COMPLEXITY REDUCTION IN DISCRETE EVENT SYSTEMS

BY

Rajinderjeet Singh Minhas

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy,
Graduate Department of Electrical and Computer Engineering,
in the University of Toronto.

Copyright © 2002 by Rajinderjeet Singh Minhas.
All Rights Reserved.

For my parents.

Complexity Reduction in Discrete Event Systems

Doctor of Philosophy Thesis
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

by Rajinderjeet Singh Minhas
September 2002

Abstract

The aim of this thesis is to explore new ways to achieve complexity reduction in discrete event systems (DES). We present three different ways of reducing complexity. First, we present a symbolic supervisory control design method for composite systems such that the complete state space never needs to be computed. Instead of a supervisor implemented by a static lookup table, we provide a function that can be efficiently and dynamically computed at each state to determine the control action. This symbolic function can be suitably modified to ensure that the system under control is free of deadlocks. Secondly, we present a heuristic algorithm to reduce the size of a (static) supervisor. Our symbolic supervision scheme is not able to guarantee non-blocking behaviour in the system under control. So to ensure non-blockingness it may be necessary to use a lookup table. Finding the smallest lookup table for a given control task is an NP-hard problem. We propose a greedy supervisor reduction algorithm based on the concept of control covers. This algorithm seems to work quite well in a large number of cases. Finally, we present a compact model of timed discrete event systems (TDES). We use local timers at each state of the TDES to model the passage of time. This model is quite robust to changes in time scale and is closed under control.

Acknowledgements

First and foremost, I want to thank Prof. Wonham for everything he has done for me over the years. Without a doubt, this work would not have been possible without his keen mathematical insight, guidance and financial support. However, I am most grateful for his counsel on things totally unrelated to this work.

I would also like to thank Dr. Bertil Brandin and Dr. Meera Sampath for the internship opportunities at Siemens and Xerox respectively. The experience I gained during these internships has been invaluable.

The students, faculty and staff at the System Control Group have been incredibly helpful and gracious over the years and for that I am very thankful. In particular, Peyman Gohari and Sean Bourdon have been a pleasure to know and work with.

Finally, I would like to thank the University of Toronto and the government of Ontario for the financial assistance they provided in the form of fellowships. Sarah Cherian went out of her way to help me and I am grateful for that.

Contents

List of Tables

List of Figures

1. Introduction

Yes, but are they practical?

This is a refrain that is heard a bit too often regarding the modelling and control of discrete event systems. A *discrete event system* (DES) is a physical system that is discrete (in time and space), asynchronous (event-driven rather than clock-driven), and in some sense generative (or nondeterministic) [Won01]. Ramadge and Wonham have provided an automaton based framework [RW82] (called RW) for the modelling and control of DES. Over the past couple of decades a great deal of literature has been published in this area. It has been shown that a DES can be controlled in a systematic and optimal manner [RW87a], [RW87b], [RW89]; modular [WR88], [WW98] and hierarchical [Zho92], [ZW90], [WW96a], [WW96b], [Pu00] decomposition schemes have been proposed; time has been incorporated into RW [OW90], [Bra93], [BW94], [LW88], [O'Y91] to expand its modelling capabilities. And yet the refrain is heard. The main reason for this is the *state explosion problem*. Most complex systems are composed of several interacting components. The state space of such a system is usually represented by the cross product of the state sets of the individual components. As a result, the state space may grow exponentially with the number of interacting components.

Modular and hierarchical decomposition schemes can often provide a stepwise approach for handling complex systems. But we end up dealing with the entire state space at one stage or the other. In the hierachical scenario, a control action computed at the high

(abstract) level has to be implemented at the low (physical) level. This implementation often requires complete knowledge of the low level state space. In the modular scenario, the control tasks can be broken down but the state space of the physical system still needs to be computed.

The situation is even worse if time is modelled explicitly. In addition to the complexity due to the interaction of components, there is complexity due to the modelling of time. Thus the state space of a timed system is usually much bigger than that of an untimed system. The flexibility of incorporating temporal information usually exacts a big price in terms of the complexity of the state space.

1.1. Goal and the Main Results

The main goal of this thesis is to make a contribution towards providing practical tools and methods for the modelling and control of discrete event systems. We aim to tackle the state explosion problem in discrete event systems. Towards that end, we first present a supervisory control design method for composite systems such that the complete state space never needs to be computed. The proposed method is symbolic in nature: a control function is provided that can be efficiently evaluated to compute the control action at any given state. This method is prone to blocking but can be augmented to provide deadlock avoidance. Supervisor synthesis is an NP-hard problem [GW00] even when blocking is not an issue. Under that backdrop, our proposed scheme provides an efficient synthesis tool under reasonably mild restrictions.

Sometimes it may not be practical to use our symbolic scheme. Among other reasons, this may happen either because the preconditions for symbolic supervision are violated or because blocking is a concern. For such scenarios, we present a heuristic supervisor

reduction algorithm. A supervisor reduction algorithm is meant to reduce the state size of the automaton (or the entries in a lookup table) that is used for controlling a physical system. It is always possible to find a minimal supervisor for a given job but the problem is NP-hard [SW01]. Our algorithm does not attempt to find a minimal supervisor. In fact, it does not even guarantee a reduced supervisor. However it seems to perform quite well in practice.

Finally we present a compact model of timed discrete event systems (TDES). Rather than using a special event to model the passage of time, we use timers instead. These timers countdown with respect to a global clock. This allows us to compactly incorporate temporal information. The model is also quite robust to changes in time scale. In other words, the size of the model usually does not increase much if the frequency of the global clock is increased. But the main advantage of the proposed model is that it is closed under control. A TDES subject to supervisory control can be modelled as another TDES. This is a very desirable property since it allows us to perform a series of control designs on a TDES.

1.2. Related Work

In this section we outline some of the related work done by other researchers. We begin with symbolic methods in discrete event systems. Most of it is quite different than our approach but the basic idea is the same: avoid enumerating the state space. Then we move on to supervisor reduction which has not been a very active area since the publication of [VW86]. It is shown in [VW86] that finding a minimal supervisor is exponential in time and that has perhaps retarded research in this area. The problem becomes a lot more tractable if the aim is simply to find a reduced supervisor rather than a minimal supervisor. Finally

we outline the related work in timed discrete event systems.

1.2.1. Symbolic Supervision

Symbolic methods have long been used for model checking and verification of properties of discrete event systems [McM92], [BCM92], [CGP01]. These methods try to exploit inherent regularities and symmetries in the system being verified and are often used for the verification of large digital hardware systems. The system representation is based on boolean decision diagrams (BDDs) [Ake78], [Bry86] which are used to represent binary functions. A large class of boolean functions can be represented by function graphs whose size is a polynomial function of the number of boolean variables. Thus BDDs can often compactly represent systems that are rich in symmetry although the size of a BDD is sensitive to the variable ordering [Bry86]. The inability of BDDs to represent general functions prevented their direct application in the field of supervisory synthesis although one such approach is presented in [HWT92b], [HWT92a].

Integer decision diagrams (IDDs) [Gun97] are an extension of BDDs that can be used to represent general finite functions. IDDs utilize the structure of integers and arithmetic operations and share a number of characteristics with BDDs. In particular, IDDs can be used to provide a compact representation of discrete event systems. IDDs have been successfully used in [Zha01] for the purpose of supervisor synthesis: a supervisor has been synthesized for a system with state size of 1.96×10^{15}.

The common theme of all the aforementioned results is that they use some sort of a decision diagram to construct the system model. The reduction in computational complexity is a direct result of the properties of the decision diagram. Heuristics are used to automatically get a *good* variable ordering; these heuristics are often quite good for loosely coupled systems. All the computations are still carried out offline and herein lies the major

weakness of these methods: the results need to be stored somewhere. As the state space increases so does the complexity of computing the control action *and* the complexity of storing it. The approach we take in this thesis is an online approach. We compute the control action whenever it is needed (upon entering a new state). The reduction in complexity is not a result of a particular representation (BDD, IDD etc) but of a decomposition defined on the system. This decomposition allows us to distribute the control task and employ a *look-ahead* control. To the best of our knowledge, no one else has used a similar decomposition. However a look-ahead policy has been implemented in [CLL92a], [CLL92b]. The authors there employ a look-ahead window of fixed size (say N). The control action at the current state is based on what can be foreseen in the next N steps. An assumption is made regarding the system behaviour beyond those N steps. The assumption can be *optimistic:* if nothing bad can be foreseen in N steps then it is assumed that nothing bad will happen after that; the assumption can be *conservative*: if safety cannot be guaranteed based on a look-ahead of N steps then it is assumed safety is in jeopardy after that. The authors present a lower bound on N that guarantees that the control is optimal (i.e. the same as in the case when all the information is available). However, in general, this bound cannot be computed without constructing the entire state space. Thus the choice of N is arbitrary and a designer must rely on intuition about the system behaviour.

A symbolic synthesis approach based on predicates is presented in [AMP95]. There the authors have presented supervisory control setup in a game theoretic framework. The aim is to find winning strategies without enumerating the entire state space. Similar work in a control theoretic framework is presented in [Ma02]. While similar in spirit, their approach is quite different than ours.

1.2.2. Supervisor Reduction

Given a plant and a supervisor, the supervisor reduction problem is to find a supervisor that has fewer states than the original supervisor but performs the same control task. A reduced supervisor is typically found by inducing it from a suitable cover [VW86] of the state set of the original supervisor. It is shown in [VW86] that it is always possible to find a minimal supervisor (there may be more than one). It can be done by simply enumerating all possible (and suitable) covers and picking a minimal one. However this is very inefficient; in fact, finding a minimal supervisor is NP-hard [SW01]. A supervisor reduction algorithm is presented in [SW00], [SW01]. This work recognizes the fact that is perhaps futile to search for an efficient algorithm that produces a minimal supervisor. So it focuses on finding just a reduced supervisor instead. A suitable congruence is defined on the state set of the original supervisor. This congruence leads to a partition of the state set and a supervisor is induced from that partition. A partition of a set can have no more elements than the set so this always leads to a reduced supervisor. Our algorithm provides no such guarantees as it induces the supervisor from a cover (which can have more elements than the set of which it is a cover). However we use a greedy heuristic that is based on similar heuristics used for finding an approximate set cover [Chv79], [Hoc82]. These set cover heuristics provide good approximate solutions to the set cover problem. Since the problem of finding a control cover is quite similar to finding a set cover, we expect our algorithm to perform well in practice. In fact, the algorithm has been able to produce reduced supervisors for a large variety of systems.

1.2.3. Timed Discrete Event Systems

The RW framework proposed by Ramadge and Wonham in [RW82] does not model time explicitly. It is possible to make statements like "The machine must be turned on *before* a workpiece can be processed" but it is not possible to infer *when* the machine is turned on. Thus we can only talk about the relative order of occurrences of various events. The work presented in [Mol86] extends RW to include temporal information by assigning time delays to events. The effect of time delays for supervisory control is explored in [LW88].

Brave and Heymann use a timer to model time in [BH88]. The timer is reset to zero upon entering a state and starts counting up with respect to a global clock. Time intervals are assigned for the possible occurrence of events and nondeterminism is resolved using the current timer value. The composition of systems is problematic in this setup but our proposed model is very close to this in spirit. We use timers as well but composition is well defined in our framework.

A theory of timed automata unrelated to RW is given in [AD92]. State transition graphs are annotated with timing information using real-valued clocks. The time of occurrence of every event is associated with it to form timed words and the system is modelled using a variant of Büchi automata. No apriori limit is placed on the temporal resolution but an equivalence relation is defined to derive a finite state space. Supervisory control is not explored but the theory is applied to the verification of real-time constraints. In a similar vein, supervisory control is explored in [WTH91]. Timed traces are defined over a dense domain and conditions are given for the existence of supervisors.

The use of temporal logics to specify and verify system properties is explored in [Pnu77], [CE81], [EH86]. Temporal logics are used for model checking in [BCM92], [Hol97], [QS81]. The aim in model checking is to verify whether a finite-state model of a system satisfies

a temporal logic specification. In a similar vein, temporal logic frameworks are used to reason about the correctness of supervisors [AHK97], [EH91]. However, not much work has been done in the area of supervisor synthesis.

Brandin [Bra93], [BW94] proposes an extension of RW (called BW) where the passage of time is explicitly modelled by using a special event called *tick*. The *tick* event represents the lapse of one time unit measured against a global clock. Time bounds are assigned to events and represent the constraints on their occurrence relative to their time of enablement. BW utilizes the timed features of timed transition models [OW90], [Ost90], [Ost89]. This framework possesses good expressiveness properties but suffers from a couple of major problems. First, the state size is very sensitive to the clock frequency: a *tick* event must be associated with the passage of each unit of time. As the clock frequency increases, so must the number of *tick* events. Secondly, BW is not closed under control: it may not be feasible to put a system under control in a form that is amenable to further control design. An attempt to rectify the first of these problems has been made in [Bra97], [Bra98]. No special event is used to model the passage of time; timers are used instead. However, this work is not quite correct as we show in Chapter 6. Despite incorrectness in the details, the idea laid out in [Bra97], [Bra98] is a good one. We explore it in detail in this thesis and it forms the basis of the new timed model presented in Chapter 7. The work in [Bra97], [Bra98] does not address the second problem since the model presented there is not closed under control either. Wong and Wonham have presented a framework for timed systems in [WW96a], [WW96b] that is closed under control. However their model still uses a *tick* event to model the passage of time. The model we propose in this thesis is also closed under control although our approach is different than the one taken in [WW96b].

1.3. A Roadmap for the Thesis

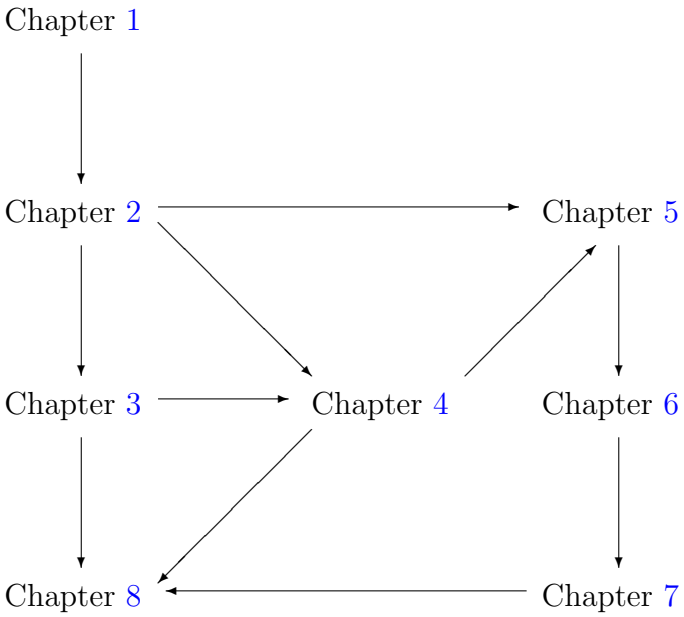
In Chapter 2 we present the general notation used in this thesis. Our symbolic supervision scheme is automaton (rather than language) based so we adapt for this thesis a number of linguistic definitions and results given in [Won01]. We also define a *join* operation for automata that will be used to construct our symbolic supervision scheme given in Chapter 3. We consider systems that comprise components that do not share any events. Under this assumption, we divide a given specification over the alphabets of the various components. A *distribution* operation (a dual of the *join* operation) allows us to automatically divide the specification. We give stepwise and modular operations on the components that help in the computation of the global control action. We show that this computation can be done in an efficient manner and provide an example to illustrate it. We then extend our supervision scheme to provide deadlock avoidance. Finally we give a complexity analysis of our symbolic scheme.

