State Based Control of Timed Discrete Event Systems using Binary Decision Diagrams

by

Ali Saadatpoor

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

Abstract

State Based Control of Timed Discrete Event Systems using Binary Decision Diagrams

Ali Saadatpoor

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2004

This thesis discusses a new synthesis approach to the supervisory control of Timed Discrete Event Systems (TDES).

The new approach is much more efficient than the existing synthesis approaches. Using this method, many practical systems can be synthesized using a personal computer. Besides, it is shown that the number of nodes in the Binary Decision Diagram (BDD) representing a TDES can be a better measurement of the complexity of the TDES than the number of states and transitions.

The structural information of the timers in a given TDES together with the reduction properties of BDDs are exploited to help this new method achieve more efficient performance. The algorithm is based on the fact that each flat structure can be divided into smaller structures.

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

Acknowledgements

I would like to sincerely thank my supervisor Professor W. M. Wonham, for providing useful insight in the area of Discrete Event Systems. I have been truly honoured by his generosity, support and guidance throughout the course of my studies, and I feel priviledged to have been his student.
I am very grateful for the support of all my friends in Toronto and specially my colleagues in Systems Control Group. I would like to acknowledge Maziar Khatir, who was highly supportive and Farid Fadaie for his help and patience. Special thanks to Ma Chuan for his complementary guidance throughout the development of this work.
Financial support, in the form of University of Toronto Fellowship and Research Assistantship from Professor Wonham, is gratefully appreciated.
At last I devote this work to my Father and Mother in Iran for their love and support.

Contents

List of Figures

Chapter 1

Introduction

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

1.1 Discrete-Event Systems

Discrete event systems (DES) 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 non-deterministic. The underlying primitive concepts include events, conditions and signals [29].

In the last two decades, DES 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 the system will remain there. Timed models are often used for performance analysis.

1.2 Timed Discrete Event Systems

Timing introduces a new dimension of discrete-event system modelling and control, of considerable applied interest but also of significant complexity. Many approaches are possible. The first that we know to have been proposed in the setting of [26] is that of Moller [21], who assigns time delays to events but does not explore their implication for control. In the framework of [26] , Li and Wonham [15] investigate the effects of temporal delays. Brave and Heyman [6] introduce time intervals for the possible occurrence of enabled events, relative to the time instant at which the current state is entered, and demonstrate how temporal features and logic-based features of behavior might be separated for independent treatment. An alternative perspective on timed models for controlled DES may be found in [10], which treats hierarchical layering induced by empirical separation of time scales.

Brandin and Wonham [5] adjoins to the structure of [26] the timing features of timed transition models (TTM)[23][22][24]. The BW framework, which we use in this thesis, retains the concept of maximally permissive supervision introduced in [26] , allows the timed modelling of DES, admits subsystem composition, admits forcing and disablement as means of control and allows modular supervision, possibly under the constraint of partial observation.

1.3 Objective of the Thesis

In Ramadge-Wonham theory, an automaton (in practice, finite) is used to model both the plant to be controlled and the specification. The RW approach successfully showed the existence and theoretical synthesis procedure of the nonblocking supervisory controller. Different kinds of synthesis methods are developed and implemented as software CTCT [1] for untimed models and TTCT for timed models to compute optimal controllers such that the controlled system not only satisfies the specifications but is also as permissive as possible. However, this implementation of RW theory can only solve problems of small state size. This is because an exhaustive list is used to represent the whole model of a system in TCT. Given that many practical systems have a great number of states, TCT has a very limited use in the synthesis of practical control problems. The inefficiency of this approach is directly related to the assumption of ignorance of all the structural information in a real world system. TCT uses the easiest way to achieve nonblocking,i.e. exhaustive search of entire reachable state space. This has been considered infeasible in its computational aspect.

In [30], the exhaustive search is optimized by using the structural properties for untimed models. This optimization was implemented by Integer Decision Diagrams using a software called STCT. For complex DES systems, STCT offers far better performance than CTCT.

The goal of this thesis is to extend the method in [30] to Timed-DES. We want to show that the structural information present in the structure of a TDES can be used to op-

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

timize the exhaustive search through the entire state space of the TDES, which grows much faster than untimed models. In order to achieve this goal, we use state-based control of Timed-DES instead of language-based control using predicates. Then we provide an algorithm for structurally searching the reachable state space. After that we use Binary Decision Diagrams in order to represent the predicates in *compressed* form. At the end of the thesis, we will show the efficiency of this new method in comparison with the previous implementation TTCT.

Chapter 2

Supervisory Control of Timed Discrete Event Systems

Timed discrete event systems (TDES) are introduced in order to deal with not only logical specifications but also temporal specifications. A variety of frameworks for the representation of TDES have been presented in computer science and engineering disciplines. However, most of them are developed for the verification of the system, such as Timed Petri Nets and Timed Transition Models. For synthesis purposes, a timed automaton [25][4][5] is introduced.

2.1 Timed Discrete Event Systems

In this section, the Brandin and Wonham TDES model will be reviewed. First, we consider a finite automaton $\mathbf{G_{act}} = (A, \Sigma_{act}, \delta_{act}, a_0, A_m)$, called an *activity transition graph* (ATG) to describe the untimed behavior of the system. In $\mathbf{G_{act}}$, A is the finite set of *activities*, Σ_{act} is the finite set of *events*, a partial function $\delta_{act} : A \times \Sigma_{act} \longrightarrow A$ is the activity *transition* function, $a_0 \in A$ is the *initial* activity, and $A_m \subseteq A$ is the set of *marked* activities.

In order to construct a TDES model, timing information is introduced into $\mathbf{G_{act}}$. Let

$\mathbb{N}$ denote the set of all nonnegative integers. In Σ_{act}, each event σ will be equipped with a *lower time bound* $l_\sigma \in \mathbb{N}$ and an *upper time bound* $u_\sigma \in \mathbb{N} \cup \{\infty\}$ such that $l_\sigma \leq u_\sigma$. Then the set of events is decomposed into two subsets $\Sigma_{spe} = \{\sigma \in \Sigma_{act} | u_\sigma \in \mathbb{N}\}$ and $\Sigma_{rem} = \{\sigma \in \Sigma_{act} | u_\sigma = \infty\}$. Σ_{spe} (respectively Σ_{rem}) is the set of *prospective* (respectively, *remote*) events whose upper time bounds are finite (respectively, infinite). The lower time bound would typically represent a delay, in communication or in control enforcement; while an upper time bound is a hard deadline, imposed by legal specification or physical necessity.

For each $\sigma \in \Sigma_{act}$, the timer interval T_σ is defined as

$$T_\sigma = \begin{cases} [0, u_\sigma] & \text{if } \sigma \in \Sigma_{spe} \\ [0, l_\sigma] & \text{if } \sigma \in \Sigma_{rem} \end{cases}$$

The TDES defined by Brandin and Wonham [5] is a finite automaton

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

which can be displayed by *timed transition graph (TTG)*. The state set Q is defined as

$$Q = A \times \prod \{T_\sigma | \sigma \in \Sigma_{act}\}$$

A state $q \in Q$ is of the form $q = (a, \{t_\sigma | \sigma \in \Sigma_{act}\})$, where $a \in A$ and $t_\sigma \in T_\sigma$. The initial state $q_0 \in Q$ is defined as $q_0 = (a_0, \{t_{\sigma,0} | \sigma \in \Sigma_{act}\})$ where

$$t_{\sigma,0} = \begin{cases} u_\sigma, & \text{if } \sigma \in \Sigma_{spe} \\ l_\sigma, & \text{if } \sigma \in \Sigma_{rem} \end{cases}$$

The set $Q_m \in Q$ is given by a subset of $A_m \times \prod\{T_\sigma | \sigma \in \Sigma_{act}\}$. The event set Σ is defined as $\Sigma = \Sigma_{act} \cup \{tick\}$,where the additional event *tick* represents the passage of one time unit. The state transition function $\delta : Q \times \Sigma \longrightarrow Q$ is defined as follows. For any $\sigma \in \Sigma$ and any $q = (a, \{t_\tau | \tau \in \Sigma_{act}\}) \in Q, \delta(q, \sigma)$ is defined, written $\delta(q, \sigma)!$, if and only if one of the following conditions holds:

- $\sigma = tick$ and $\forall \tau \in \Sigma_{spe}; \delta_{act}(a,\tau)! \Rightarrow t_\tau > 0$

  i.e. no deadline of a prospective event in q is 0.

- $\sigma \in \Sigma_{spe}$ and $\delta_{act}(a,\sigma)!$ and $0 \leq t_\sigma \leq u_\sigma - l_\sigma$

  i.e. if σ is defined at activity a in $\mathbf{G_{act}}$ and the delay in its occurrence has been passed, it can occur but it should occur before its deadline.

- $\sigma \in \Sigma_{rem}$ and $\delta_{act}(a,\sigma)!$ and $t_\sigma = 0$

  i.e. if σ is defined at activity a in $\mathbf{G_{act}}$ and the delay in occurring it has been passed, it can occur.

When $\delta(q,\sigma)!$,$q' = \delta(q,\sigma) = (a', \{t'_\tau | \tau \in \Sigma_{act}\})$ is defined as follows:

1. If $\sigma = tick$, then $a' := a$ and, for each $\tau \in \Sigma_{act}$,

   - if $\tau \in \Sigma_{spe}$,
   $$t'_\tau := \begin{cases} u_\tau, & \text{if } \delta_{act}(a,\tau) \text{ is not defined} \\ t_\tau - 1, & \text{if } \delta_{act}(a,\tau)! \text{ and } t_\tau > 0 \end{cases}$$
   - if $\tau \in \Sigma_{rem}$,
   $$t'_\tau := \begin{cases} l_\tau, & \text{if } \delta_{act}(a,\tau) \text{ is not defined} \\ t_\tau - 1, & \text{if } \delta_{act}(a,\tau)! \text{ and } t_\tau > 0 \\ 0, & \text{if } \delta_{act}(a,\tau)! \text{ and } t_\tau = 0 \end{cases}$$

2. if $\sigma \in \Sigma_{act}$, then $a' := \delta_{act}(a,\sigma)$ and for any $\tau \in \Sigma_{act}$,

   - if $\tau \neq \sigma$ and $\tau \in \Sigma_{spe}$,
   $$t'_\tau := \begin{cases} u_\tau, & \text{if } \delta_{act}(a',\tau) \text{ is not defined} \\ t_\tau, & \text{if } \delta_{act}(a',\tau)! \end{cases}$$
   - if $\tau = \sigma$ and $\tau \in \Sigma_{spe}$, $t'_\tau := u_\sigma$.

   - if $\tau \neq \sigma$ and $\tau \in \Sigma_{rem}$,
   $$t'_\tau := \begin{cases} l_\tau, & \text{if } \delta_{act}(a',\tau) \text{ is not defined} \\ t_\tau, & \text{if } \delta_{act}(a',\tau)! \end{cases}$$

- if $\tau = \sigma$ and $\tau \in \Sigma_{rem}$, $t'_\tau := l_\sigma$.

In order to prevent a *tick* transition from being preempted indefinitely by repeated execution of an activity loop within a fixed unit time interval, a TDES should be *activity-loop-free (alf)* [29] ,i.e.

$$(\forall q \in Q)(\forall s \in \Sigma_{act}^{+})\ \delta(q, s) \neq q$$

For the readers who are familiar with *zeno* loops [14], an activity loop behavior is similar to a zeno loop, in that with an activity loop, an unbounded number of events can occur in a single *tick* interval.
Let Σ^* be the set of all finite strings of elements in Σ, including the empty string ε. The function δ can be generalized to $\delta : Q \times \Sigma^* \rightarrow Q$ in the natural way. The *closed* behavior, the strings that are generated by $\mathbf{G}$, and *marked* behavior, the strings that are generated by $\mathbf{G}$ and lead us to a marker state, of the TDES $\mathbf{G}$ are defined by

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

and

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

respectively. The term closed behavior is so called because it is a prefix-closed [29] language.

2.2 Controllability of TDES

Brandin and Wonham have developed a supervisory control framework for TDES [5]. As in the untimed supervisory control framework, the set Σ_{act} is partitioned into two subsets Σ_c and Σ_u of controllable and uncontrollable events, respectively, where $\Sigma_c \subseteq \Sigma_{rem}$, because the prospective events can not be disabled and they should occur before their deadline. An event $\sigma \in \Sigma_{act}$ that can preempt the event *tick* is called a *forcible* event. The set of forcible events is denoted by Σ_{for}. A forcible event can be either controllable or uncontrollable. By forcing an enabled event in Σ_{for} to occur, we can disable the event *tick*. In this framework a supervisor decides to disable or enable each event in $\Sigma_c \cup \{tick\}$. The simplest way to visualize the behavior of a TDES **G** under supervision is first to consider the infinite reachability tree of **G** before any control is operative [29]. Each node of the tree corresponds to a unique string s of $L(\mathbf{G})$. At each node of the tree we can define the subset of *eligible* events by

$$Elig_{\mathbf{G}}(s) := \{\sigma \in \Sigma |\ s\sigma \in L(\mathbf{G})\}$$

In order to define the notion of *controllability* we should consider a language $K \subseteq L(\mathbf{G})$and write

$$Elig_K(s) := \{\sigma \in \Sigma |\ s\sigma \in \bar{K})\}$$

K is *controllable* with respect to **G** if, for all $s \in \bar{K}$

$$Elig_K(s) \supseteq \begin{cases} Elig_{\mathbf{G}}(s) \cap (\Sigma_u \cup \{tick\}), & \text{if } Elig_K(s) \cap \Sigma_{for} = \emptyset \\ Elig_{\mathbf{G}}(s) \cap \Sigma_u, & \text{if } Elig_K(s) \cap \Sigma_{for} \neq \emptyset \end{cases} \tag{2.1}$$

Thus K controllable means that an event σ (in the full alphabet Σ including *tick*) may occur in K if σ is currently eligible in **G** and either (i) σ is uncontrollable, or (ii) $\sigma = tick$ and no forcible event is currently eligible in K.

The difference between controllability in TDES and in untimed DES is in the event *tick*. Event *tick* acts as an *uncontrollable* event in the states where no forcible event is present, but in the states with one or more forcible events, *tick* can be preempted (disabled) by a forcible event. So an event σ can occur in K if σ is currently eligible in **G** and either σ is uncontrollable or $\sigma = tick$ and no forcible event is currently eligible in K.

A *supervisory* control for **G** is any map

$$V : L(\mathbf{G}) \rightarrow 2^{\Sigma}$$

such that, for all $s \in L(\mathbf{G})$,

$$V(s) \supseteq \begin{cases} \Sigma_u \cup (\{tick\} \cap Elig_{\mathbf{G(s)}}), & \text{if } V(s) \cap Elig_{\mathbf{G(s)}} \cap \Sigma_{for} = \emptyset \\ \Sigma_u, & \text{if } V(s) \cap Elig_{\mathbf{G(s)}} \cap \Sigma_{for} \neq \emptyset \end{cases}$$

Actually, V decides after each string s of $L(\mathbf{G})$ which events can be enabled: no uncontrollable event can ever be disabled, nor can *tick* be disabled if there is no forcible event defined after s. 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/$\mathbf{G}$).

That is, $V/\mathbf{G}$ only generates the strings of $L(\mathbf{G})$ that are admitted by V.

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

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

Thus the marked behavior of $V/\mathbf{G}$ consists exactly of the strings of $L_m(\mathbf{G})$ that survive under supervision by V.

2.3 Supremal Controllable Sublanguage

Our control objective is, for the given plant language $L(\mathbf{G})$ and the specification language E (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 $E \cap L_m(\mathbf{G})$ which is controllable w.r.t $\mathbf{G}$ and also nonblocking.

Let $\mathbf{G}$ be a controlled TDES with Σ partitioned as in the previous section. Let $E \subseteq \Sigma^*$. We introduce the set of all sublanguages of E that are controllable with respect to $\mathbf{G}$:

$$\mathcal{C}(E) = \{K \subseteq E |\ K \textit{ is controllable wrt } \mathbf{G}\}$$

Proposition 2.3.1 [29] There exists a unique supremal element in $\mathcal{C}(E)$, sup$\mathcal{C}(E)$, which can be described as sup$\mathcal{C}(E) = \bigcup\{K |\ K \in \mathcal{C}(E)\}$.

Proposition 2.3.2 [29] 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 $\boldsymbol{G}$ such that $L_m(V/\boldsymbol{G}) = K$.*

Thus K is (if nonempty) the maximally permissive (or minimally restrictive) solution of the problem of supervising $\mathbf{G}$ in such a way that its behavior belongs to E and control is nonblocking. For more details see [29].

Chapter 3

Introduction to Binary Decision Diagrams

Binary Decision Diagrams (BDD) were introduced by Akers [1]. Then Bryant [7] introduced operations and algorithms that utilize the ordering of the variables in BDDs. Bryant showed that an interesting subset of Boolean functions could be represented by function graphs in size polynomial rather than exponential in the number of variables. After that, many different diagrams similar to BDDs have emerged. The most important objective for these types of decision diagrams has been, so far, to derive constructions that will reduce the complexity of verifying hardware implementing arithmetic (integer) functions.
The material in this chapter is based on the article by Bryant [7] in which Boolean functions are represented by Boolean function graphs.

