that the 'Simulator' or 'IO Device' plug-in can directly work with the bddGenerator without transforming it to a cGenerator or vGenerator. The synthesis elevator example from the IO plug-in already returns a correct solution. But the performance advantage is lost because the bddGenerator has to be exported for writing the solution to a '.gen' file or simulating it.

Besides this, new BDD libraries can be adapted to the wrapper-class or replaced by a new wrapper that uses Interval Decision Diagrams (IDDs) instead as a task for the future.

Bibliography

[And97] Henrik Reif Andersen. An introduction to binary decision diagrams. 1997.

[AS07] W. M. Wonham Ali Saadatpoor. Tstate based control of timed discrete event systems using binary decision diagrams. In *Systems & Control Letters, Vol. 56*, pages 62–74, 2007.

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *DAC '91: Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 403–407, New York, NY, USA, 1991. ACM.

[BLVA06] M. Byrod, B. Lennartson, A. Vahidiand, and K. Akesson. Efficient reachability analysis on modular discrete-event systems using binary decision diagrams. *Discrete Event Systems, 2006 8th International Workshop on*, pages 288 – 293, 2006. Download: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01678444.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[bud10] Buddy, 2010. Homepage: http://sourceforge.net/projects/buddy/.

[CBM90] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373, London, UK, 1990. Springer-Verlag.

[cud09] CU Decision Diagram package. Homepage: http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html, 2009.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GV01] Jaco Geldenhuys and Antti Valmari. Techniques for smaller intermediary bdds. In *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*, pages 233–247, London, UK, 2001. Springer-Verlag.

[GVSSV07] Wilsin Gosti, Tiziano Villa, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. Fsm encoding for bdd representations. *Applied Mathematics and Computer Science*, 17(1):113–124, 2007.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event systems. *SIAM J. Control and Optimization*, 25:206–230, 1987.

[SW07] Ali Saadatpoor and W. M. Wonham. State based control of timed discrete event systems using binary decision diagrams. *Systems & Control Letters*, 56(1):62–74, January 2007.

[WR87] W. M. Wonham and P. J. Ramadge. On the supremal controllable sub language of a given language. In *SIAM J. Control and Optimization*, pages 635–659, 1987.

[WR97] W. M. Wonham and E. S. Rogers. Supervisory control of discrete-event systems. 1997.