In Chapter 4 we present a heuristic algorithm for supervisor reduction. This is a greedy algorithm that is based on the concept of control covers [VW86]. Given a plant and a supervisor, our algorithm computes another supervisor that is equivalent in control action to the original supervisor. The new supervisor is not guaranteed to be a reduced supervisor, i.e. it may not have fewer states than the original supervisor. However, our algorithm seems to perform quite well and gives reduced supervisors for a large number of examples we have tried.

In Chapter 5 we give additional notation that is needed for timed systems. Some of the concepts from Chapter 2 are redefined here (the new definitions pertain only to timed systems). In Chapter 6 we pose and solve the problem of finding shortest and longest paths in a timed discrete event system. The problem is posed in a dynamic programming

framework so we know that a solution can always be found. However the typical dynamic programming solution is recursive [Bel61], [Dre65]. We provide two greedy solutions and show that only one of them is useful for the purpose of supervisory control. This solution provides the motivation for a new model of timed systems given in Chapter 7. This model is based on timers and is therefore quite compact. However, its biggest advantage is that it is closed under control. It is also quite insensitive to changes in clock frequency.

Finally, we present our conclusions in Chapter 8. We also provide some suggestions for future work. The various chapters may be read in sequence or selectively as shown below.


Chapter 1
Chapter 2
Chapter 5
Chapter 3
Chapter 4
Chapter 6
Chapter 8
Chapter 7

2. Notation

2.1. An Overview

Here we present some notation and preliminary background material. Most of the material presented here is taken from [Won01] with minor modifications. These modifications are needed because some of the results presented in this thesis are automaton based and depend critically on the state set of the automaton representing the specification. So, for instance, we need to adapt for automata the RW notion of controllability of a language with respect to another language. In other words, we define the notion of controllability of an automaton with respect to another automaton. This sort of definition is already implicit in the work of Rudie [Rud88] and the *supcon* procedure of the TCT[Won01] software. We show that the notion of controllability of an automaton with respect to another is equivalent to the notion of controllability of the languages generated by the respective automata. We also define a *join* operation for two automata by taking the union of the underlying graphs. The language generated by such a *join* of two automata is generally bigger than the union of the languages generated by the individual automata. This property allows us to decompose the problem of verifying controllability. The decomposed subproblems can be solved quite efficiently. This allows us to present a modular framework for the synthesis of controllable automata.

2.2. Notation and Definitions

Definition 1 *An* alphabet Σ *is a finite set of distinct symbols. Let* Σ^+ *denote the set of all finite symbol sequences, of the form* $\sigma_1\sigma_2\cdots\sigma_k$ *where* $k \geq 1$ *is arbitrary and the* $\sigma_i \in \Sigma$. *Let* $\epsilon \notin \Sigma$ *be the empty sequence symbol and define* $\Sigma^* = \{\epsilon\} \cup \Sigma^+$. *An element of* Σ^* *is a* string *or* word *over the alphabet* Σ*;* ϵ *is the* empty string.

Definition 2 *An* automaton $\mathbf{G} = \left(X^G, \Sigma^G, E^G, x_o^G, X_m^G\right)$ *is a 5-tuple where*

X^G *is a (non-empty) state set,*

Σ^G *is the alphabet over which* $\mathbf{G}$ *is defined,*

$E^G = \{(x, \sigma, y)\,|x, y \in X, \sigma \in \Sigma\} \subseteq X \times \Sigma \times X$ *is the set of edges or transitions of* $\mathbf{G}$,

$x_0^G \in X^G$ *is the initial state, and*

$X_m^G \subseteq X^G$ *is the set of marker states.*

For convenience, we also define the empty automaton $\mathbf{\Phi} = (\emptyset, \Sigma, \emptyset, \_, \emptyset)$.

Unless the 5-tuple representation of any automaton $\mathbf{G}$ is explicitly given, we will assume it to be $\left(X^G, \Sigma^G, E^G, x_0^G, X_m^G\right)$. We will also assume that $\Sigma^G = \Sigma_c^G \cup \Sigma_u^G$ is the disjoint union of controllable and uncontrollable events. Accordingly, we will use E_c^G and E_u^G to respectively denote the sets of controllable and uncontrollable transitions of $\mathbf{G}$, where

$$E_c^G := \left\{(x, \sigma, y) \in E^G | \sigma \in \Sigma_c^G\right\}$$

and

$$E_u^G := \left\{(x, \sigma, y) \in E^G | \sigma \in \Sigma_u^G\right\}.$$

Henceforth in this chapter, we assume that $\mathbf{G}$, $\mathbf{G}_1$ and $\mathbf{G}_2$ refer to the automata (X, Σ, E, x_0, X_m), $(X^1, \Sigma^1, E^1, x_0^1, X_m^1)$ and $(X^2, \Sigma^2, E^2, x_0^2, X_m^2)$ respectively.

Definition 3 *Let $\eta^G : X \times \Sigma \to X$ be a partial function. Then η^G is called the* state transition function *for* $\mathbf{G}$ *if for all $x_1, x_2 \in X$ and $\sigma \in \Sigma$*

$$\eta^G(x_1, \sigma) = x_2 \Longleftrightarrow (x_1, \sigma, x_2) \in E.$$

The notation $\eta^G(x, \sigma)!$ will mean that $\eta^G(x, \sigma)$ is defined. We extend η^G to a partial function $\eta^G : X \times \Sigma^* \to X$ by the rules

$$\eta^G(x, \epsilon) = x$$
$$\eta^G(x, s\sigma) = \eta^G\left(\eta^G(x, s), \sigma\right)$$

provided $\eta^G(x, s)!$ and $\eta^G\left(\eta^G(x, s), \sigma\right)!$. Sometimes it is convenient to talk about the transition function for a set of states. Let $\overline{\eta}^G : 2^X \times \Sigma \to 2^X$ be the mapping such that

$$\overline{\eta}^G(Y, \sigma) := \left\{x \in X : (\exists y \in Y)\, \eta^G(y, \sigma) = x\right\}.$$

By an abuse of notation we will use η^G instead of $\overline{\eta}^G$; the distinction should be clear from the context.

Definition 4 *The* set of eligible events *at any state $x \in X$ is defined to be*

$$Elig(\mathbf{G}, x) := \left\{\sigma \in \Sigma | \eta^G(x, \sigma)!\right\}.$$

Definition 5 *The* closed behaviour *of* $\mathbf{G}$ *is*

$$L(\mathbf{G}) := \left\{ s \in \Sigma^* | \eta^G(x_0, s)! \right\}$$

while

$$L_m(\mathbf{G}) := \left\{ s \in L(\mathbf{G}) | \eta^G(x_0, s) \in X_m \right\}$$

is the marked behaviour *of* $\mathbf{G}$. *The marked behaviour of* $\mathbf{G}$ *is also referred to as the* language generated by $\mathbf{G}$.

Definition 6 *Let* $s \in L(\mathbf{G})$ *be any string belonging to the closed behaviour of* $\mathbf{G}$. *Then the* state in $\mathbf{G}$ corresponding to s *is* $\eta^G(x_0, s)$. *Let* $x \in X$ *be any state belonging to* $\mathbf{G}$. *Then the* set of strings in $L(\mathbf{G})$ corresponding to x *is*

$$\left\{ s \in L(\mathbf{G}) | x = \eta^G(x_0, s) \right\}.$$

These definitions are similarly extended to $L_m(\mathbf{G})$.

Definition 7 *The automaton* $\mathbf{G}_1$ *is a* subautomaton *of* $\mathbf{G}_2$ *(denoted* $\mathbf{G}_1 \leq \mathbf{G}_2$*) if* $X^1 \subseteq X^2$, $\Sigma^1 \subseteq \Sigma^2$, $E^1 \subseteq E^2$, $X_m^1 \subseteq X_m^2$, *and* $x_0^1 = x_0^2$. *We extend* $\leq$ *by declaring that* $\mathbf{\Phi} \leq \mathbf{G}$ *for any automaton* $\mathbf{G}$.

It follows that if $\mathbf{G}_1 \leq \mathbf{G}_2$ then $L(\mathbf{G}_1) \subseteq L(\mathbf{G}_2)$ and $L_m(\mathbf{G}_1) \subseteq L_m(\mathbf{G}_2)$.

Definition 8 *Let* $\mathbf{H}$ *and* $\mathbf{K}$ *be subautomata of* $\mathbf{G}$. *Then the* join *of* $\mathbf{H}$ *and* $\mathbf{K}$ *(denoted* $\mathbf{H} \vee \mathbf{K}$*) is defined as*

$$\mathbf{H} \vee \mathbf{K} = \left(X^H \cup X^K, \Sigma^H \cup \Sigma^K, E^H \cup E^K, x_0, X_m^H \cup X_m^K \right).$$

Proposition 9 *Let* $\mathbf{F}$, $\mathbf{H}$, *and* $\mathbf{K}$ *be subautomata of* $\mathbf{G}$ *such that* $\mathbf{H}, \mathbf{K} \leq \mathbf{F}$. *Then* $\mathbf{H} \vee \mathbf{K} \leq \mathbf{F}$.

Proof. Since $\mathbf{H}$ and $\mathbf{K}$ are subautomata of $\mathbf{F}$ it follows that $X^H \subseteq X^F$ and $X^K \subseteq X^F$. So $X^H \cup X^K \subseteq X^F$. Similarly $\Sigma^H \cup \Sigma^K \subseteq \Sigma^F$, $E^H \cup E^K \subseteq E^F$, and $X_m^H \cup X_m^K \subseteq X_m^F$ which gives the desired result. ■

Proposition 10 *Let* $\mathcal{G}$ *be the set of all subautomata of* $\mathbf{G}$. *Then* $< \mathcal{G}, \leq >$ *is an upper semilattice with* $\vee$ *as the join operation.*

Proof. In order to show that $< \mathcal{G}, \leq >$ is an upper semilattice, we need to show that it is a poset in which *join* is always defined.

1. (**Reflexive**): Let $\mathbf{H}$ be any subautomaton of $\mathbf{G}$. Then $\mathbf{H} \leq \mathbf{H}$ since $X^H \subseteq X^H$, $\Sigma^H \subseteq \Sigma^H$, $E^H \subseteq E^H$ and $X_m^H \subseteq X_m^H$.

2. (**Transitive**): Let $\mathbf{F}$, $\mathbf{H}$, and $\mathbf{K}$ be subautomata of $\mathbf{G}$ such that $\mathbf{F} \leq \mathbf{H}$ and $\mathbf{H} \leq \mathbf{K}$. Then $X^F \subseteq X^H$ and $X^H \subseteq X^K$ which implies that $X^F \subseteq X^K$. Similarly, $\Sigma^F \subseteq \Sigma^K$, $E^F \subseteq E^K$ and $X_m^F \subseteq X_m^K$. Thus $\mathbf{F} \leq \mathbf{K}$.

3. (**Antisymmetric**): Let $\mathbf{H}$ and $\mathbf{K}$ be subautomata of $\mathbf{G}$ such that $\mathbf{H} \leq \mathbf{K}$ and $\mathbf{K} \leq \mathbf{H}$. This implies that $X^H = X^K$, $\Sigma^H = \Sigma^K$, $E^H = E^K$, and $X_m^H = X_m^K$ which means that $\mathbf{H} = \mathbf{K}$.

4. (**Join Exists**): Let $\mathbf{H}$, $\mathbf{K}$ and $\mathbf{F}$ be subautomata of $\mathbf{G}$. Additionally, assume that $\mathbf{H}$, $\mathbf{K}$ are subautomata of $\mathbf{F}$. Then from Proposition 9 we can conclude that $\mathbf{H} \vee \mathbf{K} \leq \mathbf{F}$. Similarly, we can conclude that $\mathbf{H} \vee \mathbf{K} \leq \mathbf{G}$, i.e. $\mathbf{H} \vee \mathbf{K} \in \mathcal{G}$. ■

Definition 11 *The automaton* $\mathbf{G}_1$ *is* isomorphic *to* $\mathbf{G}_2$ *(denoted* $\mathbf{G}_1 \approx \mathbf{G}_2$*) if there exists a bijection* $f : X^1 \to X^2$ *such that* $f_1\left(X_m^1\right) = X_m^2$, $f\left(x_0^1\right) = x_0^2$, *and* $f_2\left(E^1\right) = E^2$ *where*

$$\begin{aligned} f_1\left(X_m^1\right) &= \left\{f\left(x\right) | x \in X_m^1\right\}, \text{ and} \\ f_2\left(E^1\right) &= \left\{\left(f\left(x_1\right), \sigma, f\left(x_2\right)\right) | \left(x_1, \sigma, x_2\right) \in E^1\right\}. \end{aligned}$$

If $\mathbf{G}_1 \approx \mathbf{G}_2$ *then* $L\left(\mathbf{G}_1\right) = L\left(\mathbf{G}_2\right)$ *and* $L_m\left(\mathbf{G}_1\right) = L_m\left(\mathbf{G}_2\right)$.

Thus an automaton is isomorphic to another if one can be converted into the other by relabeling the states.

Definition 12 *Let* $x_1, x_2 \in X$ *and let* $e_1, \ldots, e_n \in E$. *Then the sequence* $e_1 \cdots e_n$ *is called a* path *in* $\mathbf{G}$ *(of length* n*) between* x_1 *and* x_2 *if* $e_1 = \left(x_1, \sigma_1, y_1\right)$, $e_i = \left(y_{i-1}, \sigma_i, y_i\right)$ *for* $2 \leq i \leq n-1$, *and* $e_n = \left(y_{n-1}, \sigma_n, x_2\right)$ *for some* $y_1, \ldots, y_{n-1} \in X$ *and* $\sigma_1, \ldots, \sigma_n \in \Sigma$. *The empty sequence is called the* empty path *(of length zero).*

Definition 13 *Let* $x_1, x_2 \in X$. *Then* x_2 *is said to be* reachable from x_1 *if there exists a path from* x_1 *to* x_2. *A state is said to be* reachable *if it is reachable from* x_0. *A state is declared to be reachable from itself via the empty path.*

Definition 14 *The automaton* $\mathbf{G}_{rch} = \left(X_{rch}, \Sigma, E_{rch}, x_0, X_m \cap X_{rch}\right)$ *is the* reachable subautomaton *of* $\mathbf{G}$ *where*

$X_{rch} = \left\{x \in X | x \text{ is reachable}\right\}$, *and*

$E_{rch} = \left\{\left(x_1, \sigma, x_2\right) \in E | x_1, x_2 \in X_{rch}\right\}$.

The reachable subautomaton $\mathbf{G}_{rch}$ *has the same closed and marked behaviours as* $\mathbf{G}$, *i.e.* $L\left(\mathbf{G}\right) = L\left(\mathbf{G}_{rch}\right)$ *and* $L_m\left(\mathbf{G}\right) = L_m\left(\mathbf{G}_{rch}\right)$. *We say* $\mathbf{G}$ *is* reachable *if* $\mathbf{G} = \mathbf{G}_{rch}$.