3.1 Function Graphs

In order to define the **BDD**s, we first present the definition of a function graph.

Definition 3.1.1 *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 $index(v) \in \{1, ..., n\}$ and children $child(v, i) \in V, 0 \leq i < range(v)$, where n, $range(v) \in Z_+$. *A terminal* vertex v has as attribute a value $value(v) \in \{0, ..., M-1\}$, 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.

□

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.

In order to clarify the above definition, we present a simple example in Figure 3.1. The vertical axis to the left shows the indices for the vertices. We denote a vertex by $v_{i,j}$ where i is the index 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. If there are more than one label on the same line, then there are several edges connecting the same pair of vertices. The terminals are placed at the bottom of the graph, and indicated by a T on the vertical axis. The values of terminals are written inside the vertices. The range of all the vertices in this example is 2.

We have the following for this function graph:

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

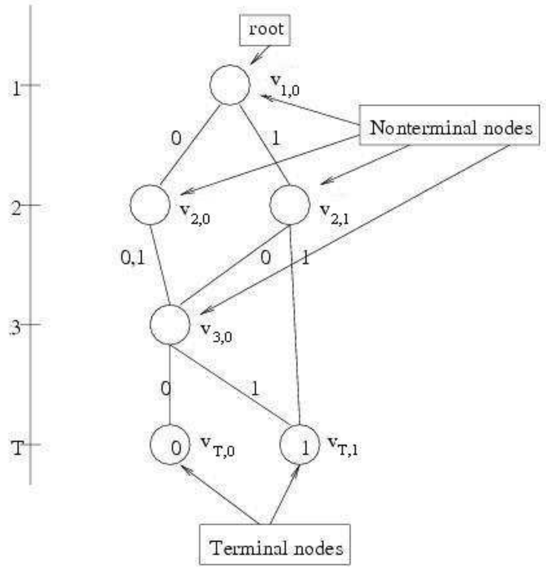


Figure 3.1: A simple function graph

Now we define the correspondence between the function graphs and functions.

Definition 3.1.2 *Function Graph to Function Connection*

A function graph G having a *root* vertex v denotes a function $f_v : R_1 \times R_2 \times ... \times R_n \rightarrow \{0, ..., M-1\}$ where $R_i = \{0, ..., range(v') - 1\}, index(v') = i$ and $i \in \{1, ..., n\}$, defined recursively as follows.

1. If v is a terminal vertex, then $f_v = value(v)$.

2. If v is a nonterminal vertex with $index(v) = i$, then $f_v : R_i \times R_{i+1} \times ... \times R_n \rightarrow \{0, ..., M-1\}$, with R_i as above, is the function

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

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

□

The function related to the graph in Figure 3.1 is shown in Table 3.1. The number of variables of the function is equal to the maximum of the indices of the vertices, namely 3 in this example. x_1, x_2 and x_3 correspond to indices 1,2 and 3. So we start from the root of the function graph and assign the edge labels to the corresponding function variable until we reach to a terminal. The value of the terminal will be the value of the function.

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 3.1: Function represented by Figure 3.1

The definition above defines f_v as an evaluation following one path from the root vertex down to a terminal vertex. In this general form we have no symbolic representation of the final f_v for the complete graph G. Symbolic 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 3.1.3 *Isomorphic Function Graphs*

The function graphs G and G' are *isomorphic* if there exists a one-to-one function θ from vertices of G onto vertices of G' such that for any vertex v if $\theta(v) = v'$, 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

$$\theta(child(v, i)) = child(v', i) \qquad 0 \leq i < range(v)$$

□

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

The mapping θ 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 Definition 3.1.1.

Definition 3.1.4 *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.

Lemma 3.1.1 If G is isomorphic to G' by the mapping θ, then for any vertex v in G, the subgraph rooted by v is isomorphic to the subgraph rooted by $\theta(v)$.

Definition 3.1.5 *Reduced Function Graph*

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

$$child(v,0) = child(v,1) = ... = child(v, range(v) - 1),$$

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

Lemma 3.1.2 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. For any function f over a finite domain with a fixed order of variables, there exists a unique (up to isomorphism) reduced function graph denoting f, and any other function graph denoting f contains more vertices. This is called a canonical form. The following theorem shows that a reduced function graph is a canonical form for the corresponding function.

Theorem 3.1.1 [27] Reduced function graph is a canonical form.

3.2 Binary Decision Diagrams

For the case of Boolean functions and variables we use a special form of the reduced function graph called binary decision diagram (BDD) [7]. The basic idea used in binary decision diagrams is to rewrite a Boolean expression in a recursive form and reuse common subexpressions. In the case of Boolean expressions this leads to highly efficient computations in most cases.

Suppose we have a Boolean expression $f(x_1, ..., x_n)$. We can then rewrite it using Shannon's expression formula:

$$f(x_1, ..., x_n) = ((\neg x_1) \wedge f(0, x_2, ..., x_n)) \ \vee (x_1 \wedge f(1, x_2, ..., x_n))$$

We continue with this recursively for each of the functions $f(0, x_2, ..., x_n)$ and $f(1, x_2, ..., x_n)$ w.r.t x_2 and then x_3,etc.. Let $g^1_0(x_2, ..., x_n) \triangleq f(0, x_2, ..., x_n), g^1_1(x_2, ..., x_n) \triangleq f(1, x_2, ..., x_n)$ and generally $g^i_{x'_1,...,x'_i}(x_{i+1}, ..., x_n) \triangleq f(x'_1, ..., x'_i, x_{i+1}, ..., x_n)$, where $x'_1, ..., x'_i$ are fixed boolean numbers. Then we obtain

$$f(x_1, ..., x_n) = (\ \neg x_1 \wedge \underbrace{(...(\underbrace{(\neg x_n \wedge \alpha)}_{g^{n-1}_{0,...,0}(x_n)} \vee \underbrace{(x_n \wedge \beta)}_{g^{n-1}_{0,...,1}(x_n)}\ ))}_{g^1_0(x_2,...,x_n)}\ ...) \vee (\ x_1 \wedge \underbrace{(...(\underbrace{(\neg x_n \wedge \gamma)}_{g^{n-1}_{1,...,0}(x_n)} \vee \underbrace{(x_n \wedge \delta)}_{g^{n-1}_{1,...,1}(x_n)}\ ))}_{g^1_1(x_2,...,x_n)}\ ...)$$

where $\alpha, \beta, \gamma, \delta \in \{0, 1\}$.

We see that we obtain several subexpressions with progressively fewer variables. In fact, all expressions $g^i_{x'_1,...,x'_i}$ above are Boolean expressions in the variables $\{x_{i+1}, ..., x_n\}$. In case some of these expressions are equal we should not have to repeat this part more than once, but instead substitute a reference to this common subexpression.

This is achieved by constructing a reduced function graph G for the function f, where each nonterminal vertex v with $index(v) = i$ corresponds to the variable x_i and the functions corresponding to the subgraphs of v are each equal to one of the subexpressions $g^i_{x'_1,...,x'_i}(x_{i+1}, ..., x_n)$.

The recursive Boolean expression from above can be visualized as a binary tree, where each vertex corresponds to the $\vee$, and where the $g^i_{x'_1,...,x'_i}$ expressions in principle are subtrees. This is the basis for the name **BDD**.

From Theorem 3.1.1 we know that a **BDD** is a canonical representation of a Boolean function for a given variable ordering. According to Definition 3.1.1 we have for **BDD**s that $range(v) = 2$ for all vertices and there are only two terminals, i.e., $M = 2$.

By changing the order in which we expand with respect to the *variables* we usually get large differences in the number of vertices (nodes). The ordering is called *variable ordering* and plays a significant role in lowering the representational complexity of the functions. A simple example is given below.

Suppose we have a 4-variable function $F : (x_1, x_2, x_3, x_4) \rightarrow \{0, 1\}$, where $x_i \in \{0, 1\}, i \in \{1, 2, 3, 4\}$. F is defined as

x_1	x_2	x_3	x_4	F
0	0	0	1	1
1	1	0	1	1

Table 3.2: Function F

The value of F for other values of (x_1, x_2, x_3, x_4) is 0.

The two **BDD**s representing F, using different variable ordering, are shown in Figure 3.2. Obviously the ordering x_1, x_2, x_3, x_4 is better than the ordering x_1, x_4, x_3, x_2 (6 nodes vs 8 nodes).

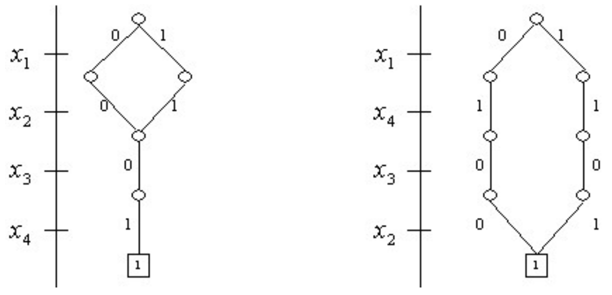


Figure 3.2: BDDs for representing function F with different variable ordering

Definition 3.2.1 *BDD to Function Connection*

A BDD G having root vertex v denotes a function f_v defined recursively as

1. If v is a terminal vertex, then $f_v = value(v) \in \{0, 1\}$.

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

$$f_v(x_i, x_{i+1}, ..., x_n) = \neg x_i \wedge \; f_{low(v)}(x_{i+1}, ..., x_n) \vee x_i \wedge \; f_{high(v)}(x_{i+1}, ..., x_n)$$

where $low(v) = child(v, 0)$ and $high(v) = child(v, 1)$.

$\square$

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

Chapter 4

Synthesis Algorithm Based on Predicates

4.1 Predicates

In order to place *conditions* on the states of the system $\mathbf{G}$, it will be convenient to use a logic formalism. A *predicate* P defined on Q is a function $P : Q \to \{0, 1\}$. P can always be identified with the corresponding state subset ("Truth Set")

$$Q_P := \{q \in Q \mid P(q) = 1\}.$$

For clarity, if $P(q) = 1$, we write $q \vDash P$, otherwise $q \nvDash P$.

Note: $Pred(Q)$ is the set of all the predicates on Q.

For $s \in \Sigma^*$ we say $t \in \Sigma^*$ is a prefix of s, and write $t \leq s$, if $s = tu$ for some $u \in \Sigma^*$. We define the *closed language* $L(\mathbf{G}, P)$ induced by a predicate $P \in Pred(Q)$ to be

$$L(\mathbf{G}, P) := \{w \in L(\mathbf{G}) \mid (\forall v \leq w)\delta(q_0, v) \vDash P\}$$

That is, the set of strings that are generated by $\mathbf{G}$ and leads us to a state which satisfies P through a path of states that all satisfy P .

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

$$L_m(\mathbf{G}, P) := \{w \in L(\mathbf{G}, P) \mid \delta(q_0, w) \in Q_m\}$$

That is, the set of strings in $L(\mathbf{G}, P)$ that lead us to the marker states.

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

Definition 4.1.1 *Reachability*

For a given predicate P, a state q is *reachable* iff

$$(\exists w \in L(\mathbf{G}, P))(\delta(q_0, w) = q)$$

That is, it can be reached from the initial state through a path of states that all satisfy P.

Definition 4.1.2 *Coreachability*

For a given predicate P, a state q is *coreachable* iff

$$(\exists w \in \Sigma^*)(\delta(q, w) \in Q_m) \textit{ and } (\forall v \leq w)(\delta(q, v) \models P)$$

That is, from that state a marker state can be reached through a path of states that all satisfy P.

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

- Plant: $\mathbf{G}_p = (Q_p, \Sigma, \delta_p, q_{p,0}, Q_{p,m})$
- Specification: $\mathbf{G}_s = (Q_s, \Sigma, \delta_s, q_{s,0}, Q_{s,m})$

Note that we consider that the event sets of the plant and specification are the same. Suppose that these event sets are different and 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 modelling, i.e., the events in $\Sigma_p - \Sigma_s$ are not known to be freely enabled or totally disabled [30]. For the common behavior of the plant and the specification, the meet[29] of $\mathbf{G}_p$ and $\mathbf{G}_s$ can be expressed as

$$\mathbf{G}_{meet} = \mathbf{meet}\,(\mathbf{G}_p, \mathbf{G}_s) = \textit{Reachable states of}\ \ (Q_p \times Q_s, \Sigma, \delta_p \times \delta_s, (q_{p,0}, q_{s,0}), Q_{p,m} \times Q_{s,m})$$

where

$$\delta_p \times \delta_s((q_p, q_s), \sigma) = (\delta_p(q_p, \sigma), \delta_s(q_s, \sigma))$$

Note: The closed behavior(marked behavior) of $\mathbf{G}_{meet}$ is the intersection of the closed behaviors(marked behaviors) of the plant and the specification.

Remark: The predicates can also be defined on a "2-dimensional" set. For example:

$$L(\mathbf{G}_{meet}, P) = \{w \in L(\mathbf{G}_{meet}) \mid (\forall v \leq w)(\delta_p(q_{p,0}, v), \delta_s(q_{s,0}, v)\ ) \models P\}$$

where $P \in \mathit{Pred}(Q_p \times Q_s)$.

Definition 4.1.3 *Controllability*

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

That is, at each state of $\mathbf{G}_{meet}$ an uncontrollable transition will lead us to a state that satisfies P. The uncontrollable transition can be by an uncontrollable event or by *tick* when there is no forcible event defined at that state.

Definition 4.1.4 *Nonblocking*

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

That is, from each state of $\mathbf{G}_{meet}$ that satisfies P you can reach to a marker state through a path of the states that all satisfy P.

In order to extend the theorems in untimed models to timed models, we define a *forcing-free* predicate.

Definition 4.1.5 *forcing-free predicate*

The forcing-free predicate $\mathcal{F}$ on $Q_p \times Q_s$ is defined as follows:

$$(q_p, q_s) \vDash \mathcal{F} \Leftrightarrow (\nexists \sigma \in \Sigma_{for})\ \delta_s(q_s, \sigma)!$$

where Q_p and Q_s are the state sets of the plant and the specification.

That is, there is no forcible event that is eligible in $\mathbf{G}_s$ in the forcing-free states of $\mathbf{G}_{meet}$.

Lemma 4.1.1 For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is controllable iff

$$(\ \forall (q_p, q_s) \vDash P, \forall \sigma \in \Sigma_u\ )(\delta_p(q_p, \sigma)!) \Rightarrow (\delta_s(q_s, \sigma)!\ \wedge\ (\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \vDash P)$$

and

$$(\ \forall (q_p, q_s) \vDash (P \wedge \mathcal{F})\ )(\delta_p(q_p, tick)!) \Rightarrow (\delta_s(q_s, tick)!\ \wedge\ (\delta_p(q_p, tick), \delta_s(q_s, tick)) \vDash P)$$

That is, at each state of $\mathbf{G}_{meet}$ that satisfies P, if an uncontrollable transition is eligible in $\mathbf{G}_p$ then it is also eligible in $\mathbf{G}_s$ and the target state also satisfies P. An uncontrollable transition occurs by an uncontrollable event or by *tick* at the forcing-free states.

Proof: For simplicity let $K := L(\mathbf{G}_{meet}, P)$ and $\mathbf{G} := \mathbf{G}_{meet}$. Obviously we have $\overline{K} = K$.
(if). We need to show

$$Elig_K(s) \supseteq \begin{cases} Elig_{\mathbf{G}}(s) \cap (\Sigma_u \cup \{tick\}), & \text{if } Elig_K(s) \cap \Sigma_{for} = \emptyset \\ Elig_{\mathbf{G}}(s) \cap \Sigma_u, & \text{if } Elig_K(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

Given a string $s \in K$, let $q_p = \delta_p(q_{p,0}, s)$ and $q_s = \delta_s(q_{s,0}, s)$. Then $(q_p, q_s) \vDash P$. Now there are two possible cases:

1. $(q_p, q_s) \vDash \mathcal{F}$,let $\sigma \in \Sigma_u \cup \{tick\}$. Now we have again two cases:

   - $\delta_p(q_p, \sigma)!$, then $\delta_s(q_s, \sigma)!$ and $(\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \vDash P$. Therefore $\sigma \in Elig_{\mathbf{G}}(s) \cap (\Sigma_u \cup \{tick\})$ and also $\sigma \in Elig_K(s)$.
   - $\delta_p(q_p, \sigma)$ is undefined. Then $\sigma \notin Elig_{\mathbf{G}}(s) \cap (\Sigma_u \cup \{tick\})$.

2. $(q_p, q_s) \nvDash \mathcal{F}$,let $\sigma \in \Sigma_u$. There are two possible cases:

   - $\delta_p(q_p, \sigma)!$, then $\delta_s(q_s, \sigma)!$ and $(\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \vDash P$. Therefore $\sigma \in Elig_{\mathbf{G}}(s) \cap (\Sigma_u \cup \{tick\})$ and also $\sigma \in Elig_K(s)$.
   - $\delta_p(q_p, \sigma)$ is undefined. Then $\sigma \notin Elig_{\mathbf{G}}(s) \cap (\Sigma_u \cup \{tick\})$.

