Smart TCT:
An Efficient Algorithm for Supervisory Control Design

by

Zhonghua Zhang

A thesis submitted in conformity with the requirements
for the Degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Minor revisions, June 2001

Smart TCT: An Efficient Algorithm for Supervisory Control Design

M.A.Sc., 2001

Zhonghua Zhang

Graduate Department of Electrical and Computer Engineering

University of Toronto

Abstract

This thesis discusses a new synthesis approach to the supervisory control for Discrete Event Systems (DES).

The new approach, named S(mart)TCT after the old CTCT, is much more efficient than the existing synthesis approaches. Using STCT, many practical systems can be synthesized using a personal computer. Besides, it is shown that the number of nodes in the Integer Decision Diagram (IDD) representing a DES can be a better measurement of the complexity of the DES than the number of states.

The structural information in a given DES is exploited to help STCT achieve efficient performance. The algorithm is based on one common feature of most practical systems – a large system is almost always built up by synchronizing many small components. In the synthesis process, no synchronization product is computed explicitly and thus there is no need to list the entire state space.

The success of our new approach is illustrated with very large versions of existing (small) examples taken from the literature.

Acknowledgement

It has been a tremendous privilege to work under the supervision of Professor W.M. Wonham. His enthusiasm for a control theory of discrete event systems has been infectious and has greatly contributed to my studies and work in this area.

I have been exceptionally fortunate to work amongst fellow students in the Systems Control Group who have on innumerable occasions shared their time and patience with me, both guiding me and laughing with me.

Financial support, in the form of University of Toronto Open Fellowships and Research Assistantships from Professor Wonham, is gratefully appreciated.

At last I am very much indebted to my family in China who love and support me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter a brief introduction to Discrete-Event Systems (DES) and related research work is provided. Section 1.1 is an introduction to Discrete-Event Systems. Section 1.2 is a brief research review. Section 1.3 presents the objective of the thesis.

1.1 Introduction

Discrete event systems encompass a wide variety of physical systems that arise in technology. These include manufacturing systems, traffic systems, logistic systems (for the distribution and storage of goods, or the delivery of services), database management systems, communication protocols, and data communication networks. Typically the processes associated with these systems may be thought of as discrete(in time and state space), asynchronous(event-driven rather than clock-driven), and in some sense nondeterministic. The underlying primitive concepts include events, conditions and signals.

In the last two decades, Discrete-Event Systems have been studied by researchers from different fields and with respect to modeling, analysis and control. Quite a few models have been proposed and investigated. These models can be classified as *untimed* DES models and *timed* DES models.

In an untimed model, when considering the state evolution, only the sequence of states visited is of concern. That is we are only interested in the logical behavior of the system, for instance, whether or not the system will enter a particular state, but we don't care when the system enters that state or how long the system remains there.

In a timed model, both logical behavior and timing information are considered. That is, we are concerned not only with the problem of whether or not the system will enter a particular state, but also with when the system enters that state and how long this system will remain there. Timed models are often used for performance analysis.

The next section provides an overview of some existing DES models. Since the models to be investigated in this thesis are untimed models, we review only untimed models.

1.2 Research Overview

This section is by no means a comprehensive overview of existing models and their related research topics. Since the DES research area is growing very fast, a comprehensive overview is beyond the scope of this thesis. The author will concentrate on the topics that are most closely related to his research.

A control theoretic approach was introduced to DES by Ramadge and Wonham [RW87]. Their work, which is based on finite automata and formal languages and then extended to Vector Discrete-Event Systems (VDES) and Petri Nets, is now known as RW theory or supervisory control theory. In RW theory, an automaton (in practice, finite) is used to model both the plant to be controlled and the specification. Different kinds of synthesis methods are developed and implemented as software (CTCT[1]) to compute optimal controllers such that the controlled systems not only satisfy the specifications but also are as permissive as possible. Several control architectures such as decentralized and hierarchical control are investigated. Timing features are incorporated into the framework. However, the existing implementation of RW theory,

[1] CTCT stands for "C based Toy Control Theory". Originally there was a TCT written in Pascal, which was given the name with "toy" because of its limited ability of dealing with large scale systems. Later it was rewritten in C thus got the name CTCT. The name STCT follows it.

i.e. CTCT, can only solve problems of a small state size. This is because an *exhaustive* list is used to represent the whole model of a system in CTCT. Given that many practical systems have more than a trillion states, CTCT has a very limited use in the synthesis of practical control problems.

In actuality, many DES have structured state space and transition functions. Such structure can be exploited to achieve modeling compactness and computational efficiency. In order to make better use of structural properties of DES, another Discrete-Event Systems model called the Vector Discrete-Event System (VDES) has been studied [LW89] [LW93] [LW94] [Hua91], [Li91]. In the VDES framework, control concepts like controllability and observability are borrowed explicitly from RW theory. Two flavors of supervisory control, namely, language-based and state-based with static and dynamic feedback have been studied. The mathematical tool employed is elementary linear algebra and, under some conditions, Integer Linear Programming is used to replace exhaustive search to reduce the computational complexity. However, an inherent shortcoming of VDES is that the nonblocking property of a DES can not be easily verified or achieved in the VDES framework. In VDES, the explicit listing of the state space is avoided. This gives VDES the advantage of performance, w.r.t. RW approach, but at the same time makes the synthesis of a nonblocking controller much more difficult, because we can no longer check each state's reachability and coreachability easily, as is done in the RW approach and its implementation CTCT.

The Petri Net model is another important model to study structured DES [Kro87] [UM88] [Den89] [GD92] [ZD93] [MA98] [ZV99]. In the Petri Net framework, state feedback control was first studied in [UM88] and [Kro87].In [Den89], the supervisory control of a Petri Net was discussed for specifications given in the form of a Petri Net. Existence of a closed-form solution for supervisors is introduced in [GD92]. Since both VDES and Petri Nets are set in the framework of vector addition systems, and some control concepts of RW theory are explicitly borrowed by specialists working on supervisory control of Petri Nets, the Petri Net framework and VDES have appreciable overlap in essence. In order to meet requirements of different application domains, various kinds of restricted, extended and modified versions of Petri Nets are defined and studied. Petri Nets are widely used to solve control problems for real-world systems, such as manufacturing systems, chemical processes and robots etc. In addition to supervisory control, some other important issues studied in the Petri Net framework are analysis and verification of the system's properties and logical behavior, like boundedness, conservativeness, liveness and deadlock, state reachability, reversibility, and language recognition. Different kinds of analysis and verification tools are investigated to address these issues. Because of the similarity between VDES and Petri Net, it is not surprising that the nonblocking problem is also a difficulty in the Petri Net framework. This task is partially covered by the topics of liveness and deadlock; however, all the work done in these areas is quite case-specific.

Besides supervisory control, some other approaches to study DES are: Max-Plus Algebra, Temporal Logic, Communicating Sequential Processes (CSP) and Calculus of Communicating Systems (CCS). Also, stochastic DES and hybrid systems are receiving more and more attention.

1.3 Objective of the Thesis

The above mentioned models, especially RW models and VDES (Petri Net) models, have been successfully used to solve many control problems. However, a fundamental control problem, the nonblocking issue, is still a remaining topic for real-world systems. The RW approach successfully showed the existence and theoretical synthesis procedure of the nonblocking supervisory controller. Unfortunately, the exhaustive search algorithm, which has been implemented in CTCT, has a time complexity of $O(\exp(2 * N))$ (N is the number of modules in the system), and therefore is hardly practical for nontrivial systems. The inefficiency of this approach is directly related to the assumption of ignorance of all the structural information in a real world system. On the other hand, the VDES and Petri Net approaches failed to come up with a general approach toward the nonblocking problem because of the following reasons: (a) too much constraint is imposed on the system itself; and (b) a search of the entire state space is intentionally avoided.

Approach	Algorithm	Generality	Efficiency
RW (CTCT)	Exhaustive Search	Good	Bad
VDES/Petri Net	Structural Analysis	Bad	Good
RW (STCT)	Structural Search	Good	Good

Table 1.1: Comparison of 3 approaches to nonblocking problem

On the other hand, the "easiest" way to achieve nonblocking is exhaustive search of the entire reachable state space, i.e. to verify that every reachable state is coreachable. And this has been considered infeasible in its computational aspect. In the VDES and Petri Net approaches, exhaustive search is intentionally avoided. Researchers in the latter fields have been trying to solve the nonblocking problem by only exploiting the structural information of the system. This approach seems to be theoretically appealing but no results in this direction have a generality comparable to the RW approach.

As we see from the above, all these approaches use only one of the two possible ways, the exhaustive search and the investigation of the structural properties of the system. By contrast, the goal of this thesis is to optimize the exhaustive search algorithm by using the structural properties. In other words, a new algorithm is presented to structurally search the entire reachable state space. It will be shown that the new approach, S(mart)TCT, will be both general and computationally feasible. Table 1.1 provides an overall comparison among the approaches mentioned above.

Chapter 2

Supervisory control of Discrete-Event Systems and its CTCT implementation

This chapter presents a quick review of supervisory control concepts, and defines the concept of state predicates. Section 2.1 reviews the theoretical framework of supervisory control. Section 2.2 briefly introduces the CTCT implementation of the synthesis procedure.

2.1 Supervisory control and the supremal controllable sublanguage [1]

The formal structure of a DES to be controlled can be represented by a *generator* which generates the language corresponding to the DES. Denote the generator by

$$\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m).$$

[1]For detailed discussion of the material of this section, please refer to [Won99].

Here Σ is a finite alphabet of symbols that we refer to as *event labels* , Q is the *state set* (at most countable), $\delta : Q \times \Sigma \to Q$ is the (partial) *transition function* , q_0 is the *initial state* , and $Q_m \subseteq Q$ is the subset of *marker states* .

For the alphabet Σ we have the partition

$$\Sigma = \Sigma_c \cup \Sigma_u$$

where the disjoint subsets Σ_c and Σ_u comprise respectively the *controllable* events and the *uncontrollable* events.

The languages associated with a DES $\mathbf{G}$ are the *closed behavior*

$$L(\mathbf{G}) = \{s \in \Sigma^* | \delta(q_0, s)!\}^1$$

and the *marked behavior*

$$L_m(\mathbf{G}) = \{s \in L(\mathbf{G}) | \delta(q_0, s) \in Q_m\}$$

Note that $\emptyset \subseteq L_m(\mathbf{G}) \subseteq L(\mathbf{G})$, and always $\epsilon \in L(\mathbf{G})$ (provided $\mathbf{G}$ is not *empty* , i.e., $Q \neq \emptyset$) . The *reachable (state) subset* of $\mathbf{G}$ is

$$Q_r = \{q \in Q | (\exists s \in \Sigma^*) \delta(q_0, s) = q\};$$

$\mathbf{G}$ is *reachable* if $Q_r = Q$. The *coreachable (state) subset* of $\mathbf{G}$ is

$$Q_{cr} = \{q \in Q | (\exists s \in \Sigma^*) \delta(q, s) \in Q_m\};$$

$\mathbf{G}$ is *coreachable* if $Q_{cr} = Q$. $\mathbf{G}$ is *trim* if it is both reachable and coreachable.

The DES $\mathbf{G}$ is *nonblocking* if every reachable state is coreachable, i.e.

$$\overline{L}_m(\mathbf{G}) = L(\mathbf{G}).$$

[1] If $f : A \to B$ is a partial function, $f(a)!$ means that f is defined at $a \in A$.

The *closure* $\overline{K}$ of a language K is the set of all prefixes of strings in K. K is said to be *closed* if $\overline{K} = K$.

Suppose we have a plant DES $\mathbf{G}$ with event set Σ and language $L(\mathbf{G})$, and a specification language L. We say L is *controllable* w.r.t. $L(\mathbf{G})$ or $\mathbf{G}$ iff

$$\overline{L}\Sigma_u \cap L(\mathbf{G}) \subseteq \overline{L}.$$

It can be easily proved that if L is controllable w.r.t. $L(\mathbf{G})$ then $L \cap L(\mathbf{G})$ is controllable w.r.t $L(\mathbf{G})$.

A *supervisory control* for $\mathbf{G}$ is any map

$$V : L(\mathbf{G}) \rightarrow \Gamma$$

where Γ is the set of *control patterns* , i.e.

$$\Gamma = \{\gamma \in Pwr(\Sigma) | \gamma \supseteq \Sigma_u\}$$

The pair $(\mathbf{G}, V)$ will be written $V/\mathbf{G}$, to suggest "$\mathbf{G}$ under the supervision of V". The *closed behavior* of $V/\mathbf{G}$ is defined to be the language $L(V/\mathbf{G}) \subseteq L(\mathbf{G})$ described as follows.

1. $\epsilon \in L(V/\mathbf{G})$

2. If $s \in L(V/\mathbf{G})$, $\sigma \in V(s)$, and $s\sigma \in L(\mathbf{G})$ then $s\sigma \in L(V/\mathbf{G})$.

3. No other strings belong to $L(V/\mathbf{G})$.

The *marked behavior* of $V/\mathbf{G}$ is

$$L_m(V/\mathbf{G}) = L(V/\mathbf{G}) \cap L_m(\mathbf{G})$$

Our control objective is, for the given plant $\mathbf{G}$ and the specification language K (in computation, also represented by an automaton), to find a supervisor such that the closed-loop language is, in the sense of set inclusion, the largest sublanguage of $K \cap L_m(\mathbf{G})$ which is controllable w.r.t $L(\mathbf{G})$ and also, nonblocking.

The basic results of supervisory control theory are the following.

Theorem 2.1 Let $E \subseteq \Sigma^*$, and $\mathcal{C}(E)$ be the set of all sublanguages of E that are controllable w.r.t $\mathbf{G}$. Then there exists a (unique) supremum in this set, which can be described as $\sup\mathcal{C}(E) = \cup\{K | K \in \mathcal{C}(E)\}$.

Theorem 2.2 Let $E \subseteq \Sigma^*$, and let $K = \sup\mathcal{C}(E \cap L_m(\mathbf{G})$. If $K \neq \emptyset$, there exists a marking nonblocking supervisory control (MNSC) V for $\mathbf{G}$ such that $L_m(V/\mathbf{G}) = K$.

2.2 The CTCT implementation

The synthesis procedure of $\sup\mathcal{C}$ is implemented in CTCT software. The earliest version of this software, which dates back to the late 1980's, was written in Pascal. Later it was rewritten in C, but the algorithm was kept the same. This section provides an overview of this implementation.

Although the original theoretical results were presented in terms of formal languages, the real computation can only be done in terms of finite state machines. In CTCT, any finite state machine, for example $\mathbf{G}$, is described in the following data structure:

An object called a *DES* is a 6-tuple(*nStates, states, init, marked, events, tlist*) where

- *nStates* is the number of elements in *states* .
- *states* is a finite set of elements
- *init* is either an element of states or undefined
- *marked* is a subset of states
- *events* is a finite set of events

- *tlist* is a set of triples, each of the form (p, σ, q) where p and q are elements in *states.*The set of all σ such that (p, σ, q) is in *tlist* is called the *real event set* of the DES and it must be a subset of *events.*Note that it is possible that an event appearing in *events* never appears in *tlist.*

We can see that each *DES* has a list of all its *states* and *transitions* . So the space needed for storing a *DES* is proportional to *nStates* . In almost all cases, the plant is modeled by a set of component finite state machines, and so is the specification.

The synthesis procedure of CTCT is as follows:

1. Synchronize all the plant component models into *one* finite state machine

$$plant = \mathbf{sync}^{1}(plant_1, plant_2, \dots)$$

   In actuality, the synchronization is done in multiple executions of

$$des_3 = \mathbf{sync}(des_1, des_2)$$

2. Synchronize all the specification models into one finite state machine

$$spec = \mathbf{sync}(spec_1, spec_2, \dots)$$

3. The supremal controllable sublanguage *sup* is computed as

$$sup = \mathbf{supcon}(plant, spec)$$

4. V is computed by

$$V = \mathbf{condat}(plant, sup)$$

As far as complexity is concerned, there are several serious issues:

[1] "**sync**" and the following "**supcon**" and "**condat**" are procedures implemented in CTCT. Further details can be found in [Won99].

1. The procedure $des_3 = \textbf{sync}(des_1, des_2)$ has a complexity of $O(des_1.nStates \times des_2.nStates)$. Theoretically, we always have $des_3.nStats \leq des_1.nstates \times des_2.nStates$. In reality, it is quite probable that

$$des_3.nStats << des_1.nstates \times des_2.nStates$$

for a nontrivial system. Therefore, we often use more space than is actually necessary. For example, given that $des_1.nStates = 10,000$, $des_2.nStates = 10,000$, we will need space of size 100,000,000, even though the result des_3 may only have $50,000$ states. What makes things worse is that the first step of the **supcon** procedure is to compute $\textbf{sync}(plant, spec)$, which actually demands a lot of space even for a moderate sized system. This step is often prohibitive in practice.

2. The time complexity of **supcon** is $O(n^2)$, where n is the number of states of $\textbf{sync}(plant, spec)$. Although this is polynomial, the n is actually an eponential function of the number of modules in the system. Thus **supcon** is still not feasible for most practical systems. It is realistic to find that $\textbf{sync}(plant, spec)$ has more than 1 million states; thus the computation time will be in on order of 1 trillion steps.

3. V is simply a number of lists of states at which each controllable event should be disabled. The size of those lists is also proportional to $sup.nStates$. For a practical system, that may be millions or even trillions. It is very difficult to store in a controller.

This complexity problem of CTCT was discovered years, even decades, ago and little work has been done to obtain a better implementation of RW theory . This thesis presents a new algorithm, STCT, which is many times faster and smaller (in the sense of memory usage) than CTCT. STCT does not need an exhaustive list of the entire state space and thus is able to achieve a performance far beyond that of CTCT. Therefore, we can see that with proper implementation, the RW approach is capable of the direct synthesis of large models.

Chapter 3

Synthesis algorithm based on structural predicates

3.1 Synthesis algorithm for simple DES

Suppose we have a simple DES system, whose plant and specification are both given in the form of a single automaton [1]:

- Plant: $\mathbf{G}_p = (X, \Sigma, x_0, \xi, X_m)$.
- Specification: $\mathbf{G}_s = (Y, \Sigma, y_0, \eta, Y_m)$.

Note that we *require* that the event sets of the plant and specification be the same. From the writer's point of view, this is reasonable and necessary. Suppose that they are different and are denoted as Σ_p and Σ_s respectively. Then we should at least have $\Sigma_p \supseteq \Sigma_s$, because it makes no sense to add "imaginary events" in the specification. And if $\Sigma_p \supset \Sigma_s$, actually it will cause an ambiguity in the modeling:

- Should we consider those events in $\Sigma_p - \Sigma_s$ to be freely enabled or totally disabled?

[1] To distinguish between the $\mathbf{G}_p$, and $\mathbf{G}_s$, different symbols are used to represent the elements in them throughout this chapter.

Thus practically, for the correctness and completeness of the model, we should have $\Sigma = \Sigma_p = \Sigma_s$, although it is not required by the theory.

As mentioned before, the meet of $\mathbf{G}_p$ and $\mathbf{G}_s$ can be expressed as

$$\mathbf{G}_{meet} = \mathbf{meet}^2(\mathbf{G}_p, \mathbf{G}_s) = (X \times Y, \Sigma, (x_0, y_0), \xi \times \eta, X_m \times Y_m).$$

In order to place 'conditions' on the state (x, y) of $\mathbf{G}_{meet}$, it will be convenient to use a logic formalism. A *predicate* P defined on $X \times Y$ is a function $P : X \times Y \to \{0, 1\}$. P can always be identified with the corresponding state subset

$$(x, y)_P := \{(x, y) \in X \times Y | P(x, y) = 1\}.$$

Remark: P is defined on a "2-dimensional" set, compared with normal predicates defined on a simple set. P holds not only information about conditions we put on the states, but also structural information about the system. And all the logic operators defined for predicates for simple sets can be easily extended to this kind of predicate. For clarity, if $P(x, y) = 1$ we write $(x, y) \models P$, otherwise $(x, y) \not\models P$.

We define the *closed language* $L(\mathbf{G}_{meet}, P)$ induced by a predicate $P \in Pred(X \times Y)$ to be

$$L(\mathbf{G}_{meet}, P) := \{w \in L(\mathbf{G}_{meet}) | (\forall v \leq w)\big(\xi(x_{p0}, v), \eta(y_{s0}, v)\big) \models P\}$$

and the *marked language* $L_m(\mathbf{G}_{meet}, P)$ induced by a predicate $P \in Pred(X \times Y)$ to be

$$L_m(\mathbf{G}_{meet}, P) := \{w \in L(\mathbf{G}_{meet}, P) | \big(\xi(x_{p0}, w), \eta(y_{s0}, w)\big) \in X_m \times Y_m\}.$$

The following definitions will help to introduce the controllability and nonblocking property related to structural predicate P.

Definition 3.1 Reachability of a given state.

[2] The **meet** procedure is defined in CTCT, as described in [Won99].

For a given P, a state (x, y) is *reachable* iff

$$(\exists w \in L(\mathbf{G}_{meet}, P))\ \big(\xi(x_0, w) = x \text{ and } \eta(y_0, w) = y\big)$$

Definition 3.2 Coreachability of a given state.

For a given P, a state (x, y) is *coreachable* iff

$$(\exists w \in \Sigma^*)\ \big(\xi(x, w) \in X_m \text{ and } \eta(y, w) \in Y_m \text{ and } (\forall v \leq w)(\xi(x, v), \eta(y, v)) \models P\big)$$

Definition 3.3 Controllability of structural predicate P.

For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is *controllable* iff $L(\mathbf{G}_{meet}, P)$ is controllable w.r.t $L(\mathbf{G})$.

Definition 3.4 Nonblocking property of structural predicate P.

For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is *nonblocking* iff $\overline{L_m(\mathbf{G}_{meet}, P)} = L(\mathbf{G}_{meet}, P)$.

Now we can provide two useful lemmas about structural predicates.

Lemma 3.1 For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is controllable iff

$$(\forall (x, y) \models P, \forall \sigma \in \Sigma_u)(\xi(x, \sigma)!) \Rightarrow \big(\eta(y, \sigma)!\ \textit{and}\ (\xi(x, \sigma), \eta(y, \sigma)) \models P\big)$$

Proof: For simplicity let $K = L(\mathbf{G}_{meet}, P)$ and write L for $L(\mathbf{G}_{meet})$. Obviously we have $\overline{K} = K$.

(If). We need to show

$$K\Sigma_u \cap L \subseteq K$$

. Given a string $s \in K$ and $\sigma \in \Sigma_u$, let $x = \xi(x_0, s)$ and $y = \eta(y_0, s)$. Then $(x, y) \models P$. Now there are two possible cases.

1. $\xi(x, \sigma)!$, then $\eta(y, \sigma)!$ and $(\xi(x, \sigma), \eta(y, \sigma)) \models P$. Therefore $s\sigma \in K$.

2. $\xi(x, \sigma)$ is undefined. Then $s\sigma \notin L$.

(Only if). Given K is controllable w.r.t. L, suppose the contrary of the conclusion, i.e.

$$(\exists (x,y) \models P, \sigma \in \Sigma_u)(\xi(x,\sigma)! \text{ and } ((\neg \eta(y,\sigma)!) \text{ or } (\xi(x,\sigma), \eta(y,\sigma)) \not\models P))$$

It is obvious that now we have $s \in K$, $s\sigma \in L$, but $s\sigma \notin K$, therefore K is not controllable w.r.t. L. This contradicts the assumption.

□

Lemma 3.2 For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is nonblocking iff

$$(\forall (x,y) \models P)((x,y) \text{ is reachable}) \Rightarrow ((x,y) \text{ is coreachable})$$

Proof: (If). Need to show that $\overline{L_m(\mathbf{G}_{meet}, P)} = L(\mathbf{G}_{meet}, P)$.

First show $\overline{L_m(\mathbf{G}_{meet}, P)} \subseteq L(\mathbf{G}_{meet}, P)$. Given a string $s \in \overline{L_m(\mathbf{G}_{meet}, P)}$, there exists $s' \in L_m(\mathbf{G}_{meet}, P)$ such that $(\exists u \in \Sigma^*) s' = su$. Thus $(\xi(x_0, s), \eta(y_0, s)) \models P$ and then $s \in L(\mathbf{G}_{meet}, P)$ because we always have $s \in L(\mathbf{G}_{meet})$.

Now show $\overline{L_m(\mathbf{G}_{meet}, P)} \supseteq L(\mathbf{G}_{meet}, P)$. Given a string $s \in L(\mathbf{G}_{meet}, P)$, we need to show there exists a string w, such that $\xi(x_0, sw) \in X_m, \eta(y_0, sw) \in Y_m$. It is enough to show $(\exists w \in \Sigma^*)\big(\xi(\xi(x_0, s), w) \in X_m, \eta(\eta(y_0, s), w) \in Y_m\big)$. This is obviously true because $(\xi(x_0, s), \eta(y_0, s))$ is reachable, and thus coreachable.

(Only if). Suppose the contrary, i.e. $(\exists (x,y) \in X \times Y)$, (x,y) is reachable but not coreachable. Then there exists s, such that $\xi(x_0, s) = x, \eta(y_0, s) = y$, but there doesn't exist any w such that $\xi(x, w) \in X_m, \eta(y, w) \in Y_m$. Thus

$$\overline{L_m(\mathbf{G}_{meet}, P)} \subset L(\mathbf{G}_{meet}, P),$$

which gives us a contradiction.

□

Now we are ready to present the main result of this chapter:

Theorem 3.1 Given $\mathbf{G}_p$ and $\mathbf{G}_s$, there exists a structural predicate P_{sup}[1] such that

1. $L_m(\mathbf{G}_{meet}, P_{sup}) = \sup\mathcal{C}(L(\mathbf{G}_p), L_m(\mathbf{G}_s))$.

2. P_{sup} is nonblocking.

Proof: The proof is done constructively.