Definition 15 *Let $x \in X$. Then x is said to be* coreachable *if there exists a path from x to some state in X_m. We say $\mathbf{G}$ is* coreachable *if x is coreachable for all $x \in X$.*

Definition 16 *An automaton is* trim *if it is both reachable and coreachable.*

Definition 17 *The* product automaton $\mathbf{G}_1 \times \mathbf{G}_2$ *is defined as*

$$\mathbf{G}_1 \times \mathbf{G}_2 = \left(X^1 \times X^2, \Sigma^1 \cap \Sigma^2, P, \left(x_0^1, x_0^2\right), X_m^1 \times X_m^2\right)$$

where

$$P = \left\{\left((x_1, y_1), \sigma, (x_2, y_2)\right) \mid (x_1, \sigma, x_2) \in E^1 \text{ and } (y_1, \sigma, y_2) \in E^2\right\}.$$

The product automaton $\mathbf{G}_1 \times \mathbf{G}_2$ generates only those strings that can be generated by both $\mathbf{G}_1$ and $\mathbf{G}_2$, i.e. $L(\mathbf{G}_1 \times \mathbf{G}_2) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2)$ and $L_m(\mathbf{G}_1 \times \mathbf{G}_2) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$.

Definition 18 *The* meet *of $\mathbf{G}_1$ and $\mathbf{G}_2$ (denoted $\mathbf{G}_1 \wedge \mathbf{G}_2$) is defined as the reachable subautomaton of the product automaton, i.e.*

$$\mathbf{G}_1 \wedge \mathbf{G}_2 = (\mathbf{G}_1 \times \mathbf{G}_2)_{rch}.$$

Clearly $L(\mathbf{G}_1 \wedge \mathbf{G}_2) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2)$ and $L_m(\mathbf{G}_1 \wedge \mathbf{G}_2) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$.

Often the behaviour of a plant is modelled as the synchronous operation of various subsystems. We now define such a synchronous behaviour of components in terms of the *meet* operation. We also define the synchronous composition of languages and show that language generated by the synchronous composition of two automata is the same as the synchronous composition of their languages.

Definition 19 *The* synchronous composition *of* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *(denoted* $\mathbf{G}_1 \| \mathbf{G}_2$*) is defined as* $\mathbf{G}_1 \| \mathbf{G}_2 = \mathbf{G}_1' \wedge \mathbf{G}_2'$ *where*

$$\begin{aligned}
\mathbf{G}_1' &= \left(X^1, \Sigma^1 \cup \Sigma^2, E', x_0^1, X_m^1\right), \\
E' &= E^1 \cup \left\{(x, \sigma, x) \,|x \in X^1 \text{ and } \sigma \in \Sigma^2 - \Sigma^1\right\}, \\
\mathbf{G}_2' &= \left(X^2, \Sigma^1 \cup \Sigma^2, F', x_0^2, X_m^2\right), \\
F' &= E^2 \cup \left\{(x, \sigma, x) \,|x \in X^2 \text{ and } \sigma \in \Sigma^1 - \Sigma^2\right\}.
\end{aligned}$$

If $\Sigma^1 = \Sigma^2$ *then* $\mathbf{G}_1 \| \mathbf{G}_2 = \mathbf{G}_1 \wedge \mathbf{G}_2$.

Definition 20 *Let* $L_1 \subseteq \Sigma^{1*}$, $L_2 \subseteq \Sigma^{2*}$. *The* synchronous composition *of* L_1 *and* L_2 *(denoted* $L_1 \| L_2$*) is defined as* $L_1 \| L_2 = P_1^{-1} L_1 \cap P_2^{-1} L_2$ *where* $P_i : (\Sigma^1 \cup \Sigma^2)^* \to \Sigma_i^*$ *are the natural projection maps.*

Proposition 21 *The marked behaviour of the synchronous composition of* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *is the same as the synchronous composition of the marked behaviours of* $\mathbf{G}_1$ *and* $\mathbf{G}_2$*, i.e.* $L_m(\mathbf{G}_1 \| \mathbf{G}_2) = L_m(\mathbf{G}_1) \| L_m(\mathbf{G}_2)$.

Proof. Let $\mathbf{G}_1'$ and $\mathbf{G}_2'$ be automata such that $\mathbf{G}_1 \| \mathbf{G}_2 = \mathbf{G}_1' \wedge \mathbf{G}_2'$. For $i \in \{1, 2\}$ let $P_i : (\Sigma^1 \cup \Sigma^2)^* \to \Sigma_i^*$ be the natural projection maps. From Definition 19 we can see that $L_m(\mathbf{G}_i') = P_i^{-1} L_m(\mathbf{G}_i)$. Thus

$$\begin{aligned}
L_m(\mathbf{G}_1 \| \mathbf{G}_2) &= L_m(\mathbf{G}_1' \wedge \mathbf{G}_2') \\
&= L_m(\mathbf{G}_1') \cap L_m(\mathbf{G}_2') \\
&= P_1^{-1} L_m(\mathbf{G}_1) \cap P_2^{-1} L_m(\mathbf{G}_2) \\
&= L_m(\mathbf{G}_1) \| L_m(\mathbf{G}_2).
\end{aligned}$$

■

2.2.1. Distribute and Join

Let a system be modelled by $\mathbf{G}$ and assume that $\mathbf{G}_1$ is a subautomaton of $\mathbf{G}$. Then $\mathbf{G}_1$ can be thought of as a partial model of the system. Perhaps $\mathbf{G}$ contains far more information than is needed for some specific purpose. In such a case a partial model might provide an uncluttered picture of the system for that specific purpose.

Example 22 *Let*

$$\mathbf{G} = \left(\begin{array}{c} \{idle, working\}, \{start, stop, run\_diagnostics\}, \\ \{(idle, start, working), (working, stop, idle), (idle, run\_diagnostics, idle)\}, \\ idle, \{idle\} \end{array} \right)$$

represent a machine (shown in Figure 2.1) that is either in idle *state or in* working *state.*

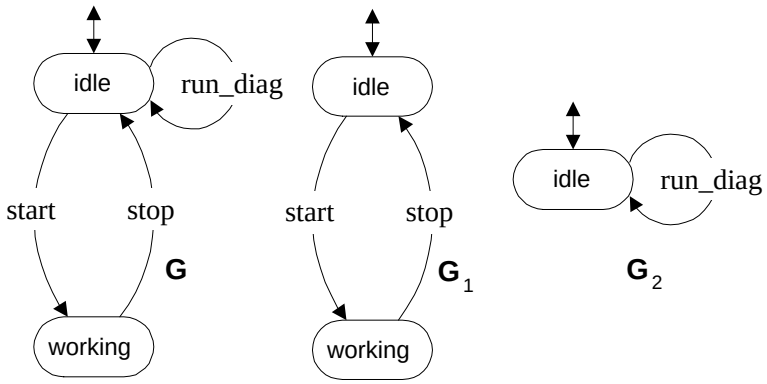


Figure 2.1: Partial Models of a Machine

If it is idle *then it can be made to run diagnostics on itself or can be made to start working. Now suppose that we are not interested in running diagnostics on the machine. In such a*

case, a partial model of the machine that does not contain diagnostic details might be more useful. Such a model may be represented as

$$\mathbf{G}_1 = \left(\begin{array}{c} \{idle, working\}, \{start, stop\}, \\ \{(idle, start, working), (working, stop, idle)\}, \\ idle, \{idle\}. \end{array} \right)$$

Similarly if we are only interested in the diagnostic operation then perhaps a partial model containing just that information would be more useful. Such a model may be represented as

$$\mathbf{G}_2 = (\{idle\}, \{run\_diagnostics\}, \{(idle, run\_diagnostics, idle)\}, idle, \{idle\}).$$

Both $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *are also shown in Figure 2.1.* □

In the above example, it is important to note that both $\mathbf{G}_1$ and $\mathbf{G}_2$ are models of the same machine but represent different aspects of the machine. In other words, $\mathbf{G}_1$ and $\mathbf{G}_2$ represent different views of the same system. It seems reasonable to assume that if we are given $\mathbf{G}$ and know the attributes of interest then we should be able to derive partial models from $\mathbf{G}$ representing those attributes. Similarly if we are given a collection of partial models of a system, it again seems reasonable that we should be able to somehow join these partial models to compute an aggregate model of the system that contains all the information of the individual partial models. We now show one way of doing this *distribution* and *joining*.

Definition 23 *Let* $S = \{\Sigma^i \subseteq \Sigma | 1 \leq i \leq n\}$ *be a set of subalphabets of* Σ. *Let* τ *be an event that does not belong to* Σ. *We now define*

$$Distribute\,(\mathbf{G}, S) = \left\{ \mathbf{G}_i = \left(X, \Sigma^i \dot{\cup} \{\tau\}, E^i, x_0, X_m \right) | 1 \leq i \leq n \right\}$$

where

$$\begin{aligned} E^i &= E^i_l \,\dot\cup\, E^i_f \\ E^i_l &= \left\{(x_j,\sigma,x_k)\,|\sigma\in\Sigma^i \text{ and } (x_j,\sigma,x_k)\in E\right\} \\ E^i_f &= \left\{(x_j,\tau,x_k)\,|\,\left(\exists\sigma\in\Sigma-\Sigma^i\right) \text{ s.t. } (x_j,\sigma,x_k)\in E \text{ and } x_j\neq x_k\right\}, \end{aligned}$$

to be the distribution *of* $\mathbf{G}$ *over* S.

Here we are given a system model $\mathbf{G}$ and a set of alphabets Σ^i. For each of the given alphabets, the aim is to produce a partial model of the system represented by $\mathbf{G}$. For any given alphabet, the corresponding partial model should ideally just contain information about the events that belong to the alphabet (i.e. the local events). However, as we will shortly show, this is not entirely feasible as some of the partial models may be rendered ambiguous or even incorrect. So a special event τ is used to retain reachability information that might have been lost otherwise. Since it models the effect of one or more events that do not belong to the given alphabet, τ is called a *foreign* event. Thus the transition set E^i is the disjoint union of transitions E^i_l and E^i_f due to the local events belonging to Σ^i and the foreign event τ respectively.

It may turn out that the distributed models are non-deterministic. However this non-determinism is restricted to the transitions caused by the foreign event τ. This is not a big concern because in a distributed model we are mainly interested in the transitions caused by the local events. The foreign event transitions merely provide additional information (like reachability).

Example 24 *Let us reconsider the machine from Example 22. It is modelled as*

$$\mathbf{G} = \left(\begin{array}{c} \{idle, working\}, \{start, stop, run\_diagnostics\}, \\ \{(idle, start, working), (working, stop, idle), (idle, run\_diagnostics, idle)\}, \\ idle, \{idle\}. \end{array} \right)$$

Let

$$S = \{\{start\}, \{stop\}, \{run\_diagnostics\}, \{start, stop\}\}.$$

Then

$$Distribution(\mathbf{G}, S) = \{\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3, \mathbf{G}_4\}$$

where

$$\mathbf{G}_1 = \left(\begin{array}{c} \{idle, working\}, \{start, \tau\}, \\ \{(idle, start, working), (working, \tau, idle)\}, \\ idle, \{idle\}, \end{array} \right)$$

$$\mathbf{G}_2 = \left(\begin{array}{c} \{idle, working\}, \{stop, \tau\}, \\ \{(idle, \tau, working), (working, stop, idle)\}, \\ idle, \{idle\}, \end{array} \right)$$

$$\mathbf{G}_3 = \left(\begin{array}{c} \{idle, working\}, \{run\_diagnostics, \tau\}, \\ \{(idle, \tau, working), (working, \tau, idle), (idle, run\_diagnostics, idle)\}, \\ idle, \{idle\}, \end{array} \right)$$

and

$$\mathbf{G}_4 = \left(\begin{array}{c} \{idle, working\}, \{start, stop, \tau\}, \\ \{(idle, start, working), (working, stop, idle)\}, \\ idle, \{idle\}, \end{array} \right)$$

Here we are given four alphabets so the distribution *of* $\mathbf{G}$ *produces four partial models:*

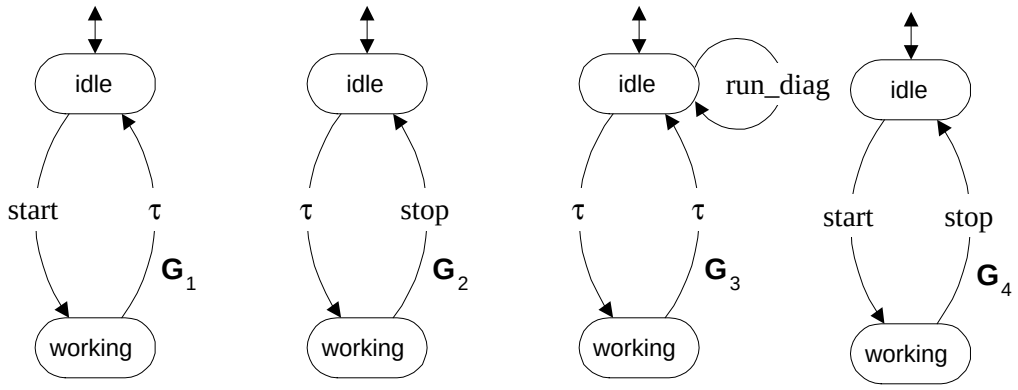


Figure 2.2: Distribution of an Automaton

$\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3$ *and* $\mathbf{G}_4$ *shown in Figure 2.2. The partial model* $\mathbf{G}_1$ *may be interpreted as follows: If the machine is in* idle *state then it may be started. If started, the machine reaches the* working *state where it remains until some "foreign" event* τ *takes it to the idle state again. If we had not included the* τ *event then the partial model* $\mathbf{G}_1$ *would have given the erroneous impression that the machine can be started only once. Similarly, if we had not included the* τ *event in the partial model* $\mathbf{G}_2$ *then it would have been unclear how the machine reaches the* working *state from the initial* idle *state. The partial model* $\mathbf{G}_3$ *tells us that the diagnostics may not be run while the machine is working. It does not give any useful information about the other modes of operation of this machine. Finally, the partial model* $\mathbf{G}_4$ *provides information about the start and stop operations of the machine.* □

The *distribution* operation takes an automaton model of a system and generates partial system models. We now define the reverse operation that takes partial models of a system and generates an aggregate system model. By an abuse of notation, we shall call this operation a *join* operation as well.

Definition 25 *Let* $D = \left\{\mathbf{G}_i = \left(X, \Sigma^i \dot{\cup} \{\tau\}, E_l^i \dot{\cup} E_f^i, x_0, X_m\right) | 1 \leq i \leq n\right\}$ *be a set of automata models of some system. Then the* join *of* D *is defined as*

$$Join\,(D) = \left(X, \cup_{i=1}^n \Sigma^i, \cup_{i=1}^n E_l^i, x_0, X_m\right).$$

We may also represent $Join\,(D)$ *as* $\mathbf{G}_1 \vee \cdots \vee \mathbf{G}_n$ *or simply as* $\vee D$.

From the definitions of *distribution* and *join* it can be seen that $Join(Distribution(\mathbf{G}, S)) = \mathbf{G}$ whenever $\cup S = \Sigma$. The *join* operation of Definition 25 is based on the *join* operation of Definition 8 but has been specialized to get rid of the τ event. The τ event is introduced in a *distribution* only to retain some reachability information and is of no use once the distribution has been put back together. The algebraic properties of the *join* operation of Definition 25 are inherited by the *join* operation of Definition 8 and this proves useful in various proofs.