(Only if). Given K is controllable w.r.t $\mathbf{G}$, suppose the contrary of the conclusion ,i.e.

$$(\exists (q_p, q_s) \vDash P, \sigma \in \Sigma_u)(\delta_p(q_p, \sigma)! \ \wedge ((\neg \delta_s(q_s, \sigma)!) \ \vee \ (\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \nvDash P))$$

It is obvious that now we have $s \in K$,$s\sigma \in L(\mathbf{G})$, but $s\sigma \notin K$, therefore K is not controllable w.r.t $L(\mathbf{G})$. This contradicts the assumption.
For the second condition, the proof is exactly the same.

□

Lemma 4.1.2 For the given $\mathbf{G}_p$ and $\mathbf{G}_s$, P is nonblocking iff

$$(\forall (q_p, q_s) \vDash P)((q_p, q_s) \ \textit{is reachable}) \Rightarrow ((q_p, q_s) \ \textit{is coreachable})$$

Proof: (If). We have to show that $\overline{L_m(\mathbf{G}_{meet}, P)} = L(\mathbf{G}_{meet}, P)$.
First we 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 $(\delta_p(q_{p,0}, s), \delta_s(q_{s,0}, s)) \vDash P$ and then $s \in L_m(\mathbf{G}_{meet}, P)$ because we always have $s \in L(\mathbf{G}_{meet})$.
Now we 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 $\delta_p(q_{p,0}, sw) \in Q_{p,m}, \delta_s(q_{s,0}, sw) \in Q_{s,m}$. It is enough to show $(\exists w \in \Sigma^*)(\delta_p(\delta_p(q_{p,0}, s), w) \in Q_{p,m}, \delta_s(\delta_s(q_{s,0}, s), w) \in Q_{s,m})$. This is obviously true because $(\delta_p(q_{p,0}, s), \delta_s(q_{s,0}, s))$ is reachable, and thus coreachable.
(Only If). Suppose the contrary, i.e. $(\exists (q_p, q_s) \in Q_p \times Q_s)), (q_p, q_s)$ is reachable but not

coreachable. Then there exists s, such that $\delta_p(q_{p,0}, s) = q_p, \delta_s(q_{s,0}, s) = q_s$, but there doesn't exist any w such that $\delta_p(q_p, w) \in Q_{p,m}, \delta_s(q_s, w) \in Q_{s,m}$. Thus

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

which gives us a contradiction.

$\square$

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

Theorem 4.1.1 Given $\mathbf{G}_p$ and $\mathbf{G}_s$, there exists a predicate P_{sup} such that

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

Proof: The proof is done constructively.
Given $\mathbf{G}_p$ and $\mathbf{G}_s$, we will present a synthesis algorithm for finding P_{sup}. As mentioned before, each predicate can be identified by its corresponding state subset. In this algorithm the predicates are determined by their truth set.

1. $P_{good1} = 1$
2. $[Q_p \times Q_s]_{P_{bad1}} := \{(q_p, q_s) \mid (q_p, q_s) \not\models P_{good1}$

   or

   $(\exists \sigma \in \Sigma_u)(\delta_p(q_p, \sigma)! \wedge \neg\delta_s(q_s, \sigma)!)$

   or

   $(q_p, q_s) \models \mathcal{F} \wedge \delta_p(q_p, tick)! \wedge \neg\delta_s(q_s, tick)!$

   or

   $(\exists \sigma \in \Sigma_u)((\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \not\models P_{good1})$

   or

   $(q_p, q_s) \models \mathcal{F} \wedge (\delta_p(q_p, tick), \delta_s(q_s, tick)) \not\models P_{good1}\}$

3. $[Q_p \times Q_s]_{P_{bad2}} := \{(q_p, q_s) \mid (\exists w \in \Sigma_u^*)(\ (\delta_p(q_p, w), \delta_s(q_s, w)) \vDash P_{bad1}\ )$

   $$or$$

   $$((q_p, q_s) \vDash \mathcal{F}) \wedge (\exists \sigma \in \Sigma_u \cup \{tick\})((\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \vDash P_{bad1})\}$$

4. $[Q_p \times Q_s]_{P_{re}} := \{(q_p, q_s) \mid (\exists w \in \Sigma^*)(\delta_p(q_{p,0}, w) = q_p, \delta_s(q_{s,0}, w) = q_s) \wedge$

   $$(\forall v \leq w)(\delta_p(q_{p,0}, v), \delta_s(q_{s,0}, v)) \nvDash P_{bad1} \vee P_{bad2}\}$$

5. $[Q_p \times Q_s]_{P_{cr}} := \{(q_p, q_s) \mid (\exists w \in \Sigma^*)(\delta_p(q_p, w) \in Q_{p,m}, \delta_s(q_s, w) \in Q_{s,m}) \wedge$

   $$(\forall v \leq w)(\delta_p(q_p, v), \delta_s(q_s, v)) \nvDash P_{bad1} \vee P_{bad2}\}$$

6. $P_{good2} = P_{re} \wedge P_{cr}$

7. If $P_{good2} \subset P_{good1}$, repeat steps 2-7 with $P_{good1} := P_{good2}$. Otherwise let $P_{sup} = P_{newgood}$ and the algorithm terminates here.

- The algorithm will terminate in finite steps. Define $|P|$ to be the number of states where a predicate P holds. Obviously in each cycle, if the algorithm does not terminate, $|P_{bad}|$ must increase by at least 1 at step 7. Therefore, the algorithm can iterate at most $|Q_p| \times |Q_s|$ times, which is finite.

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

  - P_{sup} is nonblocking, because we know that $P_{good2} = P_{re} \cap P_{cr}$,i.e. the states which satisfy P_{good2} are both reachable and coreachable, so P_{sup} is nonblocking.

  - P_{sup} is controllable.

    Suppose

    $$(\exists (q_p', q_s') \vDash P_{sup})(\exists \sigma \in \Sigma_u)(\delta_p(q_p', \sigma)! \wedge \neg \delta_s(q_s', \sigma)!)$$

    then according to step 2, $(q_p', q_s') \vDash P_{bad1}$ which contradicts $(q_p', q_s') \vDash P_{sup}$, so we have

    $$(\forall (q_p, q_s) \vDash P_{sup})(\forall \sigma \in \Sigma_u)(\delta_p(q_p, \sigma)! \Rightarrow \delta_s(q_s, \sigma)!) \tag{4.1}$$

Now suppose that

$$(\exists(q_p', q_s') \vDash P_{sup})(\exists\sigma \in \Sigma_u)((\delta_p(q_p', \sigma), \delta_s(q_s', \sigma)) \nvDash P_{sup})$$

then $(q_p', q_s') \vDash P_{bad2}$, which contradicts $(q_p', q_s') \vDash P_{sup}$, so we have

$$(\forall(q_p, q_s) \vDash P_{sup})(\forall\sigma \in \Sigma_u)((\delta_p(q_p, \sigma)! \wedge \delta_s(q_s, \sigma)!) \Rightarrow (\ (\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \vDash P_{sup}\ ) \tag{4.2}$$

For the states of $\mathbf{G}_{meet}$ that satisfy $\mathcal{F}$, by the same reasoning we can conclude that

$$(\forall(q_p, q_s) \vDash (P_{sup} \wedge \mathcal{F}))(\forall\sigma \in \Sigma_u \cup \{tick\})(\delta_p(q_p, \sigma)! \Rightarrow \delta_s(q_s, \sigma)!) \tag{4.3}$$

and

$$\begin{aligned}&(\forall(q_p, q_s) \vDash (P_{sup} \wedge \mathcal{F}))(\forall\sigma \in \Sigma_u \cup \{tick\})\\ &((\delta_p(q_p, \sigma)! \wedge \delta_s(q_s, \sigma)!) \Rightarrow (\ (\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \vDash P_{sup}\ )\end{aligned} \tag{4.4}$$

Equations 4.1,4.2,4.3,4.4 show that P_{sup} is controllable (Lemma 4.1.2).

- $L_m(\mathbf{G}_{meet}, P_{sup}) \subseteq L_m(\mathbf{G}_{meet}) = L_m(\mathbf{G}_p) \cap L_m(\mathbf{G}_s)$, so we have

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

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

We have to show that

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

Suppose the contrary. Then we can find a K', which is controllable and nonblocking but $K' \nsubseteq L_m(\mathbf{G}_{meet}, P_{sup})$ i.e. $(\exists s \in K')(s \notin L_m(\mathbf{G}_{meet}, P_{sup}))$. Let $q_p = \delta_p(q_{p,0}, s)$ and $q_s = \delta_s(q_{s,0}, s)$. We have $(q_p, q_s) \nvDash P_{sup}$.

We know that our algorithm has a loop (steps 2-7). We write $P_{good1}(i)$, $P_{good2}(i)$,$P_{bad1}(i)$

, $P_{bad2}(i)$ for their values in cycle i.

Now we want to show that:

If

$$(\exists s \in K')(\ (\delta_p(q_{p,0}, s), \delta_s(q_{s,0}, s)) \nvDash P_{good2}(n)\ )$$

then

$$(\exists s' \in K')(\ (\delta_p(q_{p,0}, s'), \delta_s(q_{s,0}, s')) \nvDash P_{good1}(n)\ )$$

In order to show the above statement we get $(\hat{q}_p, \hat{q}_s) = (\delta_p(q_{p,0}, s), \delta_s(q_{s,0}, s))$. If $(\hat{q}_p, \hat{q}_p) \nvDash P_{good1}(n)$, then we get $s' = s$. Otherwise (i.e. $(\hat{q}_p, \hat{q}_p) \vDash P_{good1}(n)$) we have $(\hat{q}_p, \hat{q}_s) \nvDash P_{re}(n)$ or $(\hat{q}_p, \hat{q}_s) \nvDash P_{cr}(n)$. Suppose we have $(\hat{q}_p, \hat{q}_s) \nvDash P_{re}(n)$. Then either

$$(\hat{q}_p, \hat{q}_s) \vDash (P_{bad1}(n) \cup P_{bad2}(n))$$

or

$$(\exists w \leq s)(\ (\delta_p(q_{p,0}, w), \delta_s(q_{s,0}, w)) \vDash (P_{bad1}(n) \cup P_{bad2}(n))\ )$$

If the former is true, let $(\tilde{q}_p, \tilde{q}_s) = (\hat{q}_p, \hat{q}_s)$ and $s_1 = s$; otherwise let $(\tilde{q}_p, \tilde{q}_s) = (\delta_p(q_{p,0}, w), \delta_s(q_{s,0}, w))$ and $s_1 = w$ ($w \in K'$).

Similar reasoning can also be applied if $(\hat{q}_p, \hat{q}_s) \nvDash P_{cr}(n)$ Now we have

$$(\exists s_1 \in K')(\ (\tilde{q}_p, \tilde{q}_s) = (\delta_p(q_{p,0}, s_1), \delta_s(q_{s,0}, s_1)), (\tilde{q}_p, \tilde{q}_s) \vDash P_{bad1}(n) \cup P_{bad2}(n)\ )$$

If $(\tilde{q}_p, \tilde{q}_s) \nvDash P_{bad1}(n)$, we must have $(\tilde{q}_p, \tilde{q}_s) \nvDash P_{bad2}(n)$ thus

$$(\exists s_2 \in \Sigma^*)(\ (\delta_p(\tilde{q}_p, s_2), \delta_s(\tilde{q}_s, s_2)) \vDash P_{bad1}(n)\ )$$

In this case let $(q'_p, q'_s) = (\delta_p(\tilde{q}_p, s_2), \delta_s(\tilde{q}_s, s_2))$ and $s_3 = s_1 s_2$. Otherwise, i.e. $(\tilde{q}_p, \tilde{q}_s) \vDash P_{bad1}(n)$, let $(q'_p, q'_s) = (\tilde{q}_p, \tilde{q}_s)$ and $s_3 = s_1$. Now we have

$$(\exists s_3 \in K')((q'_p, q'_s) = (\delta_p(q_{p,0}, s_3), \delta_s(q_{s,0}, s_3)), (q'_p, q'_s) \vDash P_{bad1}(n)\ )$$

Based on the definition of P_{bad1}, there are 5 possible cases:

- $(q'_p, q'_s) \nvDash P_{good1}(n)$. In this case s_3 is what we want.
- $(\exists \sigma \in \Sigma_u)(\ (\delta_p(q'_p, \sigma)!) \wedge (\neg \delta_s(q'_s, \sigma)!)\ )$ This is impossible because this will make K' uncontrollable.
- $((q'_p, q'_s) \vDash \mathcal{F}) \wedge (\ (\delta_p(q'_p, tick)!) \wedge (\neg \delta_s(q'_s, tick)!)\ )$This is also impossible because this will make K' uncontrollable.
- $(\exists \sigma \in \Sigma_u)(\ (\delta_p(q'_p, \sigma), \delta_s(q'_s, \sigma)) \nvDash P_{good1}(n)\ )$. In this case we must have $s_3\sigma \in K'$ because K' is controllable . Also we have $(\delta_p(q_{p,0}, s_3\sigma), \delta_s(q_{s,0}, s_3\sigma) \nvDash P_{good1}(n)$. So $s_3\sigma$ is what we want.
- $((q'_p, q'_s) \vDash \mathcal{F}) \wedge (\ (\delta_p(q'_p, tick), \delta_s(q'_s, tick)) \nvDash P_{good1}(n)\ )$. So $s_3(tick)$ is what we want.

Because $P_{good1}(n) = P_{good2}(n-1)$, this process can continue inductively until we finally reach $n = 1$,i.e. it must be true that

$$(\exists s' \in K')(\ (\delta_p(q_{p,0}, s'), \delta_s(q_{s,0}, s')) \nvDash P_{good1}(1)\ )$$

But this is impossible, because we have $P_{good1}(1) \equiv 1$.

Thus the conclusion is proved.

$\square$

4.2 Implementation of Supervisory Controller by Predicates

As mentioned before, a *supervisory* control for $\mathbf{G}$ is any map

$$V : L(\mathbf{G}) \rightarrow 2^{\Sigma}$$

such that, for all $s \in L(\mathbf{G})$,

$$V(s) \supseteq \begin{cases} \Sigma_u \cup (\{tick\} \cap Elig_{\mathbf{G}}(s)), & \text{if } V(s) \cap Elig_{\mathbf{G}}(s) \cap \Sigma_{for} = \emptyset \\ \Sigma_u, & \text{if } V(s) \cap Elig_{\mathbf{G}}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

Thus it can also be expressed as

$$V' : Q \times \Sigma \to \{0, 1\}$$

where

$$V'(q, \sigma) = \begin{cases} 1, & \text{if } \sigma \in V(s) \\ 0, & \text{Otherwise} \end{cases}$$

if $q = \delta(q_0, s)$ and $\sigma \in \Sigma_a \cup \{tick\}$.

We can express V' in the form of $n + 1$ scalar functions:

$$V' \equiv \{V_i' : Q \to \{0, 1\}, i \in \{0, 1, ..., n\}\ \}$$

where n is the number of events except $tick$, V_i' is the predicate corresponds to event σ_i, $\sigma_0 = tick$ and $\sigma_i \in \Sigma_a\ for\ i = 1, ..., n$.

Thus V' is nothing but $n + 1$ predicates. Therefore supervisory control can be implemented by predicates in a straightforward way.

The V' defined above can be obtained as follows:

Let

$$Q := Q_p \times Q_s$$

$$q = (q_p, q_s)$$

$$\Delta(q, \sigma) = (\delta_p(q_p, \sigma), \delta_s(q_s, \sigma))$$

We define

$$V_i'(q) = \begin{cases} 1, & \text{if } q \vDash P_{sup}\ \ \&\ \ \Delta(q, \sigma_i)!\ \ \&\ \ \Delta(q, \sigma_i) \vDash P_{sup} \\ 0, & \text{otherwise} \end{cases}$$

Thus the state q satisfies the predicate for event σ_i if q satisfies P_{sup} and we can reach from q to another state that satisfies P_{sup} by the event σ_i.

Obviously, for any $\sigma_i \in \Sigma_u, V_i' = 1$, because P_{sup} is controllable. If $q \vDash \mathcal{F}$, then $V_0'(q) = 1$ because P_{sup} is controllable. Also, clearly $L(V'/\mathbf{G}) = L(\mathbf{G}, P_{sup})$ and $L_m(V'/\mathbf{G}) = L_m(\mathbf{G}, P_{sup})$, since exactly the same constraints are imposed upon $\mathbf{G}$ by V' and by P_{sup}.

Chapter 5

BDD based implementation

5.1 Motivation

As shown in the previous chapter, all the computations needed for finding the supremal controllable sublanguage can be realized by using predicates. However, if we use the naive representation of predicates, the value table, we gain no advantage compared with the previous implementations like TCT and TTCT. Fortunately, Binary Decision Diagrams [8] can be used here so that we can actually represent the predicates in *compressed* form. The approach using Integer Decision Diagrams (an extended form of BDDs) has been applied to Discrete Event Systems in [13]. The focus of that work is the relational representation, not the computation. A direct and efficient algorithm based on IDDs, namely STCT, is presented in [30].