Given $\mathbf{G}_p$ and $\mathbf{G}_s$, we can construct P_{sup} as follows:

1. Initialize $P_{good} \equiv 1$.

2. $P_{bad1} = \{(x, y) \models P_{bad1}\}$ iff
   $((x, y) \not\models P_{good})$
   or $(\exists \sigma \in \Sigma_u)\big((\xi(x, \sigma)!) \text{ and } (\neg(\eta(y, \sigma)!))\big)$
   or $(\exists \sigma \in \Sigma_u)\big(\xi(x, \sigma)!, \eta(y, \sigma)! \text{ and } (\xi(x, \sigma), \eta(y, \sigma)) \not\models P_{good}\big)$

3. $P_{bad2} = \{(x, y) \models P_{bad2}\}$ iff
   $(\exists w \in \Sigma_u^*)((\xi(x, w), \eta(y, w)) \models P_{bad1}$

4. $P_{re} = \{(x, y) \models P_{re}\}$ iff
   $(\exists w \in \Sigma^*)$
   $(\xi(x_0, w) = x, \eta(y_0, w) = y),\ (\forall v < w)(\xi(x_0, v), \eta(y_0, v)) \not\models (P_{bad1} \cup P_{bad2})$

5. $P_{cr} = \{(x, y) \models P_{cr}\}$ iff
   $(\exists w \in \Sigma^*)$
   $(\xi(x, w) \in X_m, \eta(y, w) \in Y_m),\ (\forall v < w)(\xi(x, v), \eta(y, v) \not\models (P_{bad2} \cup P_{bad2})$

6. $P_{newgood} = P_{re} \cap P_{cr}$.

7. If $P_{newgood} \subset P_{good}$ repeat step 2-7 with $P_{good} = P_{newgood}$, otherwise let $P_{sup} = P_{newgood}$ and the algorithm terminates here.

Now we need to show that P_{sup} is what we want.

[1] It is possible that $P_{sup} \equiv 0$. This means $\sup\mathcal{C}(L(\mathbf{G}_p), L_m(\mathbf{G}_s)) = \emptyset$. If this happens, we know that a solution to the given control problem does not exist. In other words, the constraints are too tight for the system.

- The algorithm will terminate in finite steps.

  Define $|P|$ to be the number of states where a structural predicate P holds. Obviously in each cycle (step 3 – step 9), if the algorithm does not terminate, $|P_{bad}|$ must increase by at least 1 at step 7. Therefore, the algorithm can run at most $|Q_p| \times |Q_s|$ times, which is finite.

- $L_m(\mathbf{G}_{meet}, P_{sup}) \subseteq \sup\mathcal{C}(L(\mathbf{G}_p), L_m(\mathbf{G}_s))$.

  – P_{sup} is nonblocking.

    This is because when we terminate, we always have $P_{good} = P_{re} = P_{cr} = P_{newgood}$, i.e. every reachable state satisfying $P_{newgood}$ is coreachable.

  – P_{sup} is controllable.

    Suppose that
    $\big(\exists (x', y') \models P_{sup}\big)\big((\exists \sigma \in \Sigma_u)(\xi(x', \sigma)!) \text{ and } (\neg(\eta(y', \sigma)!))\big)$
    then according to step 1 and step 2, we have $(x', y') \models P_{bad1}$, which contradicts $(x', y') \models P_{sup}$. Thus we have
    $(\forall (x, y) \models P, \forall \sigma \in \Sigma_u)(\xi(x, \sigma)!) \rightarrow (\eta(y, \sigma)!)$

    Suppose that
    $\big(\exists (x', y') \models P_{sup}\big)$
    $\big((\exists \sigma \in \Sigma_u)((\xi(x', \sigma)!, (\eta(y', \sigma)!)(\xi(x', \sigma), \eta(y', \sigma))|{\neq} P)\big)$
    then $(x', y') \models P_{bad2}$, which contradicts $(x', y') \models P_{sup}$. Thus we have
    $\big(\forall (x', y') \models P_{sup}\big)$
    $\big((\forall \sigma \in \Sigma_u)((\xi(x', \sigma)!, (\eta(y', \sigma)!) \Rightarrow (\xi(x', \sigma), \eta(y', \sigma)) \models P)\big)$

    Thus P is controllable according to Lemma 3.1.

  – $L_m(\mathbf{G}_{meet}, P_{sup}) \subseteq L_m(\mathbf{G}_{meet}) = L_m(\mathbf{G}_p) \cap L_m(\mathbf{G}_s)$.

    This is obvious according to definitions of $L_m(\mathbf{G}_{meet}, P_{sup})$ and $L_m(\mathbf{G}_{meet})$.

- $L_m(\mathbf{G}_{meet}, P_{sup}) \supseteq \sup\mathcal{C}(L(\mathbf{G}_p), L_m(\mathbf{G}_s))$. We need to show that

$$(\forall K' \subseteq L(\mathbf{G}_{meet})(K' \text{ is controllable and nonblocking})\ K' \subseteq L_m(\mathbf{G}_{meet}, P_{sup})$$

Suppose the contrary. Then we can find a K', which is controllable and non-blocking but $K \nsubseteq L_m(\mathbf{G}_{meet}, P_{sup})$. Then $(\exists s \in K')(s \notin L_m(\mathbf{G}_{meet}, P_{sup}))$. Let $x = \xi(x_0, s)$ and $y = \eta(y_0, s)$. So $(x, y) | \neq P_{sup}$.

The algorithm is iterative and there is one loop, step 2-7. For clarity, write $P_{good(i)}$, $P_{newgood(i)}$, $P_{bad1(i)}$, and $P_{bad2(i)}$ for their values in cycle i. For example, write the $P_{bad1(2)}$ for the value of P_{bad1} in the second cycle.

Now we will show that if

$$(\exists s \in K')((\xi(x_0, s), \eta(y_0, s)) | \neq P_{newgood(n)})$$

then

$$(\exists s' \in K')((\xi(x_0, s'), \eta(y_0, s')) | \neq P_{good(n)})$$

Let $(x_1, y_1) = (\xi(x_0, s), \eta(y_0, s))$. If $(x_1, y_1) | \neq P_{good(n)}$, then $s' = s$ is what we need. Otherwise (i.e. $(x_1, y_1) \models P_{good(n)}$) we have $(x_1, y_1) | \neq P_{re(n)}$ or $(x_1, y_1) | \neq P_{cr(n)}$ or both. Suppose we have $(x_1, y_1) | \neq P_{re(n)}$. Then either $(x_1, y_1) \models (P_{bad1(n)} \cup P_{bad2(n)})$ or $(\exists w \leq s)((\xi(x_0, w), \eta(y_0, w)) \models (P_{bad1(n)} \cup P_{bad2(n)})$. If the former is true, let $(x_2, y_2) = (x_1, y_1)$ and $s_1 = s$; otherwise let $(x_2, y_2) = (\xi(x_0, w), \eta(y_0, w))$ and $S_1 = w$. In the latter case, we must have $w \in K'$ too. Similar reasoning can also be applied if $(x_1, y_1) | \neq P_{cr(n)}$. Now we have

$$(\exists s_1 \in K')((x_2, y_2) = (\xi(x_0, s_1), \eta(y_0, s_1)), (x_2, y_2) \models (P_{bad1(n)} \cup P_{bad2(n)}))$$

If $(x_2, y_2) | \neq P_{bad1(n)}$, we must have $(x_2, y_2) \models P_{bad2(n)}$ Thus

$$(\exists s_2 \in \Sigma^*)((\xi(x_2, s_2), \eta(y_2, s_2)) \models P_{bad1(n)}$$

In this case let $(x_3, y_3) = (\xi(x_2, s_2), \eta(y_2, s_2))$ and $s_3 = s_1 s_2$; otherwise (i.e. $(x_2, y_2) \models P_{bad1(n)})$, let $(x_3, y_3) = (x_2, y_2)$ and $s_3 = s_1$. Now we have

$$(\exists s_3 \in K')((x_3, y_3) = (\xi(x_0, s_3), \eta(y_0, s_3)), (x_3, y_3) \models P_{bad1(n)})$$

There are 3 possible cases now:

- $(x_3, y_3)|\neq P_{good(n)}$. Then s_3 is what we need.
- $(\exists \sigma \in \Sigma_u)\big((\xi(x_3,\sigma)!) \text{ and } (\neg(\eta(y_3,\sigma)!))\big)$. This is impossible bacause this will make K' uncontrollable.
- $(\exists \sigma \in \Sigma_u)\big(\xi(x_3,\sigma)!, \eta(y_3,\sigma)! \text{ and } (\xi(x_3,\sigma),\eta(y_3,\sigma))|\neq P_{good(n)}\big)$. Now we must have $s_3\sigma \in K'$, otherwise K' is not controllable. Also we have $(\xi(x_0, s_3\sigma), \eta(y_0, s_3\sigma))|\neq P_{good(n)}$. So $s_3\sigma$ is what we need.

Because $P_{good(n)} \equiv P_{newgood(n-1)}$, this process can continue inductively until we finally reach $n = 1$, i.e. it must be true that

$$(\exists s' \in K')((\xi(x_0, s'), \eta(y_0, s'))|\neq P_{good(1)})$$

But this is impossible, because we have $P_{good(1)} \equiv 1$. Thus the conclusion is proved.

□

3.2 Synthesis algorithm for complex DES

The algorithm described in the previous section can be easily extended to complex DES systems.

Suppose we have a complex DES system, whose plant and specification are both given in the form of the synchronous product of more than one automaton module:

- Plant: $\mathbf{G}_p = \mathbf{sync}(\mathbf{G}_{pi})$. $\mathbf{G}_{pi} = (X_i, \Sigma, x_{0i}, \xi_i, X_{mi})$. $i \in 1, \ldots, N_p$.
- Specification:$\mathbf{G}_s = \mathbf{sync}(\mathbf{G}_{sj})$. $\mathbf{G}_{sj} = (Y_j, \Sigma, y_{0j}, \eta_j, Y_{mj})$. $j \in 1, \ldots, N_s$.

Then similarly as in the previous section, it can be shown that the synthesis and implementation of the controller can be done in the form of a $N_p + N_s$-variable structural predicate:

Theorem 3.2 Given $\mathbf{G}_{pi}$ and $\mathbf{G}_{si}$ as shown above, there exists a structural predicate P_{sup} such that

1. $L_m(\mathbf{G}_{meet}, P_{sup}) = \sup\mathcal{C}(L(\mathbf{sync}(\mathbf{G}_{pi})), L_m(\mathbf{sync}(\mathbf{G}_{si})))$.

2. P is nonblocking.

Proof: The proof is identical to the proof in the previous section, except for a description of higher dimension. Also, the same constructive steps can be used to compute P_{sup}.

□

3.3 Implementation of supervisory controller by structural predicate

As mentioned before, the *supervisory control* is defined as a function

$$V : Q \rightarrow \Gamma$$

where $\Gamma = \{\gamma \in Pwr(\Sigma) | \gamma \supseteq \Sigma_u\}$. Thus it can also be expressed as

$$V' : Q \rightarrow (e_{\sigma 1}, e_{\sigma 2}, \ldots e_{\sigma n_e})$$

where $e_{\sigma i} \in \{0, 1\}$, n_e is the number of events, and $e_{\sigma i} = 1$ if $\sigma_i \in \Sigma_u$. Then we can express V' in the form of n_e scalar functions:

$$V' \equiv \left\{V'_i : Q \rightarrow \{0, 1\}, \qquad i \in \{1, 2, \ldots, n_e\}\right\}$$

This is nothing but n_e structural predicates. Therefore *supervisory control* can be implemented by structural predicates in a straightforward way.

The V' defined above can be obtained as follows.

Write

$$
\begin{aligned}
x &= (x_{p1}, \ldots, x_{pm}) \\
y &= (y_{s1}, \ldots, y_{sn}) \\
z &= (x, y) \\
\zeta(z, \sigma) &= \big(\xi(x, \sigma), \eta(y, \sigma)\big)
\end{aligned}
$$

Finally, let

$$
e_\sigma(z) = \begin{cases} 1 & \text{if } z \models P_{sup},\ \zeta(z,\sigma)!,\ \zeta(z,\sigma) \models P_{sup} \\ 0 & \text{otherwise} \end{cases}
$$

Obviously, for any $\sigma \in \Sigma_u$, $e_\sigma \equiv 1$, because P_{sup} is controllable. Also, clearly $L(V'/\mathbf{G}) = L(\mathbf{G}, P_{sup})$ and $L_m(V'/\mathbf{G}) = L(\mathbf{G}, P_{sup})$, since exactly the same constraints are imposed upon $\mathbf{G}$ by V' and by P_{sup}.

Chapter 4

Introduction to Integer Decision Diagrams (IDDs)

4.1 Overview

Binary Decision Diagrams (BDD) were introduced by Akers [Ake78]. Then Bryant [Bry86] 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 a polynomial function of the number of variables. After that many different diagrams similar to BDDs have emerged. The most important objective for these types of decision diagrams is to derive constructions that will reduce the complexity of verifying hardware implementing arithmetic (integer) functions.

To extend the Boolean representation to general finite functions has been an important task. A direct IDD representation was studied in [Gun97]. The material in this chapter is extracted from it. Interested readers are encouraged to read the book directly.

4.2 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

1. 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 $index(v) \in \{1, \ldots, n\}$ and children $chld(v, i) \in V, 0 \leq i < range(v)$, where $n, range(v) \in Z_+$. A *terminal* vertex v has as attribute a value $value(v) \in 0, \ldots, M - 1$, where $M \in Z_+$.

2. For any nonterminal vertex v, if $child(v, i)$ is also non-terminal, then $index(v) < index(child(v, i))$.

3. The range 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.

A lot of function graphs will be depicted in this thesis. Nevertheless, we will present a simple example to clarify how these depicted graphs are related to Definition 4.1.

The vertical axis to the left shows the indexes for the vertices. We denote a vertex by $v_{i,j}$ where i is the vertical order starting from the top with $i = 1$ and j is the horizontal order in the picture starting from the left with $j = 0$. The edge labels in the picture identify each child of a vertex. The terminals are placed at the bottom of the graph, and indexed by a T on the vertical axis. The values of the terminals are written inside the vertices.

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

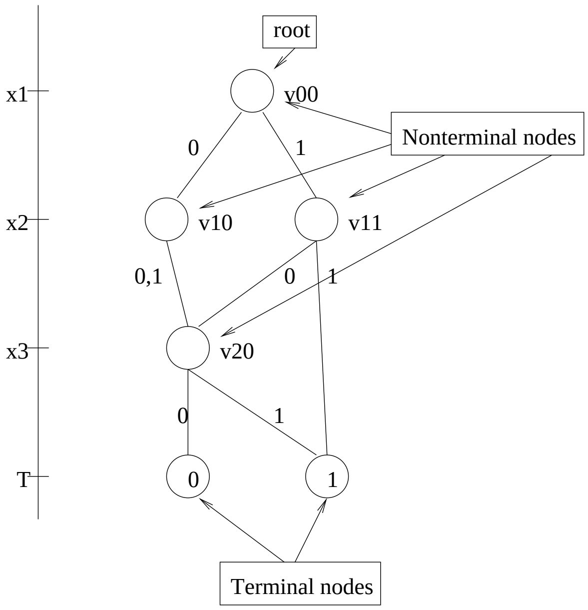


Figure 4.1: A simple function graph

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 $f_v = value(v)$.
2. If v is a nonterminal vertex with $index(v) = i$, then f_v is the function

$$f_v(x_i, x_{i+1}, \ldots, x_n) = f_{child(v,x_i)}(x_{i+1}, \ldots, x_n)$$

where the function variable $x_i \in (0, 1, \ldots, range(v) - 1)$, $1 \leq i \leq n$, and n is the maximum index of vertices in G.

From the graph depicted in Fig 4.1, we can deduce the function represented by it. The value of this function is shown in Table 4.2.

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

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 4.1: Function represented by Fig 4.2

representation of the final f_v for the complete graph G. Symbolic representation will be formulated later in next section.

To be able to form the function graphs with canonical properties (of which the definition follows) we have to provide some definitions before we can state the theorem on canonical function graph.

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 vertices of G' such that for any vertex v if $\sigma(v) = v'$, then either both v and v' are terminal vertices with

$$value(v) = value(v')$$

or both v and v' are non-terminal vertices with

$$index(v) = index(v')$$

and

$$\sigma(child(v, i)) = child(v', i) \quad 0 \leq i < range(v)$$

Remark: Note that $range(v) = range(v')$.

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 in 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 4.1.

Definition 4.4 Subgraph

For any vertex v in a function graph G, the *subgraph rooted at* v is defined as the graph consisting of v and all of its descendants.

The following lemma is immediate.

Lemma 4.1 Isomorphic Subgraphs

If G is isomorphic to G' by the mapping σ, then for any vertex v in G, the subgraph rooted by v is isomorphic to the subgraph rooted by $\sigma(v)$.

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

Definition 4.5 Reduced Function Graph

A function graph G is *reduced* if it contains no vertex v with

$$child(v, 0) = child(v, 1) = \cdots = child(v, range(v) - 1),$$

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

Lemma 4.2 Reduced Subgraphs

For every vertex v in a reduced function graph, the subgraph rooted by v is itself a reduced function graph.

We are now ready for the main result about function graphs. 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 [Bry86] for the Boolean case only, but the idea of the proof can easily be adapted to more general cases as well.

□

4.3 IDDs

In this section we will consider functions $f : A \to B$ as represented by reduced function graphs called *integer decision diagrams* (IDDs). Here A is a product of finite sets and B is a simple finite set of scalars (i.e. set of scalars, not tuples).

Example 4.1 Functions over Finite Domains

For the function $f(x_1, x_2, x_3), f : A \to B$ we have the domains

$$A = Z_2 \times Z_3 \times Z_4$$

and

$$B = Z_2.$$

This means that the values of the arguments are

$$x_1 \in \{0, 1\}, x_2 \in \{0, 1, 2\}, x_3 \in \{0, 1, 2, 3\}$$

and that $f(x_1, x_2, x_3) \in \{0, 1\}$.

If we consider the integer function as represented by a table, we can then extract some structure according to a variable ordering. To do this we need to define the *Lagrange function.*

Definition 4.6 Lagrange Function

The *Lagrange function* , denoted $L_k(x)$, is defined as

$$L_k(x) = \begin{cases} 1 & (x = k) \\ 0 & (x \neq k) \end{cases}$$

So we can write:

$$
\begin{aligned}
f(x_1, \ldots, x_n) &= L_0(x_1)\underbrace{f(0, x_2, \ldots, x_n)}_{f_0^1(x_2,\ldots,x_n)} + \cdots \\
&\quad + L_{d_1}(x_1)\underbrace{f(d_1, x_2, \ldots, x_n)}_{f_{d_1}^1(x_2,\ldots,x_n)} \\
f_0^1(x_2, \ldots, x_n) &= L_0(x_2) f_{0,0}^2(x_3, \ldots, x_n) + \cdots \\
&\quad + L_{d_2}(x_1) f_{0,d_2}^2(x_2, \ldots, x_n) \\
&\vdots \\
f^n_{d_1,\ldots,d_{n-1},0} &= c_0 \\
f^n_{d_1,\ldots,d_{n-1},d_n} &= c_{m-1}
\end{aligned}
$$

where $d_1 = range(x_i) - 1$ and the number of constant terminals c_j [1] is

$$m = \Pi_{0<i\leq n} range(x_i)$$

The function $f(x_1, \ldots, x_n)$ 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 $f(x_1, \ldots, x_n)$.

We can represent $f(x_1, \ldots, x_n)$ with a reduced function graph where identical subexpressions are represented by a single subgraph.

Two graphs representing the same function are shown in Fig 4.3. The left one is in the full form, and the right one is in reduced form.

We will call this reduced function graph for functions over finite domains an *integer decision diagram* (IDD) and define it as follows.

Definition 4.7 Integer Decision Diagrams

An *integer decision diagram* is a reduced function graph (see Def. 4.1 and 4.5) where the number of terminals M [2] is arbitrary and the variable domain $range(x_i)$ may be chosen arbitrarily for each variable x_i.

[1] Not all these terminals are diffrerent from each other.

[2] However, in this thesis, M is always 2 in this thesis since we are using the IDDs to represent predicates.

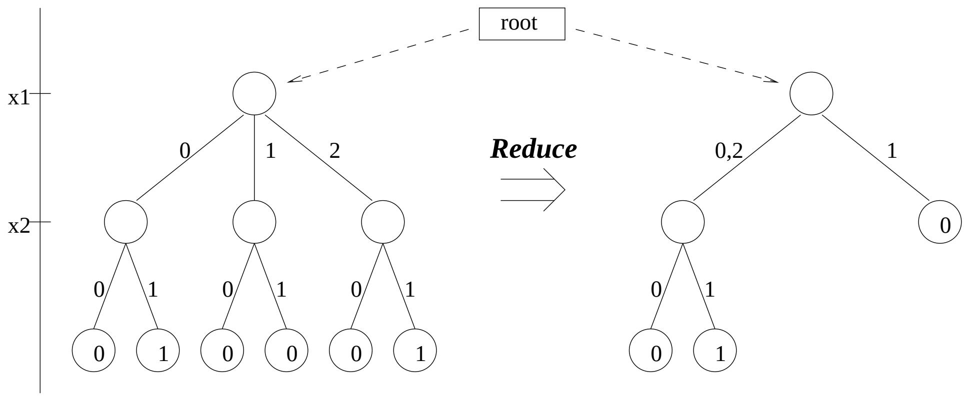


Figure 4.2: Reducing an IDD

From theorem 4.1 we know that an IDD is a canonical form.

The connection between IDDs and functions follows immediately from 4.2 but may be restated for clarity.

Definition 4.8 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 $f_v = value(v) \in 0, \ldots, M-1$.

2. If v is a nonterminal vertex with $index(v) = i$, then f_v is the function

$$
\begin{aligned}
f_v(x_1, \ldots, x_n) \quad = \quad & L_0(x_i) f_{child(v,0)}(x_1, \ldots, x_n) + \\
& L_1(x_i) f_{child(v,1)}(x_1, \ldots, x_n) + \cdots + \\
& L_{range(x_i)-1}(x_i) f_{child(v,range(x_i)-1)}(x_1, \ldots, x_n)
\end{aligned}
$$

Chapter 5

IDD based implementation and algorithm optimization

As shown in the previous chapter, all the computation needed for procedure **supcon** can be realized by using structural predicates. However, if we use the naive representation of structural predicate, the value table, we gain no advantage compared with the CTCT implementation. Fortunately, Integer Decision Diagrams can be used here so that we can actually represent the structural predicate in *compressed* form.

The approach using IDDs has been applied to Discrete Event Systems in [Gun97]. However, the focus of that work is the relational representation, not the computation. Besides, the algorithm in [Gun97] is implemented in a rather inefficient way[1] , and thus no obvious computational advantages were gained and demonstrated.

A direct and efficient algorithm based on IDDs, namely STCT, will be presented in this chapter. It has several unique features worth mentioning:

- It is a native C++ implementation and thus is very fast and compact.

[1]The algorithm in [Gun97] is implemented by first translating the control logic to logical expressions in Mathematica, and then completing the synthesis in Mathematica. Although Mathematica is considered an efficient tool for symbolic manipulations for mathematicians, its efficiency can hardly be compared with a program of native binary code. For example, the simple logical "AND" can be done by one CPU instruction in a binary code, but in Mathematica, where all objects are stored in a symbolic way, this operation needs many more CPU cycles (transformation between symbolic value and machine data, and there are quite a few function calls that must be invoked). Besides, the translation adds a considerable overhead to the synthesis procedure.

- It has the ability to find the (nearly) optimal order of the variables of IDDs.

- It uses an incremental method to improve the performance of the synthesis of a system with buffers.

This chapter consists of three sections. Section 5.1 presents the representation of structural predicates by IDDs. Section 5.2 is the detailed description of the algorithms applied to IDDs. The last section is about the optimizations used in the algorithm.

Remarks:

- Due to the complexity of the algorithm itself, it is not realistic to present it in the form of strict algebraic definitions, given the space and time allowance of this thesis. Therefore the description will be given in a form which is as strict as possible but still pseudo-code-like.

- There are numerous kinds of optimizations which can be applied to computational algorithms. For the IDD-based algorithm, *Reduction* is the most important and essential one. Without it, the IDD-based algorithm has no advantages over algorithms based on the exhaustive list of state space, such as CTCT. Thus it is "naturally" incorporated into the algorithm itself, as presented in the Section 5.2. Another optimization, *caching* , is also introduced, together with the algorithms in Section 5.2, because of its close relation to the algorithm itself. Other optimizations are introduced in Section 5.3.

5.1 Representation of structural predicates by IDDs

This can be done in a straightforward way, since structural predicates are simply functions defined on a finite domain.

From the computational point of view, an IDD is a 4-tuple $(nlevel, listNB, nodes, index)$ where

- $nlevel$ is the number of levels in the IDD.

- $listNB$ is the list of numbers of branches in all the levels. There are always $nlevel$ entries in this list, and all the elements are strictly positive integers.

- $nodes$ is a set of nodes, where a node is a triple $(type, level, list_{next})$, with $type$ and $list_next$ defined as follows.

  - $$type = \begin{cases} 0 & \text{if the node is a terminal node and its value is 0} \\ 1 & \text{if the node is a terminal node and its value is 1} \\ \text{-1} & \text{if the node is a non-terminal node} \end{cases}$$

  - $level$ is the level that the node lies in and is determined by the references of nodes. There is only 1 node whose $level = 0$ and it is called $root$ node. All the other non-terminal nodes' $level$ values are determined as follows:

    * If the current node is referenced by another node which has $level = i$, the current node's $level$ is defined to be $i + 1$. Terminal nodes always have $level = -1$.

    * If $type$ is 0 or 1, the $listnext$ is an empty set. Otherwise $listnext$ is a list of references to other nodes, which are either other non-terminal nodes in the $next$ level or terminal nodes.

  - $list_{next}$ is the list of the children of the given node.

- $index$ is additional indices for nodes, and is a list (one entry for each level) of lists (one entry for each node in the given level) of references to nodes.

Note that

- The IDD is organized strictly in levels and all the non-terminal nodes except for the $root$ node are referenced by one or more other nodes.

- The nature of node ensures that the structure formed by nodes and their references are acyclic, because a node can only reference another node if the referenced node is in the next level or it is a terminal node.

- There is no "jumping" of level allowed, i.e., a node in level i can never reference a node in level $i+2$, $i+3$ ... However, "premature termination" is allowed, i.e., a branch can be terminated at any level by a reference to a terminal node.