Example 26 *Let us continue with the machine of Example 24. There we computed the partial models corresponding to four given alphabets. Now let us join those partial models.*

$$\begin{aligned} Join\,(\{\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3, \mathbf{G}_4\}) &= \mathbf{G}_1 \vee \mathbf{G}_2 \vee \mathbf{G}_3 \vee \mathbf{G}_4 \\ &= \left(\begin{array}{c} \{idle, working\}, \{start, stop, run\_diagnostics\}, \\ \left\{ \begin{array}{c} (idle, start, working), (working, stop, idle), \\ (idle, run\_diagnostics, idle) \end{array} \right\}, \\ idle, \{idle\}. \end{array} \right) \\ &= \mathbf{G}. \end{aligned}$$

□

We now show that *product* distributes over *join*.

Proposition 27 *Let* $\mathbf{S}_1 = \left(Y, \Sigma^1 \dot{\cup} \{\tau\}, A_l^1 \cup A_f^1, y_0, Y_m\right)$, $\mathbf{S}_2 = \left(Y, \Sigma^2 \dot{\cup} \{\tau\}, A_l^2 \cup A_f^2, y_0, Y_m\right)$ *and assume that* $\Sigma^1 \cup \Sigma^2 = \Sigma$. *Then*

$$\mathbf{G} \times (\mathbf{S}_1 \vee \mathbf{S}_2) = (\mathbf{G} \times \mathbf{S}_1) \vee (\mathbf{G} \times \mathbf{S}_2).$$

Proof. Let $\mathbf{S} = \mathbf{S}_1 \vee \mathbf{S}_2$. Then

$$\mathbf{S} = \left(Y, \Sigma^1 \cup \Sigma^2, A_l^1 \cup A_l^2, y_0, Y_m\right).$$

So

$$\begin{aligned} \mathbf{G} \times (\mathbf{S}_1 \vee \mathbf{S}_2) &= \mathbf{G} \times \mathbf{S} \\ &= \left(X \times Y, \Sigma \cap \left(\Sigma^1 \cup \Sigma^2\right), E', (x_0, y_0), X_m \times Y_m\right) \\ &= (X \times Y, \Sigma, E', (x_0, y_0), X_m \times Y_m) \end{aligned}$$

where

$$\begin{aligned} E' &= \left\{((x_1, y_1), \sigma, (x_2, y_2)) \,|\, (x_1, \sigma, x_2) \in E \text{ and } (y_1, \sigma, y_2) \in A_l^1 \cup A_l^2\right\} \\ &= \left\{((x_1, y_1), \sigma, (x_2, y_2)) \,|\, (x_1, \sigma, x_2) \in E \text{ and } (y_1, \sigma, y_2) \in A_l^1\right\} \\ &\quad \cup \left\{((x_1, y_1), \sigma, (x_2, y_2)) \,|\, (x_1, \sigma, x_2) \in E \text{ and } (y_1, \sigma, y_2) \in A_l^2\right\} \end{aligned}$$

Similarly

$$\mathbf{G} \times \mathbf{S}_1 = \left(X \times Y, \Sigma^1, A^1, (x_0, y_0), X_m \times Y_m\right)$$

where

$$A^1 = \left\{((x_1, y_1), \sigma, (x_2, y_2)) \,|\, (x_1, \sigma, x_2) \in E \text{ and } (y_1, \sigma, y_2) \in A_l^1\right\},$$

and

$$\mathbf{G} \times \mathbf{S}_2 = \left(X \times Y, \Sigma^2, A^2, (x_0, y_0), X_m \times Y_m\right)$$

where

$$A^2 = \left\{((x_1, y_1), \sigma, (x_2, y_2)) \,|\, (x_1, \sigma, x_2) \in E \text{ and } (y_1, \sigma, y_2) \in A_l^2\right\}.$$

Now

$$\begin{aligned}(\mathbf{G} \times \mathbf{S}_1) \vee (\mathbf{G} \times \mathbf{S}_2) &= \left(X \times Y, \Sigma^1 \cup \Sigma^2, A^1 \cup A^2, (x_0, y_0), X_m \times Y_m\right) \\ &= (X \times Y, \Sigma, E', (x_0, y_0), X_m \times Y_m) \\ &= \mathbf{G} \times (\mathbf{S}_1 \vee \mathbf{S}_2).\end{aligned}$$

■

2.2.2. Controllability of Automata

We now define the notion of controllability of an automaton with respect to another automaton. This is essentially the translation of the RW notion of controllability of a language with respect to another language. We assume as usual that any given alphabet $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$ where Σ_c and Σ_u represent the (mutually disjoint) sets of controllable and uncontrollable events respectively.

Definition 28 *Let* $\mathbf{G}$ *and* $\mathbf{S} = (Y, \Sigma^2, F, y_0, Y_m)$ *be trim automata. We define* $\mathbf{S}$ *to be*

controllable *with respect to* $\mathbf{G}$ *if*

$$(\forall (x_1, y_1) \in \mathbf{G} \parallel \mathbf{S}) (\forall \sigma \in \Sigma_u) (\forall x_2 \in X) [(x_1, \sigma, x_2) \in E \Rightarrow (\exists y_2 \in Y) (y_1, \sigma, y_2) \in F] .$$

The definition essentially says that $\mathbf{S}$ must allow an uncontrollable event whenever such an event is eligible in $\mathbf{G}$. We now show the equivalence of automaton controllability to language controllability.

Proposition 29 *Let* $\mathbf{G}$ *and* $\mathbf{S} = (Y, \Sigma, F, y_0, Y_m)$ *be trim automata. Then* $\mathbf{S}$ *is controllable with respect to* $\mathbf{G}$ *if and only if* $L_m(\mathbf{S})$ *is controllable with respect to* $L_m(\mathbf{G})$.

Proof. We prove this by showing implication both ways.

(i) Let us assume that $\mathbf{S}$ is controllable with respect to $\mathbf{G}$. We have to show that this implies that $L_m(\mathbf{S})$ is controllable with respect to $L_m(\mathbf{G})$. Let $s \in \overline{L_m(\mathbf{G}) \cap L_m(\mathbf{S})}$ and let $\sigma \in \Sigma_u$ be such that $s\sigma \in \overline{L_m(\mathbf{G})}$. In order to show that $L_m(\mathbf{S})$ is controllable with respect to $\mathbf{G}$ we have to show that $s\sigma \in \overline{L_m(\mathbf{S})}$. Let $x_1 \in X$ and $y_1 \in Y$ be the states in $\mathbf{G}$ and $\mathbf{S}$ respectively corresponding to s. Since $s\sigma \in \overline{L_m(\mathbf{G})}$ it follows that there exists $x_2 \in X$ such that $(x_1, \sigma, x_2) \in E$. Now the controllability of $\mathbf{S}$ with respect to $\mathbf{G}$ implies that there exists a state $y_2 \in Y$ such that $(y_1, \sigma, y_2) \in F$ which implies that $s\sigma \in \overline{L_m(\mathbf{S})}$.

(ii) Let us assume that $L_m(\mathbf{S})$ is controllable with respect to $L_m(\mathbf{G})$. We have to show that this implies that $\mathbf{S}$ is controllable with respect to $\mathbf{G}$. Let $(x_1, y_1) \in \mathbf{G} \wedge \mathbf{S}$ and let $\sigma \in \Sigma_u$ and $x_2 \in X$ be such that $(x_1, \sigma, x_2) \in E$. In order to show that $\mathbf{S}$ is controllable with respect to $\mathbf{G}$ we have to show that there exists $y_2 \in Y$ such that $(y_1, \sigma, y_2) \in F$. Let s be any string in $L(\mathbf{G} \wedge \mathbf{S})$ corresponding to (x_1, y_1); then $s \in \overline{L_m(\mathbf{G}) \cap L_m(\mathbf{S})}$.

Since $(x_1, \sigma, x_2) \in E$ it follows that $s\sigma \in \overline{L_m(\mathbf{G})}$. Now the controllability of $L_m(\mathbf{S})$ with respect to $L_m(\mathbf{G})$ implies that $s\sigma \in \overline{L_m(\mathbf{S})}$. Thus there must exist $y_2 \in Y$ such that $(y_1, \sigma, y_2) \in F$. ■

Since isomorphic automata generate the same language we immediately get the next result.

Corollary 30 *Let* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *be isomorphic automata. Then an automaton* $\mathbf{S}$ *is controllable with respect to* $\mathbf{G}_1$ *if and only if it is controllable with respect to* $\mathbf{G}_2$.

We now define controllability in the presence of foreign event transitions.

Definition 31 *Let* $\mathbf{G}$ *and* $\mathbf{S} = \left(Y, \Sigma' \dot{\cup} \{\tau\}, F, y_o, Y_m\right)$ *be trim automata such that* $\Sigma' \subseteq \Sigma$. *We define* $\mathbf{S}$ *to be* locally controllable *with respect to* $\mathbf{G}$ *if*

$$(\forall (x_1, y_1) \in \mathbf{G} \parallel \mathbf{S}) \left(\forall \sigma \in \Sigma'_u\right) [(x_1, \sigma, x_2) \in E \Rightarrow (\exists y_2 \in Y)(y_1, \sigma, y_2) \in F].$$

As we shall shortly show, this notion of local controllability allows us to check the controllability of a specification by looking at its distribution instead. The specification automaton is distributed over a set of alphabets and the local controllability is checked over the distributed automata. If all of the distributed automata are locally controllable with respect to a plant then the specification automaton is controllable with respect to the same plant. There is no real advantage in doing this but it provides a glimpse into the symbolic supervision scheme introduced in Chapter 3. The symbolic supervision scheme also utilizes a distribution of the specification automaton.

Proposition 32 *Let* $\mathbf{S} = (Y, \Sigma, A, y_0, Y_m)$ *and let* $\left\{\mathbf{S}_i = \left(Y, \Sigma^i \cup \{\tau\}, A^i_l \cup A^i_f, y_0, Y_m\right) | i \in I\right\}$ *be a distribution of* $\mathbf{S}$ *over some set* $S = \{\Sigma^i | i \in I\}$ *of alphabets where* I *is a finite index*

set. Assume that $\cup S = \Sigma$. *If* $\mathbf{S}_i$ *is locally controllable with respect to* $\mathbf{G}$ *for all* $i \in I$ *then* $\mathbf{S}$ *is also controllable with respect to* $\mathbf{G}$.

Proof. Let $(x_1, y_1) \in \mathbf{G} \wedge \mathbf{S}$, $x_2 \in X$ and $\sigma \in \Sigma_u$ be such that $(x_1, \sigma, x_2) \in E$. In order to show that $\mathbf{S}$ is controllable with respect to $\mathbf{G}$ we have to show that there exists a state $y_2 \in Y$ such that $(y_1, \sigma, y_2) \in A$. Let $\sigma \in \Sigma^i$ for some $i \in I$. Since $\mathbf{S}^i$ is locally controllable with respect to $\mathbf{G}$, it follows that there must exist $y_2 \in Y$ such that $(y_1, \sigma, y_2) \in A_l^i \subseteq A$. ■

Given trim automata $\mathbf{G}$ and $\mathbf{S}$, let $\mathcal{C}(\mathbf{G}, \mathbf{S})$ be the set of all trim subautomata of $\mathbf{G} \wedge \mathbf{S}$ that are controllable with respect to $\mathbf{G}$, i.e.

$$\mathcal{C}(\mathbf{G}, \mathbf{S}) := \{\mathbf{S}' \leq \mathbf{G} \wedge \mathbf{S} | \mathbf{S}' \text{ is trim and controllable wrt } \mathbf{G}\}.$$

Proposition 33 *The set* $\mathcal{C}(\mathbf{G}, \mathbf{S})$ *is nonempty and its* join *is controllable with respect to* $\mathbf{G}$. *In other words,* $\vee\mathcal{C}(\mathbf{G}, \mathbf{S}) \in \mathcal{C}(\mathbf{G}, \mathbf{S})$.

Proof. Let $\mathbf{\Phi}$ represent the empty automaton. Since $\emptyset = L_m(\mathbf{\Phi})$ is controllable with respect to $L_m(\mathbf{G})$ it follows that $\mathbf{\Phi}$ is controllable with respect to $\mathbf{G}$. Since $\mathbf{\Phi} \leq \mathbf{G} \times \mathbf{S}$ we get $\mathbf{\Phi} \in \mathcal{C}(\mathbf{G}, \mathbf{S})$ which implies $\mathcal{C}(\mathbf{G}, \mathbf{S})$ is nonempty. From Proposition 32 we can conclude that $\vee\mathcal{C}(\mathbf{G}, \mathbf{S})$ is controllable with respect to $\mathbf{G}$. ■

We now show that the language generated by the supremal controllable subautomaton is the same as the supremal controllable sublanguage.

Proposition 34 *Let* $\mathbf{G}$ *and* $\mathbf{S}$ *be trim automata and let* $\mathbf{K} = \sup\mathcal{C}(\mathbf{G}, \mathbf{S})$ *be the supremal controllable subautomaton of* $\mathbf{G} \wedge \mathbf{S}$ *with respect to* $\mathbf{G}$. *Let* C *be the supremal controllable sublanguage of* $L_m(\mathbf{G}) \cap L_m(\mathbf{S})$ *with respect to* $L_m(\mathbf{G})$. *Then* $L_m(\mathbf{K}) = C$.

Proof. We prove this by showing language inclusion both ways.

(i) We want to show that $L_m(\mathbf{K}) \subseteq C$. Since $\mathbf{K}$ is controllable with respect to $\mathbf{G}$ it follows from Proposition 29 that $L_m(\mathbf{K}) \subseteq L_m(\mathbf{G}) \cap L_m(\mathbf{S})$ is controllable with respect to $L_m(\mathbf{G})$. This implies that $L_m(\mathbf{K}) \subseteq C$ since C is the supremal controllable sublanguage of $L_m(\mathbf{G}) \cap L_m(\mathbf{S})$ with respect to $L_m(\mathbf{G})$.

(ii) We want to show that $L_m(\mathbf{K}) \supseteq C$. Let $\mathbf{H} \leq \mathbf{G} \wedge \mathbf{S}$ be a trim automaton such that $L_m(\mathbf{H}) = C$. Such an automaton exists since $C \subseteq L_m(\mathbf{G}) \cap L_m(\mathbf{S})$. Since C is controllable with respect to $L_m(\mathbf{G}) \cap L_m(\mathbf{S})$, it follows from Proposition 29 that $\mathbf{H}$ is controllable with respect to $\mathbf{G}$. This implies that $\mathbf{H} \leq \mathbf{K}$ since $\mathbf{K}$ is the supremal controllable subautomaton of $\mathbf{G} \wedge \mathbf{S}$. Thus $C = L_m(\mathbf{H}) \subseteq L_m(\mathbf{K})$. ■

Let us recap what we have done so far. We have defined the notion of controllability of automata and shown it to be equivalent to the notion of controllability of languages. We have also presented a sufficient condition for controllability: if the entire distribution of an automaton is controllable then the automaton is also controllable.