All these works deal with untimed models. The BW framework for Timed DES has its own difficulties. A TDES can be much bigger in size compared to the corresponding untimed model. We present a very simple example to illustrate this [20]. Assume that a public parking spot is modelled as shown in Figure 5.1. It may be interpreted as follows. If the parking spot is *idle* then a car may be *parked* in it. If the parking spot is *occupied* then the car may be *unparked*.

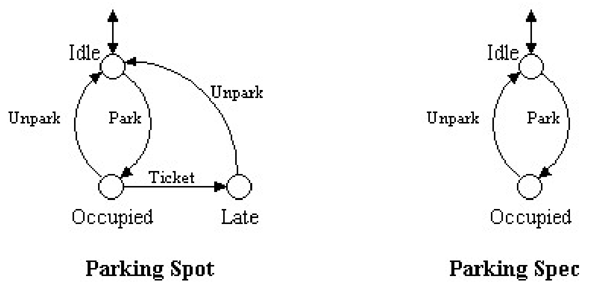


Figure 5.1: A Car parking setup and specification

However if the car remains parked for longer than a certain duration of time then a traffic officer may give it a *ticket* for traffic violation. Of course the car may be *unparked* even after receiving a ticket (we assume that the cars do not get towed). A desirable specification from the point of view of a car driver is also shown in Figure 5.1. It simply says "Avoid getting a ticket". Parking and unparking are controllable events but getting a ticket is uncontrollable. This is the case where a timed model is essential because the parking specification is uncontrollable if we use an untimed model. So let us define time bounds for each event: $(park, 30, \infty), (unpark, 7, \infty), (ticket, 950, 999)$. This may be interpreted as follows. It takes at least 30 ticks to park a car; it takes at least 7 ticks to unpark; it is safe to park for 950 ticks but it is possible not to get ticketed for a further 49 ticks. If we use the timed transition graph (TTG) of the system and assume that *unpark* is a forcible event, the parking specification is now controllable and the supervisor simply *forces* the car owner to *unpark* the car before 950 ticks. For this simple example we see that the timed transition graph has 1052 states and 2115 transitions!
So the problem of *state explosion* is more obvious in timed models, and with the previous implementations we cannot deal with big examples. For example TTCT cannot find the Timed Transition Graph(TTG) for the Activity Transition Graphs with time bounds greater than 1000. So in this work we will use Binary Decision Diagrams for reducing the size of our data structures , in order to work with much bigger examples in a fast way.

In this chapter, first we describe the method which is used in previous implementations and then describe our new approach for attacking this problem in two major steps:

- Converting an ATG to its corresponding TTG.
- Finding the supremal controllable and nonblocking sublanguage.

5.2 Previous Implementation

Although the original theoretical results in discrete event systems were presented in terms of formal languages, the actual computation can only be done in terms of finite state machines. In TCT and TTCT, any finite state machine, for example **G**, is described by an object which has a list of all its states and transitions. So the space required for storing a DES or TDES is proportional to *number of states* [11]. In almost all cases, the plant is modelled by a set of finite state machines, i.e. by synchronizing all the plant component models into one finite state machine. The specification can also be found by synchronizing smaller specifications. Finally the supremal controllable sublanguage can be computed by using the **supcon** procedure. As far as complexity is concerned in both time and space, the procedure $TDES3 = \textbf{sync}(TDES1, TDES2)$ which computes the synchronous product of two TDES (DES), has complexity of $O(Number\ of\ states\ of\ TDES1 \times Number\ of\ states\ of\ TDES2)$. Theoretically the number of states of the synchronous product of two TDES is less than or equal to the product of their number of states. But in reality it is often much less than their product for a nontrivial system. Therefore we often need to allocate much more space than is actually required to store the result. For example if two systems have 10,000 states each, we will need space of size 100,000,000, even though the result may only have 50,000 states.

What makes matters worse is that the first step in computing **supcon**(plant,specification) is computing the synchronous product of the plant and the specification, and this is often prohibitive in practice. This complexity problem for untimed DES is solved by STCT

[30], which is many times faster and smaller in the sense of memory usage than TCT. STCT does not need an exhaustive list of the entire state space and thus is able to achieve a performance far beyond that of TCT.

For a timed model this complexity problem is more critical, because as we saw in the previous section state explosion is much faster in untimed models than timed ones. Clearly for large systems an exhaustive list of the entire state space cannot be used.

5.3 Converting ATG to TTG

Constructing the TTG from the ATG of the TDES is based on the definition of $\delta(q, \sigma)$. Actually we should find the reachable states of the TDES. As we saw in chapter 2, each state of the TDES has two kinds of component: activity and timer of each event. Thus each state would be an $n + 1$ tuple, where n is the number of events in the system.

The natural way to find these states is to start from the initial state $q_0 = (a_0, \{t_{\sigma,0} | \sigma \in \Sigma_{act}\})$ where

$$t_{\sigma,0} = \begin{cases} u_\sigma, & \text{if } \sigma \in \Sigma_{spe} \\ l_\sigma, & \text{if } \sigma \in \Sigma_{rem} \end{cases}$$

and find the reachable states by the rules given in the definition of δ. In this method, we should go through all the states and check the conditions to find the next states in the graph. If the time bounds of events are big, the number of states will be much bigger. Using this exhaustive search method would not be efficient and fast. Additionally, using the tables for storing these states would be much too memory-consuming. Instead we convert the rules for defining the transitions in TDES to logical operations that can be applied to predicates. By these logical operations, all the possible transitions in the system which satisfy the rules, can be found in the form of predicate. After finding the transition predicates, the reachable states can be found by applying suitable logical operations as we will discuss later.

5.3.1 Transition Predicates

All the transitions of the TDES can be presented by a predicate. For example; in case $q' = \delta(q, \sigma)$, then this transition can be expressed as:

$$T \equiv (current = q) \wedge (event = \sigma) \wedge (next = q')$$

where *current*, *event* and *next* are variables. If we have more transitions in the system, we can disjunct them together. For the TDES in Figure 5.2 we can write:

$$\begin{aligned} T \equiv &(\ (current = 1) \wedge (event = tick) \wedge (next = 2)\ ) \\ &\vee (\ (current = 2) \wedge (event = \alpha) \wedge (next = 3)\ ) \\ &\vee (\ (current = 3) \wedge (event = \beta) \wedge (next = 1)\ ) \\ &\vee (\ (current = 3) \wedge (event = tick) \wedge (next = 4)\ ) \end{aligned}$$

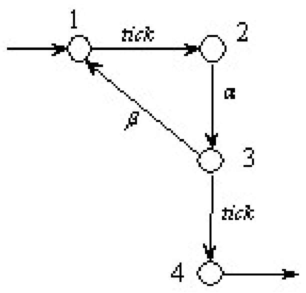


Figure 5.2: Transition predicate

For our purpose, the predicate would be more complex, because each state as mentioned before has some timers. So we have

$$current = (act = a) \wedge (t_{\sigma_1} = t_1) \wedge ... \wedge (t_{\sigma_n} = t_n)$$

and

$$next = (act' = a') \wedge (t'_{\sigma_1} = t'_1) \wedge ... \wedge (t'_{\sigma_n} = t'_n)$$

where $a, a' \in \Sigma_{act}$; if $\sigma \in \Sigma_{rem},\ t_\sigma \in [0, l_\sigma]$; and if $\sigma \in \Sigma_{spe},\ t_\sigma \in [0, u_\sigma]$

Now we have to figure out how to change the conditions of transitions in TDES to logical operations on these predicates. This will be done for prospective events, remote events and *tick* independently.

If $q = (a, \{t_\tau | \tau \in \Sigma_{act}\}), q' = (a', \{t'_\tau | \tau \in \Sigma_{act}\}) \in Q$, we have:

- $\sigma = tick$.

  $\delta(q, tick)! \Leftrightarrow (\ (\forall \tau \in \Sigma_{spe}) \delta_{act}(a, \tau)! \Rightarrow t_\tau > 0\ )$

  Thus the timer value of the prospective events that are defined in a should be greater than zero for occurring event *tick*. So the current state in the transition predicate for event *tick* will have the form:

  $$current = (act = a) \wedge (\bigwedge_{\tau \in \Sigma_{spe}, \delta_{act}(a,\tau)!} (t_\tau > 0)\ )$$

  We will reach state q' from q after occurring event *tick*:

  $\delta(q, tick) = q' \Leftrightarrow a' = a$ and for each $\tau \in \Sigma_{act}$,

  – if $\tau \in \Sigma_{spe}$, $t'_\tau := \begin{cases} u_\tau, & \text{if } \delta_{act}(a,\tau) \text{ is not defined} \\ t_\tau - 1, & \text{if } \delta_{act}(a,\tau)! \text{ and } t_\tau > 0 \end{cases}$

  – if $\tau \in \Sigma_{rem}$, $t'_\tau := \begin{cases} l_\tau, & \text{if } \delta_{act}(a,\tau) \text{ is not defined} \\ t_\tau - 1, & \text{if } \delta_{act}(a,\tau)! \text{ and } t_\tau > 0 \\ 0, & \text{if } \delta_{act}(a,\tau)! \text{ and } t_\tau = 0 \end{cases}$

  Thus the activity of the next state (act') will not change and the timer value of the events which are not defined at activity a in $\mathbf{G_{act}}$ will be their default value in the next state. The timer value of the events that are defined at activity a will be decreased by one unit in the next state if it is positive and will remain zero if it is zero (the latter case is only for the remote events). So the next state in the transition predicate for event *tick* will have the form:

  $$next = (act' = a) \wedge (\bigwedge_{\delta_{act}(a,\tau)! \wedge t_\tau > 0} (t'_\tau = t_\tau - 1)\ ) \wedge (\bigwedge_{\neg \delta_{act}(a,\tau)!} (t'_\tau = t_{\tau,0})\ ) \wedge (\bigwedge_{\delta_{act}(a,\tau)! \wedge t_\tau = 0} (t'_\tau = 0)\ )$$

  The transition predicate of *tick* for the states of the TDES whose activity is "a" will be $current \wedge (event = tick) \wedge next$. In order to find the transition predicate for all the *tick* transitions of the TDES, we should find the disjunction of the transition predicates of *tick* transitions for the states with different activities. So T_{tick} can be defined according to

$$\begin{aligned} T_{tick} = \bigvee_{\forall a \in A} \Big\{ \{(act = a) \wedge (\bigwedge_{\tau \in \Sigma_{spe}, \delta_{act}(a,\tau)!} (t_\tau > 0)) \Big\} \\ \wedge \Big\{ (event = tick) \Big\} \\ \wedge \Big\{ (act' = a) \wedge (\bigwedge_{\delta_{act}(a,\tau)! \wedge t_\tau > 0} (t'_\tau = t_\tau - 1)) \wedge \\ (\bigwedge_{\neg \delta_{act}(a,\tau)!} (t'_\tau = t_{\tau,0})) \wedge (\bigwedge_{\delta_{act}(a,\tau)! \wedge t_\tau = 0} (t'_\tau = 0)) \Big\} \Big\} \end{aligned}$$

- σ is prospective.

  $\delta(q,\sigma)! \Leftrightarrow \delta_{act}(a,\sigma)! \ \wedge \ 0 \leq t_\sigma \leq u_\sigma - l_\sigma$

  That is, a prospective event can occur in state q if that event is defined in the untimed model at activity a and the delay for occurring it has been passed($t_\sigma \leq u_\sigma - l_\sigma$). So the current state in the transition predicate for prospective event σ is:

$$current = (act = a) \wedge (0 \leq t_\sigma \leq u_\sigma - l_\sigma)$$

Event σ will lead us to a state whose activity is $a' = \delta_{act}(a,\sigma)$ and the timer value of σ will reset to its default value. The timer value of other events will be remaind unchanged if they are defined at activity a' and will reset to their default value if they are not defined at a'.

$\delta(q,\sigma) = q' \Leftrightarrow a' = \delta_{act}(a,\sigma)$ and for any $\tau \in \Sigma_{act}$,

- if $\tau \neq \sigma$, $t'_\tau := \begin{cases} t_{\tau,0}, & \text{if } \delta_{act}(a',\tau) \text{ is not defined} \\ t_\tau, & \text{if } \delta_{act}(a',\tau)! \end{cases}$
- if $\tau = \sigma$, $t'_\tau := u_\sigma$.

So the next state in the transition predicate for the prospective event σ will have the form:

$$next = (act' = a'\ ) \wedge (\bigwedge_{\tau\neq\sigma, \neg\delta_{act}(a',\tau)!} (t'_\tau = t_{\tau,0})\ ) \wedge (\bigwedge_{\tau\neq\sigma,\ \delta_{act}(a',\tau)!} (t'_\tau = t_\tau)\ ) \wedge (t'_\sigma = u_\sigma)$$

The transition predicate of prospective event σ for the states of the TDES whose activity is "a" will be $current \wedge (event = tick) \wedge next$. In order to find the transition predicate for all the σ transitions of the TDES, we should find the disjunction of the transition predicates of σ transitions for the states of the TDES with different activities. According to these rules the transition predicate can be computed for σ:

$$\begin{aligned} T_\sigma = \bigvee_{a\in A\ \wedge\delta_{act}(a,\sigma)!} \Big\{ & \big\{(act = a) \wedge (0 \le t_\sigma \le u_\sigma - l_\sigma)\big\} \\ & \wedge \big\{(event = \sigma)\big\} \\ & \wedge \big\{(act' = a'\ ) \wedge (\bigwedge_{\tau\neq\sigma,\neg\delta_{act}(a',\tau)!} (t'_\tau = t_{\tau,0})\ )\wedge \\ & (\bigwedge_{\tau\neq\sigma,\ \delta_{act}(a',\tau)!} (t'_\tau = t_\tau)\ ) \wedge (t'_\sigma = u_\sigma)\ \big\}\ \Big\} \end{aligned}$$

- Remote events.

  $\delta(q,\sigma)! \Leftrightarrow \delta_{act}(a,\sigma)!\ \wedge\ t_\sigma = 0$

  Thus the remote event σ will occur at state q if it is defined at activity "a" in $\mathbf{G_{act}}$ and its delay has been passed. By occurring σ we will reach the state q':

  $\delta(q,\sigma) = q' \Leftrightarrow a' = \delta_{act}(a,\sigma)$ and for any $\tau \in \Sigma_{act}$,

  - if $\tau \neq \sigma$, $t'_\tau := \begin{cases} t_{\tau,0}, & \text{if } \delta_{act}(a',\tau) \text{ is not defined} \\ t_\tau, & \text{if } \delta_{act}(a',\tau)! \end{cases}$
  - if $\tau = \sigma$, $t'_\tau := l_\sigma$.

  The next state's activity will be $a' = \delta_{act}(a,\sigma)$ and the timer value of σ will reset to its default value. The timer value of other events will be remaind unchanged if they are defined at activity a' and will reset to their default value if they are not defined at a'. So the transition predicate for remote event σ will be constructed in the same way as what we did for prospective events :

$$T_\sigma = \bigvee_{a \in A \;\wedge \delta_{act}(a,\sigma)!} \Big\{ \; \{(act = a) \wedge (t_\sigma = 0)\}$$
$$\wedge \{(event = \sigma)\}$$
$$\wedge \{(act' = a' \;) \wedge (\bigwedge_{\tau \neq \sigma, \neg\delta_{act}(a',\tau)!} (t'_\tau = t_{\tau,0}) \;) \wedge$$
$$(\bigwedge_{\tau \neq \sigma, \;\delta_{act}(a',\tau)!} (t'_\tau = t_\tau) \;) \wedge (t'_\sigma = l_\sigma) \;\} \;\Big\}$$

After finding the transition predicate for each event, the transition predicate of the TDES would be the disjunction of all of them.

$$T = T_{tick} \vee (\bigvee_{\sigma \in \Sigma_{act}} T_\sigma) \tag{5.1}$$

But T is not what we want because it may contain transitions from states which are not reachable from initial state. Now we have to find the reachable states from this predicate by applying logical operations. Thus we have to define a function for finding the destination states from a set of states and a set of events, i.e.

$$\Omega(X, \Gamma) = \{y \in Q | (\exists x \in X)(\exists \gamma \in \Gamma)\delta(x, \gamma) = y\} \subseteq Q$$

where $X \subseteq Q$ and $\Gamma \subseteq \Sigma$.

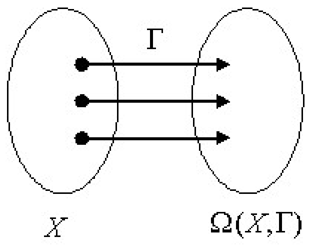


Figure 5.3: Target states

The target states can be computed by conjunction and existential quantification (called *relational product* in symbolic model checking).

$$\exists(next) \left[(\bigvee_{x \in X} (current = x) \wedge \bigvee_{\gamma \in \Gamma} (event = \gamma) \;) \wedge T(current, event, next) \right]$$

In order to do these computations we employ binary decision diagrams (**BDD**s) as the representation of predicates.