A simple IDD is given in Fig 5.1 to illustrate the definition above. In the graph, the given IDD has 3 levels, and the numbers of branches are (3,3,3). All the nodes except for the root can be accessed in two ways: through its parent (solid line arrows) or the index of its level (dashed line arrows). The root node can be accessed directly from outside. In order to find the corresponding value of a given vector x_0, x_1, x_2, we travel from the root, choosing the proper branch at each level until we reach a terminal node. For instance, for (0,0,1), we start from the root node, take branch 0 at level 0, branch 0 at level 1, branch 1 at level 2, and then reach the terminal node T, i.e. (0,0,1) satisfies the predicate represented by the IDD in Fig 5.1. There are also "premature" terminations in the graph (dotted lines). For example, we can see that all state vectors like (2,2,*) do not satisfy the IDD by starting from the root, taking branch 2 at level 0, branch 2 at level 0, and reaching terminal node F.

5.2 Computation based on IDDs

5.2.1 Operations necessary to complete the synthesis

The computational procedure was shown in the proof of Theorem 3.1. For clarity, it is listed below again.

1. Initialize $P_{good} \equiv 1$.

2. $P_{bad1} = \{(x,y) \models P_{bad1}\}$ iff

$$\begin{aligned} & ((x,y) \not\models P_{good}) \\ \text{or}\quad & (\exists \sigma \in \Sigma_u)\big((\xi(x,\sigma)!) \text{ and } (\neg(\eta(y,\sigma)!))\big) \\ \text{or}\quad & (\exists \sigma \in \Sigma_u)\big(\xi(x,\sigma)!, \eta(y,\sigma)! \text{ and } (\xi(x,\sigma),\eta(y,\sigma)) \not\models P_{good}\big) \end{aligned}$$

3. $P_{bad2} = \{(x,y) \models P_{bad2}\}$ iff

$(\exists w \in \Sigma_u^*)((\xi(x,w),\eta(y,w)) \models P_{bad1}$

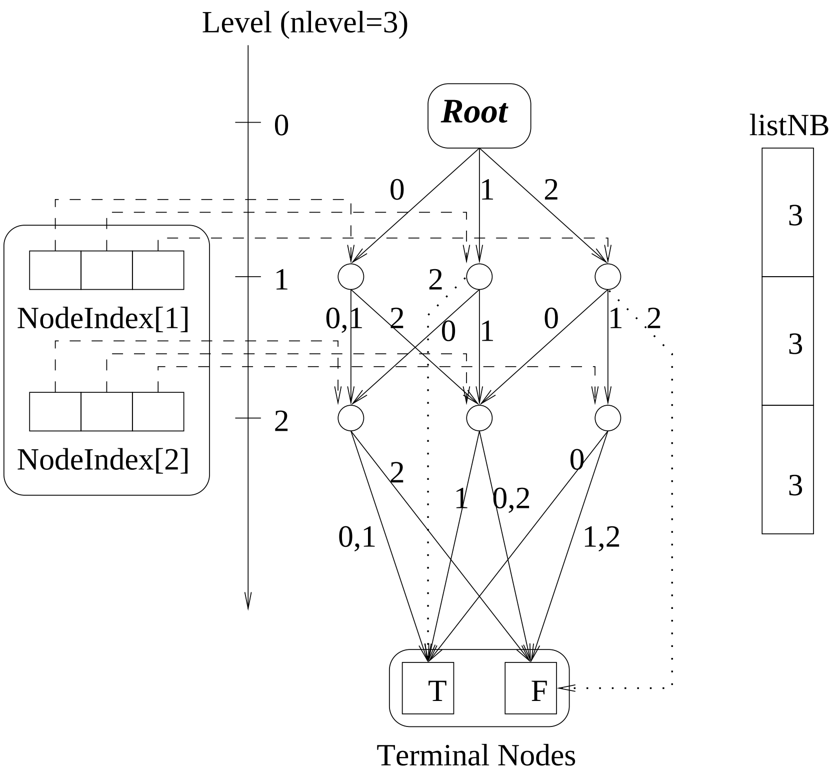


Figure 5.1: A simple example of the data structure of IDD

4. $P_{re} = \{(x, y) \models P_{re}\}$ iff
   $(\exists w \in \Sigma^*)$
   $(\xi(x_0, w) = x, \eta(y_0, w) = y),\ s(\forall v < w)(\xi(x_0, v), \eta(y_0, v)) |\not= (P_{bad1} \cup P_{bad2})$

5. $P_{cr} = \{(x, y) \models P_{cr}\}$ iff
   $(\exists w \in \Sigma^*)$
   $(\xi(x, w) \in X_m, \eta(y, w) \in Y_m),\ s(\forall v < w)(\xi(x, v), \eta(y, v) |\not= (P_{bad2} \cup P_{bad2})$

6. $P_{newgood} = P_{re} \cap P_{cr}$.

7. If $P_{newgood} \subset P_{good}$ repeat step 2-7 with $P_{good} = P_{newgood}$, otherwise let $P_{sup} = P_{newgood}$ and the algorithm terminates here.

Step 1,2,6,7 can be easily realized by logical operations, i.e. AND, OR, and NEGATION.

Steps 3,4,5 needs more consideration. We define the operation $R : Pred(Y) \rightarrow Pred(Y)$ [2] as follows. Let $P_1 = R < P_0, \Sigma', P' >$ be the reachable predicate starting from P_0, while enabling all $\sigma \in \Sigma'$ and satisfying predicate P', i.e.

$$y \models P_1 \Leftrightarrow (\exists y' \models P_0, s \in \Sigma'^*)\big((\forall v \leq s)(\eta(y', v) \models P')\big)$$

Similarly we define operation CR to produce the coreachable predicate. Thus steps 3,4,5 can be realized by R and CR, i.e.

- Step 3. $P_{bad2} = CR < P_{bad1}, \Sigma_u, true >$.
- Step 4. $P_{re} = R < P_0, \Sigma, \neg(P_{bad1} \cup P_{bad2}) >$
- Step 5. $P_{cr} = CR < P_m, \Sigma, \neg(P_{bad1} \cup P_{bad2}) >$

Here P_0 is true for and only for the initial state, and P_m is true for and only for the marker state(s).

[2]For simplicity, the following discussion uses notations for simple predicates i.e. predicates defined on a non-structured set, and does not distinguish between X and Y. All the results can be easily generalized to structural predicates.

Define the operation $N_\sigma : Pred(Y) \to Pred(Y)$ to produce the *one-step reachable predicate* . That is to say, $P' = N_\sigma(P)$ is defined as

$$(y' \models P') \Leftrightarrow (\exists y \models P)(\eta(y,\sigma)!, \eta(y,\sigma) = y')$$

The operation $P_1 = R < P_0, \Sigma', P >$ can then be realized by using N_σ and logical operations as follows.

1. $P_1 = P_0$.

2. $P_1' = P_1$

3. For each $\sigma \in \Sigma'$ compute $P_1 = P_1 \cup N_a(P_1)$.

4. If $P_1 \not\equiv P_1'$ goto step 1. Otherwise the procedure terminates and P_1 is what we want.

Similarly, define M_σ to be the *one-step coreachable predicate* operation, i.e.

$$(y' \models M_\sigma(P)) \Leftrightarrow (\exists y \models P)(\eta(y',\sigma)!, \eta(y',\sigma) = y)$$

and then the operation CR can be done using logical operations and M_σ.

As we can see, there are two types of computation that need to be done on IDDs:

1. Logical operations, i.e., AND, OR, and NEG(ATION).

2. The transformations N_σ and M_σ.

5.2.2 Logical operations

Negation

The negation operation on IDDs is trivial. Given a vector $v = (x_1, \ldots, x_n)$, the value of $f_{IDD}(v)$ is decided by whether the route from the root node finally reaches the terminal node of type 0 or 1. Therefore, the negation of a given IDD can be easily obtained by interchanging the types of the two terminal nodes in the IDD.

AND and OR

These two operations are the most complicated part of the algorithm and are implemented recursively and only applicable to two IDDs with the same number of levels and identical $listNB$s: that is obvious because the logical operation can only be applied to two structural predicates defined on identical finite domains.

The ideas behind these two algorithms are very similar. Generally speaking, it is a depth-first traverse algorithm. The reason why the AND and OR operations are complicated is that the computation must be based only on the reduced IDDs, i.e., it is not feasible to first expand two IDDs into unreduced ones, do the computation, and then reduce them. In that way, we have to store a full IDD tree, which will nullify all the advantages of the IDD based implementation.

The fundamental difficulty in computing the AND (or OR) of two reduced IDDs is that the two IDDs are (almost always) not reduced in the same manner. The basic idea of the computation is "expand and reduce as you go", i.e., we do expand reduced branches, but as soon as we finish computing one branch, we reduce it, so that we never have to store a fully unreduced IDD.

In order to realize the ongoing reduction, indices of all the nodes in each level of the destination IDD are built. The destination IDD is constructed in a bottom-to-top manner, and as soon as a new node is computed, it is checked against the index of the level it is in, and if an "identical" node is found, the new node will be deleted and a new reference to the existing node will be used instead of this new node. In this way, it is guaranteed that there are no isomorphic subgraphs in the destination IDD, i.e, it is reduced while computed.

Another important way to improve the performance is caching. When computing the AND(OR) of two reduced IDDs, it is quite possible that the logical operation on two identical subgraphs will be computed more than once. Fig 5.2.2 illustrates a possible situation. As shown, the AND(OR) operation of subgraphs rooted at node A and node B will be computed at least twice: first when following branch (0,0, ...), then (1,0, ...). Obviously there is a duplication of computation. The solution to

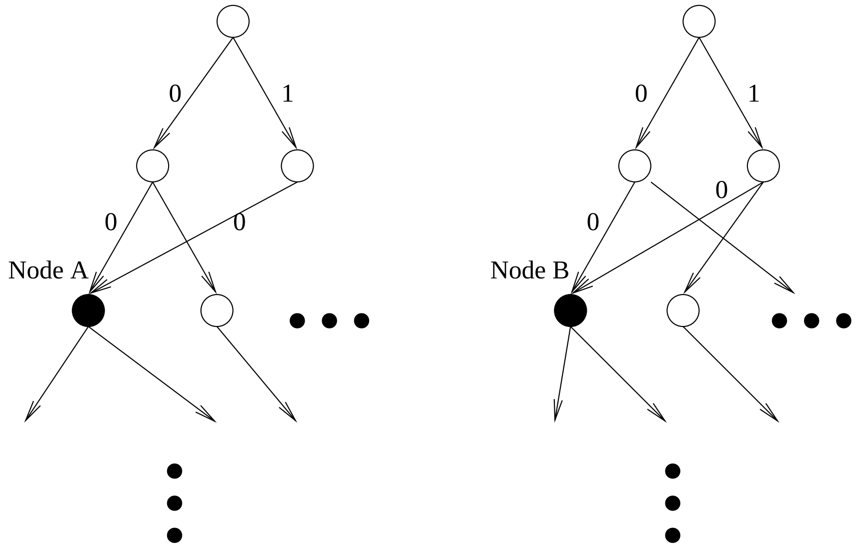


Figure 5.2: An example of duplicated computation of subgraphs

this is to maintain a cache for each level and put all computed pairs of nodes as well as the result of each pair into the cache. Before doing the computation of one pair of nodes, first check the cache; if the pair is already computed, then the result can be simply obtained from the cache.

The above mentioned indices and caches are all unique to the destination IDD. When the computation begins, the indices (one per level) and caches (two per level, one for the logical operation and another for the copy operation (see below)). For the recursive functions doing the computing, these indices and caches are global for the computation of a given destination IDD. Therefore, they can be considered as one part of the destination IDD's data. Actually, they are treated in this way in the implementation of STCT.

In the following pseudo-code, some simple functions are used:

- *cloneIDD(IDDA)* creates a new IDD with the same *listNB* as *IDDA* .
- *cloneNode(aNode)* will create a new node with the same structure as *aNode* .

- *delete(aNode)* will delete *aNode* .
- *result=checkCopyCache[i](aNode)* checks the copy cache of level i for a key value of *aNode* . The *result* contains both whether *aNode* is found and then the corresponding value if found.
- *addToCopyCache[i](aKeyNode, aValueNode)* adds one new entry to the copy cache of level i with the key *aKeyNode* and the value *aValueNode* .
- *result=checkComputeCache[i](aNodeA, aNodeB)* checks the compute cache of level i for a key value of *(aNodeA, aNodeB)* . The *result* contains both whether *aNode* is found and then the corresponding value if found.
- *addToComputeCache[i](aKeyNodeA, aKeyNodeB, aValueNode)* adds one new entry to the compute cache of level i with the key *(aKeyNodeA, aKeyNodeB)* and the value *aValueNode* .
- *result=checkNodeIndex[i](aNode)* checks the node index of level i for *aNode* . The *result* contains whether *aNode* is found.
- *addToNodeIndex[i](aKeyNode, aValueNode)* adds *aNode* to the node index of level i;

A very important procedure *CopySubGraph* is widely used to copy a part of an IDD (a subgraph) to another IDD, and reduce as much as possible. This is not trivial.

Definition 5.1 Procedure CopySubGraph is defined as
void[3] CopySubGraph(IDDA, nodeA, IDDC, nodeC, level);
where IDDA and IDDC are the source and destination IDDs; nodeA and nodeC are the starting nodes in those two IDDs respectively; level is the level where those nodes reside. Note that the two IDDs must have the same domain, and the source and destination nodes must lie in the same level. The procedure is executed recursively as follows:

[3] "void" is a keyword in C/C++ language, which means the function does not return any return value. A "void function" in C/C++ is the counterpart of "procedure" in Pascal.

```
void CopySubGraph(IDDA, nodeA, IDDC, nodeC, level) {

    i=0;
    while(i<listNB[level]) do{
        next_nodeA=nodeA.childList[i];
        if(next_nodeA.type=0) {
                nodeC.childList[i]=IDDC.nodeFalse;
                i=i+1;
                continue to next cycle;
        }
        if(next_nodeA.type=1) {
                nodeC.childList[i]=IDDC.nodeTrue;
                i=i+1;
                continue to next cycle;
        }
        result=checkCopyCache[i](next_nodeA);
        if(result.found=1) {
                nodeC.childList[i]=result.value;
                i=i+1;
                continue to next cycle;
        }
        next_nodeC=cloneNode(next_nodeA);
        copySubGraph(IDDA, next_nodeA, IDDC, next_nodeC, level+1);
        result=checkNodeIndex[level](next_nodeC);
        if(result.found=1) {
                delete(nextNodeC);
                nodeC.childList[i]=result.value;
                addToCopyCache[level](next_nodeA, next_nodeC);
                i=i+1;
                continue to next cycle;
```

```
        }
        nodeC.childList[i]=next_nodeC;
        addToNodeIndex[level](next_nodeC);
        addToComputeCache[level](next_nodeA, next_nodeC);
        i=i+1;
        continue to next cycle;
    }
}
```

Now we are ready to define the procedures AND and OR.

Definition 5.2 Procedure OR is prototyped as
IDDResult=OR(IDDA, IDDB);
Note that IDDA and IDDB must have the same structure, i.e., identical listNB. The procedure is:

```
IDDResult=OR(IDDA, IDDB)
{
    IDDC=cloneIDD(IDDA);
    computeOR(IDDA, IDDA.root, IDDB, IDDB.root, IDDC, IDDC.root, 0);
    return IDDC;
}
```

The procedure computeOR is to compute the conjunction of the two structural predicates represented by the two subgraphs.

Definition 5.3 Procedure computeOR is prototyped as void computeOR(IDDA, nodeA, IDDB, nodeB, IDDC, nodeC, level); The procedure is:

```
void computeOR(IDDA, nodeA, IDDB, nodeB, IDDC, nodeC, level)
{
    i=0;
    while(i<listNB[level]) do{
```

```
next_nodeA=nodeA.childList[i];
next_nodeB=nodeB.childList[i];
if(next_nodeA.type=0 and next_nodeB.type=0) {
        nodeC.childList[i]=IDDC.nodeFalse;
        i=i+1;
        continue to next cycle;
}
if(next_nodeA.type=1 or next_nodeB.type=1) {
        nodeC.childList[i]=IDDC.nodeTrue;
        i=i+1;
        continue to next cycle;
}
if(next_nodeA.type=1) {
        result=checkCopyCache[i](next_nodeB);
        if(result.found=1) {
                nodeC.childList[i]=result.value;
                i=i+1;
                continue to next cycle;
        }
        next_nodeC=cloneNode(next_nodeB);
        copySubGraph(IDDB, next_nodeB, IDDC, next_nodeC,
                                                  level+1);
        result=checkNodeIndex[level](next_nodeC);
        if(result.found=1) {
                delete(next_nodeC);
                nodeC.childList[i]=result.value;
                addToCopyCache[level](next_nodeB, next_nodeC);
                i=i+1;
                continue to next cycle;
        }
```

```
        addToNodeIndex[level](next_nodeC);
        addToCopyCache[level](next_nodeB, next_nodeC);
        i=i+1;
        continue to next cycle;
}
if(next_nodeB.type=1) {
        result=checkCopyCache[level](next_nodeA);
        if(result.found=1) {
                nodeC.childList[i]=result.value;
                i=i+1;
                continue to next cycle;
        }
        next_nodeC=cloneNode(next_nodeA);
        copySubGraph(IDDA, next_nodeA, IDDC, next_nodeC,
                                                level+1);
        result=checkNodeIndex[level](next_nodeC);
        if(result.found=1) {
                delete(next_nodeC);
                nodeC.childList[i]=result.value;
                addToCopyCache[level](next_nodeA, next_nodeC);
                i=i+1;
                continue to next cycle;
        }
        addToNodeIndex[level](next_nodeC);
        addToCopyCache[level](next_nodeA, next_nodeC);
        i=i+1;
        continue to next cycle;
}
result=checkComputeCache[level](next_nodeA, next_nodeB);
if(result.found=1) {
```

```
                nodeC.childList[i]=result.value;
                i=i+1;
                continue to next cycle;
        }
        next_nodeC=cloneNode(next_nodeA);
        computeOR(IDDA, next_nodeA, IDDB, next_nodeB,
                                IDDC, next_nodeC, level+1);
        result=checkNodeIndex[level](next_nodeC);
        if(result.found=1) {
                delete(nextNodeC);
                nodeC.childList[i]=result.value;
                addToComputeCache[level](next_nodeA, next_nodeB, next_nodeC);
                i=i+1;
                continue to next cycle;
        }
        nodeC.childList[i]=next_nodeC;
        addToNodeIndex[level](next_nodeC);
        addToComputeCache[level](next_nodeA, next_nodeB, next_nodeC);
        i=i+1;
        continue to next cycle;
    }
}
```

The procedure AND is defined identically to the procedure OR, except that the logical operations in the procedure are all conjunctions instead of disjunctions.

Definition 5.4 Procedure AND is prototyped as
IDDResult=AND(IDDA, IDDB); Note that IDDA and IDDB must have the same structure, i.e., identical listNB. The procedure is:

```
IDDResult=AND(IDDA, IDDB)
{
```

```
    IDDC=cloneIDD(IDDA);
    computeAND(IDDA, IDDA.root, IDDB, IDDB.root, IDDC, IDDC.root, 0);
    return IDDC;
}
```

where procedure *cloneIDD(IDDA)* creates a new IDD with the same *listNB* of *IDDA*
.

The procedure computeAND is to compute the conjunction of the two structural predicates represented by the two subgraphs.

Definition 5.5 Procedure computeAND is prototyped as void computeAND(IDDA, nodeA, IDDB, nodeB, IDDC, nodeC, level); The procedure is:

```
void computeAND(IDDA, nodeA, IDDB, nodeB, IDDC, nodeC, level)
{
    i=0;
    while(i>listNB[level]) do{
        next_nodeA=nodeA.childList[i];
        next_nodeB=nodeB.childList[i];
        if(next_nodeA.type=0 or next_nodeB.type=0) {
                nodeC.childList[i]=IDDC.nodeFalse;
                i=i+1;
                continue to next cycle;
        }
        if(next_nodeA.type=1 and next_nodeB.type=1) {
                nodeC.childList[i]=IDDC.nodeTrue;
                i=i+1;
                continue to next cycle;
        }
        if(next_nodeA.type=0) {
                result=checkCopyCache[i](next_nodeB);
```

```
        if(result.found=1) {
                nodeC.childList[i]=result.value;
                i=i+1;
                continue to next cycle;
        }
        next_nodeC=cloneNode(next_nodeB);
        copySubGraph(IDDB, next_nodeB, IDDC, next_nodeC,
                                                    level+1);
        result=checkNodeIndex[i](next_nodeC);
        if(result.found=1) {
                delete(next_nodeC);
                nodeC.childList[i]=result.value;
                addToCopyCache(next_nodeB, next_nodeC);
                i=i+1;
                continue to next cycle;
        }
        addToNodeIndex[i](next_nodeC);
        addToCopyCache(next_nodeB, next_nodeC);
        i=i+1;
        continue to next cycle;
}
if(next_nodeB.type=0) {
        result=checkCopyCache[i](next_nodeA);
        if(result.found=1) {
                nodeC.childList[i]=result.value;
                i=i+1;
                continue to next cycle;
        }
        next_nodeC=cloneNode(next_nodeA);
        copySubGraph(IDDA, next_nodeA, IDDC, next_nodeC,
```

```
                                                level+1);
        result=checkNodeIndex[i](next_nodeC);
        if(result.found=1) {
                delete(next_nodeC);
                nodeC.childList[i]=result.value;
                addToCopyCache(next_nodeA, next_nodeC);
                i=i+1;
                continue to next cycle;
        }
        addToNodeIndex[i](next_nodeC);
        addToCopyCache(next_nodeA, next_nodeC);
        i=i+1;
        continue to next cycle;
}
result=checkComputeCache[i](next_nodeA, next_nodeB);
if(result.found=1) {
        nodeC.childList[i]=result.value;
        i=i+1;
        continue to next cycle;
}
next_nodeC=cloneNode(next_nodeA);
computeAND(IDDA, next_nodeA, IDDB, next_nodeB,
                               IDDC, next_nodeC, level+1);
result=checkNodeIndex[i](next_nodeC);
if(result.found=1) {
        delete(nextNodeC);
        nodeC.childList[i]=result.value;
        addToComputeCache(next_nodeA, next_nodeB, next_nodeC);
        i=i+1;
        continue to next cycle;
```

```
        }
        nodeC.childList[i]=next_nodeC;
        addToNodeIndex[i](next_nodeC);
        addToComputeCache(next_nodeA, next_nodeB, next_nodeC);
        i=i+1;
        continue to next cycle;
    }
}
```

5.2.3 Transformations N_σ and M_σ [4]

Procedure *PostPred* is used to implement the predicate transformer *one-step reachable predicate*, i.e. $PostPred(IDD, \mathrm{a}) = N_a(P)$[5].

Definition 5.6 Procedure PostPred is prototyped as
void PostPred(IDDA, a);
where IDDA is the IDD representing the predicate P and a is the given event. After the execution of the procedure, IDDA will represent $N_a(P)$. The procedure is:

```
void PostPred(IDDA, a)
{
    IDDB=EnableCondition(a);
    IDDA=AND(IDDA, IDDB);
    for each l in 0..(IDDA.nLevels-1) do {
        for each nd in IDDA.nodeIndex[l] do {
                newNode=copyNode(nd)
                for each j in 0..(nd.nChildren) do {
                        newNode.listChild[j]=nd.listChild[j];
                        nd.listChild[j]=IDDA.nodeFalse;
```

[4]The definitions of N_σ and M_σ can be found in section 5.2.1.
[5]For simplicity, "a" is used to represent one certain event instead of the Greek character "σ" in this section. In pseudocode, "a" is used.

```
                }
                listTransition=getListTrans(a, level);
                for each t in listTransition do{
                        nd.listChild[t.to]=newNode.listChild[t.from];
                }
        }
    }
}
```

where

- *EnableCondition(a)* returns a predicate E_a defined as

$$(x_1, x_2, \dots, x_N) \models E_a \Leftrightarrow (\forall i \in \{1 \dots N\})\eta_i(x_i)!$$

  i.e., a state satisfies E_a if event a is enabled at that state.

- *copyNode(aNode)* creates an identical new node as *aNode* . In procedure PostPred *newNode* is created to keep a backup copy of *nd* .

- *getListTrans(a, level)* returns the list of transitions of event a in the automaton model of component *level* . If all transitions of a in that component are selfloops and a is enabled at all states, *getListTran(a, level)* will return an empty list.

Procedure *PrePred* is used to implement the predicate transformer *one-step coreachable predicate*, i.e. $PrePred(IDD, a) = M_a(P)$. This can be done in an identical way to the procedure *PostPred* , except that all transitions are applied in a "reversed" manner. For instance, a transition $\eta(5, \sigma) = 6$ will be applied as $\eta(6, \sigma) = 5$ to compute $M_a(P)$.

5.2.4 Implementation of controller by IDDs

As shown in the previous chapter, the controller can be implemented using $|\Sigma_c|$ structural predicates. Therefore, by replacing those predicates by corresponding IDDs, an

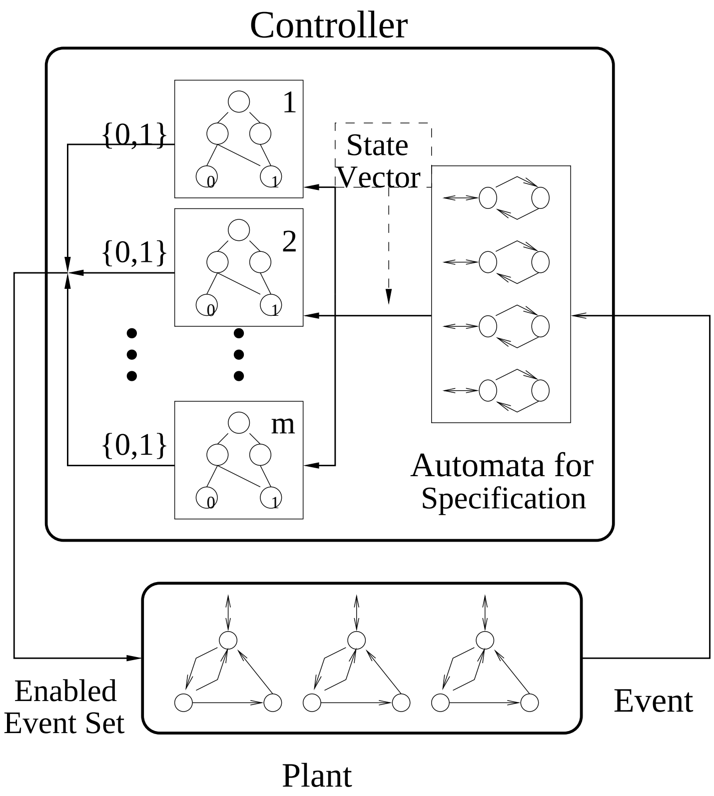


Figure 5.3: A sketch of a system with IDD based controller

implementation of controller by IDDs can be easily obtained. A diagram of the controlled system is shown in Fig 5.3.