3. Symbolic Supervision

3.1. An Overview

In this chapter we consider complex systems whose complexity is the result of the interactions of a number of subsystems. For instance, assume that a specification $\mathbf{S}$ is given for a plant $\mathbf{G} = \mathbf{G}_1 \| \cdots \| \mathbf{G}_n$ comprising n subsystems. Further assume that both the plant and the specification are defined over the same alphabet Σ. For such a system, we present a modular approach to checking the controllability of the specification with respect to the plant. The space complexity of checking the controllability of the specification with respect to the plant can be reduced from $O\left(\left|X^{G_i}\right|^n \left|X^S\right|\right)$ to $O\left(n \left|X^{G_i}\right| \left|X^S\right|\right)$. We also present a modular approach to the synthesis of the supremal controllable subautomaton of a specification automaton with respect to a plant automaton.

Our approach is automaton based and depends critically on the state set of the automaton representing the specification. We show that the controllability of a specification with respect to a plant can be verified by checking the controllability of a suitable distribution of the specification with respect to the same plant. In other words, we present a modular method to check the controllability of the specification with respect to the plant. Then we build on this modular method to present the main result of this chapter: modular synthesis of the supremal controllable subautomaton. We then show how

to implement a supervisor that enforces the behaviour of this supremal controllable sub-automaton on the plant. This implementation is intensional rather than extensional, i.e., the supervisor is not implemented using an explicit look-up table (see 4.2, page 80 for an illustration of the look-up table implementation of a supervisor). Rather than enumerating the entire state-space and specifying the control action at each state, we infer the control action based on local information. In this sense, our approach is symbolic in nature. The space required to store an explicit look-up table is $O\left(\left|X^{G_i}\right|^n \left|X^S\right| + \left|E^{G_i}\right|^n \left|E^S\right|\right)$ [Rud88]. In contrast, the space required to implement the proposed symbolic supervisor is $O\left(n\left|X^{G_i}\right|\left|X^S\right| + n\left|E^{G_i}\right|\left|E^S\right|\right)$. The time complexity to compute the look-up table is $O\left(\left|X^{G_i}\right|^n \left|X^S\right| + \left|E_u^{G_i}\right|^n \left|E_u^S\right|\right)$ while the time complexity to evaluate the symbolic supervisor for any given input is $O\left(n\left|E_u^{G_i}\right|\left|E_u^S\right|\right)$. So the proposed symbolic supervisor affords a substantial savings in space but requires a runtime computation. The time required for the runtime computation is proportional to the number of uncontrollable events. This confirms the general intuition that a system with fewer uncontrollable events should be easier to control than a system with more uncontrollable events.

The proposed approach does not guarantee nonblocking and this is its biggest drawback. However we extend our proposed approach to solve a lesser problem: deadlock avoidance. In general, even deadlock avoidance may be highly inefficient under the proposed scheme. This is an inherent property of deadlock avoidance [Gol78],[ASK77] and has little to do with the proposed approach. We offer a heuristic indicator (the number of controllable transitions) that may be used to decide whether the proposed scheme should be used for deadlock avoidance.

We present a number of examples that illustrate our approach.

3.2. Modular Checking of Controllability

In this section we assume that a plant is composed of independent components, i.e. components with no shared events. Under this assumption, we can check the controllability of a specification automaton with respect to a plant automaton by checking the controllability of a suitable distribution of the specification with respect to the component automata of the plant.

Lemma 35 *Let* $\mathbf{G}_1 = (X, \Sigma^1, E, x_0, X_m)$, $\mathbf{G}_2 = (Y, \Sigma^2, F, y_0, Y_m)$, $\mathbf{S} = (Z, \Sigma^1 \cup \Sigma^2, A, z_0, Z_m)$ *and assume that* $\Sigma^1 \cap \Sigma^2 = \emptyset$. *Let* $\left\{\mathbf{S}_i = \left(Z, \Sigma^i \cup \{\tau\}, A_l^i \cup A_f^i, z_0, Z_m\right) | i \in \{1,2\}\right\}$ *be the distribution of* $\mathbf{S}$ *over* $\{\Sigma^1, \Sigma^2\}$. *If* $\mathbf{S}_1$ *is locally controllable with respect to* $\mathbf{G}_1$ *and* $\mathbf{S}_2$ *is locally controllable with respect to* $\mathbf{G}_2$ *then* $\mathbf{S}$ *is controllable with respect to* $\mathbf{G}_1 \| \mathbf{G}_2$.

Proof. Let $(x_1, y_1, z_1) \in X^{G_1 \| G_2 \| S}$ and assume that there exists $\sigma \in \Sigma_u$ such that $((x_1, y_1), \sigma, (x_2, y_2)) \in E^{G_1 \| G_2}$. In order to show that $\mathbf{S}$ is controllable with respect to $\mathbf{G}_1 \| \mathbf{G}_2$ we need to show that there exists $z_2 \in Z$ such that $(z_1, \sigma, z_2) \in A$. Without loss of generality, let us assume that $\sigma \in \Sigma_1$. Then $(x_1, \sigma, x_2) \in E$. Since $\mathbf{S}_1$ is controllable with respect to $\mathbf{G}_1$ it follows that there must exist a state $z_2 \in Z$ such that $(z_1, \sigma, z_2) \in A_l^1 \subseteq A$.

■

Lemma 36 *Let* $\mathbf{G}_1 = (X, \Sigma^1, E, x_0, X_m)$, $\mathbf{G}_2 = (Y, \Sigma^2, F, y_0, Y_m)$, $\mathbf{S} = (Z, \Sigma^1 \cup \Sigma^2, A, z_0, Z_m)$ *and assume that* $\Sigma_1 \cap \Sigma_2 = \emptyset$. *Let* $\left\{\mathbf{S}_i = \left(Z, \Sigma^i \cup \{\tau\}, A_l^i \cup A_f^i, z_0, Z_m\right) | i \in \{1,2\}\right\}$ *be the distribution of* $\mathbf{S}$ *over* $\{\Sigma^1, \Sigma^2\}$. *If* $\mathbf{S}$ *is controllable with respect to* $\mathbf{G}_1 \| \mathbf{G}_2$ *then* $\mathbf{S}_1$ *is locally controllable with respect to* $\mathbf{G}_1$ *and* $\mathbf{S}_2$ *is locally controllable with respect to* $\mathbf{G}_2$.

Proof. We just have to show that $\mathbf{S}_1$ is controllable with respect to $\mathbf{G}_1$ since the proof can be repeated for $\mathbf{S}_2$ in an analogous manner. Let $(x_1, z_1) \in X^{G_1 \| S_1}$ and assume that there

exists $\sigma \in \Sigma_u^1$ such that $(x_1, \sigma, x_2) \in E$. In order to show that $\mathbf{S}_1$ is controllable with respect to $\mathbf{G}_1$ we need to show that there exists a state $z_2 \in Z$ such that $(z_1, \sigma, z_2) \in A_l^1$. Since $(x_1, z_1) \in X^{G_1 \| S_1}$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$, there must exist some $y_1 \in Y$ such that $(x_1, y_1, z_1) \in X^{G_1 \| G_2 \| S_1}$ which implies that $(x_1, y_1, z_1) \in X^{G_1 \| G_2 \| S}$. Since $\mathbf{S}$ is controllable with respect to $\mathbf{G}_1 \| \mathbf{G}_2$, there must exist a state $z_2 \in Z$ such that $(z_1, \sigma, z_2) \in A$. Since $A = A_l^1 \cup A_l^2$, $\sigma \in \Sigma_u^1 \subseteq \Sigma_1$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$, it cannot be the case that (z_1, σ, z_2) belongs to A_l^2 which implies that $(z_1, \sigma, z_2) \in A_l^1$. ■

We can now conclude the following result from Lemmas 35 and 36.

Theorem 37 *Let* $\mathbf{G}_1 = (X, \Sigma^1, E, x_0, X_m)$, $\mathbf{G}_2 = (Y, \Sigma^2, F, y_0, Y_m)$, $\mathbf{S} = (Z, \Sigma^1 \cup \Sigma^2, A, z_0, Z_m)$ *and assume that* $\Sigma_1 \cap \Sigma_2 = \emptyset$. *Let* $\left\{\mathbf{S}_i = \left(Z, \Sigma^i \cup \{\tau\}, A_l^i \cup A_f^i, z_0, Z_m\right) | i \in \{1, 2\}\right\}$ *be the distribution of* $\mathbf{S}$ *over* $\{\Sigma^1, \Sigma^2\}$. *Then* $\mathbf{S}$ *is controllable with respect to* $\mathbf{G}_1 \| \mathbf{G}_2$ *if and only if* $\mathbf{S}_1$ *is locally controllable with respect to* $\mathbf{G}_1$ *and* $\mathbf{S}_2$ *is locally controllable with respect to* $\mathbf{G}_2$.

Corollary 38 *Let* $\mathbf{G}_i = (X^i, \Sigma^i, E^i, x_0^i, X_m^i)$ *and* $\mathbf{S} = (Z, \cup_{i \in I} \Sigma^i, A, z_0, Z_m)$ *be trim automata for all* i *in some index set* I. *Assume that* $\Sigma^i \cap \Sigma^j = \emptyset$ *for all* $i, j \in I$ *whenever* $i \neq j$. *Let* $\left\{\mathbf{S}_i = \left(Z, \Sigma^i \cup \{\tau\}, A_l^i \cup A_f^i, z_0, Z_m\right) | i \in I\right\}$ *be the distribution of* $\mathbf{S}$ *over* $\{\Sigma^i | i \in I\}$. *Then* $\mathbf{S}$ *is controllable with respect to* $\|_{i \in I} \mathbf{G}_i$ *if and only if* $\mathbf{S}_i$ *is locally controllable with respect to* $\mathbf{G}_i$ *for all* $i \in I$.

So it can be seen from Corollary 38 that the task of checking controllability can be greatly reduced by utilizing this modular scheme. If we assume that $|X^i| = m$, $|Z| = s$, and $|I| = n$ then we can break down the task of checking controllability of a specification (of size s) with respect to a plant (of size m^n) into n subtasks of checking controllability of specifications (of size s each) with respect to components of size m each. Thus the space complexity can be reduced from $O(m^n s)$ to $O(mns)$.

Example 39 *Let*

$$\begin{aligned}
\mathbf{G}_1 &= (\{x_0, x_1\}, \{\alpha_1, \beta_1\}, \{(x_0, \alpha_1, x_1), (x_1, \beta_1, x_0)\}, x_0, \{x_0\}), \\
\mathbf{G}_2 &= (\{y_0, y_1\}, \{\alpha_2, \beta_2\}, \{(y_0, \alpha_2, y_1), (y_1, \beta_2, y_0)\}, y_0, \{y_0\}), \\
\mathbf{G}_3 &= (\{z_0, z_1\}, \{\alpha_3, \beta_3\}, \{(z_0, \alpha_3, z_1), (z_1, \beta_3, z_0)\}, z_0, \{z_0\})
\end{aligned}$$

represent the three machines of a small factory [Won01] as shown in Figure 3.1. For $1 \leq i \leq 3$, *the event* α_i *represents machine* $\mathbf{G}_i$ *starting its work and the event* β_i *represents* $\mathbf{G}_i$ *finishing its work. Assume that machines* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *take raw products from an infinite source and process them. Upon finishing they deposit the partially finished product into a one slot buffer. The machine* $\mathbf{G}_3$ *then takes this partially finished product and applies finishing touches to it and deposits the final product into an infinite sink. It is desired that the buffer should neither overflow nor underflow. An automaton* $\mathbf{S}$ *representing this specification is shown in Figure 3.2. Distributing* $\mathbf{S}$ *over the alphabets of* $\mathbf{G}_1$, $\mathbf{G}_2$ *and* $\mathbf{G}_3$ *gives the local specification automata* $\mathbf{S}_1$, $\mathbf{S}_2$ *and* $\mathbf{S}_3$ *shown in Figure 3.3. The local specifications* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *may be roughly interpreted as follows: if the buffer is full (the local specification is in state* w_1*) either due to a local or a foreign action, do not deposit anything in it until a foreign event clears the buffer (and brings the local specification into state* w_0*). Similarly the local specification* $\mathbf{S}_3$ *may be roughly interpreted as follows: do not start operation until a foreign event deposits a product into the buffer (and brings the local specification into state* w_1*). The synchronous products of* $\mathbf{G}_1$, $\mathbf{G}_2$ *and* $\mathbf{G}_3$ *with* $\mathbf{S}_1$, $\mathbf{S}_2$ *and* $\mathbf{S}_3$ *respectively are shown in Figure 3.4. Let us assume* $\alpha_1, \alpha_2, \alpha_3$ *to be controllable and* $\beta_1, \beta_2, \beta_3$ *to be uncontrollable. Then the automaton* $\mathbf{S}_1$ *is not locally controllable with respect to* $\mathbf{G}_1$ *because* β_1 *is eligible to occur at state* x_1 *in* $\mathbf{G}_1$ *but not eligible to occur at state* (x_1, w_1) *in* $\mathbf{G}_1 \| \mathbf{S}_1$. *As can be seen,* (x_1, w_1) *is unreachable in* $\mathbf{G}_1 \| \mathbf{S}_1$ *via any sequence of the local events* α_1 *and* β_1*; it is*

reachable only upon the occurrence of a foreign event (which represents the occurrence of β_2 *in* $\mathbf{G}_2$*). Similarly,* $\mathbf{S}_2$ *is not locally controllable with respect to* $\mathbf{G}_2$*. So despite the fact that* $\mathbf{S}_3$ *is locally controllable with respect to* $\mathbf{G}_3$ *it turns out that* $\mathbf{S}$ *is uncontrollable with respect to* $\mathbf{G}_1 \| \mathbf{G}_2 \| \mathbf{G}_3$*. A physical explanation of this can perhaps be given in the following manner. The behaviours of* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *can be "bumped away" from their desired behaviours (defined by* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *respectively) due to the actions of other machines while* $\mathbf{G}_3$ *can maintain its desired behaviour (defined by* $\mathbf{S}_3$*) despite the actions of other machines. Since* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *cannot maintain their desired behaviours it follows that* $\mathbf{G}_1 \| \mathbf{G}_2 \| \mathbf{G}_3$ *cannot maintain its desired behaviour which is defined by* $\mathbf{S}_1 \vee \mathbf{S}_2 \vee \mathbf{S}_3$*.* □