The algorithm for finding the reachable states is as follows:

```
$ReachableStates := InitialState;$
DO
{
    $tmp := ReachableStates;$
    $new := NextStates(ReachableStates, T);$
    $ReachableStates := ReachableStates \vee new;$
}
While $tmp \neq ReachableStates;$
```

Actually, we have started from the initial state and find all the states we can reach from it.

As stated in Chapter 3, ordering of the variables in a BDD is one of the most important optimizations. The variables we have in finding the transition predicate are: $act, t_{\sigma_1}, ..., t_{\sigma_n}, act', t'_{\sigma_1}, ..., t'_{\sigma_n}$. Since the number of elements in the domain of each variable is greater than 2, each variable itself consists of a number of **BDD** variables. The more closely coupled two variables are, the "closer" the two components should be placed in the **BDD**. In our algorithm, the variables representing t_σ and t'_σ should be adjacent because in most of the transitions t_σ and t'_σ differ only in one or zero unit of time. If we use a different ordering of variables, the number of **BDD** nodes will be increased and the algorithm may become much too slow.

One of the main problems in working with **BDD**s is the problem of "intermediate node number explosion" [30][18], i.e. although the number of **BDD** nodes in the final result may not be large, it could be tens of times larger during the synthesis process, or more accurately, during the generation of reachable states. In order to reduce this problem in finding reachable states, instead of using one transition predicate for all the transitions we can use one transition predicate for each event, i.e. the variable *event* will be removed from transition predicate. We must also change the algorithm for finding the reachable states.

```
*ReachableStates := InitialState;*
**DO**
{
    $tmp_1 := ReachableStates;$
    $for\ \ \forall\sigma \in \Sigma \cup \{tick\}$
    {
        **DO**
        {
            $tmp_2 := ReachableStates;$
            $new := NextStates(ReachableStates, T_\sigma);$
            $ReachableStates := ReachableStates \vee new;$
        }
        **While** $tmp_2 \neq ReachableStates;$
    }
}
**While** $tmp_1 \neq ReachableStates;$
```

5.4 BDD based Computation of Supremal Controllable Sublanguage

The algorithm given in Theorem 4.1.1 can be implemented using binary decision diagrams. First of all, we should construct the transition predicate for the plant and the specification, i.e. $T_{p,\sigma}(v_p, v'_p), T_{s,\sigma}(v_s, v'_s), \sigma \in \Sigma_{act} \cup \{tick\}$.

A *set* can also be represented by a predicate. For example the set $A := \{a, b, c\}$ can be represented by:

$$P_A := (v = a) \vee (v = b) \vee (v = c)$$

where v is a **BDD** variable.

The state set $Q_p \times Q_s$ can be described by the tuple (q_p, q_s) where $q_p \in Q_p$ and $q_s \in Q_s$. This set can be represented by the predicate S:

$$S(v_p, v_s) := (0 \leq v_p \leq n_p - 1) \wedge (0 \leq v_s \leq n_s - 1)$$

where $\{0, ..., n_p - 1\}$ are the states of G_p and $\{0, ..., n_s - 1\}$ are the states of G_s.

The set of states (q_p, q_s) where $(q_p, q_s) \vDash \mathcal{F}$ can be described by

$$S_{\mathcal{F}}(v_p, v_s) := S \wedge \neg(\exists(v_p, v_s)(S \wedge T_{s,\ f}))$$

where

$$T_{s,\ f} = \bigvee_{\sigma \in \Sigma_{for}} T_{s,\ \sigma}$$

and $(\exists(v_p, v_s))(S \wedge T_{s,\ f})$ [1] will be the set of states (q_p, q_s) where $\exists \sigma \in \Sigma_{for},\ \delta_s(q_s, \sigma)!$. Actually the existential quantifier quantifies out the variable v_s' which is present in $T_{s,\ f}$ [9] [19]. Now we will present the realization of each step of algorithm in Theorem 4.1.1.

Step 1: $P_{good1} := 1$.

[1]This can be done using the Buddy package function *bdd_exist*(Appendix A)

Step 2: $[Q_p \times Q_s]_{P_{bad1}} := \{(q_p, q_s) \mid$

$i.\ \ (q_p, q_s) \not\models P_{good1}$

or

$ii.\ \ (\exists \sigma \in \Sigma_u)(\delta_p(q_p, \sigma)! \wedge \neg\delta_s(q_s, \sigma)!)$

or

$iii.\ \ (q_p, q_s) \models \mathcal{F} \wedge \delta_p(q_p, tick)! \wedge \neg\delta_s(q_s, tick)!$

or

$iv.\ \ (\exists \sigma \in \Sigma_u)((\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \not\models P_{good1})$

or

$v.\ \ (q_p, q_s) \models \mathcal{F} \wedge (\delta_p(q_p, tick), \delta_s(q_s, tick)) \not\models P_{good1}\}$

In order to find the **BDD** realization of this predicate we have to define two functions to replace the source state variables with the target state variables and vice versa. We assume that F is the set of functions defined on source variables (i.e. v_i) and F' is the set of functions defined on target variables (i.e. v'_i). Now we define two functions:

- $RepSource : F' \rightarrow F$

  $RepSource(f(v'_1, ..., v'_n)) = f(v_1, ..., v_n)$

- $RepTarget : F \rightarrow F'$

  $RepTarget(f(v_1, ..., v_n)) = f(v'_1, ..., v'_n)$

We also define two transition predicates:

- $T_u(v_p, v'_p, v_s, v'_s) := \bigvee_{\sigma \in \Sigma_u}(T_{p,\sigma} \wedge T_{s,\sigma})$

- $T_{tick}(v_p, v'_p, v_s, v'_s) := T_{p,tick} \wedge T_{s,tick}$

Now we will find the **BDD** realization for this step:

i. $P_{bad1} := S \wedge \neg P_{good1}$

ii. For all $\sigma \in \Sigma_u : P_{bad1} := P_{bad1} \vee ((\exists v_p)T_{p,\sigma}(v_p, v'_p) \wedge (\exists v_s)\neg T_{s,\sigma}(v_s, v'_s))$

iii. $P_{bad1} := P_{bad1} \vee ((\exists v_p)T_{p,\ tick}(v_p, v'_p) \wedge (\exists v_s)\neg T_{s,\ tick}(v_s, v'_s))$

iv. $X \triangleq (\exists(v'_p, v'_s)S(v_p, v_s) \wedge T_u(v_p, v'_p, v_s, v'_s)\ )$ is the set of states that are targeted by an uncontrollable event. So

$$P_{bad1} := P_{bad1} \vee \{\exists(v_p, v_s)(\ T_u \wedge RepTarget(\neg P_{good1} \wedge RepSource(X))\ )\}$$

v. $Y \triangleq (\exists(v'_p, v'_s)S_{\mathcal{F}}(v_p, v_s) \wedge T_{tick}(v_p, v'_p, v_s, v'_s)\ )$ is the set of states that are targeted by the *tick* event. So

$$P_{bad1} := P_{bad1} \vee \{\exists(v_p, v_s)(\ T_{tick} \wedge RepTarget(\neg P_{good1} \wedge RepSource(Y))\ )\}$$

Step 3: $[Q_p \times Q_s]_{P_{bad2}} := \{(q_p, q_s) \mid (\exists w \in \Sigma_u^*)(\ (\delta_p(q_p, w), \delta_s(q_s, w)) \vDash P_{bad1}\ )$

or

$((q_p, q_s) \vDash \mathcal{F}) \wedge (\exists \sigma \in \Sigma_u \cup \{tick\})((\delta_p(q_p, \sigma), \delta_s(q_s, \sigma)) \vDash P_{bad1})\}$

This step needs more consideration. We define the operation **R** as follows. Let $P_1 = \mathbf{R}(P_0, P, \Sigma)$ be the reachable predicate starting from P_0, while enabling all $\sigma \in \Sigma$ and satisfying predicate P, i.e.

$$x \vDash P_1 \Leftrightarrow (\exists x' \vDash P_0\ , s \in \Sigma^*)\big((\delta(x', s) = x) \wedge (\forall v \leq s)(\delta(x', v) \vDash P)\big)$$

Similarly, we define operation **CR** to produce the coreachable predicate. Let $P_2 = \mathbf{CR}(P_m, P, \Sigma)$ be the coreachable predicate starting from P, while enabling all $\sigma \in \Sigma$ and satisfying predicate P reaches the set of states satisfying P_m, i.e.

$$x \vDash P_2 \Leftrightarrow (\exists x' \vDash P_m\ , s \in \Sigma^*)\big((\delta(x, s) = x') \wedge (\forall v \leq s)(\delta(x, v) \vDash P)\big)$$

For simplicity, the recent definitions use notations for simple predicates, i.e. predicates defined on a simple set X. All the results can be easily generalized to our states of the form $(x, y) \in X \times Y$.

Now we should find the **BDD** realization of the reachable and coreachable predicates. In these functions instead of using Σ we use :

$$T_\Sigma := \bigvee_{\sigma \in \Sigma} (T_{p,\sigma} \wedge T_{s,\sigma})$$

Thus the function $\mathbf{R}(P_0, P, \Sigma)$ can be obtained by the following algorithm:

1. $P_1 := P_0 \wedge P$
2. $P_{i+1} := P_i \vee \big(P \wedge \mathit{RepSource}(\exists(v'_p, v'_s)(T_\Sigma \wedge P_i)\ )\big)$
3. If $P_{i+1} \equiv P_i$ then $\mathbf{R}(P_0, P, T_\Sigma) := P_i$. Otherwise go back to step 2.

The coreachable function $\mathbf{CR}(P_m, P, \Sigma)$ can also be computed similarly:

1. $P_1 := P_m \wedge P$
2. $P_{i+1} := P_i \vee \big(P \wedge (\exists(v_p, v_s)(T_\Sigma \wedge \mathit{RepTarget}(P_i)\ )\big)$
3. If $P_{i+1} \equiv P_i$ then $\mathbf{CR}(P_m, P, T_\Sigma) := P_i$. Otherwise go back to step 2.

After defining these functions step 3 can easily be described by two terms:

$$P_{bad2} := \mathbf{CR}(P_{bad1}, S, \Sigma_u) \vee \mathbf{CR}(P_{bad1}, S_{\mathcal{F}}, \Sigma_u \cup \{tick\})$$

Step 4: $[Q_p \times Q_s]_{Pre} := \{(q_p, q_s) \mid (\exists w \in \Sigma^*)(\delta_p(q_{p,0}, w) = q_p, \delta_s(q_{s,0}, w) = q_s) \wedge$
$(\forall v \leq w)(\delta_p(q_{p,0}, v), \delta_s(q_{s,0}, v)) \nvDash P_{bad1} \vee P_{bad2}\}$

This step could also be realized by **BDD**s using the above defined functions. First we define the predicate P_0 for presenting the initial state:

$$P_0 \equiv (v_p = q_{p,0}) \wedge (v_s = q_{s,0})$$

Now P_{re} can be computed as follows:

$$P_{re} := \mathbf{R}(P_0, \neg(P_{bad1} \vee P_{bad2}), \Sigma_{act} \cup \{tick\})$$

Step 5: $[Q_p \times Q_s]_{P_{cr}} := \{(q_p, q_s) \mid (\exists w \in \Sigma^*)(\delta_p(q_p, w) \in Q_{p,m}, \delta_s(q_s, w) \in Q_{s,m}) \wedge$

$$(\forall v \leq w)(\delta_p(q_p, v), \delta_s(q_s, v)) \not\models P_{bad1} \vee P_{bad2}\}$$

By defining the P_m for representing the marker states,

$$P_m \equiv (\bigvee_{q \in Q_{p,m}} (v_p = q) \ ) \wedge (\bigvee_{q \in Q_{s,m}} (v_s = q) \ )$$

P_{cr} will be realized using coreachable function

$$P_{cr} := \mathbf{CR}(P_m, \neg(P_{bad1} \vee P_{bad2}), \Sigma_{act} \cup \{tick\})$$

Steps 6 and 7 can easily be implemented using logical *and* and comparison. So we have done all the steps in finding the supremal controllable and nonblocking supervisor by **BDD**s and operations on them.

5.4.1 Optimization in the algorithm

In the previous section, we could compute the supremal controllable sublanguage of a TDES using **BDD**s. But the problem with this implementation is that there is no *structure* in our **BDD**s. We have only four variables: two for the plant (current and next state) and two for the specification (current and next state); and there is no coupling between them. As mentioned in Section 5.3, the more closely coupled two variables are, the "closer" the two components should be placed in the **BDD**. Each state of a TDES consists of an activity and some timers, so we can employ this structure for our **BDD**s. The order of variables would be the same as what we used in Section 5.3.

Using this method may give rise to the problem that we usually want to place some time constraints on the system which we directly represent in the form of a TTG, so our specification will not have timer information. It has been shown by Wong and Wonham in [28] that the BW framework is not closed under control. For example if we look at the TTG in Figure 5.4, there does not exist an ATG such that the corresponding TTG generates the TTG in Figure 5.4 because the time bound of the first α is $(\alpha, 0, 0)$ while the time bound of the second α is $(\alpha, 0, 1)$.

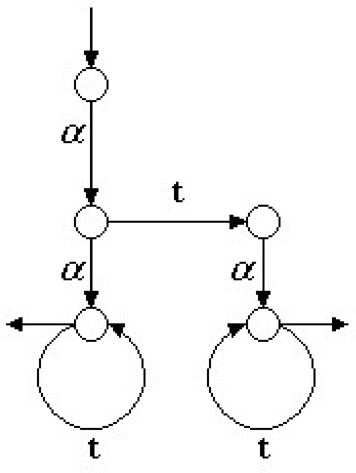


Figure 5.4: BW framework is not closed

So in the cases where our specification is implemented by a TTG, we use the unstructured variables for specification whereas the variables for the plant are structured as before. In the next chapter we will see that using this structure will speed up our algorithm.

5.4.2 Implementation of Controller by BDDs

The means of controlling the plant in TDES are disablement of controllable events and forcing of forcible events (in case of disablement of event *tick*). As shown in section 4.2, the controller can be implemented using $n + 1$ predicates where n is the number of events except *tick*. Therefore, by replacing those predicates by corresponding **BDD**s, an implementation of the controller by **BDD**s can be easily obtained. A diagram of the controlled system is shown in Figure 5.5. The enabled event set in the states where *tick* is disabled shows the forcing of events which are forcible.

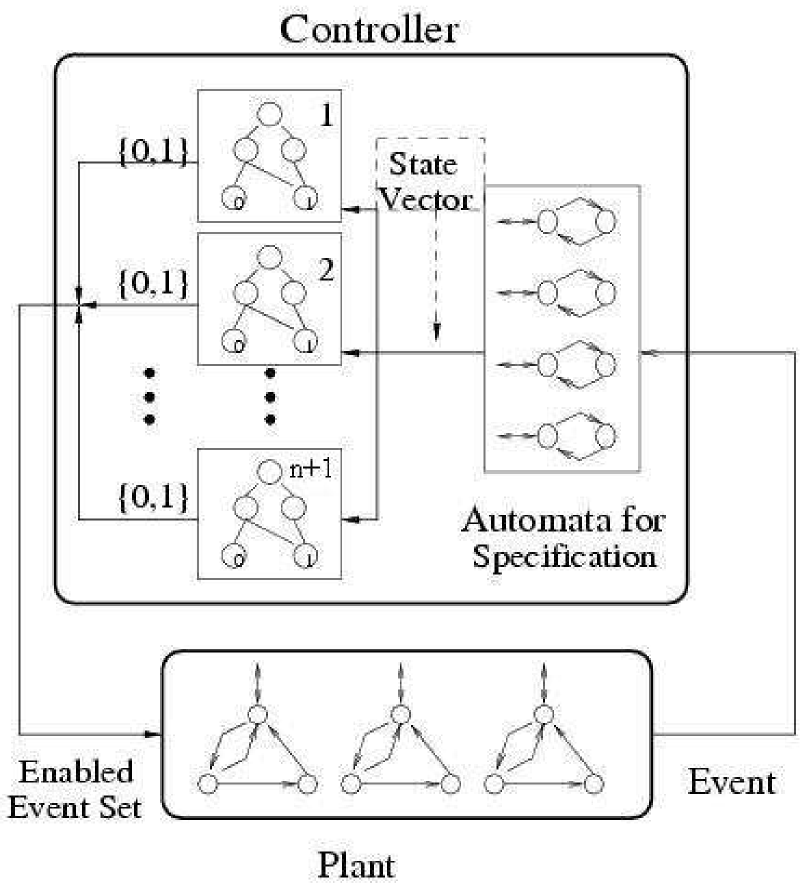


Figure 5.5: A sketch of a system with BDD based controller

The size of an **BDD**-based controller is much smaller than an original control data table generated by TTCT. A comparison between sizes is shown in Chapter 6.

5.5 Modular Supervision of TDES

Let

$$\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m), \quad \mathbf{S} = (X, \Sigma, \xi, x_0, X_m)$$

be TDES, with $\Sigma = \Sigma_{act} \cup \{tick\}$. We assume that **S** can be used as a supervisor for **G**. We write $\mathbf{S} \wedge \mathbf{G}$ for the conjunction of **S** and **G**:

$$L_m(\mathbf{G} \wedge \mathbf{S}) = L_m(\mathbf{G}) \cap L_m(\mathbf{S}), \qquad L(\mathbf{G} \wedge \mathbf{S}) = L(\mathbf{G}) \cap L(\mathbf{S})$$

We say that **S** is a proper *supervisor* for **G** if

(i) **S** is trim (reachable and coreachable).