There are two major advantages to implementing controllers by IDDs:

1. The size of an IDD-based controller is much smaller than an original control data table generated by CTCT. A comparison between the sizes is shown in Chapter 6.

2. The computation needed to get the Γ map is very modest so that it can be done in real time quite easily. Actually, the time complexity to get the Γ map is $O(N_{events} \times N_{components})$. This is because we need at most $N_{components}$ step to get the enabling information for each controllable event. For a practical system, this can be done by several thousand CPU instructions, which is a simple task for any modern computer (requiring time $< 10\mu s$) .

5.3 Algorithm optimization

5.3.1 Ordering of variables

Ordering of variables in an IDD (corresponding to the ordering of the components in the system) is the most important optimization other than reduction. In fact, an effective reduction relies on a good ordering of all the variables.

From an IDD's point of view, the reason why a reduction is almost always possible is that the information given in level i, is not always correlated[6] to the information given in another level j. If the information from two different levels is not correlated, we can expect a reduction to happen, otherwise, we normally can't. A simple example is given below.

Suppose we have a 3-variable structural predicate $P : (x_1, x_2, x_3) \to \{0, 1\}$, where $x_i \in \{0, 1, 2\}, i \in \{1, 2, 3\}$. P is defined as

x_1	x_2	x_3	P
0	0	0	1
1	1	0	1
2	2	0	1

No other values of (x_1, x_2, x_3) satisfy P.

We can see that, the information carried by x_1 and x_2 is correlated, while the information carried by x_1 and x_3 is not. The two IDDs representing P, using different variable ordering, are shown in Fig 5.4. Obviously the ordering x_1, x_2, x_3 is better than the ordering x_1, x_3, x_2 (5 nodes vs. 7 nodes).

For most systems in practice, it will be true that in P_r, the predicate representing its reachable state space, not all the components' information contents are correlated to each other. Intuitively, we can get a hint of whether two components' information contents will be correlated by looking at their events. If they have shared event(s), it will be likely that their information will be correlated, and vice versa. A similar idea

[6]Please see Appendix A for more explanation on "correlation".

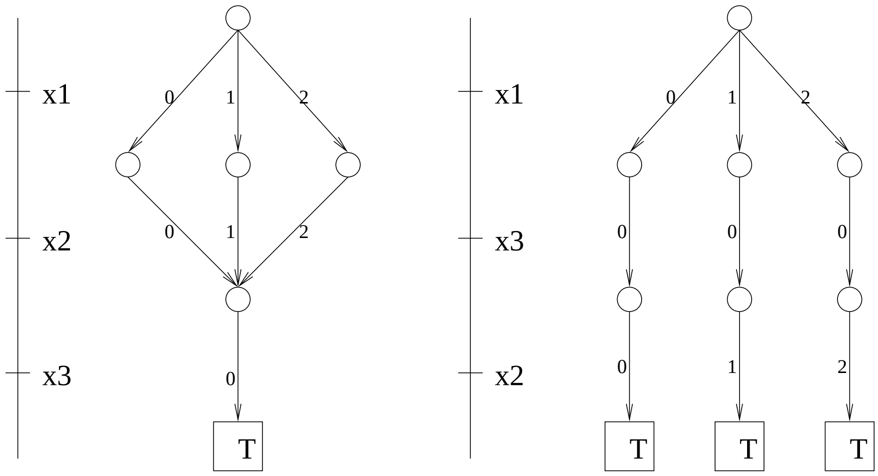


Figure 5.4: IDDs representing same P but with different variable ordering

is also applicable to the coreachable-state specification predicate.

Therefore, the basic idea behind the ordering of variables in the STCT algorithm is that

- the more closely coupled two components are, the "closer" the two components should be placed in the IDD.

A simplified description of this idea is "If two components have shared event(s), they should be adjacent (or as adjacent as possible) in the IDD".

It will be shown in Chapter 6, that this method of ordering does improve the performance dramatically.

As we can see, in order to improve the efficiency of an IDD, we should reduce the separation of two coupled components as much as we can. From another point of view, this is equivalent to reducing the crossing over of components. "Crossing over" of a given component i means that in the given ordering, there is a component in front of i that is coupled to a component behind i. For an "ideal" ordering of an "ideal" system, each component is only coupled with the two components in front of and behind it, thus the overall "crossing" is 0.

The overall crossing can serve as a good measurement of the quality of a given ordering. Thus we can define a function to evaluate a given ordering as follows.

Definition 5.7 Connection Matrix.
$C_{i,j}$=the number of events shared by $Component_i$ and $Component_j$.

For a given ordering of N components, $O = (O_1, O_2, \ldots, O_N), O_i \in 0, \ldots, N-1$, the number of crossings over of $Component_i, i \in 1 \ldots N$ is defined as

$$Cross_i = \sum_{\substack{j,k \\ Rank(j)<Rank(i) \\ Rank(k)>Rank(i)}} C_{j,k}$$

where $Rank(i)$ is the rank of component i and always satisfies $Rank(O(i)) = i$.

Then the overall crossing, as the measurement of the optimality of the given ordering $Eval(O)$ is defined as

$$Eval(O) = \sum_i 4^{Cross_i}.$$

This definition is empirical rather than theoretical [7]. When we try to reduce the overall crossing in the IDD, we practically try to reduce those larger $Cross_i$.

Now the ordering is an optimization problem. Given the nonlinear nature of the measurement function, it is very unlikely that we can find a global optimal solution in feasible time. For a system with N components, there are $N!$ possible orderings; thus it is impossible to do an exhaustive search over them. A randomized simulated-annealing algorithm, which is listed below, is used instead and yields quite satisfactory results.

Another function, $Force$, is defined to facilitate the computation. Given ordering O, $Force_i(O)$ is defined as

$$Force_i(O) = \sum_{\substack{j \\ C_{i,j} \neq 0}} (Rank(j) - Rank(i))$$

[7]The number "4" isn't crucial at all. The sole purpose is to make the evaluation function grow fast with $Cross_i$.

In other words, *Force* evaluates how "likely" a given component should be moved to another location in O and in which direction it should go . For instance, if a component i has all its coupled peers after it, $Force_i(O)$ will have a large positive value (all the summed values are positive), thus it is strongly indicated that i should be moved to the end of the ordering.

The algorithm to find a (near) optimal ordering is:

```
order findBestOrder()
{
    overallResult=+INFINITY;
    oldResult=+INFINITY;
    do (3*N times) {
        Randomly initilize Order[i];
        for(step=N/2; step>0;step--) {
            changed=true;
            while(changed) do{
                changed=false;
                Force=Force(O);
                while(1) do{
                    find i such that Force[i]=max(Force);
                    if(Force[i]==0) break;
                    if(Force[i]>0) {
                        newPosition=i+step;
                        if(newPosition>N-1) newPosition=N-1;
                    }
                    esle {
                        newPosition=i-step;
                        if(newPosition<0) newPosition=0;
                    }
                    swap element i and newPosition in Order;
                    newResult=eval(Order);
```

```
                    if(newResult<oldResult) {
                        changed=true;
                        oldResult=newResult;
                        step=N/2;
                        break;
                    }
                    else {
                        switch element i and newPosition in Order;
                    }
                    Force[i]=0;
                }
            }
        }
        if(oldResult<overallResult) {
            bestOrder=Order;
            overallResult=oldResult;
        }
    }
    return bestOrder;
}
```

5.3.2 Incremental computation on buffer sizes

For a system with quite a few nontrivial components, a problem of "intermediate node number explosion" will become quite serious. That's to say, although the number of nodes in the final results is not large, it will be tens of times larger during the synthesis process, or more accurately, during the generation of reachable or coreachable structural predicates. A graph of $|nodes|$[8] vs. time is shown in Fig 6.13. This has an obvious impact on the performance, since the time complexity of the algorithm is

[8] The number of nodes in a given IDD.

directly based on $|nodes|$.

This is kind of an intrinsic problem of IDD. An IDD is actually a classification of states based on their structural information and it is obtained by an iterative process. In other words, the IDD is formed "little by little", therefore we just cannot expect it to "stay" in the optimal form all the way.

Fortunately, for a system with a lot of buffer-like components, as many practical systems have, there is a simple way to deal with this problem.

Actually, we *can* get good results for the final IDDs; the problem is the IDDs become too large at the half-way point of the computation. Thus if we can always keep those IDDs near to the final form, we would probably get a good result. This can be achieved by enlarging the system step by step. For a system with a maximum buffer size of M, we can start with a system with the same structure but with all buffers' (and buffer-like components,) sizes set to 1. After we complete the system with buffer size of 1, we enlarge all buffers to the size 2, and generate the IDDs again. After that we enlarge the buffers to size 3 ... until every buffer reaches its final size. In this way, we always keep the system near the "final" stage. The performance improvement gained by the incremental approach will be reported in Chapter 6.

We can extend the incremental approach to a finer grained one, i.e., when expanding the size of buffers, we enlarge those buffers one by one. However, this introduces two many "idle" loops in the synthesis procedure, and thus only rarely improves the performance.

For non-buffer-like components, if their size is large, we can also apply the incremental approach to them. Although there is no "size" defined for such components, we can introduce the distance of one state from the initial state or the marking state. Such distance can be defined as the smallest number of transitions needed to bring the system from the initial state to the given state (or from the given state to the marking state). And then the "size" can be defined as the maximum value of the distance in one component so that the incremental approach is applicable. This is normally not required because non-buffer-like components are often quite small.

5.4 A conceptual complexity analysis

Unfortunately, it is very difficult to do an analytical analysis of complexity of this algorithm, because the time and space complexity are not directly related to any given factors or their easily computable functions. In previous sections, it is shown that the complexity is actually directly related to $|nodes|$ in the IDD representation of the structural predicates. However, $|nodes|$ can not be easily decided since it is related to both the parameters (such as number of components, numbers of states of each component, number of events) and the underlying structure of the system(especially the pattern of event sharing). It is probably true that the determination of $|nodes|$ is as hard as the actual computation of the IDDs.

However, it is possible to analyze the complexity conceptually, that's to say, to estimate by intuition. The only thing needed to be done is the estimation of $|nodes|$, which is related to the structure of the system. There are three IDDs in STCT: the ones representing reachable, coreachable and forbidden predicates, respectively. They are all generated by a similar procedure and principally they are all "reachable" predicates, in the sense that coreachable predicates are actually "reversed reachable" ones, and forbidden predicates are actually "uncontrollably reachable" ones. Thus, only the IDD corresponding to the reachable predicate will be discussed here. The same discussion can be applied to the other two kinds of IDDs without substantial modification.

5.4.1 Degree of coupling

A natural measurement of the complexity of the structure of a system is the degree of coupling. A topology of a system can be drawn, according to the scheme of events sharing of the system. As an example, the topology of a Small Factory[9] is shown in Fig 5.5. In a topology graph, each node represents one component of the system (plant component or specification component), and the edges represent the number of share events between each pair of components. Those edges of 0 sharing are omitted

[9]Detailed information about the Small Factory can be found in [Won99]

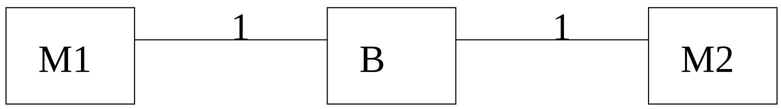


Figure 5.5: Topology of the Small Factory

from the graph. Obviously, the more edges in the graph, the more closely coupled the system is.

One possible and very "rough" definition of the degree of coupling can be

Definition 5.8 Degree of coupling of a system

$$DC = \frac{\sum_{i,j,i \neq j} M_{i,j}}{N}$$

where

$$M_{i,j} = \begin{cases} 1 & \text{component i and j have shared events} \\ 0 & \text{otherwise} \end{cases}$$

and N is the number of components in the system.

In other words, the degree of coupling is the average number of components coupled with a given component. For the Small Factory, it is easily seen that $DC = 1.33$.

5.4.2 Estimate of $|nodes|$

As a system becomes more complicated (e.g. a production line becomes longer), the topology graph will certainly become larger. However, we might expect that the degree of coupling remains constant. This is simply because of the nature of

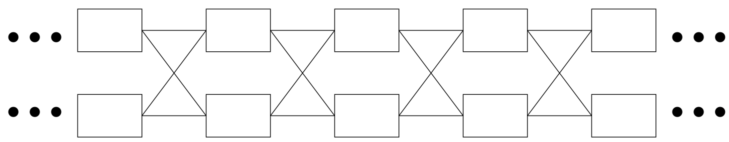


Figure 5.6: Topology of a production line

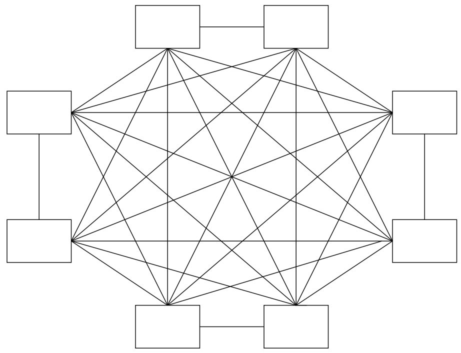


Figure 5.7: Topology of a production manifold

human intelligence – we are able and willing to build a "production line" instead of a "production manifold"(in which components are densely cross-linked). In other words, most man-made systems are probably more like Fig 5.6 than Fig 5.7. Thus we can expect DC to be constant when N increases.

Suppose the following:

1. Each component has exactly K states, i.e., there are exactly K branches in each non-terminal node of the IDD.

2. In the IDD, the component of $level_i$ is only coupled with components in the adjacent C levels.

Remark: The first assumption is made for simplicity only, and it places no actual constraint on the system. The second one is not true for all systems. Although we will always find that the value of DC is always much less than N, the value of C, which is closely related to DC, is probably much larger than DC. Bad examples can

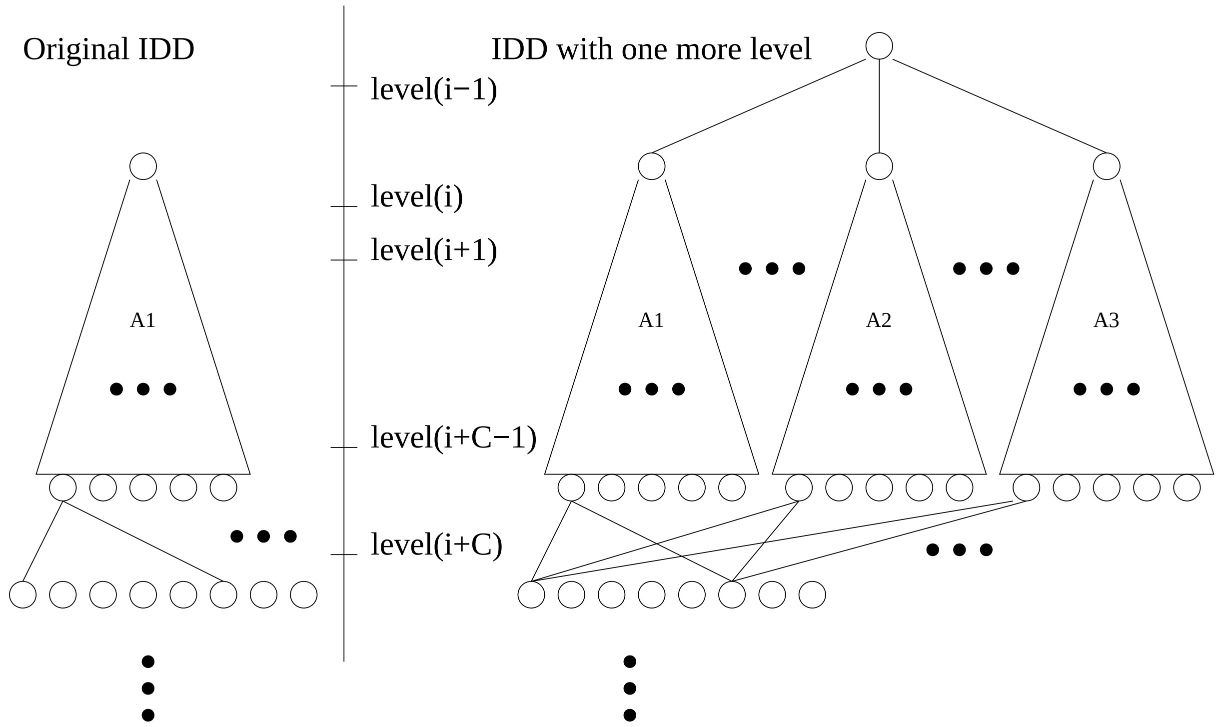


Figure 5.8: The growing of an IDD

be easily found, such as a system in the form of a loop. In such a system, every two components are actually "coupled". However, for most practical systems, those loops are local, i.e. they cover only a small section of the whole system. And even for a system with a huge loop, there is still "partial modularity" for most of the branches in the IDD. In short, for a practical system, we should be able to find such a constant C, at least for almost all of the branches.

With the above assumption, it can be estimated that

$$|nodes| = O(N \times K^C)$$

This is illustrated in Fig 5.8.

We can (theoretically) build the IDD by adding levels one by one. Assume we have an IDD with N levels and we want to add another level to the top of the IDD. With the above assumption, the added level can at most have a close relationship with the top C levels, therefore we can expect those "far down" levels will not be severely disturbed by the adding of the top level. The adding of top level can have a big impact on the top levels, but at most only K^C more nodes can be added to the IDD. (This is because each Ai shown in the graph can contain at most $(K^C - 1)/(K - 1)$ nodes and there are at most $K - 1$ such Ai that can be added to the IDD.) Thus one more level means K^C more nodes, and we have the above formula.

Here we get a very interesting implication from this estimate:

- For a practical system, the control problem may be not exponential w.r.t. N.

Of course this estimate is rather inaccurate and ideal. However it does give us some idea about the complexity of a system.

Chapter 6

Examples

Three examples are provided in this chapter to show the performance of the STCT algorithm. The first section is a tutorial example using the Transfer Line. It provides a detailed illustration of the whole computational procedure, the optimizations, and the complexity issue. The second example is due to Michael Seidl and actually was proposed during a discussion[1] between Dr. Michael Seidl and the author's supervisor Prof. Wonham. The third example is directly adopted from Section 8.2 of [DAJ95] and is the most complex example in that book.

In examples, a lot of "bufferizing" is used to efficiently model multiple instances of the same entity. A detailed explanation of "bufferizing" can be found in Appendix B.

6.1 Transfer Line – a tutorial example

We provide a simple example to illustrate the synthesis algorithm presented in the previous sections. The predicates will be presented as tables.

The automaton model of a simple Transfer Line is shown in Fig 6.1. Transfer Line consists of two machines M1 and M2, followed by a test unit TU, linked by buffers B1 and B2, in the configuration shown. A workpiece tested by TU may be accepted or rejected; if accepted, it is released from the system; if rejected, it is returned to

[1]This discussion was stimulated by Seidl's paper in WODES2000 [SS00].

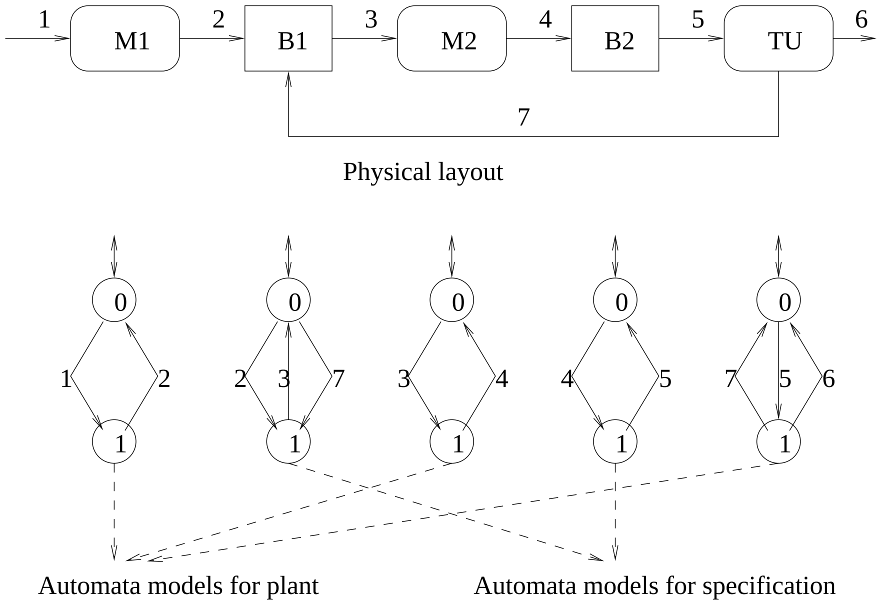


Figure 6.1: Transfer Line

B1. Thus the structure incorporates "material feedback". The specification is simply that B1 and B2 must be protected against underflow and overflow. The events are defined as shown in Table 6.1:

6.1.1 A conceptual walk-through of the synthesis procedure

First let see how the procedure works conceptually.

In CTCT, the first step is to synchronize everything together. The whole transition

event	controllable	definition
1	Y	M1 takes a raw workpiece into the system
2	N	M1 sends one workpiece to B1
3	Y	M2 takes a workpiece
4	N	M2 sends a workpiece to B2
5	Y	TU takes one workpiece from B2
6	N	TU releases one workpiece
7	N	TU returns one workpiece to B1

Table 6.1: Event definition in Transfer Line

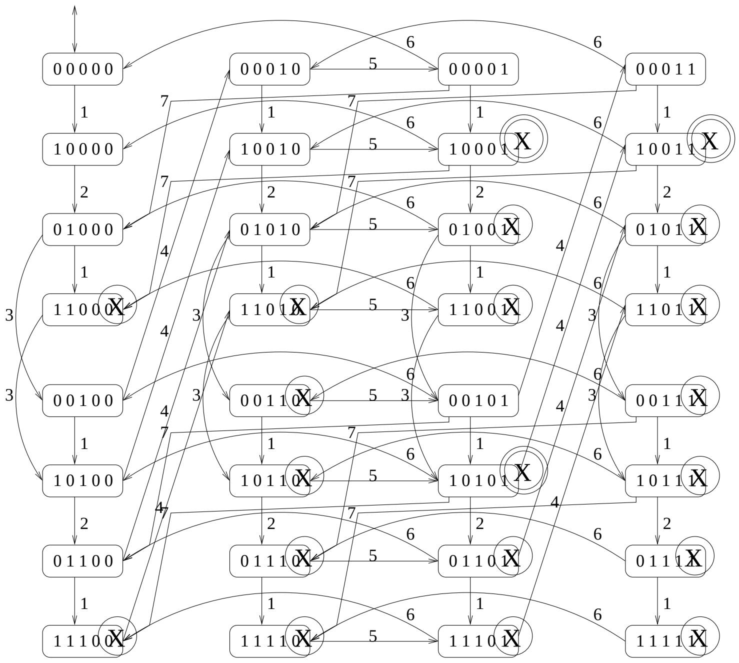


Figure 6.2: Synchronized Transfer Line during synthesis (1)

graph of the synchronized Transfer Line is shown in Fig 6.2.

Note that every state in the transition graph is labeled with the unique vector of 5 state variables corresponding to the five individual modules. For example, the initial state corresponds to the (0,0,0,0,0) state combination, i.e. each initial module is in its own initial state.

Then those states which violate the controllability property have to be forbidden. That's to say, if at a given state there is an uncontrollable event disabled by the specification but enabled by the plant, this state must be forbidden. All these states are marked by a single-circled "X"in Fig 6.2.

Then we should remove those states which enable uncontrollable event(s) going to a forbidden states. These states are marked by a double-circled "X" in Fig 6.2.

After removing all the forbidden states and the associated transitions, we get the transition graph as shown in 6.3. Then we should try to verify the nonblocking property. It is discovered that one state is reachable but not coreachable, which is marked by a triple-circled "X". Now we have some states which have uncontrollable

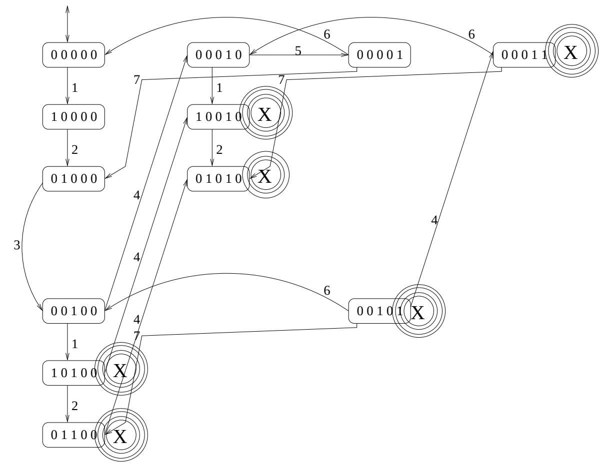


Figure 6.3: Synchronized Transfer Line during synthesis (2)

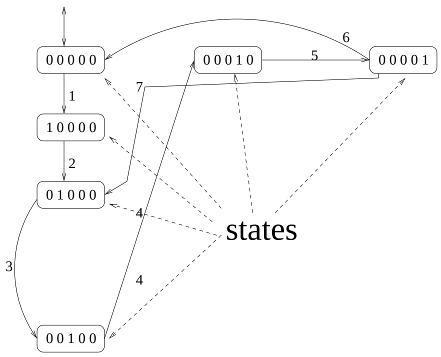


Figure 6.4: Synchronized Transfer Line after synthesis

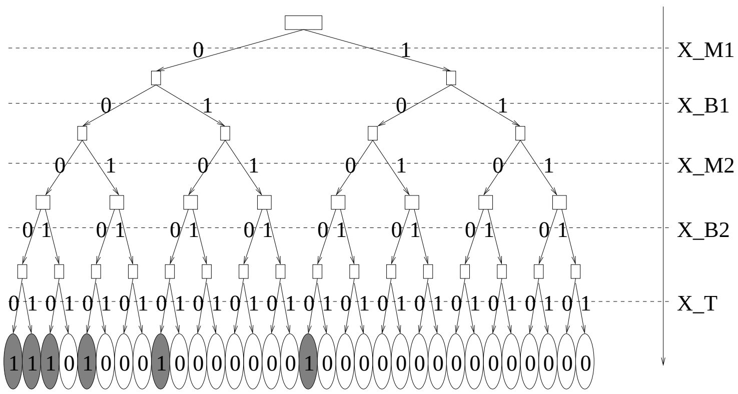


Figure 6.5: Removals of forbidden states represented by predicates

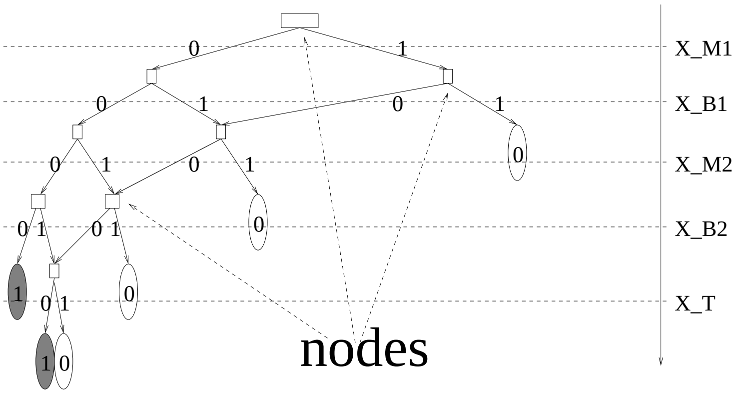


Figure 6.6: Synthesized Transfer Line represented by an IDD

X_{M1}	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
X_{B1}	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
X_{M2}	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
X_{B1}	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
X_T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
P_1				X				X				X	X	X	X	X			X	X			X	X			X	X	X	X	X	X
P_2																		X				X				X						
P_3											X																					
P_4						X	X			X											X				X							
P	1	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.2: Forbidden states removal procedure

events leading to one of the forbidden states again. They are marked by a quadruple-circled "X".