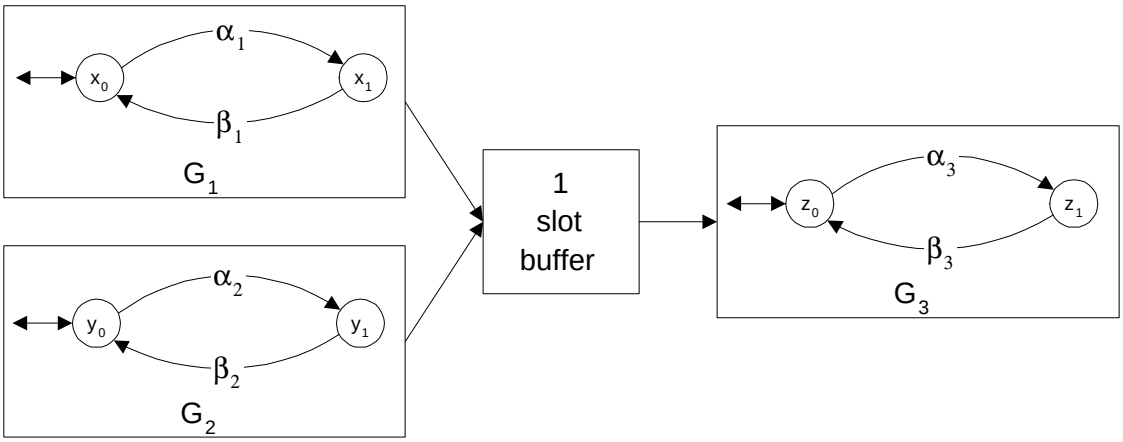


Figure 3.1: Small Factory

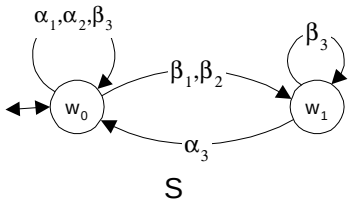


Figure 3.2: Buffer under/overflow specification

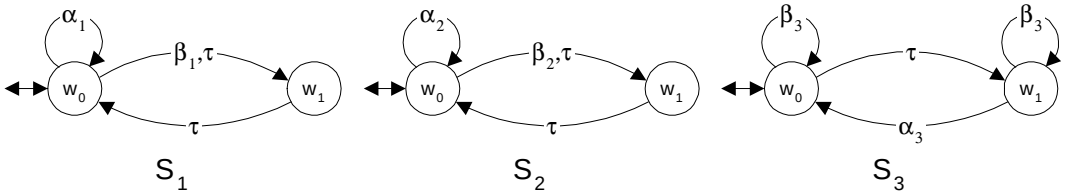


Figure 3.3: Distributed buffer specifications

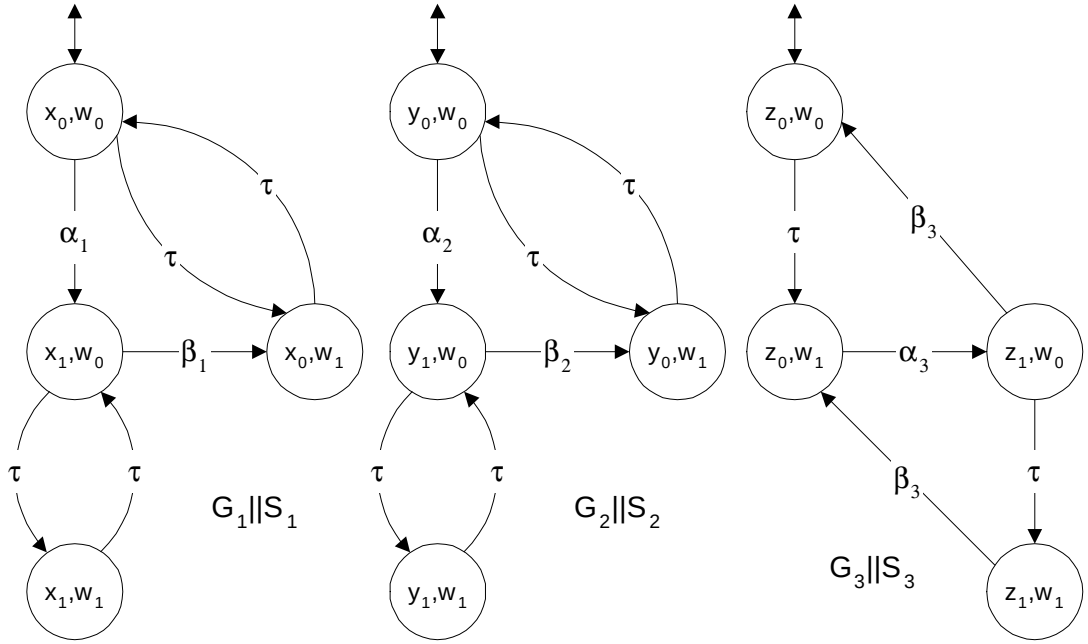


Figure 3.4: Synchronous product of distributed specifications with respective plant components

3.3. Synthesis of Supremal Controllable Subautomaton

In the previous section we showed how we can check controllability in a modular manner. If the given specification turns out to be uncontrollable, we are interested in computing the supremal controllable subautomaton of the given specification automaton. Assume that a plant is represented as the synchronous composition of n automata of m states each and assume that the specification automaton has s states. Then the space complexity of storing

the resultant synchronous product automaton is $O\left(m^n s + e\right)$ where e represents the number of transitions in this automaton. Since the automaton is reachable it has at least as many transitions as the number of states minus one although it will generally have a lot more. The time complexity of computing its supremal controllable subautomaton with respect to the plant is $O\left(m^n s + \sigma_u e\right)$ [Rud88] where σ_u is the number of uncontrollable events in the alphabet. Since both these complexities grow exponentially with n (the number of plant components), it is highly desirable to come up with a modular scheme for synthesizing the supremal controllable subautomaton. In this section we show how that may be done.

Assume that $\mathbf{G} = (X, \Sigma, E, x_o, X_m)$ is a plant automaton while $\mathbf{S} = (Y, \Sigma, F, y_0, Y_m)$ is a specification automaton. In this section we only consider safety properties so we assume that $X = X_m$ and $Y = Y_m$. We are interested in the case where the plant comprises n components $\mathbf{G}_i = \left(X^i, \Sigma^i, E^i, x_0^i, X_m^i\right)$, $1 \leq i \leq n$, such that $\mathbf{G} = \mathbf{G}_1 \| \cdots \| \mathbf{G}_n$. We assume that $\Sigma^i \cap \Sigma^j = \emptyset$ whenever $i \neq j$. Let $\left\{\mathbf{S}_i = \left(Y, \Sigma^i \cup \{\tau\}, F_l^i \cup F_f^i, y_0, Y_m\right) | 1 \leq i \leq n\right\}$ be the distribution of $\mathbf{S}$ over $\left\{\Sigma^i | 1 \leq i \leq n\right\}$.

Definition 40 *A state $(x, y) \in X^{G\|S}$ is a* primary bad state *of $\mathbf{G} \parallel \mathbf{S}$ if there exist $\sigma \in \Sigma_u$ and $x_1 \in X$ such that $(x, \sigma, x_1) \in E$ but there exists no $y_1 \in Y$ such that $(y, \sigma, y_1) \in F$. Let $PB^{G\|S}$ represent the set of all primary bad states of $\mathbf{G} \parallel \mathbf{S}$.*

Definition 41 *A state $(x, y) \in X^{G\|S}$ is a* secondary bad state *of $\mathbf{G} \parallel \mathbf{S}$ if there exists a primary bad state in $\mathbf{G} \parallel \mathbf{S}$ that is reachable from (x, y) via an uncontrollable path (where each transition is due to an uncontrollable event).*

Definition 42 *A state $(x, y) \in X^{G\|S}$ is a* bad state *of $\mathbf{G} \parallel \mathbf{S}$ if it is either a primary bad state or a secondary bad state. Let $B^{G\|S}$ represent the set of all bad states of $\mathbf{G} \parallel \mathbf{S}$.*

A state in the synchronous product of a plant and a specification is a primary bad state

if the specification prohibits the plant from performing an uncontrollable transition at that state. By its very nature, an uncontrollable event cannot be prohibited from occurring so it follows that its occurrence causes the plant to exhibit undesirable behaviour. In other words, once a system reaches a primary bad state it may exhibit behaviour that violates the given specification. A primary bad state is reachable from a secondary bad state via the occurrence of a sequence of uncontrollable events. Therefore if the desired behaviour is to be maintained, a supervisor should take control actions (i.e. disable appropriate events) so that all the bad states are unreachable [Rud88]. It turns out that the supremal controllable subautomaton of $\mathbf{G} \parallel \mathbf{S}$ can be characterized completely in terms of the bad states of $\mathbf{G} \parallel \mathbf{S}$.

Lemma 43 *Let* $\mathbf{C} = (X \times Y, \Sigma, H, (x_0, y_0), X_m \times Y_m)_{rch}$ *be the subautomaton of* $\mathbf{G} \parallel \mathbf{S}$ *where*

$$H = E^{G\|S} - \left\{(x, \sigma, x_b) \mid \ (x, \sigma, x_b) \in E^{G\|S} \wedge x_b \in B^{G\|S}\right\}.$$

Then $\mathbf{C}$ *is the supremal controllable subautomaton of* $\mathbf{G} \parallel \mathbf{S}$ *with respect to* $\mathbf{G}$.

Proof. We prove this in two parts: we first show that $\mathbf{C}$ is controllable with respect to $\mathbf{G}$ and then show that if any automaton $\mathbf{C}' \leq \mathbf{G} \parallel \mathbf{S}$ is controllable with respect to $\mathbf{G}$ then $\mathbf{C}' \leq \mathbf{C}$. This allows us to conclude the desired result since $\mathbf{C}$ is trim by construction.

Part 1: Let $(x, x, y) \in X^{G\|C}$ and let $\sigma \in \Sigma_u$ be such that $(x, \sigma, x_1) \in E$ for some $x_1 \in X$. In order to show that $\mathbf{C}$ is controllable with respect to $\mathbf{G}$, we need to show that there exists $y_1 \in Y$ such that $((x, y), \sigma, (x_1, y_1)) \in H$. However by the definition of H, if $((x, y), \sigma, (x_1, y_1))$ does not belong to H then either it does not belong to $E^{G\|S}$ or (x_1, y_1) is a bad state in $\mathbf{G} \parallel \mathbf{S}$.

Case 1: Let us consider the case where $((x, y), \sigma, (x_1, y_1))$ does not belong to $E^{G\|S}$. This would imply that (x, y) is a bad state of $\mathbf{G} \parallel \mathbf{S}$ since an uncontrollable event,

σ, is ineligible here. Thus it should be unreachable in $\mathbf{C}$ by the definition of H. If (x, y) is unreachable in $\mathbf{C}$ then $(x, x, y) \notin X^{G\|C}$ which is contradictory to our assumption.

Case 2: Now let us consider the case where (x_1, y_1) is a bad state in $\mathbf{G} \parallel \mathbf{S}$. Then from the definition of a bad state it would follow that (x, y) is a bad state of $\mathbf{G} \parallel \mathbf{S}$. As in Case 1, that would imply that (x, y) is unreachable in $\mathbf{C}$ and $(x, x, y) \notin X^{G\|C}$ which is contradictory to our assumption.

Thus from the above two cases we can conclude that $((x, y), \sigma, (x_1, y_1)) \in H$ which implies that $\mathbf{C}$ is controllable with respect to $\mathbf{G}$.

Part 2: Let $\mathbf{C}' \leq \mathbf{G} \parallel \mathbf{S}$ be controllable with respect to $\mathbf{G}$. We need to show that $\mathbf{C}' \leq \mathbf{C}$. Since $\mathbf{C}'$ is controllable with respect to $\mathbf{G}$ it follows that it cannot have any bad states of $\mathbf{G} \parallel \mathbf{S}$, i.e.

$$X^{C'} \cap B^{G\|S} = \emptyset$$

which implies that

$$E^{C'} \cap \left\{(x, \sigma, x_b) \,|\, (x, \sigma, x_b) \in E^{G\|S} \wedge x_b \in B^{G\|S}\right\} = \emptyset.$$

Since $\mathbf{C}$ contains all the reachable non-bad states of $\mathbf{G} \parallel \mathbf{S}$, it follows that $X^{C'} \subseteq X^C$ and $E^{C'} \subseteq E^C$. Since $x_0^C = x_0^{C'} = x_0^G$ and $\Sigma^C = \Sigma^{C'} = \Sigma$ we can conclude that $\mathbf{C}' \leq \mathbf{C}$. ■

This characterization of the supremal controllable subautomaton of $\mathbf{G} \parallel \mathbf{S}$ in terms of the bad states of $\mathbf{G} \parallel \mathbf{S}$ allows us to define the control action needed to synthesize this subautomaton.

Definition 44 *Let* $\mathbf{K}$ *be an automaton and let* $V_H : X^K \to 2^{\Sigma^K}$ *be a partial map. Let an automaton* $\mathbf{H}$ *be defined as follows:*

$$\mathbf{H} = \left(X^K, \Sigma^K, H, x_0^K, X_m^K\right)_{rch}$$

where

$$H = E^K - \left\{(x, \sigma, x_1) \,|\, (x, \sigma, x_1) \in E^K \wedge \sigma \in V_H(x)\right\}.$$

Then $\mathbf{H}$ *is called the subautomaton of* $\mathbf{K}$ induced under the map V_H. *If* V_H *maps* X^K *into* $2^{\Sigma_c^K}$ *then it is called a* disablement *map.*

A disablement map asks for the disablement of controllable events only and can therefore be used to specify the control action needed to synthesize a controllable automaton.

Lemma 45 *Let* $V_C : X^{G\|S} \to 2^{\Sigma_c}$ *be a disablement map defined as follows:*

$$\left(\forall x \in X^{G\|S}\right)\left(\forall \sigma \in \Sigma_c\right)\left[\begin{array}{c} \sigma \in V_C(x) \\ \Updownarrow \\ \left(\exists x_b \in X^{G\|S}\right)\left((x, \sigma, x_b) \in E^{G\|S} \wedge x_b \in B^{G\|S}\right) \end{array}\right].$$

Let $\mathbf{C}$ *be the subautomaton of* $\mathbf{G} \parallel \mathbf{S}$ *induced under* V_C. *If* $x_0^{G\|S}$ *is not a bad state then* $\mathbf{C}$ *is the supremal controllable subautomaton of* $\mathbf{G} \parallel \mathbf{S}$ *with respect to* $\mathbf{G}$.

Proof. The disablement map V_C requires the disablement of a controllable transition if and only if the transition leads to a bad state. The initial state is presumed to be not a bad state. Thus the transition set of $\mathbf{C}$ includes all the transitions of $\mathbf{G} \parallel \mathbf{S}$ except those that lead to bad states. Now Lemma 43 gives the desired result. ■

The control action needed to synthesize the supremal controllable subautomaton involves disabling those transitions that lead to bad states. In other words, a transition is disabled if it leads to a state from which a primary bad state may be reached via an uncontrollable path.

Definition 46 *Let* $\mathbf{K}$ *be an automaton and let* $x \in X^K$. *Then*

$$U(x) := \left\{ x' \in X^K \middle| \left(\exists s \in \Sigma_u^{K*} \right) \left[\eta(x, s) = x' \right] \right\}$$

is defined to be the uncontrollable span *of state* x.

Now the above lemma may be restated as follows.

Lemma 47 *Let* $V_C : X^{G\|S} \to 2^{\Sigma_c}$ *be a disablement map defined as follows:*

$$\left(\forall x \in X^{G\|S} \right) \left(\forall \sigma \in \Sigma_c \right) \left[\sigma \in V_C(x) \Leftrightarrow \left(\exists x_b \in U(x) \right) \left(x_b \in PB^{G\|S} \right) \right].$$

Let $\mathbf{C}$ *be the subautomaton of* $\mathbf{G} \parallel \mathbf{S}$ *induced under* V_C. *If* $x_0^{G\|S}$ *is not a bad state then* $\mathbf{C}$ *is the supremal controllable subautomaton of* $\mathbf{G} \parallel \mathbf{S}$ *with respect to* $\mathbf{G}$.

This lemma provides an inefficient but simple method for the computation of the disablement map that induces the supremal controllable subautomaton: At each state, simply disable those events that lead to states whose uncontrollable span includes a primary bad state. This method requires the complete knowledge of $\mathbf{G} \parallel \mathbf{S}$. Additionally, it requires a search at each state to check whether any successor state contains a primary bad state in its uncontrollable span. As mentioned earlier, we are interested in the scenario where the plant is represented as the synchronous composition of n automata of m states each and the specification automaton has s states. We would like to be able to compute the

disablement map V_C without having to perform the synchronous composition. This should be possible if we are able to compute uncontrollable spans and identify primary bad states without performing the synchronous composition. We now show how that may be done.