(ii) **S** is controllable with respect to **G**.

(iii) $\mathbf{S} \wedge \mathbf{G}$ is nonblocking.

Since by (iii), $\overline{L_m(\mathbf{S} \wedge \mathbf{G})} = L(\mathbf{S} \wedge \mathbf{G})$, (ii) means that

$$Elig_{L(\mathbf{S}\wedge\mathbf{G})}(s) \supseteq \begin{cases} Elig_{\mathbf{G}}(s) \cap (\Sigma_u \cup \{tick\}), & \text{if } Elig_{L(\mathbf{S}\wedge\mathbf{G})}(s) \cap \Sigma_{for} = \emptyset \\ Elig_{\mathbf{G}}(s) \cap \Sigma_u, & \text{if } Elig_{L(\mathbf{S}\wedge\mathbf{G})}(s) \cap \Sigma_{for} \neq \emptyset. \end{cases}$$

The following definition extracts the feature of controllability that expresses the preemption of *tick* by a forcible event.

Definition 5.5.1 *Coerciveness*

Let $K \subseteq L(\mathbf{G})$. We say that K is *coercive* with respect to **G** if

$$(\forall s \in \overline{K})\ tick \in Elig_{\mathbf{G}}(s) - Elig_K(s) \Rightarrow Elig_K(s) \cap \Sigma_{for} \neq \emptyset$$

i.e.

$$(\forall s \in \overline{K})\ Elig_K(s) \cap \Sigma_{for} = \emptyset \ \&\ \ tick \in Elig_{\mathbf{G}}(s) \Rightarrow tick \in Elig_K(s)$$

We say that languages $K_1, K_2 \subseteq L(\mathbf{G})$ are *jointly coercive* with respect to **G** if $K_1 \cap K_2$ is coercive with respect to **G**. Now let $\mathbf{S1}, \mathbf{S2}$ be proper supervisors for **G**.

Theorem 5.5.1 *[29]* $\mathbf{S1} \wedge \mathbf{S2}$ *is a proper supervisor for* **G** *if*

(i) $\mathbf{S1} \wedge \mathbf{S2}$ is trim.

(ii) $L_m(\mathbf{S1} \wedge \mathbf{G}), L_m(\mathbf{S2} \wedge \mathbf{G})$ are nonconflicting.

(iii) $L_m(\mathbf{S1} \wedge \mathbf{G}), L_m(\mathbf{S2} \wedge \mathbf{G})$ are jointly coercive with respect to $\mathbf{G}$.

In order to extend our symbolic computations to modular supervision, we need to be able to check the conditions of Theorem 5.5.1 symbolically.

First of all we should implement $\wedge$ operator symbolically. (This function is called **meet** in **TTCT**.) It is straightforward to express this function using the transition **BDD**s of the two TDES. First we find the transition **BDD** of each TDES, i.e. $T_{\mathbf{S1}}, T_{\mathbf{S2}}$; then the transition **BDD** of $\mathbf{S1} \wedge \mathbf{S2}$ would be:

$$T_{\mathbf{S1} \wedge \mathbf{S2}} = \bigvee_{\sigma \in \Sigma_{\mathbf{S1}} \cap \Sigma_{\mathbf{S2}}} T_{\mathbf{S1},\ \sigma} \wedge T_{\mathbf{S2},\ \sigma}$$

To check the condition (i), we should check whether $\mathbf{S1} \wedge \mathbf{S2}$ is reachable and coreachable. By definition of $\wedge$, we know that $\mathbf{S1} \wedge \mathbf{S2}$ is reachable and we can check the coreachability using the function **CR** defined in section 5.4.

Condition (ii) is equal to the nonblocking property of $L_m(\mathbf{S1} \wedge \mathbf{G}) \cap L_m(\mathbf{S2} \wedge \mathbf{G})$ which can be checked using the reachable (**R**) and coreachable (**CR**) functions defined in previous section, because a DES is nonblocking if it is trim.

For condition (iii), we need to be able to check coerciveness. Let $S_{\mathcal{F}}(v_{\mathbf{G}}, v_K)$ be the predicate representing the set of states of $Q_{\mathbf{G}} \times Q_K$ at which no forcible event is defined in $q_k \in Q_K$ where $(q_{\mathbf{G}}, q_K) \in Q_{\mathbf{G}} \times Q_K$ and $T_{\mathbf{G},tick}(v_{\mathbf{G}}, v'_{\mathbf{G}}), T_{K,tick}(v_K, v'_K)$ are the transition **BDD**s of $\mathbf{G}$ and K for event *tick*. Now we define two sets of states:

$$F_1 = (\exists (v_{\mathbf{G}}, v_K)\ )\ S_{\mathcal{F}}(v_{\mathbf{G}}, v_K) \wedge T_{\mathbf{G},tick}(v_{\mathbf{G}}, v'_{\mathbf{G}})$$

i.e. the set of states in $S_{\mathcal{F}}$ where *tick* is defined in $\mathbf{G}$, and

$$F_2 = (\exists (v_{\mathbf{G}}, v_K)\ )\ S_{\mathcal{F}}(v_{\mathbf{G}}, v_K) \wedge T_{K,tick}(v_K, v'_K)$$

i.e. the set of states in $S_{\mathcal{F}}$ where *tick* is defined in K.

Because $K \subseteq L(\mathbf{G})$, to check coerciveness it is enough to check whether F_1 and F_2 are equal or not. If F_1 and F_2 are equal, then in all the states where no forcible event is defined in K if *tick* is defined in $\mathbf{G}$, it will be defined in K too.

So for condition (iii) we should check if $L_m(\mathbf{S1} \wedge \mathbf{G}) \cap L_m(\mathbf{S2} \wedge \mathbf{G})$ is coercive with respect to $\mathbf{G}$.

Hence we can represent the modular supervision of TDES with predicates.

Chapter 6

Examples

In this chapter, we present some examples to compare our results with previous ones.

6.1 Tutorial Example

The following example shows the transition **BDD**s for all the events of a small TDES. Let

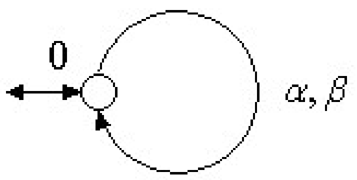
$$\mathbf{G_{act}} = (A, \Sigma_{act}, \delta_{act}, a_0, A_m)$$

with

$$\Sigma_{act} = \{\alpha, \beta\}, \quad A = A_m = \{0\}, \quad a_0 = 0$$

$$\delta_{act}(0, \alpha) = \delta_{act}(0, \beta) = 0$$

and timed events $(\alpha, 1, 1), (\beta, 2, 3)$, both prospective. The ATG for $\mathbf{G_{act}}$ is simply:



Thus α, β are always enabled. The state set for $\mathbf{G}$ is

$$Q = \{0\} \times T_\alpha \times T_\beta = \{0\} \times [0,1] \times [0,3]$$

and has size $|Q| = 8$. We take $Q_m = \{(0,[1,3])\}$. The TTG for $\mathbf{G}$ is easily constructed based on the rules given in chapter 2 and is displayed in Figure 6.1; it has 11 transitions, over the event set $\{\alpha, \beta, tick\}$; the pairs $[t_\alpha, t_\beta]$ corresponding to the states $(0, \{t_\alpha, t_\beta\})$ of $\mathbf{G}$ are listed below. The event α is pending at states 0,2,5,7 and eligible at states 1,3,4,6, while β is pending at 0,1,2,4 and eligible at 3,5,6,7. Notice that *tick* is preempted by α or β if either of these events has deadline 0.

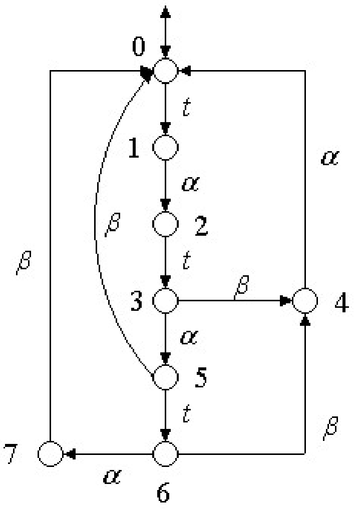


Figure 6.1: Timed Transition Graph

The states of $\mathbf{G}$(nodes of TTG) are as follows $(act, \{t_\alpha, t_\beta\})$:

0 : (0,$\{1,3\}$)

1 : (0,$\{0,2\}$)

2 : (0,$\{1,2\}$)

3 : (0,$\{0,1\}$)

4 : (0,$\{0,3\}$)

5 : (0,$\{1,1\}$)

6 : (0,$\{0, 0\}$)

7 : (0,$\{1, 0\}$)

In the naive method the transitions of the system are stored in a table. Each transition needs 3 cells of the table, 1 for the current state, 1 for the event and 1 for the next state.

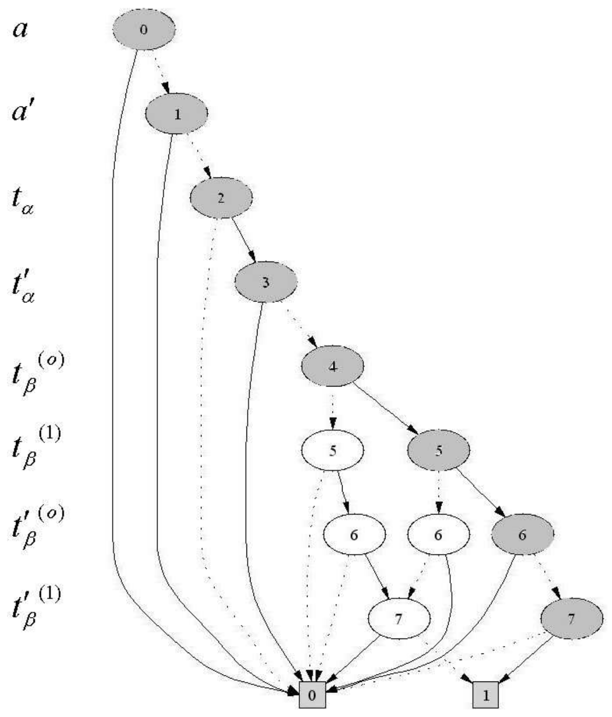


Figure 6.2: Transition BDD for event *tick*

As mentioned in the previous chapter, for the transition **BDD** of TTG in this example we have 6 main variables: $a, a', t_\alpha, t'_\alpha, t_\beta, t'_\beta$ that $a, a' \in \{0\}$,$t_\alpha, t'_\alpha \in \{0, 1\}$ and $t_\beta, t'_\beta \in \{0, 1, 2, 3\}$. So with the exception of t_β, t'_β that need 2 **BDD** variables, all the other variables need 1 **BDD** variable, because t_β, t'_β can have four different values which need 2 binary variables for representation. The less significant a bit is in a timer value, the closer to the root it is in the **BDD** tree. The transition **BDD**s of each event of the TDES are presented in Figures 6.2,6.3 and 6.4. The dotted lines in these figures represent the value "0" for the variables and solid lines represent the value "1". $t_\beta^{(0)}, t_\beta^{'(0)}$ are the least significant bits of t_β, t'_β while $t_\beta^{(1)}, t_\beta^{'(1)}$ are the most significant. Each route that leads us from root to terminal "1" is a representation of a transition in TDES.

For example in Figure 6.2 the route which is determined by grey nodes represents the existence of an event *tick* between states 0 and 1 of the TTG in Figure 6.1 because in this route we have: source state: $(a = 0, t_\alpha = 1, t_\beta = (t_\beta^{(1)} t_\beta^{(0)})_2 = (11)_2 = 3)$ and target state: $(a' = 0, t'_\alpha = 0, t'_\beta = (t_\beta^{'(1)} t_\beta^{'(0)}) = (10)_2 = 2)$.
Similarly, the transition $1 \xrightarrow{\alpha} 2$ can be traced on the **BDD** in Figure 6.3 by this route: $1 \rightarrow (a = 0, t_\alpha = 0, t_\beta = (t_\beta^{(1)} t_\beta^{(0)})_2 = (10)_2 = 2), 2 \rightarrow (a' = 0, t'_\alpha = 1, t'_\beta = (t_\beta^{'(1)} t_\beta^{'(0)}) = (10)_2 = 2)$.

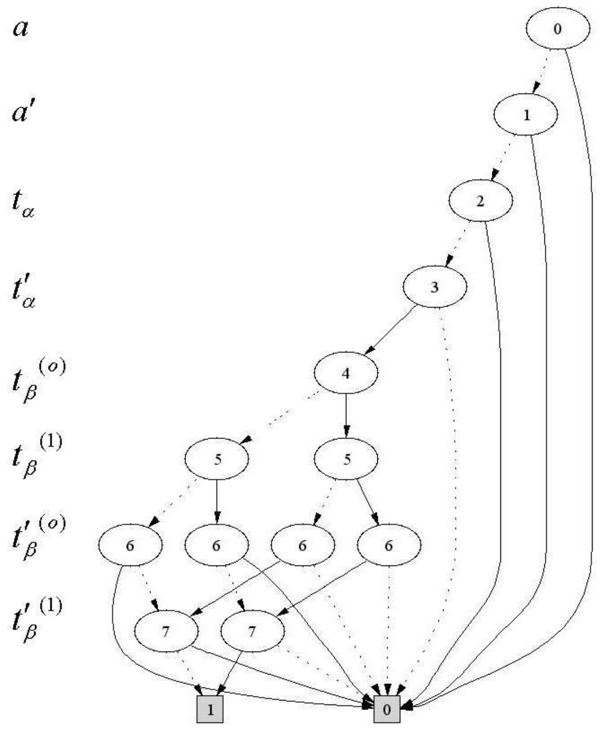


Figure 6.3: Transition BDD for α

We can see that a reduction has been made in the transition **BDD** for β in Figure 6.4 as there is no node for variable 4 $(t_\beta^{(0)})$ in this **BDD**. In this **BDD**, one route represents two transitions in TTG graph. For example the route : $(a = 0, a' = 0, t_\alpha = 0, t'_\alpha = 0, t_\beta^{(1)} = 0, t_\beta^{'(0)} = 1, t_\beta^{'(1)} = 1)$ represents these two transitions : $3 \xrightarrow{\beta} 4$, $6 \xrightarrow{\beta} 4$ because

$t_\beta^{(0)}$ can have both values “0” and “1” .

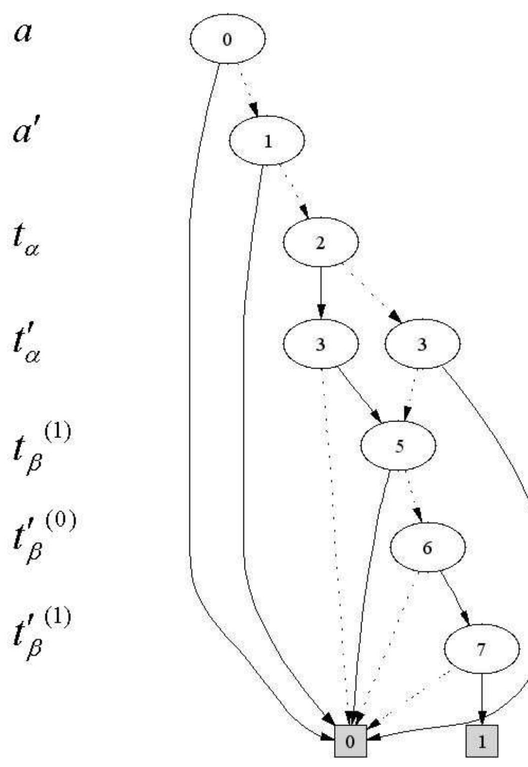


Figure 6.4: Transition BDD for β

6.2 Manufacturing Cell

The manufacturing cell of Figure 6.5 consists of machines **Mach1**,**Mach2**, with an input conveyor **Conv1** as an infinite source of workpieces and output conveyor **Conv2** as an infinite sink. Each machine may process two types of parts, P1 and P2; and for simplicity, the transfer of parts between machines will be absorbed as a step in machine operation. The machine ATGs (identical up to event labelling) are displayed in Figure 6.6. Here α_{ij} is the event " **Mach***i starts work on a Pj part*", while β_{ij} is "**Mach***i finishes working on a Pj part*".

$$\Sigma_{for} = \{\alpha_{ij} \mid i,j = 1,2\}, \quad \Sigma_u = \{\beta_{ij} \mid i,j = 1,2\}, \quad \Sigma_{hib} = \Sigma_{for}$$

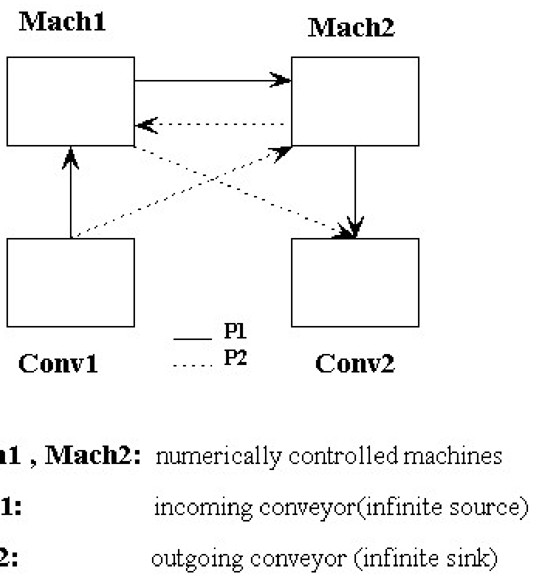

Mach1
Mach2
Conv1
Conv2
P1
P2
Mach1 , Mach2: numerically controlled machines
Conv1: incoming conveyor(infinite source)
Conv2: outgoing conveyor (infinite sink)

Figure 6.5: The manufacturing cell

6.2.1 Converting ATG to TTG

The time bounds of events of the machines can vary according to the machine specifications. So they can be small or large. For small values of time bounds, there is no problem using the previous methods but if the time bounds increase, we can not use

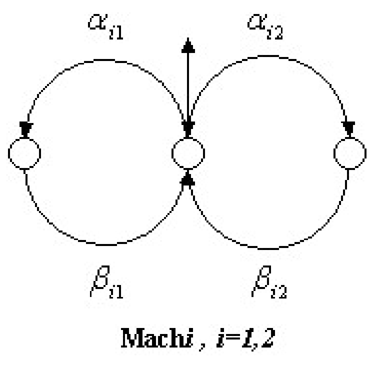


Figure 6.6: The ATG of the machines

these methods. For example **TTCT** can only compute the timed graph for ATGs with time bounds not greater than 1000. Here we present a comparison between our results and the **TTCT** results. We specify the timing information for the machines as follows:

Mach1: $(\alpha_{11}, L, \infty), (\beta_{11}, 3, U), (\alpha_{12}, L, \infty), (\beta_{12}, 2, U)$

Mach2: $(\alpha_{21}, L, \infty), (\beta_{21}, 1, U), (\alpha_{22}, L, \infty), (\beta_{22}, 4, U)$

We define the cell's open-loop behavior as the composition **Mach** of **Mach1** and **Mach2**:

Mach=comp(Mach1,Mach2)

Then we find the TTG of **Mach** using the TTCT procedure **TimedGraph** and our **BDD** program. The comparison between the computation time and space can easily be done by reference to Table 6.1.