Then we verify the nonblocking property again. We can find that every reachable state is coreachable. Thus the system is both controllable and nonblocking. The synthesis procedure stops here. The resulting transition graph is shown in Fig 6.4.

Actually we are simply removing forbidden states in successive batchs. Since there is a unique mapping between all the states in the synchronized transition structure and the vectors of the 5 state variables, we can represent the batches of removed forbidden states by structural predicates defined on the five state variables. The four stages of batch removal and the remaining states are shown in the Table 6.2 (P_1, P_2, P_3, P_4 and P respectively).

In the real computation, all the structural predicates are represented by IDDs. For example, the predicate representing the final result can be expressed by a function graph as shown in Fig 6.5. It can be verified easily that this function graph holds the same information as listed in Table 6.2. Then this function graph can be reduced to an IDD as shown in Fig 6.6. It is quite easy to see that the unreduced and reduced graphs store the same information.

Note that we never construct the full function graph (as shown in Fig 6.5) or the table of structural predicates (as shown in Table 6.2) in the real computation. All the computation is done on IDDs using the procedures defined in the previous chapter.

6.1.2 A computation walk-through of the synthesis procedure

Now we show how the computational procedure works to complete the synthesis.

1. P_{bad1} in the first cycle is shown in Table 6.3[2]. As we see, 17 states should not be reached, because at each of them there is an uncontrollable event which is enabled by the plant but disabled by the specification. Those states are $(1, 1, *, *, *)$ for event 2, $(*, *, 1, 1, *)$ for event 4, and $(*, 1, *, *, 1)$ for event 7. These states are those marked by a single-circled "X" within in Fig 6.2.

2. Next, P_{bad2} in the first cycle is shown in Table 6.4. Three states should not be reached, because at each of them there is uncontrollable event enabled and leading to a forbidden state: $(1, 0, 0, 0, 1) \rightarrow (1, 1, 000)$ by event 7, $(1, 0, 1, 0, 1) \rightarrow (1, 1, 1, 0, 0)$ by event 7, and $(1, 0, 0, 1, 1) \rightarrow (1, 1, 0, 1, 0)$ by event 7. These states are those marked by a double-circled "X" in Fig 6.2

3. In the first cycle, the computed P_{re} is shown in Table 6.5. Actually all non-forbidden states here are reachable.

[2]This and the remaining tables should be read in the following way: the table itself is separated into two halves (left and right) in order to save space. In each half there are 6 columns. The left 5 columns are the value of the elements in the state vector, while the last column indicates whether the given predicate is satisfied by the vector specified by the left 5 column. For example, the first row of the right half means $(0, 0, 0, 0, 1) | \neq P_{bad}$.

4. In the first cycle, the computed P_{cr} is shown in Table 6.6. Here, one state, $(0, 1, 0, 1, 0)$, is not forbidden but not coreachable. This state is marked by triple-circled "X" in Fig 6.3

5. In the first cycle, the value of $P_{newgood}$ is shown in Table 6.7. Clearly $P_{newgood} \subset P_{good}$; thus we need to do another cycle.

6. The second time P_{bad1} is shown in Table 6.8. The newly found forbidden state $(0, 1, 0, 1, 0)$ is added to P_{bad1}.

7. The second time P_{bad2} is shown in Table 6.9. 5 states are added to P_{bad} because they lead to forbidden states uncontrollably. These states are those marked by quadruple-circled "X" in Fig 6.3

8. The second time P_{re} is shown in Table 6.10. Now only 6 states are reachable.

9. The second time P_{cr} is shown in Table 6.11. 6 states are coreachable.

10. The second time $P_{newgood}$ is shown in Table 6.12. As we see, all reachable states are coreachable. However we still have $P_{newgood} \subset P_{good}$, so another cycle is necessary.

11. The third time P_{bad1} is shown in Table 6.13.

12. The third time P_{bad2} is shown in Table 6.14.

13. The third time P_{re} is shown in Table 6.15.

14. The third time P_{cr} is shown in Table 6.16.

15. The third time $P_{newgood}$ is shown in Table 6.17. As we see, all reachable states are coreachable. Finally we have $P_{newgood} = P_{good}$, so $P_{sup} = P_{newgood}$ and the algorithm terminates here. This corresponds to the IDD shown in Fig 6.6.

The transition structure can be easily drawn, as shown in Fig 6.7. This agrees with the result from CTCT, which is shown in Fig 6.4.

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	1	0
0	1	0	0	0	0	0	1	0	0	1	1
1	1	0	0	0	1	1	1	0	0	1	1
0	0	1	0	0	0	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0	1	0
0	1	1	0	0	0	0	1	1	0	1	1
1	1	1	0	0	1	1	1	1	0	1	1
0	0	0	1	0	0	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	1	1	0
0	1	0	1	0	0	0	1	0	1	1	1
1	1	0	1	0	1	1	1	0	1	1	1
0	0	1	1	0	1	0	0	1	1	1	1
1	0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1

Table 6.3: P_{bad1} in the first loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	1	1*
0	1	0	0	0	0	0	1	0	0	1	1
1	1	0	0	0	1	1	1	0	0	1	1
0	0	1	0	0	0	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0	1	1*
0	1	1	0	0	0	0	1	1	0	1	1
1	1	1	0	0	1	1	1	1	0	1	1
0	0	0	1	0	0	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	1	1	1*
0	1	0	1	0	0	0	1	0	1	1	1
1	1	0	1	0	1	1	1	0	1	1	1
0	0	1	1	0	1	0	0	1	1	1	1
1	0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1

Table 6.4: P_{bad2} in the first loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	1	1
1	0	1	0	0	1	1	0	1	0	1	0
0	1	1	0	0	1	0	1	1	0	1	0
1	1	1	0	0	0	1	1	1	0	1	0
0	0	0	1	0	1	0	0	0	1	1	1
1	0	0	1	0	1	1	0	0	1	1	0
0	1	0	1	0	1	0	1	0	1	1	0
1	1	0	1	0	0	1	1	0	1	1	0
0	0	1	1	0	0	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1	1	1	1	0

Table 6.5: P_{re} in the first loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	1	1
1	0	1	0	0	1	1	0	1	0	1	0
0	1	1	0	0	1	0	1	1	0	1	0
1	1	1	0	0	0	1	1	1	0	1	0
0	0	0	1	0	1	0	0	0	1	1	1
1	0	0	1	0	1	1	0	0	1	1	0
0	1	0	1	0	0*	0	1	0	1	1	0
1	1	0	1	0	0	1	1	0	1	1	0
0	0	1	1	0	0	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1	1	1	1	0

Table 6.6: P_{cr} in the first loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	1	1
1	0	1	0	0	1	1	0	1	0	1	0
0	1	1	0	0	1	0	1	1	0	1	0
1	1	1	0	0	0	1	1	1	0	1	0
0	0	0	1	0	1	0	0	0	1	1	1
1	0	0	1	0	1	1	0	0	1	1	0
0	1	0	1	0	0*	0	1	0	1	1	0
1	1	0	1	0	0	1	1	0	1	1	0
0	0	1	1	0	0	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1	1	1	1	0

Table 6.7: $P_{newgood}$ in the first loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	1	1
0	1	0	0	0	0	0	1	0	0	1	1
1	1	0	0	0	1	1	1	0	0	1	1
0	0	1	0	0	0	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0	1	1
0	1	1	0	0	0	0	1	1	0	1	1
1	1	1	0	0	1	1	1	1	0	1	1
0	0	0	1	0	0	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	1	1	1
0	1	0	1	0	1*	0	1	0	1	1	1
1	1	0	1	0	1	1	1	0	1	1	1
0	0	1	1	0	1	0	0	1	1	1	1
1	0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1

Table 6.8: P_{bad1} in the second loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	1	1
0	1	0	0	0	0	0	1	0	0	1	1
1	1	0	0	0	1	1	1	0	0	1	1
0	0	1	0	0	0	0	0	1	0	1	1*
1	0	1	0	0	1*	1	0	1	0	1	1
0	1	1	0	0	1*	0	1	1	0	1	1
1	1	1	0	0	1	1	1	1	0	1	1
0	0	0	1	0	0	0	0	0	1	1	1*
1	0	0	1	0	1*	1	0	0	1	1	1
0	1	0	1	0	1	0	1	0	1	1	1
1	1	0	1	0	1	1	1	0	1	1	1
0	0	1	1	0	1	0	0	1	1	1	1
1	0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1

Table 6.9: P_{bad2} in the second loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0	1	0
0	1	1	0	0	0	0	1	1	0	1	0
1	1	1	0	0	0	1	1	1	0	1	0
0	0	0	1	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	1	1	0
0	1	0	1	0	0	0	1	0	1	1	0
1	1	0	1	0	0	1	1	0	1	1	0
0	0	1	1	0	0	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1	1	1	1	0

Table 6.10: P_{re} in the second loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0	1	0
0	1	1	0	0	0	0	1	1	0	1	0
1	1	1	0	0	0	1	1	1	0	1	0
0	0	0	1	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	1	1	0
0	1	0	1	0	0	0	1	0	1	1	0
1	1	0	1	0	0	1	1	0	1	1	0
0	0	1	1	0	0	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1	1	1	1	0

Table 6.11: P_{cr} in the second loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0	1	0
0	1	1	0	0	0	0	1	1	0	1	0
1	1	1	0	0	0	1	1	1	0	1	0
0	0	0	1	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	1	1	0
0	1	0	1	0	0	0	1	0	1	1	0
1	1	0	1	0	0	1	1	0	1	1	0
0	0	1	1	0	0	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1	1	1	1	0

Table 6.12: $P_{newgood}$ in the second loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	1	1
0	1	0	0	0	0	0	1	0	0	1	1
1	1	0	0	0	1	1	1	0	0	1	1
0	0	1	0	0	0	0	0	1	0	1	1
1	0	1	0	0	1	1	0	1	0	1	1
0	1	1	0	0	1	0	1	1	0	1	1
1	1	1	0	0	1	1	1	1	0	1	1
0	0	0	1	0	0	0	0	0	1	1	1
1	0	0	1	0	1	1	0	0	1	1	1
0	1	0	1	0	1	0	1	0	1	1	1
1	1	0	1	0	1	1	1	0	1	1	1
0	0	1	1	0	1	0	0	1	1	1	1
1	0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1

Table 6.13: P_{bad1} in the third loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	1	1
0	1	0	0	0	0	0	1	0	0	1	1
1	1	0	0	0	1	1	1	0	0	1	1
0	0	1	0	0	0	0	0	1	0	1	1
1	0	1	0	0	1	1	0	1	0	1	1
0	1	1	0	0	1	0	1	1	0	1	1
1	1	1	0	0	1	1	1	1	0	1	1
0	0	0	1	0	0	0	0	0	1	1	1
1	0	0	1	0	1	1	0	0	1	1	1
0	1	0	1	0	1	0	1	0	1	1	1
1	1	0	1	0	1	1	1	0	1	1	1
0	0	1	1	0	1	0	0	1	1	1	1
1	0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1

Table 6.14: P_{bad2} in the third loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0	1	0
0	1	1	0	0	0	0	1	1	0	1	0
1	1	1	0	0	0	1	1	1	0	1	0
0	0	0	1	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	1	1	0
0	1	0	1	0	0	0	1	0	1	1	0
1	1	0	1	0	0	1	1	0	1	1	0
0	0	1	1	0	0	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1	1	1	1	0

Table 6.15: P_{re} in the third loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0	1	0
0	1	1	0	0	0	0	1	1	0	1	0
1	1	1	0	0	0	1	1	1	0	1	0
0	0	0	1	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	1	1	0
0	1	0	1	0	0	0	1	0	1	1	0
1	1	0	1	0	0	1	1	0	1	1	0
0	0	1	1	0	0	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1	1	1	1	0

Table 6.16: P_{re} in the third loop

Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}	Q_{M1}	Q_{B1}	Q_{M2}	Q_{B2}	Q_{TU}	P_{bad}
0	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	0	1	1	0	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	1	0
1	0	1	0	0	0	1	0	1	0	1	0
0	1	1	0	0	0	0	1	1	0	1	0
1	1	1	0	0	0	1	1	1	0	1	0
0	0	0	1	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	0	0	1	1	0
0	1	0	1	0	0	0	1	0	1	1	0
1	1	0	1	0	0	1	1	0	1	1	0
0	0	1	1	0	0	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1	1	1	1	0

Table 6.17: $P_{newgood}$ in the third loop

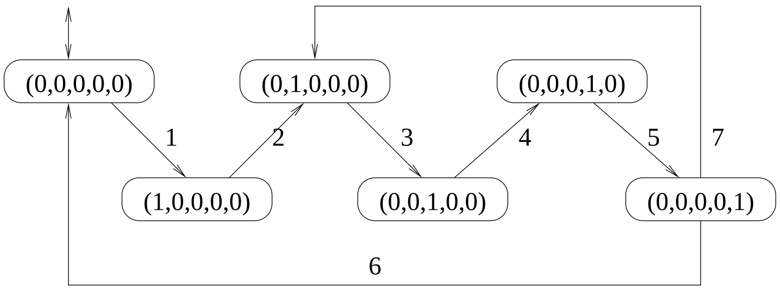


Figure 6.7: Transition structure of the controlled simple Transfer Line

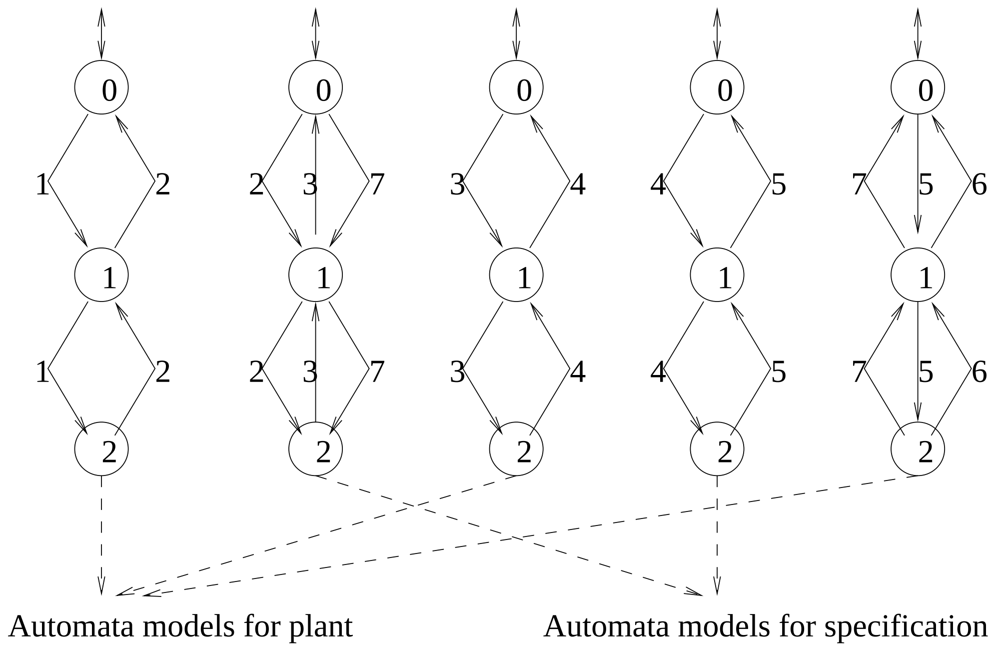


Figure 6.8: Automaton model for Transfer Line with M=2

6.1.3 Horizontal expansion of Transfer Line

As we see, Transfer Line can be easily modeled in RW approach. Also, it can be easily computed by the CTCT approach. The STCT doesn't show any considerable advantage in the previous section.

However, real-life systems are not always so simple. A typical FMS comprises numerous resources - buffers, machines, robots, automatic guided vehicles. And normally, those resources have multiple instances. For example, we may have two identical machines to perform M1's task, and thus increase the efficiency of system. Also the buffers B1 and B2 can have larger capacities. If we generalize Transfer Line by allowing everything in it to have multiple instances, it will become more complicated. For example, assuming everything has two instances, we will get the automaton model of Transfer Line as shown in Fig 6.8. For simplicity of explanation, we denote the number of instances as M. Thus the system being studied is denoted as *Transfer Line with M=2* from now on.

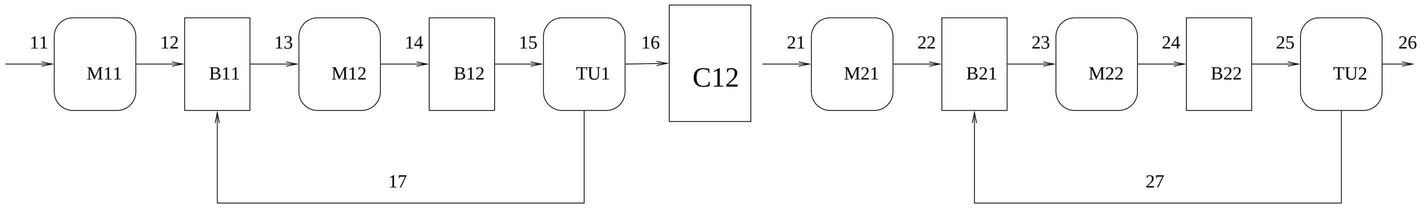


Figure 6.9: A simple Production Line consisting of two Transfer Lines

Transfer Line can always be further extended to allow more instances. The number of states of the system will increase accordingly, and so does the computational burden.

6.1.4 Vertical expansion of Transfer Line

We can also generalize Transfer Line in another direction. Assume our production line consists of 2 Transfer Lines, as shown in Fig 6.9. This system consists of 2 Transfer Lines, which are working independently except that the output of Transfer Line 1 is connected to the input of Transfer Line 2.

All the names of the components and events are prefixed by the number of the Transfer Line. For example, event "11" corresponds the original event '1' in Transfer Line one. The two Transfer Lines are connected by a simple buffer "C12" (connector $1 \rightarrow 2$).

6.1.5 STCT performance on different configurations of Transfer Lines

We can still apply the horizontal expansion on this already vertically expanded Transfer Line, thereby getting a system with variable complexity to try out STCT. A performance comparison between STCT and CTCT is given in Table 6.18.

Obviously STCT is much more efficient than CTCT and there is much potential of STCT unused to compute this system. In order to get a larger example, we connect

M	$\|Q\|$	$\|nodes\|$	CTCT Time	STCT Time	CTCT Mem	STCT Mem
1	66	19	<< 1sec	< 1sec	< 1M	< 1M
2	4536	36	0.5sec	< 1sec	2M	< 1M
3	84800	57	60sec	1sec	10M	1M
4	821250	82	> 5000sec	1.5sec	120M	1.5M
5	5.29×10^6	111	n/a	2sec	n/a	1.6M
7	1.02×10^8	181	n/a	8sec	n/a	2M
10	2.81×10^9	316	n/a	20sec	n/a	3M

Table 6.18: Performance of CTCT and STCT vs. M on expanded Transfer Line (L=2)

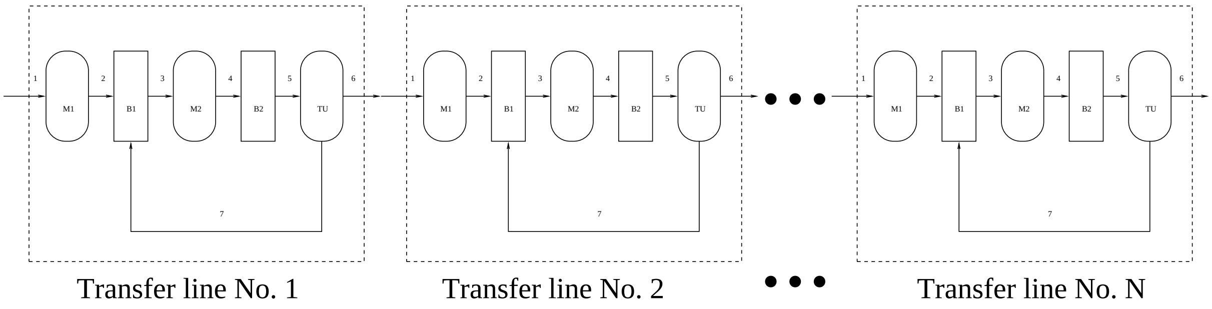


Figure 6.10: Production Line

arbitrary numbers of Transfer Lines together, as shown in Fig 6.10.

As before, we denote the number of components in the total system by N. For a simple transfer line, $N = 5 = 3 + 2$ (No. of plant modules + No. of specification modules). For a systems built by connecting L simple Transfer Lines, we have

$$N = 5L + L - 1,$$

because each Transfer Line has L components and we need $L - 1$ buffers to connect them together.

Fixing $M = 5$ and $M = 10$, we can obtain $|nodes|$ in the IDDs representing the controlled system state space as shown in Fig 6.11.

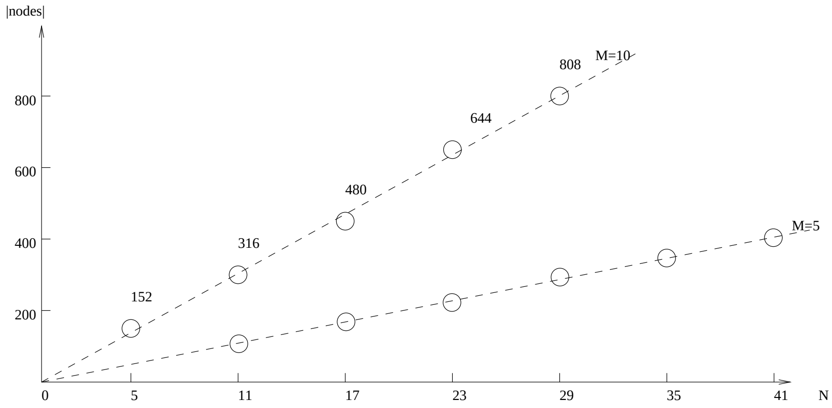


Figure 6.11: $|nodes|$ vs. M and N

M	$\|Q\|$	$\|nodes\|$	STCT Time	STCT Memory (Bytes)
1	7986	41	6sec	$<$ 1M
2	5.29×10^7	76	11sec	1M
3	2.38×10^{10}	119	15sec	1M
4	2.74×10^{12}	170	62sec	1.8M
5	1.34×10^{14}	229	76sec	2.5M
7	6.58×10^{16}	371	98sec	4M
10	6.74×10^{19}	644	340sec	8M
15	2.59×10^{23}	1259	955sec	15M

Table 6.19: Performance of STCT vs. M on expanded Transfer Line (L=4)

We can see a clear linear relationship between $|nodes|$ and N. This shows a sharp contrast to the exponential relationship between $|Q|$ and N [3].

The relationship between N and the computing time is a bit more complex. Some typical values are shown in Fig 6.12.

Fixing $L = 4$, we can get some idea about STCT's performance vs. M, as shown in Table 6.19.

[3] $|Q|$ is the number of reachable states in the controlled system.

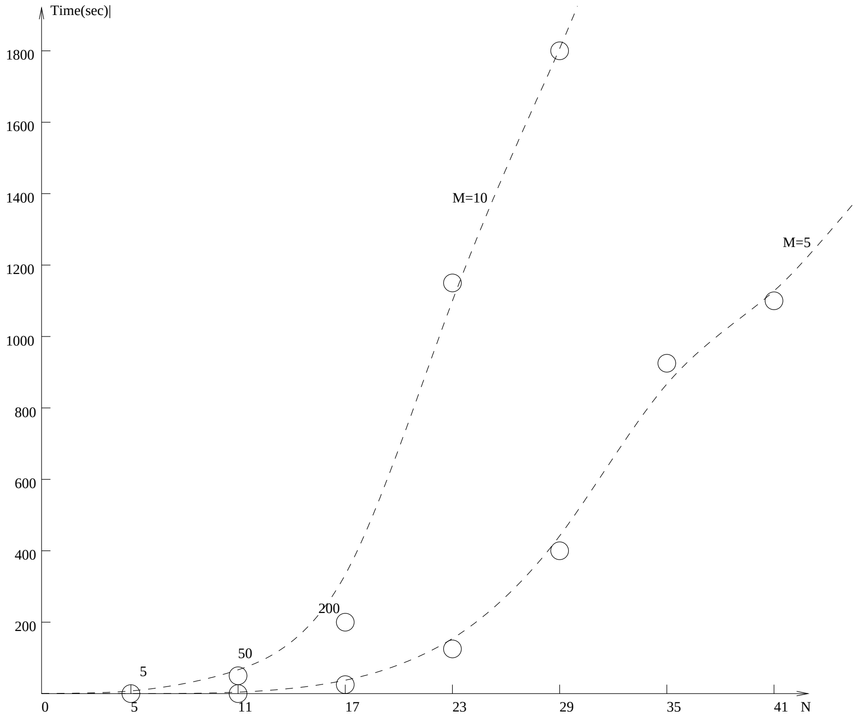


Figure 6.12: Time usage vs. M and N

M	$\|Q\|$	$\|nodes\|$ (optimal)	$\|nodes\|$ (random)	Time (optimal)	Time (random)	Memory (optimal)	Memory (random)
1	66	19	41	$<$ 1sec	$<$ 1sec	$<$ 1M	$<$ 1M
2	4536	36	115	$<$ 1sec	2sec	1M	1.7M
5	5.29×10^6	111	727	3sec	27sec	1.6M	5M
7	1.02×10^8	181	1610	8sec	70sec	2M	10M
10	2.81×10^9	316	3947	20sec	300sec	3M	30M

Table 6.20: Performance of ordered and unordered STCT vs. M (L=2)

6.1.6 Effect of optimizations

The performance improvement given by various types of optimizations is shown in this section.