Definition 48 *Let* $x = (x^1, \ldots, x^n, s) \in X^{G\|S}$. *Then the* iterative uncontrollable span *of* x *is defined as follows:*

$$
\begin{aligned}
&IU(x) \quad := \{x\} \\
&\textit{for} \qquad i := 1 \textit{ to } n \\
&\qquad\quad temp := IU\,(x) \\
&\qquad\quad IU\,(x) := temp \cup \left\{ \begin{array}{c} \left(x_1^1, \ldots, x_1^{i-1}, x_2^i, x_1^{i+1}, \ldots, x_1^n, s_2\right) \in X^{G\|S} | \\ \left(\exists\,(x_1^1, \ldots, x_1^n, s_1) \in IU(x)\right)\left(\exists w \in \Sigma_u^{i*}\right) \\ \left[\eta^{G_i\|S_i}\left((x_1^i, s_1), w\right) = (x_2^i, s_2)\right] \end{array} \right\} \\
&\textit{end} \qquad \textit{for}
\end{aligned}
$$

The iterative uncontrollable span of a state in $\mathbf{G}\|\mathbf{S}$ is computed using all the $\mathbf{G}_i\|\mathbf{S}_i$. The process begins by initializing the span with x and computing all the states that may be reachable from x via uncontrollable events from Σ^1. These states, along with x, form a partial span. Now all those states are computed that may be reachable from any of the states of this partial span via uncontrollable events from Σ^2, and so on. After the n^{th} iteration we get the complete iterative uncontrollable span of x. We will shortly show that the iterative uncontrollable span of a state is equal to its uncontrollable span. But first we present a result that will be useful in the proof.

Lemma 49 *Let* $\mathbf{J} = \mathbf{J}_1\|\mathbf{J}_2$ *be a plant automaton and let* $\mathbf{K}$ *be a specification automaton. Let* $(x_1, y_1, z_1), (x_2, y_2, z_2) \in X^{J_1\|J_2\|K}$ *and let* s *be an uncontrollable path from* (x_1, y_1, z_1) *to* (x_2, y_2, z_2). *Assume that* $\Sigma_u^{J_1} \cap \Sigma_u^{J_2} = \emptyset$. *If* (x_1, y_1, z_1) *is not a bad state in* $\mathbf{J}_1\|\mathbf{J}_2\|\mathbf{K}$

then there exists an uncontrollable path s' from (x_1, y_1, z_1) to (x_2, y_2, z_2) such that $s' = s_1 s_2$ where $s_1 \in \Sigma_u^{J_1 *}$ and $s_2 \in \Sigma_u^{J_2 *}$.

Proof. Let $s = \sigma_1 \cdots \sigma_m$. Let $\sigma_1 \cdots \sigma_i$ be the longest prefix of s belonging to $\Sigma_u^{J_1 *}$. If there does not exist a $\dot{j} > i$ such that $\sigma_j \in \Sigma_u^{J_1 *}$ then the lemma is trivially true. So let us assume that there does exist a $j > i$ such that $\sigma_j \in \Sigma_u^{J_1 *}$. Let (x_3, y_1, z_3) and (x_3, y_4, z_4) be the states that are reached from (x_1, y_1, z_1) via the strings $\sigma_1 \cdots \sigma_i$ and $\sigma_1 \cdots \sigma_{j-1}$ respectively. Since σ_j is eligible at (x_3, y_4, z_4) it follows that it must also be eligible at (x_3, y_1, z_3) because if it were not then (x_3, y_1, z_3) would be a bad state (which would imply that (x_1, y_1, z_1) is a bad state). A similar argument tells us that $\sigma_1 \cdots \sigma_i \sigma_j \sigma_{i+1} \cdots \sigma_{j-1} \sigma_{j+1} \cdots \sigma_m$ must be a path from $(x_1, y_{1,} z_1)$ to (x_2, y_2, z_2). The whole process can now be repeated, if needed, to get the desired result. ■

Corollary 50 *Let $(x_1^1, \ldots, x_1^n, z_1), (x_2^1, \ldots, x_2^n, z_2) \in X^{G\|S}$ and let s be an uncontrollable path from $(x_1^1, \ldots, x_1^n, z_1)$ to $(x_2^1, \ldots, x_2^n, z_2)$. Assume that $\Sigma_u^{J_1} \cap \Sigma_u^{J_2} = \emptyset$. If $(x_1^1, \ldots, x_1^n, z_1)$ is not a bad state in $\mathbf{G}\|\mathbf{S}$ then there exists an uncontrollable path $s' = s_1 \cdots s_n$ from $(x_1^1, \ldots, x_1^n, z_1)$ to $(x_2^1, \ldots, x_2^n, z_2)$ such that $s_i \in \Sigma_u^{i*}$ for $1 \leq i \leq n$.*

This result tells us that if a state is reachable via a sequence of uncontrollable events then that state is also reachable using a permutation of those events such that all the events belonging to a subalphabet are clustered together. This result relies on the fact that the subalphabets share no common events.

Lemma 51 *Assume that $x = (x^1, \ldots, x^n, s) \in X^{G\|S}$ is not a bad state in $\mathbf{G}\|\mathbf{S}$. Then $U(x) = IU(x)$, i.e. the uncontrollable span of x is the same as its iterative uncontrollable span.*

Proof. We prove this by set inclusion both ways.

Part 1: $(\mathbf{IU}(x) \subseteq \mathbf{U}(x))$: Let $y \in IU(x)$. Then from Definition 48 we can conclude that there must exist a path $s_1 \cdots s_n \in \Sigma_u^*$ from x to y such that $s_i \in \Sigma_u^{i*}$. Since $s_1 \cdots s_n \in \Sigma_u^*$, from Definition 46 it follows that $y \in U(x)$.

Part 2: $(\mathbf{U}(x) \subseteq \mathbf{IU}(x))$: Let $y \in U(x)$. Then there must exist a path $s \in \Sigma_u^*$ from x to y. From Corollary 50 we can conclude that there must exist a path $s_1 \cdots s_n \in \Sigma_u^*$ from x to y such that $s_i \in \Sigma_u^{i*}$. Thus from Definition 48 it follows that $y \in IU(x)$. ■

Lemma 52 *Let $x = (x^1, \ldots, x^n, z) \in X^{G\|S}$ be a primary bad state in $\mathbf{G}\|\mathbf{S}$. Then $(x^i, z) \in X^{G_i\|S_i}$ must be a primary bad state in $\mathbf{G}_i\|\mathbf{S}_i$ for some $1 \leq i \leq n$.*

Proof. Let $\sigma \in \Sigma_u$ be such that $((x^1, \ldots, x^n), \sigma, (x_1^1, \ldots, x_1^n)) \in E^G$ for some $(x_1^1, \ldots, x_1^n) \in X^G$ but $(z, \sigma, z_1) \notin E^S$ for any $z_1 \in X^S$. Assume that $\sigma \in \Sigma^i$ for some $1 \leq i \leq n$. Since $(z, \sigma, z_1) \notin E^S$ it follows that $(z, \sigma, z_1) \notin E^{S_i}$. But $(x^i, \sigma, x_1^i) \in E^{G_i}$ which implies that (x^i, z) is a primary bad state in $\mathbf{G}_i\|\mathbf{S}_i$. ■

Lemma 53 *Let $(x^i, z) \in X^{G_i\|S_i}$ be a primary bad state in $\mathbf{G}_i\|\mathbf{S}_i$ for some $1 \leq i \leq n$. If $x \in X^{G\|S}$ such that $x = (\ldots, x^i, \ldots, z)$ then x is a primary bad state in $\mathbf{G}\|\mathbf{S}$.*

Proof. Let $\sigma \in \Sigma_u^i$ be such that $(x^i, \sigma, x_1^i) \in E^{G_i}$ for some $x_1^i \in X^{G_i}$ but $(z, \sigma, z_1) \notin E^{S_i}$ for any $z_1 \in X^{S_i}$. Then it must be the case that $(z, \sigma, z_1) \notin E^S$ for any $z_1 \in X^S$. Thus if there exists a state $x \in X^{G\|S}$ such that $x = (\ldots, x^i, \ldots, z)$ then it must be a bad state in $\mathbf{G}\|\mathbf{S}$. ■

Corollary 54 *Any state $x = (x^1, \ldots, x^n, z) \in X^{G\|S}$ is a primary bad state in $\mathbf{G}\|\mathbf{S}$ if and only if $(x^i, z) \in X^{G_i\|S_i}$ is a primary bad state in $\mathbf{G}_i\|\mathbf{S}_i$ for some $1 \leq i \leq n$.*

We have presented modular ways to identify the primary bad states of $\mathbf{G}\|\mathbf{S}$ and compute the uncontrollable span of any of its states. We now illustrate the concept with the help of an example.

Example 55 *Let us reconsider the setup of Example 39. It was shown there that the buffer underflow/overflow specification is uncontrollable. The automaton* $\mathbf{G} = \mathbf{G}_1\|\mathbf{G}_2\|\mathbf{G}_3$ *representing the concurrent operation of the three machines* $\mathbf{G}_1$, $\mathbf{G}_2$ *and* $\mathbf{G}_3$ *is shown in Figure 3.5. The automaton* $\mathbf{G}\|\mathbf{S}$ *is partly shown in Figure 3.6. Let* $\mathbf{C}$ *be the supremal controllable subautomaton of* $\mathbf{G}\|\mathbf{S}$ *with respect to* $\mathbf{G}$. *Now* (x_1, y_0, z_0, w_1) *is a primary bad state in* $\mathbf{G}\|\mathbf{S}$ *because* β_1 *is ineligible there while it is eligible at* (x_1, y_0, z_0) *in* $\mathbf{G}$. *Since it is reachable from* (x_1, y_1, z_0, w_0) *via the uncontrollable event* β_2 *it follows that* (x_1, y_1, z_0, w_0) *is a secondary bad state in* $\mathbf{G}\|\mathbf{S}$. *Neither of these states can belong to* $\mathbf{C}$ *so it follows that* $\alpha_1 \in V_C((x_0, y_1, z_0, w_0))$ *and* $\alpha_2 \in V_C((x_1, y_0, z_0, w_0))$. *Intuitively this may be interpreted as follows. Since the buffer has only one slot,* $\mathbf{G}_1$ *should not start if* $\mathbf{G}_2$ *has already started, and vice-versa. Now let us see how these control actions may be derived in a modular manner.*

The only primary bad state of $\mathbf{G}_1\|\mathbf{S}_1$ *is* (x_1, w_1). *This implies that all global states of the form* $(x_1, \_, \_, w_1)$ *are primary bad states. Similarly, the only primary bad state of* $\mathbf{G}_2\|\mathbf{S}_2$ *is* (y_1, w_1). *Therefore all global states of the form* $(\_, y_1, \_, w_1)$ *are primary bad states. These are the only primary bad states since* $\mathbf{G}_3\|\mathbf{S}_3$ *has no primary bad states.*

The initial states of $\mathbf{G}_1\|\mathbf{S}_1$, $\mathbf{G}_2\|\mathbf{S}_2$, *and* $\mathbf{G}_3\|\mathbf{S}_3$ *are* (x_0, w_0), (y_0, w_0) *and* (z_0, w_0) *respectively. This corresponds to the global state* (x_0, y_0, z_0, w_0). *Since this is not a bad state, we may implement our supervision scheme.*

The transition $((x_0, w_0), \alpha_1, (x_1, w_0))$ *may take place in* $\mathbf{G}_1\|\mathbf{S}_1$ *causing the global state to change to* (x_1, y_0, z_0, w_0). *The uncontrollable span of* (x_1, y_0, z_0, w_0) *may be computed as*

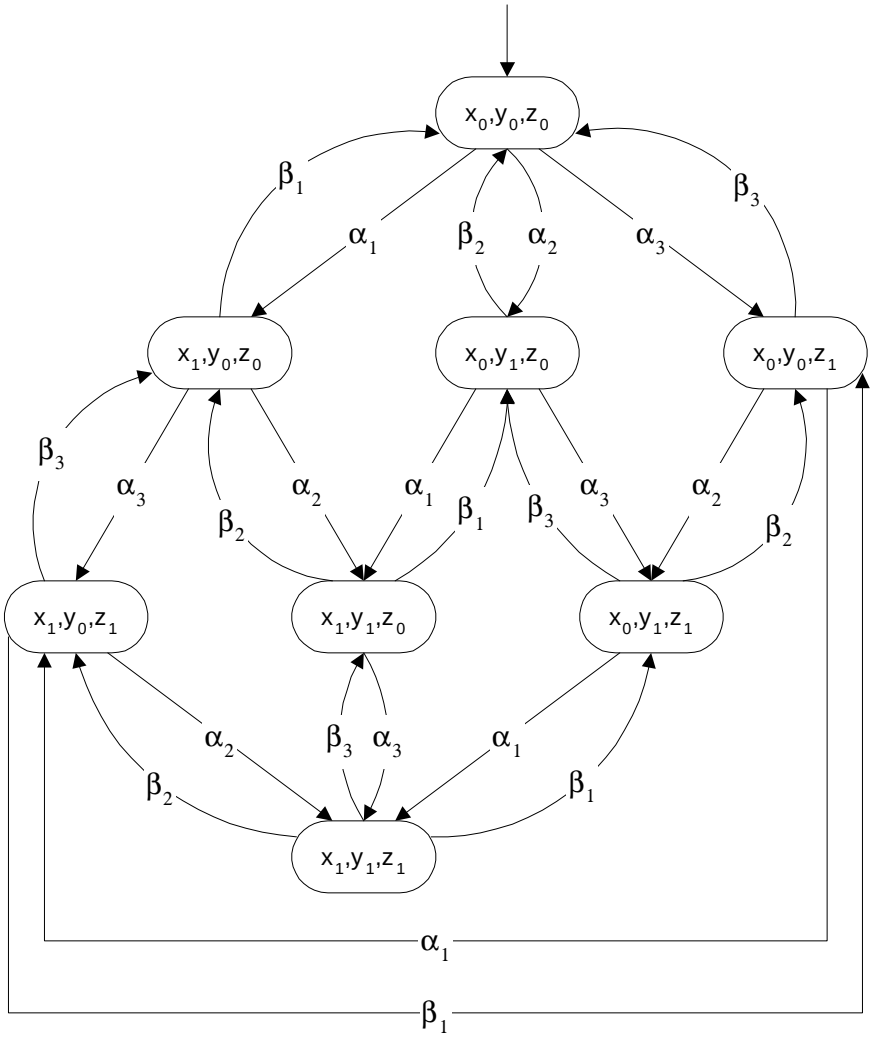


Figure 3.5: $\mathbf{G}_1 \| \mathbf{G}_2 \| \mathbf{G}_3$

follows:

Iteration 1: $U((x_1, y_0, z_0, w_0)) := \{(x_1, y_0, z_0, w_0)\} \cup \{(x_0, y_0, z_0, w_1)\}$

Iteration 2: $U((x_1, y_0, z_0, w_0)) := \{(x_1, y_0, z_0, w_0), (x_0, y_0, z_0, w_1)\} \cup \emptyset$