L	$\lvert Q\rvert$	$\lvert T\rvert$	TTCT	BDD Program		
			t_1	t_2	t_3	$\lvert nodes\rvert$
5	324	580	0	0	0	1460
50	23,409	53,095	9	0	1	10,006
100	91,809	211,195	258	0	4	19,081
150	205,209	474,295	1148	1	12	31,556
200	363,609	724,808	2516	1	22	37,010
250	567,009	1,131,008	6100	1	34	42,250
300	815,409	1,893,600	N/A	1	59	61,795
500	2,259,010	5,256,000	N/A	1	199	83,287
700	4,422,610	10,298,400	N/A	1	321	132,751
1000	9,018,010	21,012,000	N/A	1	726	165,148
1500	20,277,009	47,267,995	N/A	2	1837	274,609
2000	36,036,000	84,024,000	N/A	3	3458	328,657
3000	81,054,000	189,036,000	N/A	5	9387	547,414

Table 6.1: TTCT and BDD program results for manufacturing cell

In this table, for computation purposes we consider that L and U in the time bounds of the events are the same and given by the first column of the table. $|Q|$ is the number of states of the TDES, $|T|$ is the number of Transitions, t_1 is the computation time of TTCT in seconds , t_2 is the time our program needs to compute the transition **BDD** in seconds(T in Equation 5.1), t_3 is the time for finding the reachable states in our program in seconds, and $|nodes|$ is the number of nodes of the transition **BDD**.
The space required for storing a TDES is proportional to the number of states and transitions but the space our program needs is proportional to the number of **BDD**

nodes. The space requirements of TTCT and the **BDD** program for this example are compared in Figure 6.7. As mentioned before TTCT can not compute the TTG for the time bounds greater than 250, so in Figure 6.7 the values of space for time bounds greater than 250 represent the space required if it could compute the TTG (dotted line). It is obvious that we have achieved significant savings in space. Moreover, TTCT can not compute the timed graph for time bounds greater than or equal to 250 in this example.

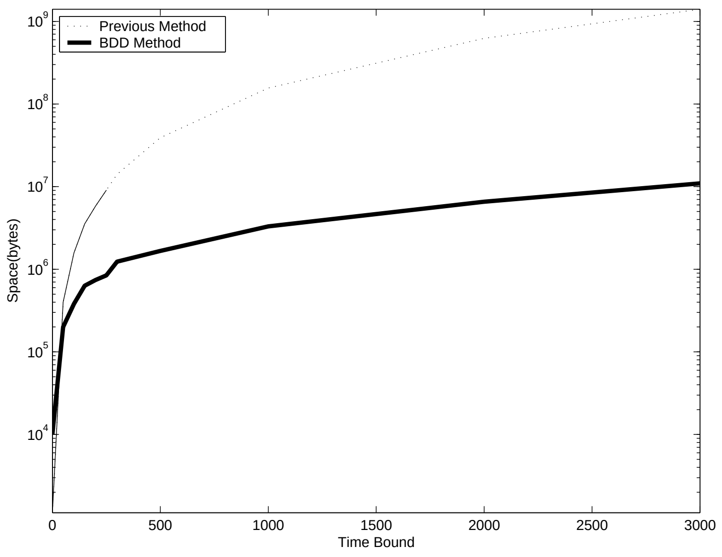


Figure 6.7: Comparison between the space required in both methods.

As shown in Figure 6.8, we find a roughly linear relationship between $|nodes|$ and the time bound L. Also, in Figure 6.9, we can see that $|nodes|$ increases much slower than $|Q|$ which satisfies $|Q| \propto L^n$ where n is the number of events.

The relationship between L and computing time is a bit more complex. For TTCT the computing time is shown in Figure 6.10. In the **BDD** Program the important time for us which will be used in the second part of the program for finding the controller would be only t_2, which does not vary too much for different values of L. As we can see from Table 6.1, t_2 ranges between 0-5 for L in 0...3000. So a big time saving of this method would be in this part, i.e. converting ATG to TTG is much faster in comparison to the previous method.

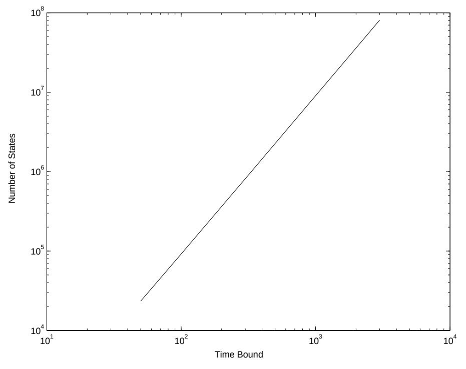


Figure 6.8: Number of states of the TDES vs time bound.(Both axes are logarithmic)

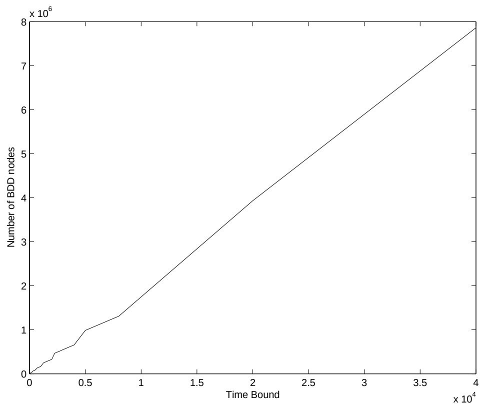


Figure 6.9: Number of the nodes of the transition BDD vs time bound.

6.2.2 Finding a Controller for Manufacturing Cell

We impose some *Logic-based* specifications on the manufacturing cell:

- A given part can be processed by just one machine at a time.
- A P1 part must be processed first by **Mach1** and then by **Mach2**
- A P2 part must be processed first by **Mach2** and then by **Mach1**
- One P1 part and one P2 part must be processed in each production cycle

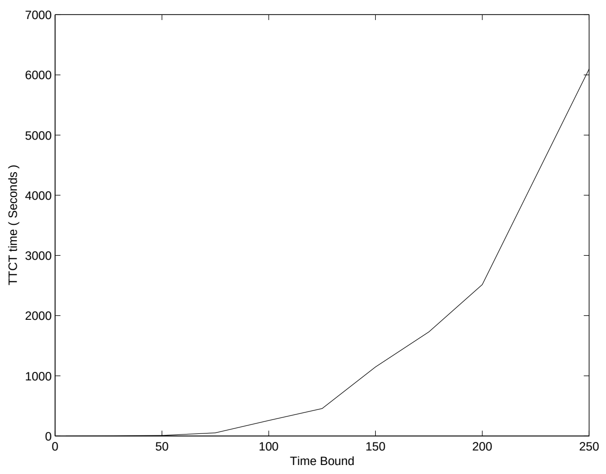


Figure 6.10: Computing time vs time bound for TTCT.

These specifications are formalized as TDES **Spec1**,**Spec2** and **Spec3** displayed in Figures 6.11,6.12 and 6.13.

The complete logic-based specification can be represented by the intersection of these three TDES:

$$\textbf{Spec} = \textbf{meet} \ (\ \textbf{Spec1}, \ \textbf{Spec2}, \ \textbf{Spec3} \ )$$

Spec has 16 states and 72 transitions.

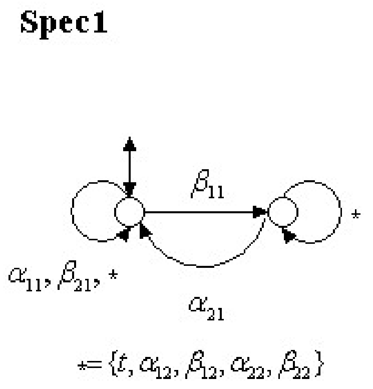

Spec1
β_{11}
$*$
$\alpha_{11}, \beta_{21}, *$
α_{21}
$* = \{t, \alpha_{12}, \beta_{12}, \alpha_{22}, \beta_{22}\}$

Figure 6.11: Spec1 TTG

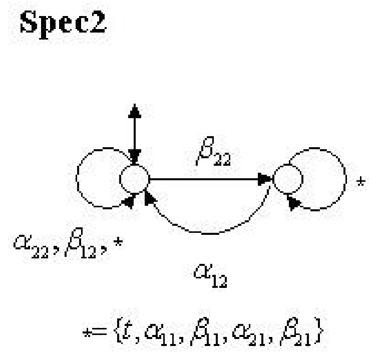

Spec2
β_{22}
*
$\alpha_{22}, \beta_{12}, *$
α_{12}
$* = \{t, \alpha_{11}, \beta_{11}, \alpha_{21}, \beta_{21}\}$

Figure 6.12: Spec2 TTG

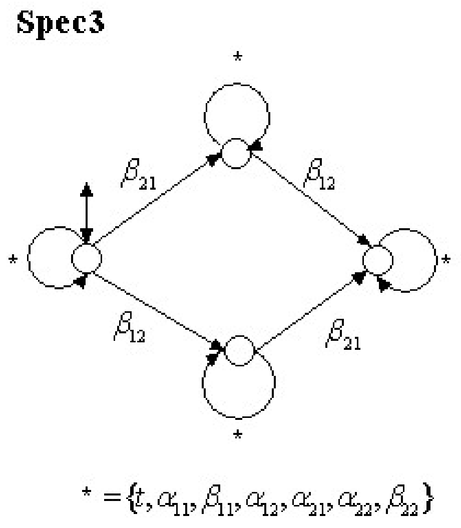

Spec3
*
β_{21}
β_{12}
*
*
β_{12}
β_{21}
*
$* = \{t, \alpha_{11}, \beta_{11}, \alpha_{12}, \alpha_{21}, \alpha_{22}, \beta_{22}\}$

Figure 6.13: Spec3 TTG

Here and below we write **G3 = supcon (G1,G2)** to denote the operation that returns a TDES **G3** whose marked behavior $L_m(\mathbf{G3})$ is the supremal controllable sublanguage $sup\mathcal{C}(L_m(\mathbf{G1}), L_m(\mathbf{G2}))$, while its closed behavior $L(\mathbf{G3}) = \bar{L}_m(\mathbf{G3})$.
The maximally permissive proper supervisor for **Mach** that enforces **Spec** can now be computed as:

Super = supcon(Mach,Spec)

Our new algorithm for finding supremal controllable sublanguage which was described in the previous chapter, is compared with the previous method, used in TTCT . Table

6.2 compares the efficiency and performance of the new method with the previous one. As described in the previous section, for different values of time bounds the TDES will have different number of states. So our open loop manufacturing cell will have a large number of states for large values of the time bounds and the previous method can not compute the controlled behavior of this large system. In this table, for computation purposes we consider that L and U in the time bounds of the events are the same and equal to the first column of the table. $|Q|_{OpenLoop}$ is the number of states of the open loop cell, $|Q|_{ClosedLoop}$ is the number of states of the controlled system, t_1 is the computation time of TTCT in seconds , t_2 is the time our program needs to compute the supremal controllable sublanguage **BDD** in seconds, and $|nodes|$ is the number of nodes of the **BDD** containing the states of the controlled system.

L	$\lvert Q\rvert_{OpenLoop}$	$\lvert Q\rvert_{ClosedLoop}$	TTCT	BDD Program	
			t_1	t_2	$\lvert nodes\rvert$
5	324	470	0	0	642
50	23,409	31,475	7	7	7775
100	91,809	122,925	92	37	17,225
200	363,609	485,825	1337	221	38,062
250	567,009	757,275	N/A	561	46,910
500	2,259,010	3,014,530	N/A	6684	103,396
750	5,076,009	6,771,780	N/A	42929	171,925

Table 6.2: TTCT and BDD program results for finding the controller for manufacturing cell

A better comparison in time and space can be made using the Figures 6.14 and 6.15. It is obvious that our new method saves both time and space in comparison to TTCT although its saving in this part of the algorithm (SUPCON) is not as significant as the

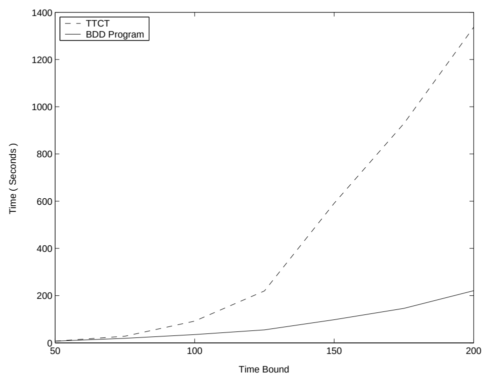


Figure 6.14: SUPCON computation time comparison between TTCT and BDD program

saving in the previous part of the algorithm (ATG $\longrightarrow$ TTG). We should mention that in our new algorithm we find the supremal controllable sublanguage directly from the activity transition graph of the plant but TTCT computes it from the timed graph of the plant. So for the total computation time of TTCT we should add the time it needs to find the TTG from the ATG of the plant which was discussed in the previous section.

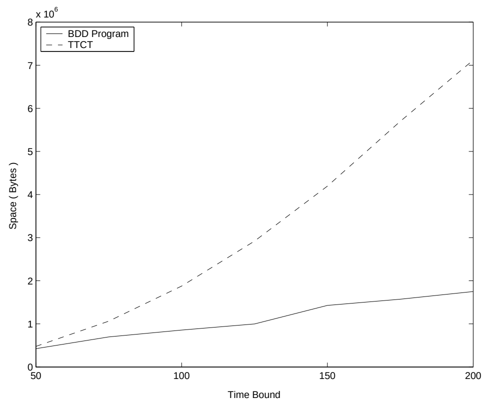


Figure 6.15: SUPCON computation space comparison between TTCT and BDD program

6.3 Two-Machine Workcell

Let us consider the workcell shown in Figure 6.16. Two machines $\mathbf{M_1}$ and $\mathbf{M_2}$ and a buffer of size one constitute the cell. The machines are represented in the form of activity transition diagrams in Figure 6.17. Furthermore, $\Sigma_u = \{\beta_1, \lambda_1, \eta_1, \beta_2, \lambda_2, \eta_2\}$, $\Sigma_{hib} = \Sigma_{for} = \{\alpha_1, \alpha_2, \mu_1, \mu_2\}$. The following timing information applies:

$$\mathbf{M_1} : (\alpha_1, L, \infty), (\beta_1, 1, U), (\lambda_1, 0, U), (\mu_1, L, \infty), (\eta_1, L, \infty)$$

$$\mathbf{M_2} : (\alpha_2, L, \infty), (\beta_2, 1, U), (\lambda_2, 0, U), (\mu_2, L, \infty), (\eta_2, L, \infty)$$

A workpiece produced by $\mathbf{M_1}$ is placed in the buffer and is consequently available for further work by $\mathbf{M_2}$. Both machines may either be idle, working or down. Once a work cycle has begun, the machines either finish working or break down, in which case they are repaired. The following production specifications are considered:

- the buffer must not overflow or underflow,
- if both machines are broken down, the repair of $\mathbf{M_2}$ must be initialized before the repair of $\mathbf{M_1}$.