First let's look at the contribution of variable ordering. Consider the configuration with $L = 2$. Table 6.20 shows the difference between an optimized ordering and a random ordering.

Obviously we can get big performance gain by a good ordering of the components. And we can see this gain becomes larger when the system becomes more complicated.

For a complex system, the incremental approach against M becomes very important. For a system with $M = 10$ and $L = 2$, using the optimal ordering, the variation of $|nodes|$ during the first computation of reachability state space is shown in Fig 6.13.

Also, this difference becomes larger as the system becomes more complicated (i.e. as L increases). It is quite difficult to compute the system with $M = 10$ and $L = 4$ directly – the maximum of $|nodes|$ exceeds 11500, i.e. more than 20 times greater than the final $|nodes|$. Actually, for a simpler system such as the one with $M = 10$ and $L = 2$, the incremental approach is slower, because of more "idle" cycles. However, for a larger system, the incremental approach is much faster. For the system with $M = 10$ and $L = 4$, the difference is more than 20 times the time usage (200 seconds vs. more than 5000 seconds). Also, the finer grained incremental approach, will show the advantage after L becomes considerably large. The variations of $|nodes|$ during the computation of reachability state space are shown in Fig 6.14. As we can see, the finer

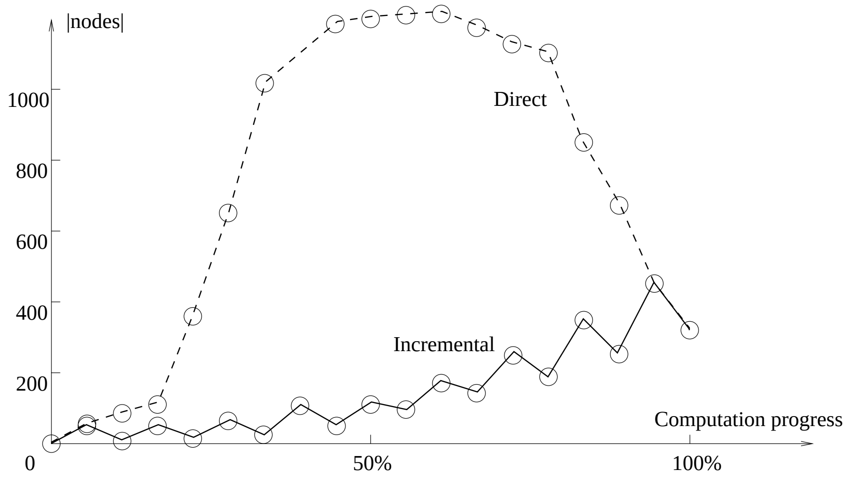


Figure 6.13: Variation of $|nodes|$ during direct and incremental computations

grained incremental approach can practically eliminate the problem of intermediate nodes explosion. However, a finer grained incremental approach also increases the $|nodes|$ "idle" cycles greatly, so generally the simple incremental approach has a better performance.

6.1.7 Empirical data vs. the complexity estimate

In the previous chapter we estimated that

$$|nodes| = O(N \times K^C)$$

This estimate can be tested against the data collected from Transfer Line example[4].

As we see in Fig 6.11, there is a rather obvious linear relationship between $|nodes|$ and N, i.e,

$$|nodes| \propto N$$

For a fixed L, i.e. fixed N, given k_1, K_2 and the corresponding $|nodes|$ (denoted

[4]Note that $K$is the number of states in the automaton while M is the number of instances. For a simple buffer or bufferized (see Appendix B) model, as Transfer Line, we always have $K = M + 1$.

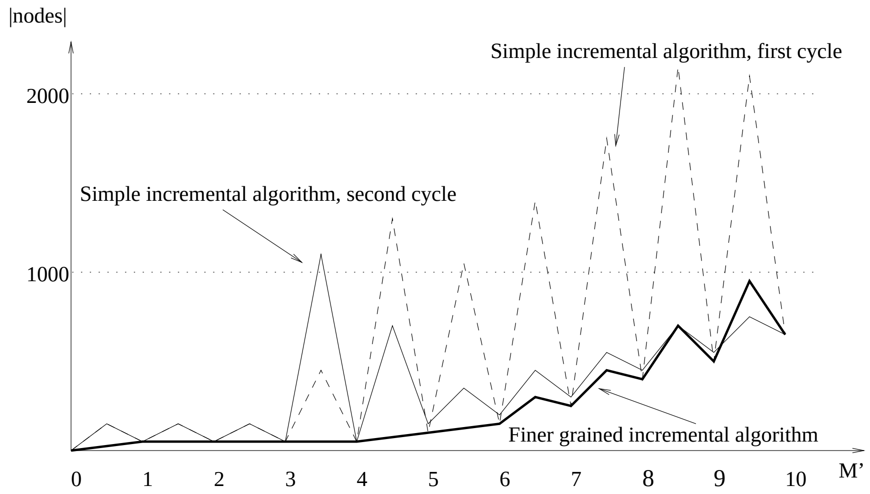


Figure 6.14: Variation of $|nodes|$ during computation

$|nodes|_{k_1}$ and $|nodes|_{k_2}$ respectively), we can estimate C by computing [5]

$$\hat{C} = \frac{log(\frac{|nodes|_{k_1}}{|nodes|_{k_2}})}{log(\frac{k_1}{k_2})}$$

For $L = 2, 4$, i.e. $N = 11, 23$, the values of $\hat{C}$ are shown in Table 6.21 and 6.22.

We can see that the values of $\hat{C}$ are pretty much the same for $N = 23$ and $N = 11$, that's to say, C is nearly constant w.r.t. N. If we plot the values as shown in Fig 6.15, we can estimate that $C \leq 1.8$.

From the above analysis, we can see that the estimate

$$|nodes| = O(N \times M^C)$$

fits the empirical data of Transfer Line very well. Although this example is somewhat artificial, it does suggest that it is quite possible to find a polynomial relationship

[5] This is obtained as follows. Assuming that $|nodes|_{k_1} = N * (k_1)^{\hat{C}}$, $|nodes|_{k_2} = N * (k_2)^{\hat{C}}$, divide the former by the latter and then apply log to both side of the equation, to get $log(\frac{|nodes|_{k_1}}{|nodes|_{k_2}}) = \hat{C} \times log(\frac{k_1}{k_2})$. Thus we can find $\hat{C}$ easily.

M	$\|nodes\|$	$\hat{C}$
1	19	N/A
2	36	0.92
3	57	1.13
4	82	1.27
5	111	1.35
7	181	1.45
10	316	1.56
15	621	1.66
30	2136	1.78

Table 6.21: Fitting of data $|nodes|$ vs. M, L=2 (N=11)

M	$\|nodes\|$	$\hat{C}$
1	41	N/A
2	76	0.89
3	119	1.10
4	170	1.25
5	229	1.33
7	371	1.43
10	644	1.54
15	1259	1.65

Table 6.22: Fitting of data $|nodes|$ vs. M, L=4 (N=23)

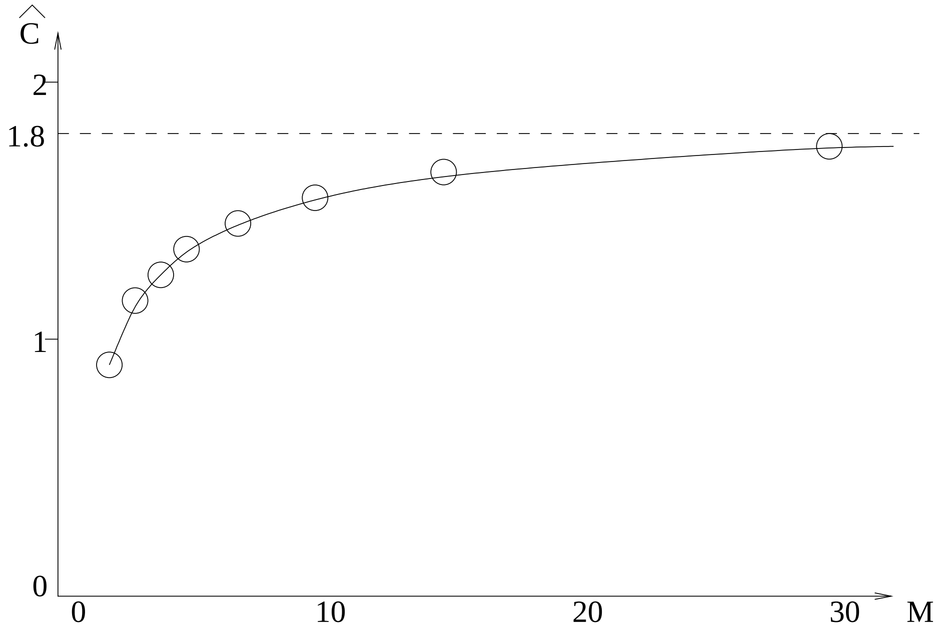


Figure 6.15: $\hat{C}$ vs. M for Transfer Line

between the real complexity of a system and the number of components in it.

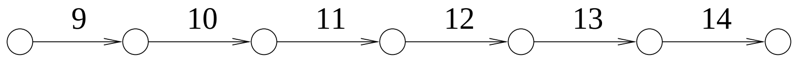
6.2 Workcell [6]

Fig. 6.16 shows the layout of Workcell consisting of 4 machines, M1, M2, M3, M4, and 2 robots, R1 and R2. All the six components can be easily modeled using automata, as shown in 6.17.

There are two kinds of products being manufactured in Workcell, namely product *Red* , and product *Green* . The events are defined as in Table 6.23.

The two kinds of products are produced according to the following recipes:

- Recipe for *Red* :



[6]This example is due to Michael Seidl and was proposed during a discussion between him and the author's supervisor Prof. Wonham.

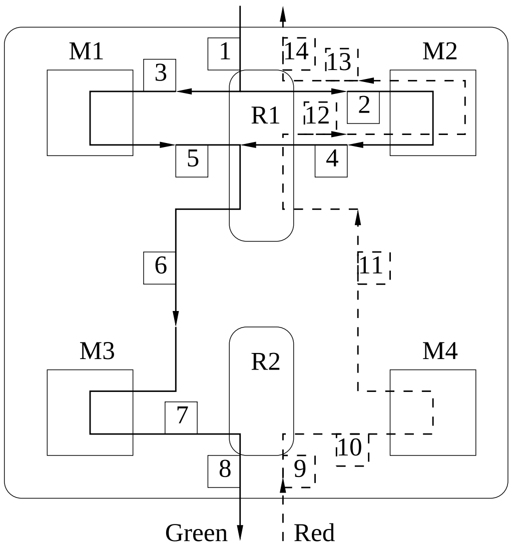


Figure 6.16: Physical layout of Workcell

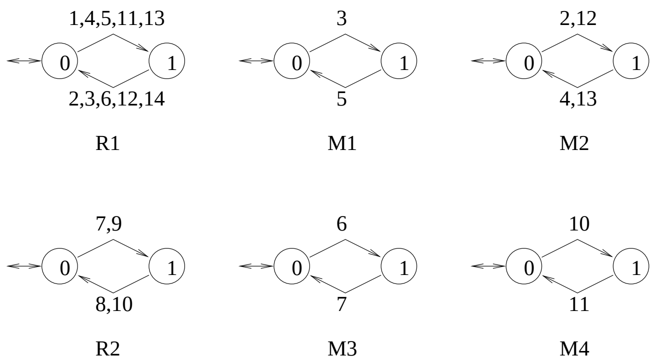
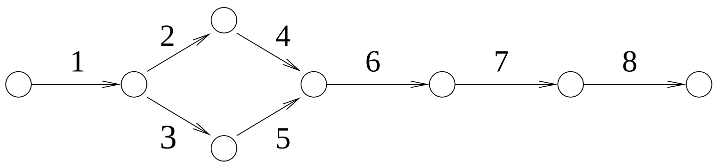


Figure 6.17: Automaton model for the plant of Workcell

event label	action
1	R1 gets a Green from outside
2	R1 sends a Green to M2
3	R1 sends a Green to M1
4	R1 gets a Green from M2
5	R1 gets a Green from M1
6	R1 sends a Green to M3
7	R2 gets a Green from M3
8	R2 sends a finished Green to outside
9	R2 gets a Red from outside
10	R2 sends a Red to M4
11	R1 gets a Red from M4
12	R1 sends a Red to M2
13	R1 gets a Red from M2
14	R1 sends a finished Red to outside

Table 6.23: Event definition in Workcell

- Recipe for *Green* :



Those recipes are actually constraints that the production procedure must follow, and thus they are modeled as specifications in the RW approach. Eleven automata are needed to represent the specification, 6 for *Green* and 5 for *Red* , as shown in 6.18.

As we see, Workcell can be easily modeled in RW approach. Assuming that every event is controllable,[7] we can now proceed to compute the supremal nonblocking sublanguage of the system. This works out fine too. Using TCT, it can be easily computed that the controlled Workcell will have 209 states.

However, real-life systems are not always so simple. A typical FMS comprises numerous finite resources - buffers, machines, robots, automatic guided vehicles. And

[7] Considering all events in a manufacturing system controllable is a common practice in Petri net literature, although it is not always appropriate. For instance, breakdown events (if any) and operation-completion events could be uncontrollable. Here all events are considered controllable to keep the automaton model comparable to its Petri net counterpart.

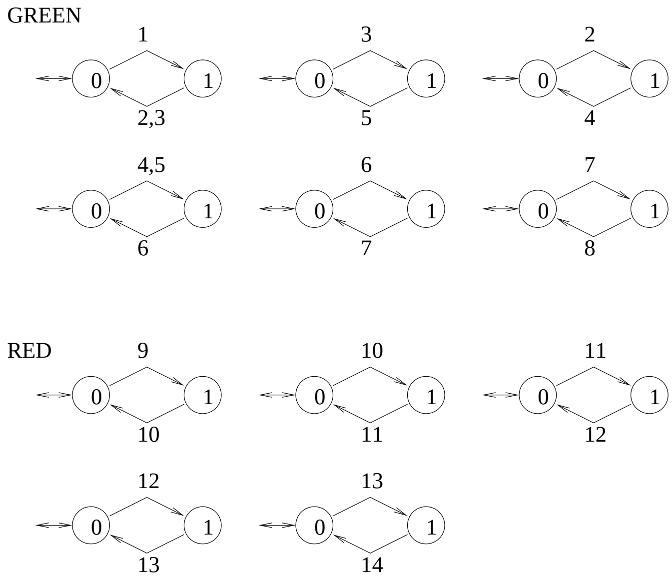


Figure 6.18: Automaton model for the specification of Workcell

normally, those resources have multiple instances. For example, we may have two identical robots to perform R1's task, thus increasing the efficiency of system. Besides, a lot of components in a FMS are buffer-like, such as conveyors. Suppose after R1 takes one raw *Green* from outside, it puts the *Green* on a conveyor so that the *Green* will be automatically taken to M1. In such a case, we should almost always allow more than one *Green* to appear on the conveyor, i.e, the conveyor behaves like a buffer with capacity > 1. Therefore we can always generalize Workcell by allowing multiple instance for each buffer-like component (actually every module is buffer-like in this example), as we have done in the Transfer Line example. As usual, we use M to denote the number of instances allowed and for simplicity of representation, every component is assumed to have same M value.

This time the increasing of M imposes an even larger penalty on CTCT. For $M = 2$, CTCT takes several minutes, hundreds of megabytes of memory and then yields a controlled automaton with 12912 states.

For a "real" system, we can expect that the buffers have capacities more than 2. But if we simply make everything in Workcell have 3 instances, we will get a

M	$\lvert Q \rvert$	$\lvert nodes \rvert$	CTCT Time	STCT Time	CTCT Mem	STCT Mem
1	205	79	$<$ 1sec	1sec	$<$ 1M	1M
2	12912	101	210sec	2sec	$>$ 500M	2M
4	1.9×10^6	192	n/a	6sec	n/a	3M
8	7.3×10^8	444	n/a	17sec	n/a	4M
10	5.8×10^9	666	n/a	30sec	n/a	5M
15	2.9×10^{11}	1116	n/a	78sec	n/a	7M
30	3.4×10^{14}	3546	n/a	691sec	n/a	17M

Table 6.24: Performance of CTCT and STCT vs. M on Workcell

system that can not computed by CTCT, given the current available computational power. As mentioned in chapter 2, for a system with a state size of N, the time and space complexities of CTCT are both $O(N^2)$. Actually, the controlled Workcell with M=3 will have 214892 states (computed by STCT). This is around 17 times the state size of Workcell with M=2. Thus it would need roughly $17^2 = 289$ times of time and memory compared with the case M=2. Clearly it is hardly possible with CTCT (needing more than 10 hours and 100G memory).

However, STCT can compute the case M=3 in several seconds. Actually, STCT can synthesize the system even when M is as large as 30 (even larger, if necessary). The results of the performance of CTCT and STCT are summarized in Table 6.24.

We can safely say that STCT can deal with the system given any practical setup. Of course, if we put $M = 1000$, STCT would not be able to produce the result in a feasible time, but then it is very unlikely that we will encounter a system with such huge buffers.

6.3 FMS: a flexible manufacturing system case study[1]

This section presents a case study of the implementation of an automata based controller for the operation of a flexible manufacturing system, FMS. The basic approach starts with the development of an automaton model for the system (plant). After that, the STCT is used to solve the synthesis problem of a nonblocking and maximally

[1]This example is directly adopted from Section 8.2 of [DAJ95] and it is the most complex example in that book.

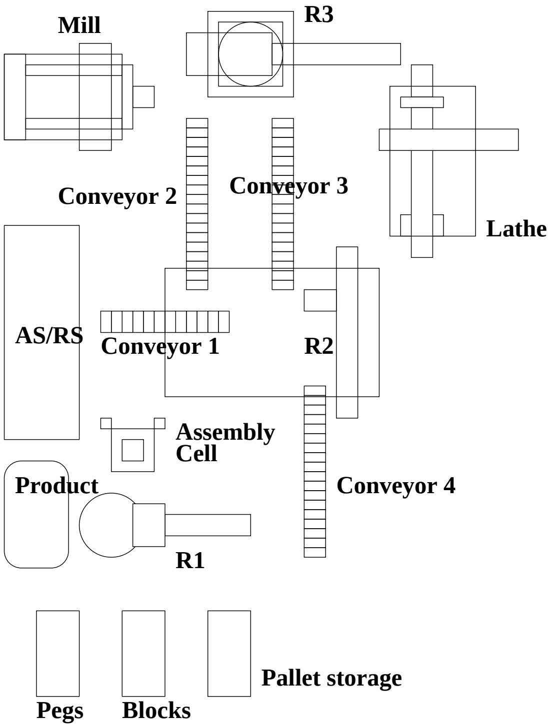


Figure 6.19: A flexible manufacturing system

permissive controller.

This laboratory system consists of six workstations: a lathe, mill, conveyor system, automatic storage and retrieval system, and two robotic workstations. The system, shown in Fig 6.19, starts with raw stock for two different part-types; it machines one into a cylindrical peg and the other into a block with a hole in it. The finished product is the peg and block mated together. No human operators are required.

The lathe is a high-precision, fully automatic computerized numerical control lathe equipped with an automatic pneumatic chuck. The lathe is used to convert the raw stock into the properly dimensioned peg.

The mill is a three-axis vertical spindle machine. It is equipped with a heavy-duty precision spindle and a pneumatic workpiece fixturing device. The mill takes raw stock and machines it into a cube and removes the proper amount of material to allow insertion of the peg.

The conveyor system, or automated material transfer system, consists of four bidirectional conveyors and a three-axis Cartesian robot (R2). This is a shared resource that transports pallets that carry raw stock or finished products throughout FMS. Conveyor 1 moves pallets to and from the automatic storage and retrieval system (AS/RS). Conveyor 2 moves pallets between the mill and the gantry robot. Conveyor 3 moves pallets between the lathe and the gantry robot. Conveyor 4 carries pallets from the gantry robot to the assembly and finished product storage area. Note that the gantry robot can transfer pallets between the ends of any of the four conveyors.

The AS/RS is a buffer (5×4) where pallets carrying raw materials and intermediate parts can be stored.

The five-axis Microbot robot (R3) is a shared resource that is used to load and unload the materials between the lathe and Conveyor 3, and between the milling machine and Conveyor 2.

The five-axis Scorbot robot (R1) is a shared resource located near the raw stock storage bins and assembly area. It has several functions: it transfers pallets between the pallet storage rack and Conveyor 4; retrieves raw stock from storage and places them into pallets; picks finished blocks and pegs from pallets (on Conveyor 4) and places them at the assembly station; performs the assembly operations; and also transfers the finished product to the storage carousel.

The material flow in this system can take several possible routes. In any case, there are four distinct operations that can take place: storage, machining, material transfer, and assembly.

As usual[8] in the study of manufacturing systems, all events in this system are considered controllable. Therefore, there is no reason to distinguish the automata models for plant and those for the specification.

We start by analyzing the specific sequence of operations that are required to manufacture one finished product.

First pallets and (raw) pegs and blocks must be available. The pallets have been designed to carry either one part of raw stock or one finished part.

Operations begin when R1 picks a pallet from the pallet storage and places it on

[8]Please see the footnote on page 89.

Conveyor 5. At this point there are two choices: R1 picks either a block or a peg from storage.

Similarly, R3 has several options. It can move the raw material(s) to the Milling machine and lathe for loading, processing and unloading. It can also move pallets (with material) to Conveyor 1 and AS/RS.

After the raw material is processed, R3 can transfer it to the next step of manufacturing or to Conveyor 1 and AS/RS. If it goes to the assembly station, finished material is removed from the pallet by R1, and the pallet is returned to the pallet storage rack by R1.

In the assembly stage, a finished block should be in the station first, then a peg is inserted into the block to complete the finished product. This can be represented easily by an automaton.

Since the AS/RS can store finished parts and raw parts waiting to be processed, there are numerous choices that will have to be captured by the modeling.

We assume the unlimited availability of raw parts, as is common in modeling manufacturing systems.

The events in the system are defined as in Table 6.25:

Now we are ready to model the system part by part as follows.

1. R1. R1 has 7 functions and at any time can perform no more than one of them. This can be easily modeled as shown in Fig 6.20.

2. Pallet storage. The number of pallets is limited so a buffer-like model is used to prevent the overflow and underflow of used pallets, as shown in Fig 6.21.

3. Conveyor 4. There are 6 different types of stocks carried by Conveyor 4 and each type should be protected from overflow and underflow. This results in 6 buffer-like models, shown in Fig 6.22.