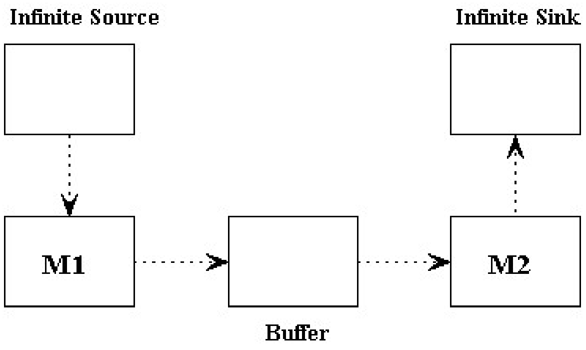


Figure 6.16: Two-machine workcell setup

First, the cell open loop behavior **M** is obtained by composing $\mathbf{M_1}$ and $\mathbf{M_2}$. After that we should find the timed transition graph of **M**. Similar to the previous example, for different time bounds of events we will have a different number of states in the timed graph of **M**. The previous method (TTCT) can not compute the TTG for time bound values (L, U) of 300 or more. Our method can compute the TTG for values of time bound much greater than that.

The above mentioned specifications are translated into the form of automata R_1 and R_2 which enforce the buffer and breakdown specifications respectively. R_1 and R_2 are shown in Figure 6.18. Accordingly, the corresponding specification languages are provided by $E_1 = L_m(R_1)$ and $E_2 = L_m(R_2)$ respectively.

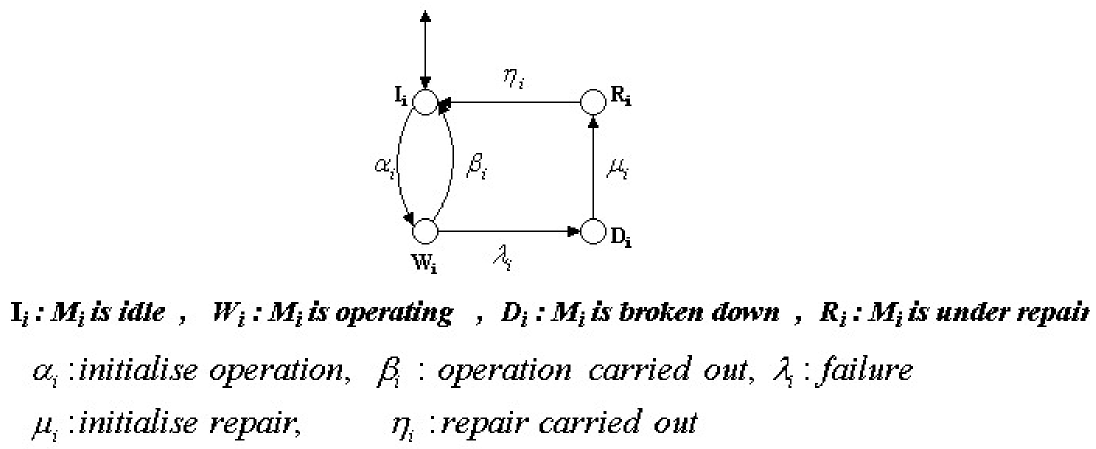

η_i
$\mathbf{I_i}$
$\mathbf{R_i}$
α_i
β_i
μ_i
$\mathbf{W_i}$
λ_i
$\mathbf{D_i}$
I_i : M_i is idle , W_i : M_i is operating , D_i : M_i is broken down , R_i : M_i is under repair
α_i :initialise operation, β_i : operation carried out, λ_i :failure
μ_i :initialise repair, η_i :repair carried out

Figure 6.17: The activity transition graph for $M_i, i = 1, 1$

R_1 and R_2 are combined together through the **meet** operation $\wedge$, i.e. $R = R_1 \wedge R_2$. Accordingly we have $E = E_1 \cap E_2 = L_m(R)$.

The cell closed loop behavior, which meets the specifications in the freest possible way, is given by $sup\mathcal{C}(E \cap L(M)))$, i.e. the supremal controllable sublanguage of the cell with respect to the combined specifications. The TDES whose marked behavior is $sup\mathcal{C}(E \cap L(M)))$ is computed by **SUPER = supcon (M,R)** .

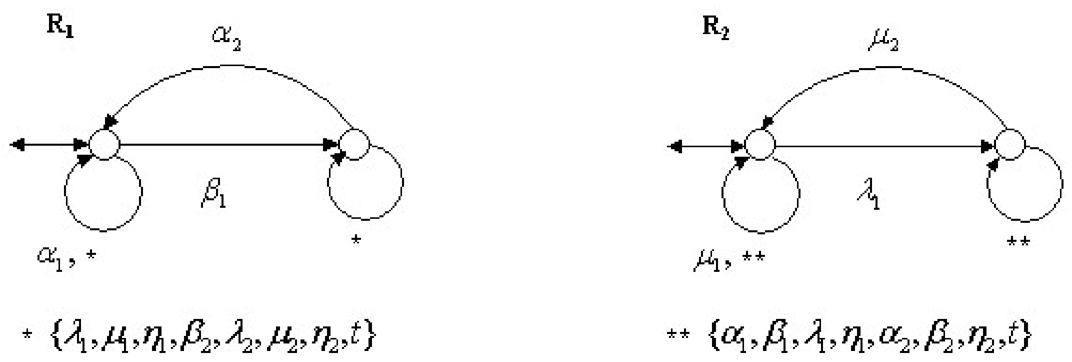

R_1
α_2
β_1
α_1, *
*
* $\{\lambda_1, \mu_1, \eta_1, \beta_2, \lambda_2, \mu_2, \eta_2, t\}$
R_2
μ_2
λ_1
μ_1, **
**
** $\{\alpha_1, \beta_1, \lambda_1, \eta_1, \alpha_2, \beta_2, \eta_2, t\}$

Figure 6.18: R_1 and R_2

The results of our algorithm and TTCT are displayed in Table 6.3.

L	Open loop		Closed loop			BDD Program		TTCT	
	$\|Q\|$	$\|nodes\|$	$\|Q\|$	$\|T\|$	$\|nodes\|$	t_{TTG}	t_{supc}	t_{TTG}	t_{supc}
20	7056	9541	6718	11781	3787	0	1	0	0
50	41,616	18,770	39,268	68,391	10,063	0	5	79	4
100	163,216	34,908	153,518	266,741	22,311	0	37	1014	40
200	646,416	66,626	607,018	1,053,441	49,410	1	176	16,429	362
250	1,008,016	75,867	946,268	1,641,791	61,111	1	272	38,827	2,116
300	1,449,616	110,435	1,360,520	2,360,141	83,132	2	500	N/A	N/A
500	4,016,016	148,245	3,767,520	6,533,541	134,887	2	5,288	N/A	N/A
750	9,024,016	244,874	8,463,770	14,675,291	224,838	2	35,388	N/A	N/A

Table 6.3: TTCT and BDD program results for two-machine workcell

As mentioned in the previous example, the space required in the previous method is proportional to number of states and transitions while in our method the space is only dependent on the number of **BDD** nodes. The space requirements of the two methods are compared in Figure 6.19.

The total time each program needs to compute the closed loop behavior from ATG to TTG and then finding the supremal controllable sublanguage is shown in Figure 6.20

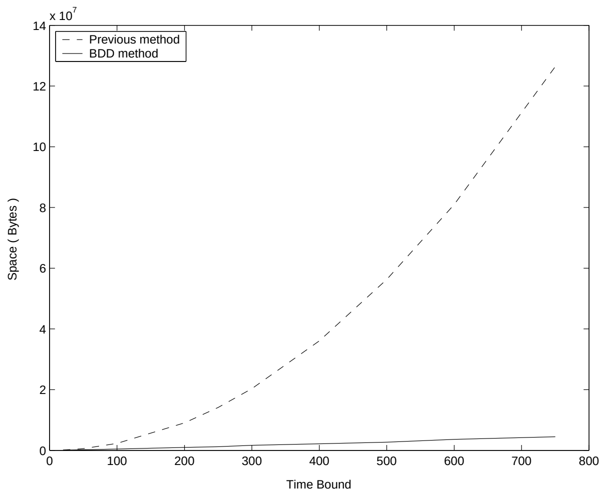


Figure 6.19: The space requirement of each method

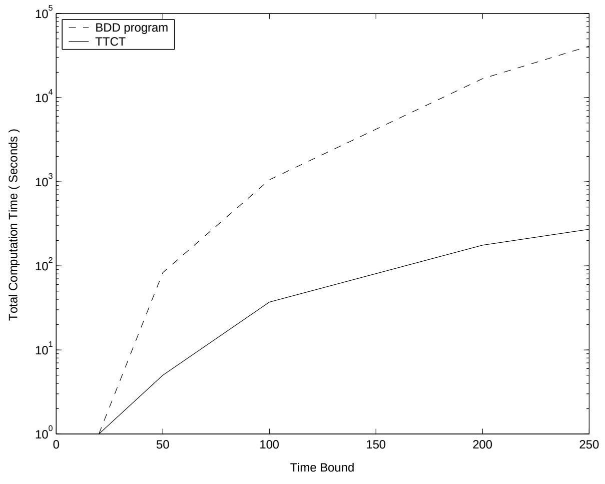


Figure 6.20: The time requirement of each method

Chapter 7

Conclusions and Future Research

This thesis presents a new synthesis algorithm for the Brandin-Wonham (BW) approach to the control of Timed Discrete Event Systems (TDES). It is easily seen that for large systems, this new method offer far better performance than the previous TTCT implementation.

This algorithm demonstrates the importance of structural information in a TDES system in the form of event timers. We can see that the computation complexity in finding the timed transition graph can be dramatically reduced. Our new algorithm also showed that this structural information can lead us to find the controller for much bigger problems. Also, it is shown that the BW framework, 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 the state space, can be used to evaluate the complexity of the system. One of these parameters, which we used in this thesis, is the number of nodes of the **BDD** representing the system's state space. It is shown that the number of **BDD** nodes, which practically decides the computational complexity of synthesis, can be linear with respect to time bounds of the events in the system. The ordering of variables can also change the number of **BDD** nodes as discussed in Section 5.3. This suggests that future research in this direction may be quite promising. Possible future research in this field can be:

- Finding a systematic way for choosing a better ordering of variables can reduce the size of the **BDD**s and therefore the complexity of the problem.

- Using other symbolic tools like Integer Decision Diagrams (IDDs) [31],Bounded Model Checking (BMC)[2] can be explored.

- Extending this method for Synchronous Product Timed Systems, Hierarchical Control of Timed DES [28] and Supervisory Control of Timed-DES with partial observation [16]

- Finding an appropriate structure for the specification automata, in the case where we do not have a timing structure for these automata, in order to convert them from a flat model to a model with some structure. This is important because in a flat model we need to encode the variable for representing the states with a large number of **BDD** variables.

- Combining our new method with other methods for finding reduced supervisors for TDES [12][3].

Appendix A

BuDDy Package

BuDDy [17] is a Binary Decision Diagram package that provides all of the most used functions for manipulating **BDD**s. The package also includes functions for integer arithmetic such as addition and relational operators.

BuDDy started as a technology transfer project between the Technical University of Denmark and Bann Visualstate. The **BDD** package presented here was made as part of a Ph.D. project by Jrn Lind-Nielsen on model checking of finite state machines. The package has evolved from a simple introduction to **BDD**s to a full blown **BDD** package with all the standard **BDD** operations, reordering and a wealth of documentation.

First of all a program needs to initialize the **BDD** package. Getting the most out of any **BDD** package is not always easy. It requires some knowledge about the optimal order of the **BDD** variables. If we allocate as much memory as possible from the very beginning, then BuDDy does not have to waste time trying to allocate more whenever it is needed.

Included in the **BDD** package is a set of functions for manipulating values of finite domains, like for example finite state machines. These functions are used to allocate blocks of **BDD** variables to represent integer values instead of only true and false. Those functions which are called "fdd" are used in our **BDD** program. The functions which are used in our program are listed below. More details about this package can be found in the package documentations, available at : *http://www.itu.dk/research/buddy*

- bdd-init
- bdd-autoreorder
- bdd-nodecount
- bdd-satcountset
- bdd-ithvar
- bdd-done
- bdd-exist
- bdd-relprod
- bdd-replace
- bdd-setpairs
- fdd-domain
- fdd-extdomain
- fdd-ithvar
- fdd-makeset
- fdd-equals

Bibliography

[1] S. B. Akers. *Functional testing with Binary Decision Diagrams. Eighth Annual Conference on Fault-Tolerant Computing*, pages 75–82, 1978.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. *Symbolic model checking using SAT procedures instead of BDDs. Design Automation Conference (DAC 99)*, 1999.

[3] S. E. Bourdon, W. M. Wonham, and M. Lawford. *Invariance under scaling of time bounds in discrete-event systems. Proceedings of the 38th Annual Allerton Conference on Communication, Control, and Computing*, 2:1145–1154, 2000.

[4] B. Brandin. *Real-time supervisory control of automated manufacturing ststems. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto*, 1993.

[5] B. Brandin and W. M. Wonham. *Supervisory control of timed discrete event systems. IEEE Transactions on Automatic Control*, 39(2):329–342, 1994.

[6] Y. Brave and M. Heymann. *Formulation and control of real-time discrete event processes. Proceedings 27th conference on Decision and Control, Austin*, pages 1131–1132, December 1988.

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

[8] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. *Symbolic Model Checking: 10^{20} states and beyond. Information and Computation*, 98:142–170, June 1992.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking. The MIT Press*, 2002.

[10] S. B. Gershwin. *A hierarchical framework for discrete event scheduling in manufacturing systems. Proceedings IIASA workshop on Discrete Event Systems, Soporn, Hungary*, August 1987.

[11] P. Gohari and W. M. Wonham. *On the complexity of supervisory control design in the RW framework. IEEE Transactions on Systems, Man and Cybernetics, Special Issue on DES*, 30(5):643–652, 2000.

[12] P. Gohari and W. M. Wonham. *Reduced Supervisors for Timed Discrete-Event Systems. IEEE Trans. on Automatic Control*, 48(7), July 2003.

[13] J. Gunnarsson. *Symbolic Methods and Tools for Discrete Event Dynamic Systems. Linkping Studies in Science and Technology, Dissertion No. 477, Linkping University*, 1997.

[14] K. H. Johansson, J. Lygeros, S. Sastry, and M. Egerstedt. *Simulation of a Zeno Hybrid Automata. IEEE conference on Decision and Control*, 1999.

[15] Y. Li and W. M. Wonham. *On supervisory control of real-time discrete event systems. Information Science*, (44):159–183, 1988.

[16] F. Lin and W. M. Wonham. *Supervisory control of timed discrete-event systems under partial observation. IEEE Trans on Automatic Control*, 40(3):558–562, March 1995.

[17] J. Lind-Nielsen. *BuDDy: Binary Decision Diagram Package. IT-University of Copenhagen (ITU) http://www.it-c.dk/research/buddy/*, 2002.

[18] C. Ma. *Nonblocking Supervisory Control of State Tree Structures. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto*, 2004.

[19] K. L. McMillan. *Symbolic Model Checking. Kluwer Academic Publishers*, 1993.

[20] R. S. Minhas. *Complexity Reduction in Discrete Event Systems. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto*, 2002.

[21] P. Moller. *Introducing real time in the algebraic theory of finite automata. Working paper WP-86-49, International Institute for applied system Analysis, Laxenburg, Austria*, September 1986.

[22] J. S. Ostroff. *Temporal logic for real-time Systems. Research Studios Press Ltd*, 1989.

[23] J. S. Ostroff. *Deciding properties of timed transition models. IEEE Transactions on Parallel and Distributed Systems*, 1(2):170–183, April 1990.

[24] J. S. Ostroff and W. M. Wonham. *A framework for real-time discrete event control. IEEE Transactions on Automatic Control*, 35(4):386–397, April 1990.

[25] S. O'Young. *On the synthesis of the supervisors for timed discrete event processes. Systems Control Group Report 9107, Department of Electrical and Computer Engineering, University of Toronto*, 1991.

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

[27] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. *Algorithms for discrete function manipulation. IEEE international conference on Computer-Aided Design*, November 1990.

[28] Kai C. Wong and W. M. Wonham. *Hierarchical control of timed discrete-event systems. Discrete Event Dynamic Systems: Theory and Applications*, 6(3):275–306, July 1996.

[29] W. M. Wonham. *Notes on control of Discrete-Event Systems. Department of Electrical and Computer Engineering, University of Toronto*, 2003.

[30] Z. Zhang. *An efficient algorithm for supervisory control design. MASc thesis, Department of Electrical and Computer Engineering, University of Toronto*, 2001.

[31] Z. H. Zhang and W. M. Wonham. *STCT: An Efficient Algorithm for Supervisory Control Design. Symposium on Supervisory Control of Discrete Event Systems (SCODES2001), Paris*, pages 82–93, July 2001.