4. R2. R2 moves 4 different types of stocks among Conveyor 1, Conveyor 2, Conveyor 3 and Conveyor 4. Note that after picking up any one stock, it can send it to the appropriate next stage of manufacturing (e.g. picking up a raw block

Event label	Controllable	Definition
R1_g_pa	Y	R1 gets a pallet from its storage
R1_p_pa_c4	Y	R1 puts a pallet on Conveyor 4
R1_g_bl	Y	R1 gets a raw block from outside
R1_p_bl_c4	Y	R1 puts a raw block in a pallet on Conveyor 4
R1_g_pe	Y	R1 gets a raw peg from outside
R1_p_pe_c4	Y	R1 puts a raw peg in a pallet on Conveyor 4
R1_g_paf_c4	Y	R1 gets a used pallet from Conveyor 4
R1_p_paf	Y	R1 puts a pallet back to its storage
R1_g_blf_c4	Y	R1 gets a finished block from Conveyor 4
R1_p_blf_asm	Y	R1 puts a finished block to the Assembly Cell
R1_g_pef_c4	Y	R1 gets a finished peg from Conveyor 4
R1_p_pef_asm	Y	R1 puts a finished peg to the Assembly Cell
R2_g_bl_c4	Y	R2 gets a raw block from Conveyor 4
R2_p_bl_c1	Y	R2 puts a raw block in a pallet on Conveyor 1 (AS/RS)
R2_p_bl_c2	Y	R2 puts a raw block to Conveyor 2
R2_g_bl_c1	Y	R2 gets a raw block in a pallet on Conveyor 1 (AS/RS)
R2_g_pe_c4	Y	R2 gets a raw peg from Conveyor 4
R2_p_pe_c1	Y	R2 puts a raw peg in a pallet on Conveyor 1 (AS/RS)
R2_p_pe_c3	Y	R2 puts a raw peg to Conveyor 3
R2_g_pe_c1	Y	R2 gets a raw peg in a pallet on Conveyor 1 (AS/RS)
R2_g_blf_c2	Y	R2 gets a finished block from Conveyor 2
R2_p_blf_c1	Y	R2 puts a finished block in a pallet on Conveyor 1 (AS/RS)
R2_p_blf_c4	Y	R2 puts a finished block to Conveyor 4
R2_g_blf_c1	Y	R2 gets a finished block in a pallet on Conveyor 1 (AS/RS)
R2_g_pe_c3	Y	R2 gets a finished peg from Conveyor 3
R2_p_pe_c1	Y	R2 puts a finished peg in a pallet on Conveyor 1 (AS/RS)
R2_p_pe_c4	Y	R2 puts a finished peg to Conveyor 4
R2_g_pe_c1	Y	R2 gets a finished peg in a pallet on Conveyor 1 (AS/RS)
R3_g_bl_c2	Y	R1 gets a raw block from Conveyor 2
R1_p_bl_L	Y	R1 puts a raw block on the Lathe
R1_g_pe_c3	Y	R1 gets a raw peg from Conveyor 3
R1_p_pe_c4	Y	R1 puts a raw peg in a the Milling machine
R1_g_blf_L	Y	R1 gets a finished block from the Lathe
R1_p_blf_c2	Y	R1 puts a finished block to Conveyor 2
R1_g_pef_M	Y	R1 gets a finished peg from the Milling Machine
R1_p_pef_c3	Y	R1 puts a finished peg to Conveyor 3
L_f_bl	Y	the Lathe finishes one block
M_f_pe	Y	the Milling machine finishes one peg

Table 6.25: Event definition in FMS

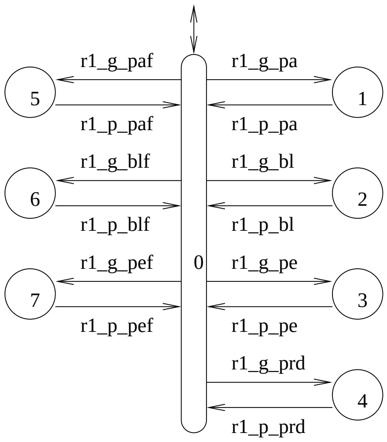


Figure 6.20: Automaton model of R1

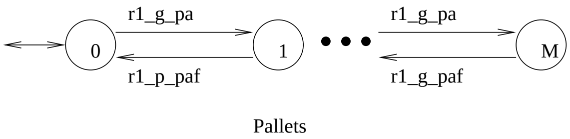


Figure 6.21: Automaton model of pallets

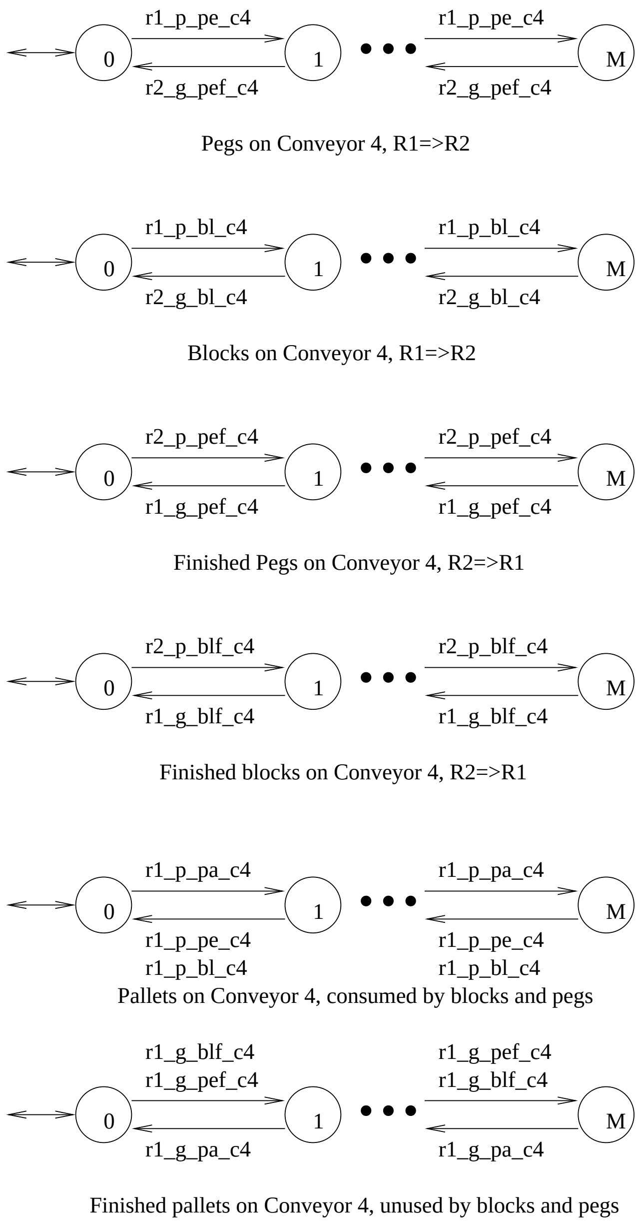


Figure 6.22: Automaton model of Conveyor 4

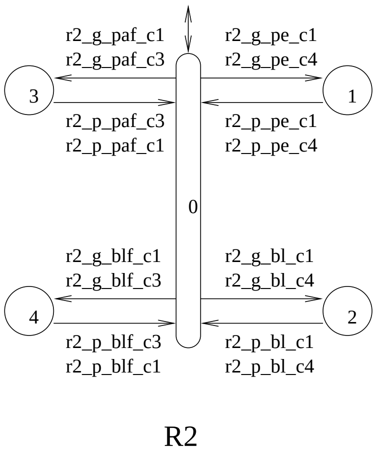


Figure 6.23: Automaton model of R2

with pallet from Conveyor 4 and sending it to Conveyor 2), or sending it to Conveyor 1 to be put in the AS/RS. This gives us 4 "two choice" buffers, as shown in Fig 6.23.

5. Conveyor 1 and AS/RS. Conveyor 1 and AS/RS together can be easily modeled as 4 buffers, as shown in Fig 6.24. Although the AS/RS has fixed capacity, the capacity of Conveyor 1 can be variable, thus we still model them using variable size.

6. Conveyor 2. There are two different types of stocks carried by Conveyor 2, thus it can be modeled using 2 buffers, as shown in 6.25.

7. Conveyor 3. Similar to Conveyor 2, Conveyor 3 is also modeled by two buffers, as shown in 6.25.

8. R3. Similar to R1, R3 is shared among several resources and is modeled as shown in Fig 6.27.

9. The Lathe and Milling machine. Each of these two machines performs only one

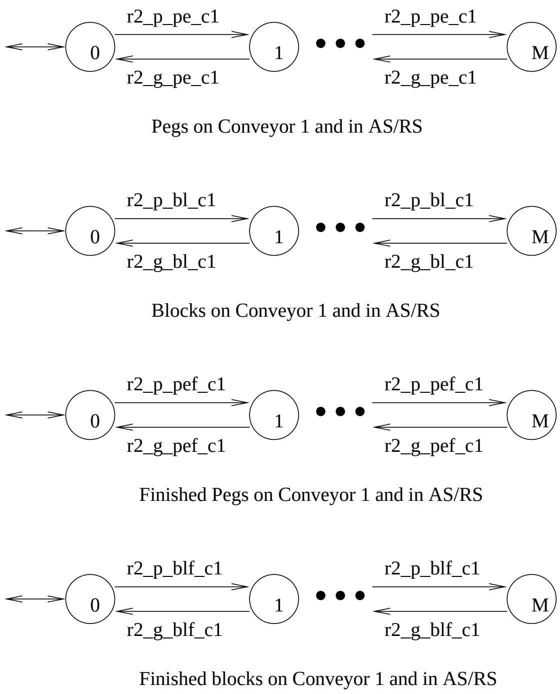


Figure 6.24: Automaton model of Conveyor 1 and AS/RS

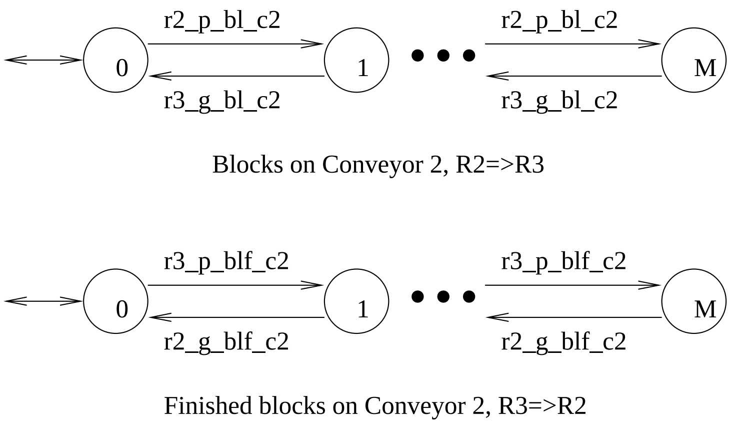


Figure 6.25: Automaton model of Conveyor 2

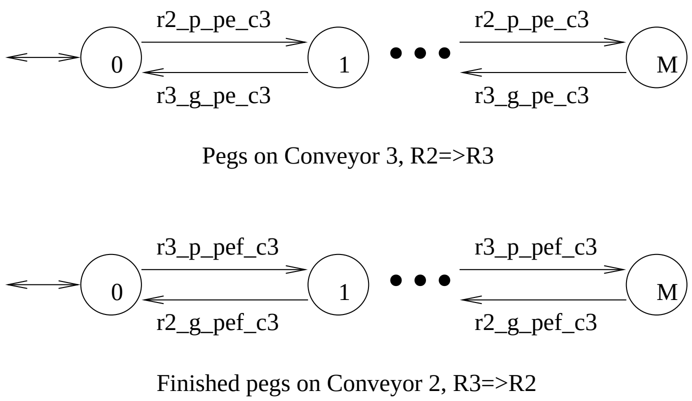


Figure 6.26: Automaton model of Conveyor 3

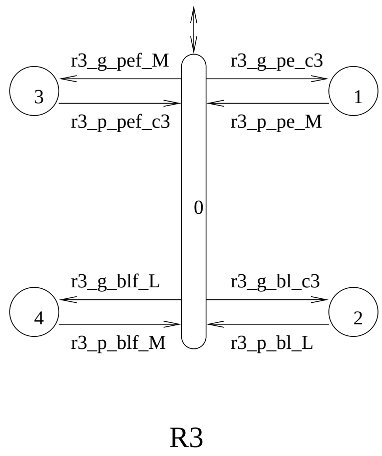


Figure 6.27: Automaton model of R3

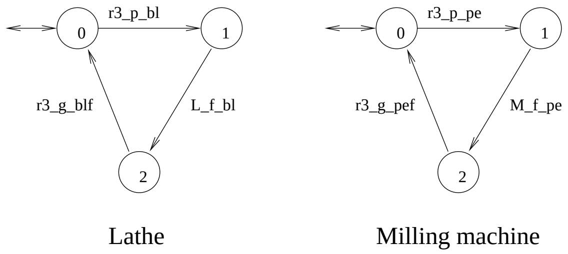


Figure 6.28: Automaton model of the Lathe and Milling machine

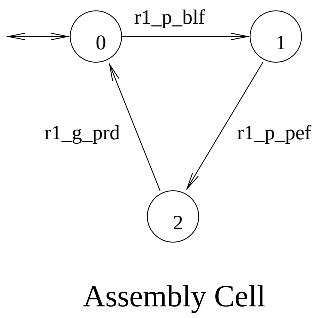


Figure 6.29: Automaton model of Assembly Cell

action – finishing its manufacturing task. Thus they are modeled as shown in Fig 6.28.

10. Assembly Cell. When assembling the finished blocks and pegs, first one block needs to be placed on the Assembly Cell and then one peg is placed to form the final product, then the product can be removed again by R1. This precedence is modeled as shown in Fig 6.29.

We have an automaton model of the system consisting of 21 components. Because all events are controllable, there is no need to worry about which components belong to the plant or the specification. All the limited resources are protected by buffers and all the shared resources are protected by their automaton models accordingly. The

Ma	Mb	Mp	$\|Q\|$	$\|nodes\|$	STCT Time	STCT Memory
2	5	6	2.48×10^6	360	26sec	4M
5	5	10	2.87×10^8	541	62sec	6M
7	5	20	4.21×10^{11}	1346	272sec	8M
14	10	40	1.96×10^{15}	2853	1425sec	18M

Table 6.26: Performance of STCT vs. M on FMS

only additional requirement is nonblocking operation of system. This can be done by STCT without too much difficulty. Because of the complexity of the system, CTCT fails to produce any result. The performance of STCT vs. various sizes of the buffers in this FMS is shown in 6.26. In the table, Ma is the sum of the capacity of AS/RS and Conveyor 1 for each component. All the other buffers have size Mb. Mp is the total number of available pallets. Those non-buffer-like components are unchanged all the time.

Chapter 7

Conclusions and future research

7.1 Conclusions

This thesis presents a new synthesis algorithm, STCT, for the RW approach to the control of DES systems. It is easily seen that for complex DES systems, STCT offers far better performance than the CTCT implementation. On the other hand, STCT is much more systematic, straightforward and general than Petri Net approaches.

The STCT algorithm again demonstrates the importance of structural information in a DES system. Given the nature of most practical systems, which are highly structured, the computational complexity can be dramatically reduced while the advantages of the algorithm, such as straightforwardness and generality, are untouched.

Also, it is shown that the RW approach, which is based on automata models, is able to deal with practical problems, even if the system has a huge state space. Besides, it is shown that parameters other than the size of state space, should be used to evaluate the complexity of a system. Two of those parameters, the degree of coupling, and the number of nodes of the IDD representing the system's state space, are discussed in the thesis. It is shown that the number of nodes, which practically decides the computational complexity of synthesis, can be polynomial in the number of components in the system. This suggests that future research in this direction may be quite promising.

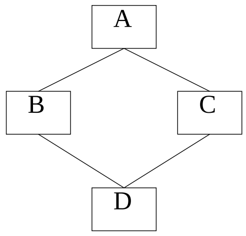


Figure 7.1: A difficult topology for ordering

7.2 Where do we go from here?

It is still possible to optimize the STCT algorithm further by explicitly synchronizing closely coupled components before the generation of structural predicates. But this will not provide a substantial boost to the performance of the algorithm.

Actually, STCT is only the first step toward this direction of the control of DES systems. There are still lots of interesting things which can be done. Obvious ones among them include the detailed analysis of the complexity and the possibility of applying a similar approach to hierarchical controllers. Two interesting possibilities are discussed in the following sections.

7.2.1 The natural utilization of the topology

The STCT algorithm utilizes the topology of the system by ordering the components in the IDD. However, this is a very "rough" utilization. The components have to be arranged in a 1-dimensional list to be presented in a IDD. While this is efficient compared with the "naive" approach of CTCT, there is still much room for improvements. Consider a system with the topology shown in Fig 7.1. There is no "best" ordering at all. The orders "A,B,C,D" and "A,C,B,D" seem to be equally "good" (or bad). This is a big problem when the system becomes larger and the degree of coupling becomes higher. In such cases, every possible ordering is actually far from "optimal".

If we can find a more powerful data structure to represent the structural predicates,

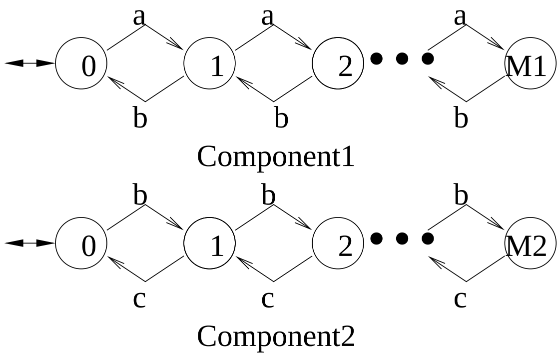


Figure 7.2: *Square* Model

therefore to reduce the redundancy further, we can then push the performance of the synthesis algorithm to a whole new level.

7.2.2 Specific treatment for buffer-like components and explicit structural analysis

In STCT, both buffer-like and non-buffer-like components are treated mostly in the same way. Actually, these two kinds of components should and probably can be processed differently. In the STCT algorithm, the "inner" structure of buffer-like components is mostly ignored. However, buffer-like components are extremely common in manufacturing systems. This makes the further utilization of the "inner" structure almost mandatory.

Strictly speaking, STCT is a modified RW approach, which incorporates structural analysis, which is more Petri Net like. But this structural analysis is done in an implicit way, i.e., it is included "automatically" in the IDDs. This yields a simpler algorithm, compared with an "explicit" structural analysis. However, an explicit structural analysis ought to have better performance. Here "explicit" does not mean the approach that is used by researchers in Petri Net field, instead, it is a geometric view of the reachable / coreachable state space.

A "linear" (totally buffer-like) system is shown in Fig 7.2. If we draw a graph of the entire theoretical state of space of it, and paint all the reachable states black and non-reachable states white, we will get a picture as shown in Fig 7.3. Similarly, for

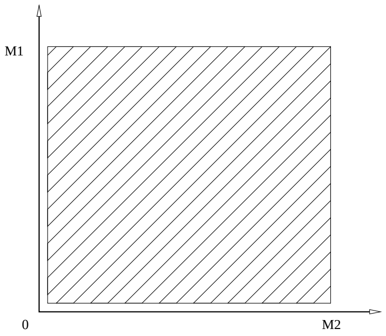


Figure 7.3: Picture of reachable state space of the *Square* Model

the system depicted in Fig 7.4, we can draw a picture of (projected) reachable state space as in Fig 7.5.

For more complicated systems (with much more than 2 components), we will get "pictures" with higher dimensions. It is the author's intuition that we can study those "shapes" to gain insight into the nonblocking property of a controlled system. There is an interesting comparison which can be made among CTCT, STCT, and the Petri Net approaches in the sense of this kind of picture. CTCT draws the picture, but never thinks there is anything other than a set of individual points on it. STCT draws the picture too, with more geometric knowledge – STCT knows about hyperplanes implicitly. Petri Net approaches are simply trying to explain a property of this picture (all points painted in the reachability picture are also painted in the coreachability picture), but without even trying to draw it out!! Therefore, a better approach might be achieved, if we could effectively exploit the geometry (as certainly we could).

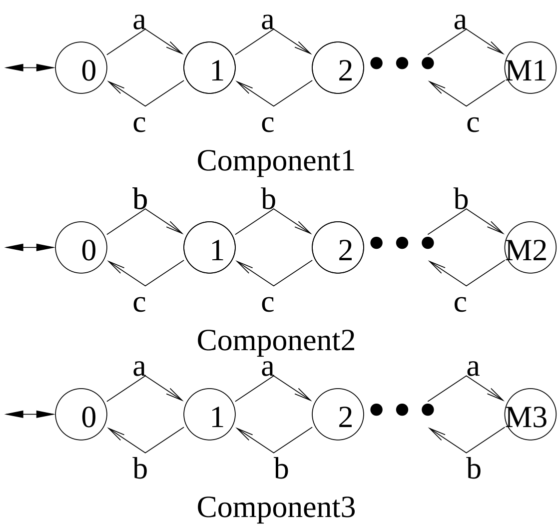


Figure 7.4: *Triangle* Model

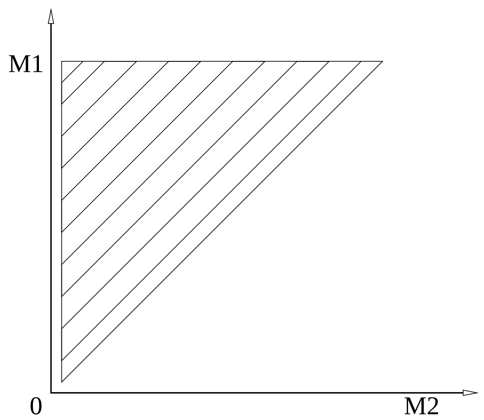


Figure 7.5: Picture of reachable state space of the *Triangle* Model (projected to component 1 vs. component 2 plane)

7.2.3 IDD based controller reduction

It was shown in Chapter 5 that all the control actions can be represented by IDDs and this representation is computationally efficient. However, this representation is not conceptually efficient. In other words, although the IDD representation of controllers is good for computers, it is hard for humans to understand. Normally we can not get any intuition about the system by looking at the IDDs generated by STCT. [1] Thus it is highly desirable to find a way to make the result of STCT more comprehensible to humans.

Which form is the simplest one for human beings is a hard question. But definitely a linear inequality will be good enough for the control of DES. Unfortunately the linear inequality is a little bit too far from the IDDs and one intermediate step is necessary. In STCT everything is represented by structural predicates. And among all predicates there is one kind that is quite easy to be understood. Given the definition domain $Z_1 \times \ldots \times Z_N$, we say a predicate P is 'reduced' if for each $i \in 1 \ldots N$, either

$$(\forall z = (z_1, \ldots z_n))(z_i \neq C_i \Rightarrow z \not\models P)$$

or

$$(\exists z' = (z'_1, \ldots z'_i, \ldots z'_n) \models P) \Rightarrow (\forall z_i)((z'_1, \ldots z_i, \ldots z'_n) \models P)$$

where C_i is a constant. And we can denote a reduced predicate by a vector consisting of C_is and '*', by replacing those variables satisfying the first condition by C_i and those satisfying the second one by '*'. In other words, either P holds for one particular value of a given variable, or P does not care about the value of the given variable at all. For a reduced predicate P, we define the number of variables satisfying the former condition by RN_P. Obviously, the smaller RN_P is, the simpler P is.

For example, given the definition domain $\{1,2,3\} \times \{1,2\} \times \{1,2,3\}$, if a predicate

[1]This may seem to be even worse then the situation of CTCT, in which we 'can' get some information from the control data table. But in actuality, STCT is slightly better than CTCT in the sense of representation transparency. The inference of result of STCT seems to be harder simply because the system itself is much more complicated. Note that CTCT will easily fail on non-trivial systems.

holds exactly on (1,1,1) and (1,2,1), then it is reduced and can be written as (1,*,1). Also we have $RN_P = 2$.

Given a (controllable) event σ, suppose we have the enabling condition of it denoted by E_σ, the disabling condition of it denoted by D_σ, and the controlled reachable state space represented by P_{re}; then we can freely disable anything from D_σ to $(\neg(P_{re} \cap E_\sigma))$, in the sense of set inclusion. Therefore we can (probably) find a predicate P_{dis} which satisfies $D_\sigma \preceq P_{dis}$ and $P_{dis} \preceq \big(\neg(P_{re} \cap E_\sigma)\big)$ and has a 'simpler' form. Given all the above predicates and a state vector $v \models D_\sigma$, a predicate P is said to be 'optimally reduced' w.r.t the above given predicates, if and only if

1. P is reduced.

2. $v \models P$, i.e. P contains the real disabling condition.

3. $(P \cap (P_{re} \cap E_\sigma)) \equiv 0$, i.e. P doesn't disable anything that must be enabled.

4. RN_P is minimum among all P satisfying the above three conditions.

and we denote such a P by P_{od}.

In order to find P_{od}, it is actually necessary to test all the reduced predicates which hold on v, thus giving us the complexity of 2^N, where N is the number of variables in IDDs. This computation is impossible if we have a considerably large N, for instance 40. Thus we have to settle for near optimality. If we test only those reduced predicates which have no overlapping in the supporting definition domain space, the complexity will be greatly reduced to the linear function of N. And in practice, we will still be able to get the optimal reduction in most cases. This can be done simply by scanning the variables from the first one to the last one, trying to replace them all by '*'. If after one replacement, the resulting predicate violates condition 3 in the definition of P_{od}, undo this replacement and proceed to the next variable. Thus after one iteration, we get one reduced predicate which satisfies all the conditions except for the last in the definition of P_{od}. Then we replace all the remaining C_is by '*' and all the '*' by their original values in v, and begin the procedure again. This time we may or may not get a reduced predicate because we start with a vector containing '*'. If we do, compare the new result with the old one, take the better, then replace all the C_is by '*' and those newly generated '*' by their original value and do the next cycle. If we can not find a reduced predicate any more, the procedure stops. The best result we ever get in the procedure is a good guess for the optimally reduced predicate.

A simple example is given below to illustrate this procedure. Suppose we have a system with 6 variables (no. of components), and we find we should disable a certain event σ on the state (1,3,4,2,5,6). Then we create a predicate which is true for this state and for this state only. Next, we try to replace all the numbers in the state vector by '*'. First we test (*,3,4,2,5,6) (this predicate is actually enlarged in the sense of supporting set); if it doesn't include any states on which σ must be enabled, we accept this '*'; otherwise, we have to change the '*' back to 1. For the example's sake, just assume that we accept this '*'. After this, we try to change the second variable, now having a value 3, into a '*' using the same criterion. After this we try to change the third, fourth, fifth ... into '*' until we have tried all the variables. After we have processed all 6 variables, we get a reduced predicate which probably contains some '*' in it. For this example, assume we get (*,3,*,2,*,*,*). This means we can disable any state which satisfies (*,3,*,2,*,*,*). Then we change all the fixed values in this reduced predicate into '*' and each newly converted '*' into its original value (this time all stars are newly converted). We get (1,*,4,*,2,5,6). This time we try to convert all numbers into '*' again, in a one-by-one fashion and with the same criterion. Assume this time we end up with another reduced predicate (1,*,*,*,*,*). Then we change all the fixed numbers into stars and all newly created stars into numbers, getting (*,*,4,*,2,5,6). Now try to convert numbers into stars again. Assume this time we make no progress, i.e. we can not change any number into a star without violating the rule - this predicate can not contain any state on which σ must be enabled. We stop here. We have got two candidates of the optimally reduced predicate, (*,3,*,2,*,*,*) and (1,*,*,*,*,*). Clearly the latter has a smaller RN value, thus we take (1,*,*,*,*,*) as our final result.

Given a D_σ, we can represent it by a disjunction of P_{od}s by repeating the following procedure until $D_\sigma \equiv 0$:

- Find a $v \models D_\sigma$.
- Find P_{od} containing v.
- $D_\sigma = D_\sigma \cap (\neg P_{od})$.

The reduced control actions of Example 6.2 are listed as follows (M=5). Given a state vector, we should disable an event if this state vector satisfies any of the reduced predicates listed below that event.

```
nothing needs to be disabled for event No. 0(13)
nothing needs to be disabled for event No. 1(14)
nothing needs to be disabled for event No. 2(12)
nothing needs to be disabled for event No. 3(2)
Rule for event No. 4(4)found:
--------------------------------------------------
 *  *  *  *  *  *  *  4  *  *  *  5  *  *  *  5  5
------------------------------------------------
nothing left to be disabled for event No. 4(4)
Rule for event No. 5(1)found:
--------------------------------------------------
 *  *  *  5  *  *  *  4  *  5  *  5  *  *  *  5  5
 *  *  *  5  *  *  1  3  *  5  *  5  *  *  *  5  5
 *  *  *  5  *  *  2  2  *  5  *  5  *  *  *  5  5
 *  *  *  5  *  *  3  1  *  5  *  5  *  *  *  5  5
 *  *  *  5  *  *  4  *  *  5  *  *  *  *  *  *  *
 *  *  *  5  1  *  *  3  *  5  *  5  *  *  *  5  5
 *  *  *  5  1  *  1  2  *  5  *  5  *  *  *  5  5
 *  *  *  5  1  *  2  1  *  5  *  5  *  *  *  5  5
 *  *  *  5  1  *  3  *  *  5  *  *  *  *  *  *  *
 *  *  *  5  2  *  *  2  *  5  *  5  *  *  *  5  5
```

```
 *  *  *  5  2  *  1  1  *  5  *  5  *  *  *  5  5
 *  *  *  5  2  *  2  *  *  5  *  *  *  *  *  *  *
 *  *  *  5  3  *  *  1  *  5  *  5  *  *  *  5  5
 *  *  *  5  3  *  1  *  *  5  *  *  *  *  *  *  *
 *  *  *  5  4  *  *  *  *  5  *  *  *  *  *  *  *
 *  *  5  *  *  *  *  4  *  5  *  5  *  *  *  5  5
 *  *  5  *  *  *  1  3  *  5  *  5  *  *  *  5  5
 *  *  5  *  *  *  2  2  *  5  *  5  *  *  *  5  5
 *  *  5  *  *  *  3  1  *  5  *  5  *  *  *  5  5
 *  *  5  *  *  *  4  *  *  5  *  *  *  *  *  *  *
 *  *  5  *  1  *  *  3  *  5  *  5  *  *  *  5  5
 *  *  5  *  1  *  1  2  *  5  *  5  *  *  *  5  5
 *  *  5  *  1  *  2  1  *  5  *  5  *  *  *  5  5
 *  *  5  *  1  *  3  *  *  5  *  *  *  *  *  *  *
 *  *  5  *  2  *  *  2  *  5  *  5  *  *  *  5  5
 *  *  5  *  2  *  1  1  *  5  *  5  *  *  *  5  5
 *  *  5  *  2  *  2  *  *  5  *  *  *  *  *  *  *
 *  *  5  *  3  *  *  1  *  5  *  5  *  *  *  5  5
 *  *  5  *  3  *  1  *  *  5  *  *  *  *  *  *  *
 *  *  5  *  4  *  *  *  *  5  *  *  *  *  *  *  *
-------------------------------------------------
nothing left to be disabled for event No. 5(1)
nothing needs to be disabled for event No. 6(3)
Rule for event No. 7(5)found:
---------------------------------------------------
 *  *  *  *  *  *  *  4  *  *  *  5  *  *  *  5  5
 *  *  *  5  *  *  1  3  *  *  *  5  *  *  *  5  5
 *  *  *  5  *  *  2  2  *  *  *  5  *  *  *  5  5
 *  *  *  5  *  *  3  1  *  *  *  5  *  *  *  5  5
 *  *  *  5  *  *  4  *  *  *  *  5  *  *  *  5  5
```

```
 *  *  5  *  *  *  1  3  *  *  *  5  *  *  *  5  5
 *  *  5  *  *  *  2  2  *  *  *  5  *  *  *  5  5
 *  *  5  *  *  *  3  1  *  *  *  5  *  *  *  5  5
 *  *  5  *  *  *  4  *  *  *  *  5  *  *  *  5  5
-----------------------------------------------
nothing left to be disabled for event No. 7(5)
nothing needs to be disabled for event No. 8(6)
Rule for event No. 9(11)found:
-------------------------------------------------
 *  *  *  5  *  *  4  *  *  *  *  *  *  *  *  *  *
 *  *  *  5  1  *  3  *  *  5  *  *  *  *  *  *  *
 *  *  *  5  2  *  2  *  *  5  *  *  *  *  *  *  *
 *  *  *  5  3  *  1  *  *  5  *  *  *  *  *  *  *
 *  *  *  5  4  *  *  *  *  5  *  *  *  *  *  *  *
 *  *  5  *  *  *  4  *  *  *  *  *  *  *  *  *  *
 *  *  5  *  1  *  3  *  *  5  *  *  *  *  *  *  *
 *  *  5  *  2  *  2  *  *  5  *  *  *  *  *  *  *
 *  *  5  *  3  *  1  *  *  5  *  *  *  *  *  *  *
 *  *  5  *  4  *  *  *  *  5  *  *  *  *  *  *  *
-----------------------------------------------
nothing left to be disabled for event No. 9(11)
nothing needs to be disabled for event No. 10(7)
nothing needs to be disabled for event No. 11(8)
Rule for event No. 12(9)found:
-------------------------------------------------
 *  *  *  *  *  *  *  5  *  *  *  5  *  *  *  5  4
 *  *  *  5  *  *  1  4  *  *  *  5  *  *  *  5  4
 *  *  *  5  *  *  2  3  *  *  *  5  *  *  *  5  4
 *  *  *  5  *  *  3  2  *  *  *  5  *  *  *  5  4
 *  *  *  5  *  *  4  1  *  *  *  5  *  *  *  5  4
```

```
 *  *  *  5  *  5  *  4  *  5  *  5  *  *  *  5  4
 *  *  *  5  1  *  1  3  *  5  *  5  *  *  *  5  4
 *  *  *  5  1  *  2  2  *  5  *  5  *  *  *  5  4
 *  *  *  5  1  *  3  1  *  5  *  5  *  *  *  5  4
 *  *  *  5  2  *  *  3  *  5  *  5  *  *  *  5  4
 *  *  *  5  2  *  1  2  *  5  *  5  *  *  *  5  4
 *  *  *  5  2  *  2  1  *  5  *  5  *  *  *  5  4
 *  *  *  5  3  *  *  2  *  5  *  5  *  *  *  5  4
 *  *  *  5  3  *  1  1  *  5  *  5  *  *  *  5  4
 *  *  *  5  4  *  *  1  *  5  *  5  *  *  *  5  4
 *  *  5  *  *  *  1  4  *  *  *  5  *  *  *  5  4
 *  *  5  *  *  *  2  3  *  *  *  5  *  *  *  5  4
 *  *  5  *  *  *  3  2  *  *  *  5  *  *  *  5  4
 *  *  5  *  *  *  4  1  *  *  *  5  *  *  *  5  4
 *  *  5  *  *  5  *  4  *  5  *  5  *  *  *  5  4
 *  *  5  *  1  *  1  3  *  5  *  5  *  *  *  5  4
 *  *  5  *  1  *  2  2  *  5  *  5  *  *  *  5  4
 *  *  5  *  1  *  3  1  *  5  *  5  *  *  *  5  4
 *  *  5  *  2  *  *  3  *  5  *  5  *  *  *  5  4
 *  *  5  *  2  *  1  2  *  5  *  5  *  *  *  5  4
 *  *  5  *  2  *  2  1  *  5  *  5  *  *  *  5  4
 *  *  5  *  3  *  *  2  *  5  *  5  *  *  *  5  4
 *  *  5  *  3  *  1  1  *  5  *  5  *  *  *  5  4
 *  *  5  *  4  *  *  1  *  5  *  5  *  *  *  5  4
------------------------------------------------
nothing left to be disabled for event No. 12(9)
nothing needs to be disabled for event No. 13(10)
```

Then we can proceed to reduce this to a linear inequality. Currently this can only done by reasoning, which is an easy task for us. But still it will not be too difficult to make a program to do it. It is simply a solution to the linear equation sets. The above rules can be further reduced to (again, we should disable an event at a state if the corresponding state vector satisfies any of the rules for that event.):

- Event No. 0, (13): nothing.
- Event No. 1, (14): nothing.
- Event No. 2, (12): nothing.
- Event No. 3, (2): nothing.
- Event No. 4, (4):
  $(x_8 = 4, x_{12} = 5, x_{16} = 5, x_{17} = 5)$.
- Event No. 5, (1):
  $(x_4 = 5, x_{10} = 5, x_{12} = 5, x_{16} = 5, x_{17} = 5,\ x_5 + x_7 + x_8 = 5)$.
  $(x_3 = 5, x_{10} = 5, x_{12} = 5, x_{16} = 5, x_{17} = 5,\ x_5 + x_7 + x_8 = 5)$.
- Event No. 6, (3): nothing.
- Event No. 7, (5):
  $(x_4 = 5, x_{12} = 5, x_{16} = 5, x_{17} = 5,\ x_7 + x_8 = 4)$.
  $(x_3 = 5, x_{12} = 5, x_{16} = 5, x_{17} = 5,\ x_7 + x_8 = 4)$.
- Event No. 8, (6): nothing.
- Event No. 9, (11):
  $(x_4 = 5, x_{10} = 5, x_5 + x_7 = 4)$.
  $(x_3 = 5, x_{10} = 5, x_5 + x_7 = 4)$.
- Event No. 10,(7): nothing.
- Event No. 11,(8): nothing.
- Event No. 12,(9):
  $(x_4 = 5, x_{10} = 5, x_{12} = 5, x_{16} = 5, x_{17} = 4,\ x_5 + x_7 + x_8 = 5)$.
  $(x_3 = 5, x_{10} = 5, x_{12} = 5, x_{16} = 5, x_{17} = 4,\ x_5 + x_7 + x_8 = 5)$.

$(x_4 = 5, x_6 = 5, x_8 = 4, x_{10} = 5, x_{12} = 5, x_{16} = 5, x_{17} = 4)$.

$(x_3 = 5, x_6 = 5, x_8 = 4, x_{10} = 5, x_{12} = 5, x_{16} = 5, x_{17} = 4)$.

- Event No. 13,(10): nothing.

Bibliography

[Ake78] S.B. Akers. Functional testing with binary decision diagrams. *Eighth Annual Conference on Fault-Tolerant Computing*, pages 75–82, 1978.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 12(8):677–691, August 1986.

[DAJ95] Alan A. Desrochers and Robert Y. Al-Jaar. *Applications of Petri Nets in Manufacturing Systems, Modeling, Control and Performance Analysis.* IEEE, 1995.

[Den89] M.J. Denham. A Petri net approach to discrete event control. In *IEE Colloquium on Modelling, Simulation and Control of Discrete Event Systems*, December 1989.

[GD92] A. Giua and F. DiCesare. On the existence of Petri net supervisors. In *Proc. of 31st IEEE Conf. Decision and Control*, pages 3380–3385, Tucson, AZ, USA, December 1992.

[Gun97] Johan Gunnarsson. *Symbolic Methods and Tools for Discrete Event Dynamic Systems.* Linkoping Studies in Science and Technology, Dissertations, 1997.

[Hua91] N. Huang. Supervisory control of vector discrete-event systems. Master's thesis, Dept. of El. Eng., Univ. of Toronto, Canada, April 1991.

[Kro87] B.H. Krogh. Controlled Petri nets and maximally permissive feedback logic. In *Proc. of 25th Allerton Conf. on Communication, Control and Computing*, volume 1, pages 317–326, Monticello, IL, USA, 1987.

[Li91] Y. Li. *Control of Vector Discrete-Event Systems.* PhD thesis, Dept. of El. Eng., Univ. of Toronto, Canada, July 1991.

[LW89] Y. Li and W.M. Wonham. Composition and modular state-feedback control of vector discrete-event systems. In *Proc. of 23rd Conf. on Information Sciences and Systems*, pages 103–110, Princeton, NJ, USA, March 1989.

[LW93] Y. Li and W.M. Wonham. Control of vector discrete-event systems I – the base model. *IEEE Trans. Autom. Control*, 38(8):1214–1227, August 1993.

[LW94] Y. Li and W.M. Wonham. Control of vector discrete-event systems II – controller synthesis. *IEEE Trans. Autom. Control*, 39(3):512–531, March 1994.

[MA98] John Moody and Panos Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets.* Kluwer Academic Publishers, 1998.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, January 1987.

[SS00] Michael Seidl and Günther Schmidt. Avoiding deadlocks in flexible manufacturing systems. In *Discrete event systems: analysis and control*, pages 149–158. Kluwer Academic Publishers, 2000.

[UM88] T. Ushio and R. Matsumoto. State feedback and modular control synthesis in controlled Petri nets. In *Proc. of 27th IEEE Conf. Decision and Control*, pages 1502–1507, Austin, TX, USA, December 1988.

[Won99] W. M. Wonham. *Notes on Control of Discrete-Event Systems.* Department of Electrical and Computer Engineering, University of Toronto, December 1999.

[ZD93] M.C. Zhou and F. DiCesare. *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems.* Kluwer Academic Publishers, 1993.

[ZV99] M.C. Zhou and K. Venkatesh. *Modeling, Simulation, and Control of Flexible Manufacturing Systems : a Petri Net Approach.* World Scientific, 1999.

Appendix A

Definition of correlation

For a given IDD, which represents a predicate $P : X_1, \times \ldots, \times X_N \rightarrow \{0, 1\}$, we can define the correlation between variable x_i and x_j, denoted $COR_{i,j}(P)$, as follows.

First consider the situation that $N = 2$. In this case we have only two correlations to be defined, $COR_{1,2}(P)$ and $COR_{2,1}(P)$. Let's consider $COR_{1,2}(P)$, and then $COR_{2,1}(P)$ can defined similarly. P is a function of x_1 and x_2. Given a fixed value of $x_1 = a$, we get $Q_a(x_2) = P(a, x_2)$, a function of x_2 only. Suppose the number of elements in the set of functions

$$\{Q_a(x_2)\ | a \in X_1, Q_a(x_2) \not\equiv false\}$$

is k. Then the correlation $COR_{1,2}(P)$ is defined to be k. Similarly, we can define $COR_{2,1}(P)$. Several examples are shown in Fig 1.1 to illustrate this definition.

In Fig 1.1 (a), we can see that for all the possible values of x_1, there is only one kind of $Q(x_2) \equiv (x_2 \in \{0, 1, 2, 3, 4\} \Leftrightarrow Q(x_2) = 1)$ which is not always false, thus $COV_{1,2}(P) = 1$. Similarly $COV_{2,1}(P) = 1$ too. Also it should be noted that 1 is the smallest value which is possible. For (b) and (c), the correlations are all 1 too. For

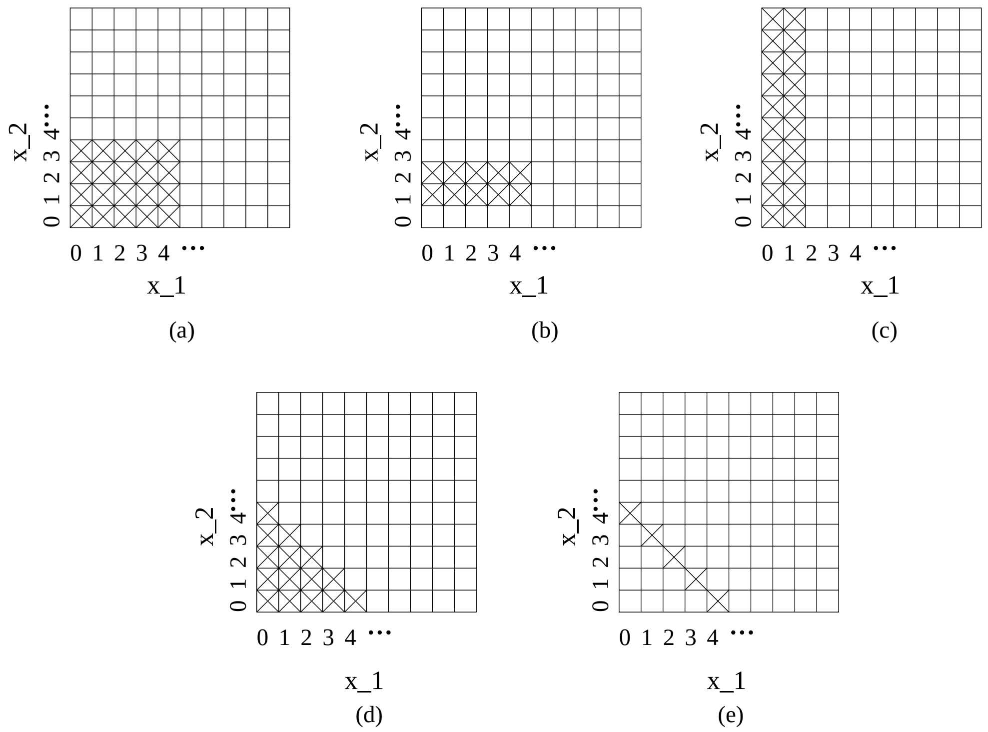


Figure 1.1: Simple examples to define correlation

(d), given x_1, there are 5 different Q_as:

$$
\begin{array}{lcllcl}
Q_4 & \equiv & (x_2 = 0) & \Leftrightarrow & Q_4(x_2) = 1 \\
Q_3 & \equiv & (x_2 \in \{0, 1\}) & \Leftrightarrow & Q_3(x_2) = 1 \\
Q_2 & \equiv & (x_2 \in \{0, 1, 2\}) & \Leftrightarrow & Q_2(x_2) = 1 \\
Q_1 & \equiv & (x_2 \in \{0, 1, 2, 3\}) & \Leftrightarrow & Q_1(x_2) = 1 \\
Q_0 & \equiv & (x_2 \in \{0, 1, 2, 3, 4\}) & \Leftrightarrow & Q_0(x_2) = 1
\end{array}
$$

thus $COV_{1,2}(P) = 5$. Similarly, $COV_{2,1}(P)$=5. For (e), we have the same level of correlation. Note here that predicates like those in (a), (b), and (c) are "good" ones, i.e. the IDD representing them are simple. On the contrary, (d) and (e) are "bad" examples.

For $N > 2$ we can define the correlation similarly. Suppose we want to find the correlation between x_i and x_j, where $i, j \in \{1, \ldots, N\}$ and $i \neq j$. Let v be the vector of all the variables other than x_i and x_j. Given a fixed v, P can be considered a function of x_i and x_j, namely $P'_v : X_i \times X_j \to \{0, 1\}$. Then we can find the correlation $COR_{i,j}(P'_v)$ for each v that satisfies $P'_v \not\equiv 0$. Then the correlation $COR_{i,j}(P)$ can be defined as the average of P'_v over all the values of v such that $P'_v \not\equiv 0$.

Also, if the two modules corresponding to x_i and x_j don't have any shared events, normally we will get a reachable or coreachable predicate like those in Fig 1.1 (a), (b), (c). If they are coupled together, we will possibly get a reachable or coreachable predicate like (d) and (e) in Fig 1.1. Therefore, we say two components (or variables) are *correlated* if they have a correlation larger than 1. This is the basic idea behind the optimization of ordering of variables.

Appendix B

Bufferization

Multiple instances of the same entity are quite common in practice, especially in manufacturing systems. A good example is several identical machines working together. The most primitive example is the simple buffer.

The model of a 1-slot buffer can be easily extended to represent a multi-slot one. However, for a general automaton model of a non-buffer unit, it is not so straightforward to expand it to model multiple instances. The first thought might be to model each instance by an automaton with the same structure with all events renamed so that each instance has a unique set of events. This approach does give us one correct solution, but it is a rather inefficient one. Given that each machine has K states and L events, this model of N instances will have K^N states and $L \times N$ events.

In fact, this complexity is not necessary because we don't need all the information which has been modeled by such an approach. When we are modeling multiple instances, we usually do *not* care which state a particular instance is in, we *only* care *how many* instances are in a given state.

Thus we can use another approach – bufferization. Suppose we are given an automaton which has K states and satisfies the following two conditions:

- All the events are uniquely labeled. While this may seem restrictive, it is often the case. If not, we can always relabel all the events to make them uniquely labeled.

- There is only one marked state. Again, restrictive as it looks, it is almost always appropriate for modeling manufacturing systems.

Then we can always represent it by K automata, each of which has only 2 states.

This can be done as follows. Suppose we have $\mathbf{G} = (Y, \Sigma, y_0, \eta, Y_m)$ First, rename events in $\mathbf{G}$ such that all events are uniquely labeled. For each given state y other than y_0 in the original model, create a two state automaton $\mathbf{G}'_y$ with states labeled by 0 and 1 respectively. For each event $\sigma \in \Sigma$, if $\eta(y, \sigma)!$, add the transition $(1, \sigma, 0)$ to $\mathbf{G}'_y$; if $(\exists y' \in Y)\eta(y', \sigma) = y$, add the transition $(0, \sigma, 1)$; otherwise add $(0, \sigma, 0)$ and $(1, \sigma, 1)$.

For state y_0, a $\mathbf{G}'_{y_0}$ is created similarly but the two kinds of transitions (other than selfloops) are added in a swapped way, i.e. For each event $\sigma \in \Sigma$, if $\eta(y_0, \sigma)!$ add the transition $(0, \sigma, 1)$ to $\mathbf{G}'_{y_0}$; if $(\exists y' \in Y)\eta(y', \sigma) = y_0$, add the transition $(1, \sigma, 0)$; otherwise add $(0, \sigma, 0)$ and $(1, \sigma, 1)$. All the $\mathbf{G}'$s have state 0 as initial state. If in $\mathbf{G}$, state $y \neq y_0$ is marked, then $\mathbf{G}'_y$ has its state 1 marked. All other $\mathbf{G}'$s have their state 0 marked. If in $\mathbf{G}$, y_0 is marked, then all $\mathbf{G}'$s have state 0 marked. The synchronous product of the K created automata will be identical to G.

The reason is that for any given time, if $\mathbf{G}$ is in state y_0, then all $\mathbf{G}'_y$s must be in state 0. If $\mathbf{G}$ is in a state $y \leq y_0$, then $\mathbf{G}'_y$ will be in state 1 and all the other modules of $\mathbf{G}'$ in state 0. It can be very easily verified in any given state y, the events are enabled and disabled in $\mathbf{G}$ and $\mathbf{G}'$ identically.

A single automaton with K states now has been converted to K 1-slot buffers. We can model multiple instances by simply increasing the sizes of all the "buffers". When bufferized, the current state of each $\mathbf{G}'_y$ actually shows how many units are in states y (Again, state y_0 is special: the current state of $\mathbf{G}_{y_0}$ is how many units are *not* in initial state). Hence the correctness of this approach is rather obvious.

A bufferization of the model of a simple machine is shown in Fig 2.1. Part 1(a) shows the model of the machine. Part 1(b) is the bufferized model, also with only one instance. Part 2(a) is the model of two instances of the machine without bufferization.

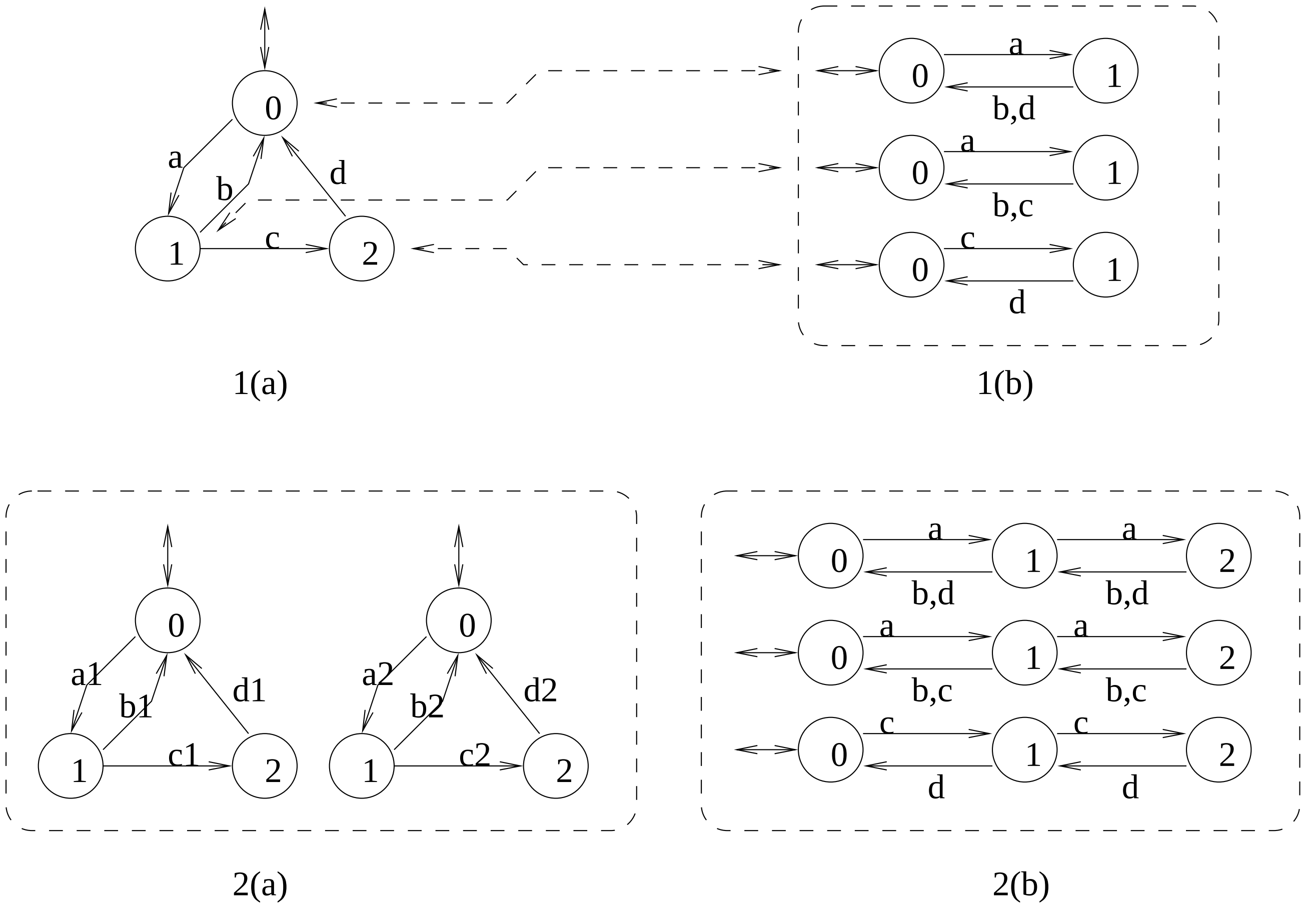


Figure 2.1: Bufferization of a simple model

Part 2(b) is the bufferized model of the two instances case. For this simple machine, if there are N instances, the un-bufferized model will have 3^N states while the bufferized one has only $\frac{(N+2)\times(N+1)}{2}$ states.