Iteration 3: $U((x_1, y_0, z_0, w_0)) := \{(x_1, y_0, z_0, w_0), (x_0, y_0, z_0, w_1)\} \cup \emptyset$

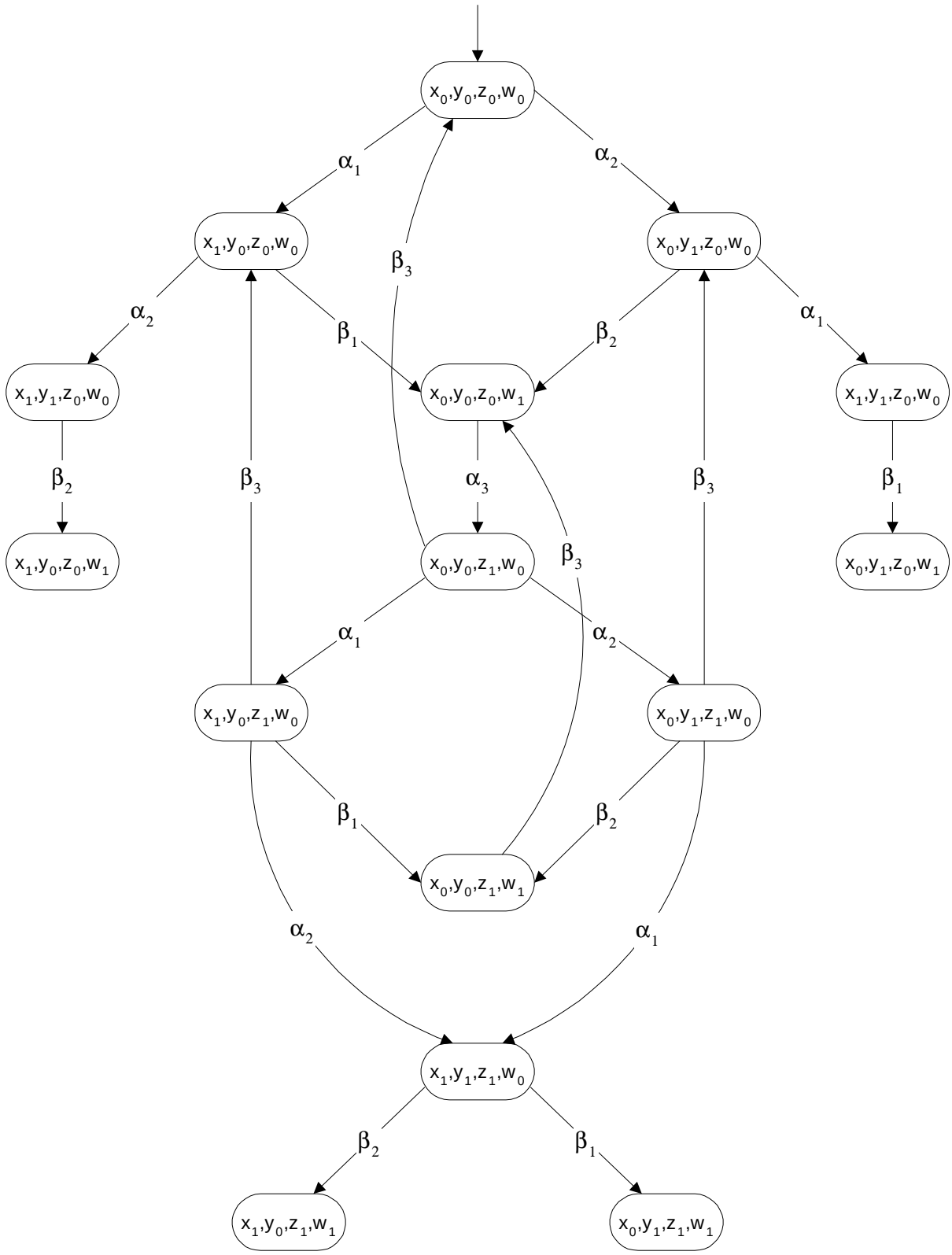


Figure 3.6: $\mathbf{G}_1 \| \mathbf{G}_2 \| \mathbf{G}_3 \| \mathbf{S}$

Since $U((x_1, y_0, z_0, w_0)) = \{(x_1, y_0, z_0, w_0), (x_0, y_0, z_0, w_1)\}$ *does not contain any primary bad states, it follows that* α_1 *does not need to be disabled. Now assume that* α_1 *does indeed occur and the global state becomes* (x_1, y_0, z_0, w_0). *There is no eligible controllable event in* $\mathbf{G}_1 \| \mathbf{S}_1$ *so no control needs to be computed here. However the transition* $((y_0, w_0), \alpha_2, (y_1, w_0))$ *may occur in* $\mathbf{G}_2 \| \mathbf{S}_2$ *causing the global state to change to* (x_1, y_1, z_0, w_0). *The uncontrollable span of* (x_1, y_1, z_0, w_0) *may be computed as follows:*

Iteration 1: $U((x_1, y_1, z_0, w_0)) := \{(x_1, y_1, z_0, w_0)\} \cup \{(x_0, y_1, z_0, w_1)\}$

Iteration 2: $U((x_1, y_1, z_0, w_0)) := \{(x_1, y_1, z_0, w_0), (x_0, y_1, z_0, w_1)\} \cup \{(x_1, y_0, z_0, w_1)\}$

Iteration 3: $U((x_1, y_1, z_0, w_0)) := \{(x_1, y_1, z_0, w_0), (x_0, y_1, z_0, w_1), (x_1, y_0, z_0, w_1)\} \cup \emptyset$

We know that (x_0, y_1, z_0, w_1) *and* (x_1, y_0, z_0, w_1) *are primary bad states. Since they are both contained in* $U((x_1, y_1, z_0, w_0))$ *it follows that* $\alpha_2 \in V_C((x_1, y_0, z_0, w_0))$. *In fact we could have drawn this conclusion after only the first iteration.* □

The above example shows how we may compute V_C without actually ever forming the synchronous composition representation of $\mathbf{G}$. Based on this approach, we have computed the disablement maps for various configurations of the small factory setup of Example 39. In each case, the aim was to prevent overflow and underflow. The physical setup of the small factory is emulated by randomly generating a sequence of events. Each generated event is fed to the program as its input. The program computes the current state and produces a list of events that have to be disabled there. The program is written in C++; Table 3.1 below shows the results when the program is run on a Linux workstation with a 1 GHz Athlon processor and 150 MB of free RAM.

The efficiency of this program might be further improved by using more efficient data structures (heaps or trees rather than lists).

Buffer Size	# Machines Before Buffer	# Machines After Buffer	Approximate Size of State Space	Average Time to Compute Control
300	25	25	2^{58}	0.001 sec
300	50	50	2^{108}	0.03 sec
300	100	100	2^{208}	0.305 sec
300	150	250	2^{408}	2.5 sec
300	200	200	2^{408}	2.6 sec

Table 3.1: Computation Times for the Symbolic Supervision of Small Factory

3.4. Deadlock Avoidance

The problem of deadlock arises when two or more processes vie for the same resources. A generally accepted abstract definition of deadlock [RF96],[Pet81] is that it is a situation in which there exists a set of processes such that every process in the set is waiting for something that can only be done by some other process in the set. Thus no events are possible in a system when it is deadlocked. In the same spirit, we give the following definition of deadlock in a system under supervision. Again we assume that $\mathbf{G} = (X, \Sigma, E, x_o, X_m)$ is a plant automaton while $\mathbf{S} = (Y, \Sigma, F, y_0, Y_m)$ is a specification automaton.

Definition 56 *Let V_C be a disablement map for $\mathbf{G} \parallel \mathbf{S}$ that induces an automaton $\mathbf{C} \leq \mathbf{G} \parallel \mathbf{S}$. A state $(x, y) \in X^C \subseteq X^{G\|S}$ is defined to be a* deadlock state *if $Elig\,(\mathbf{C}, (x, y)) = \emptyset$ while $Elig\,(\mathbf{G}, x) \neq \emptyset$. The automaton $\mathbf{C}$ is said to be* deadlock-free *if it has no deadlock states. If $\mathbf{C}$ is deadlock-free then we also say that V_C is deadlock-free. We extend this definition to the empty automaton $\mathbf{\Phi} \leq \mathbf{G} \parallel \mathbf{S}$ by declaring it to be deadlock-free.*

Thus a state in a closed loop system is a deadlock state if the control action of the supervisor prevents any event from occurring at that state. This necessarily means that no uncontrollable event is eligible in the plant because a supervisory control action can never prevent an uncontrollable event from occurring. According to this definition, a marked

state can also be a deadlock state. However there may be scenarios where one does not care if the system is stuck in a marked (and hence desired) state. In such a scenario, the above definition can be suitably modified to exclude marked states.

The symbolic supervisor scheme described in the previous section handles safety properties only. It does not tackle any nonblocking or deadlock issues. It is very easy to construct examples of deadlock-free systems that block. Nonetheless, experience suggests that nonblocking can be ensured in a number of systems simply by making them deadlock free. In this section we first discuss the intuitive reason why the detection of nonblocking is difficult while deadlock detection is a lot easier. Then we extend our results to ensure that the proposed supervisory scheme never causes deadlock in the controlled system.

In a nonblocking system every reachable state is coreachable, i.e. there exists a path from any reachable state to some marked state. In order to verify that a given state is coreachable we have to demonstrate the existence of a path to a marked state. Herein lies the main difficulty in verifying nonblocking: demonstration of the existence of such a path may require the knowledge of the entire state space in general. In contrast, the verification of the controllable or the deadlock-free nature of a given state only requires the knowledge of eligible events at that state itself. We now illustrate this distinction between deadlock and nonblocking detection with the help of an example.

Example 57 *Let* **G** *shown in Figure 3.7 represent a plant and assume that all the events*

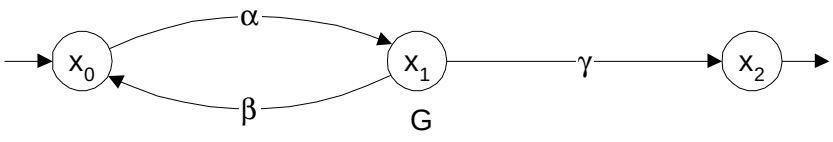


Figure 3.7: Blocking versus Deadlock Detection

are controllable. Assume that a safety specification requires the disablement of γ*. The*

resultant closed loop behaviour of **G** *(shown in Figure 3.8) is blocking. In particular,* x_0

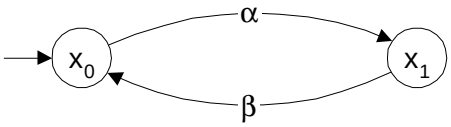


Figure 3.8: Blocking versus Deadlock Detection: Scenario 1

is a non-coreachable state. This is because the only marked state, x_2*, has been rendered unreachable by the disablement of* γ*. However, it is not possible to infer this just by looking at the eligible events at* x_0 *- we have to ensure that there no paths from* x_0 *to any marked states. In this instance, the supervisory action does not cause deadlock as it possible for an event to occur at both* x_0 *and* x_1*. Now assume that the safety specification additionally requires the disablement of* β *as well. The resultant closed loop behaviour (shown in Figure 3.9) is blocking as well as deadlocking. Here* x_1 *is a deadlock state because supervisory*

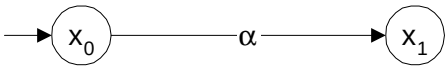


Figure 3.9: Blocking versus Deadlock Detection: Scenario 2

control has disabled both the events eligible there. This is readily identifiable at x_1 *itself. Once again, the fact that* x_0 *is a non-coreachable state cannot be verified using only the information available at* x_0*.* □

The easy identification of deadlock states does not mean that deadlock avoidance is easy to achieve: it has been shown to be NP-complete [Gol78],[ASK77]. We take advantage of the local nature of deadlock detection and extend our symbolic supervision technique to avoid deadlock. Let us first look at a couple of classical examples where deadlock occurs.

Example 58 *(Deadly Embrace) [Won01, page 108] Consider two users of two shared resources. To carry out her task each user needs both resources simultaneously although the resources may be acquired in either order. The users are modelled as* $\mathbf{User}_1$ *and* $\mathbf{User}_2$ *as shown in Figure 3.10. The overall interactions of the two users are modelled*

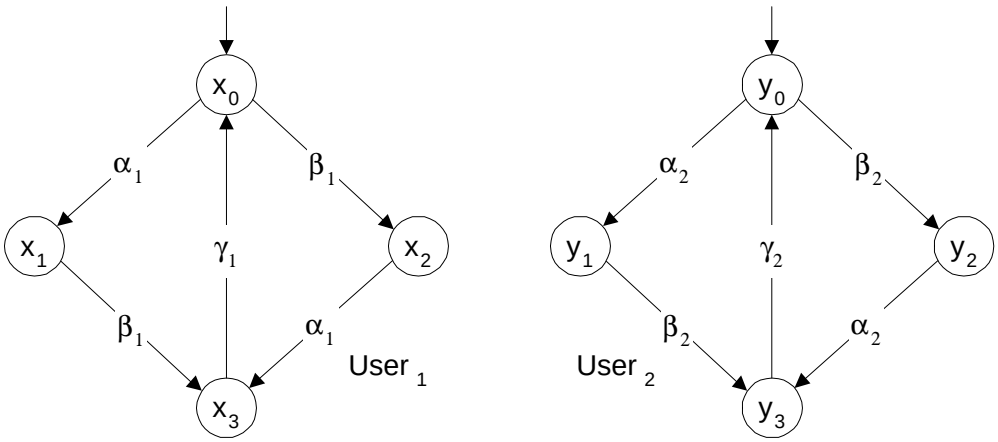


Figure 3.10: Two Users Competing for Resources

as $\mathbf{User} = \mathbf{User}_1 \parallel \mathbf{User}_2$. *One way to specify the usage of the resources is shown in Figure 3.11. This specification,* $\mathbf{S}$, *says that once a user has acquired a resource then the*

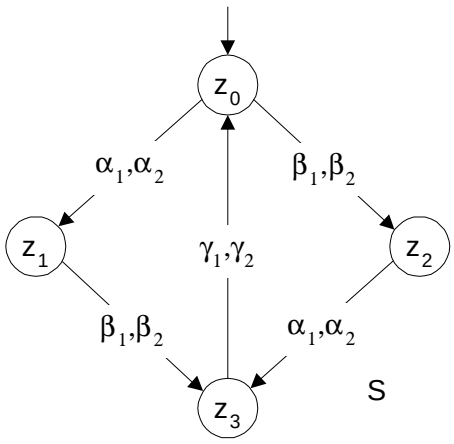


Figure 3.11: Resource Sharing Specification for Users

other user should not be allowed to acquire that resource until the first user has relinquished it. Here α_i *represents* $\mathbf{User}_i$ *acquiring the first resource while* β_i *represents* $\mathbf{User}_i$ *acquiring the second resource;* γ_i *represents* $\mathbf{User}_i$ *relinquishing both the resources. We assume* α_i *and* β_i *to be controllable and* γ_i *to be uncontrollable. Thus we have*

$$\begin{aligned} \Sigma^{User_1} &= \{\alpha_1, \beta_1, \gamma_1\}, \\ \Sigma^{User_2} &= \{\alpha_2, \beta_2, \gamma_2\} \end{aligned}$$

and distributing $\mathbf{S}$ *over* $\left\{\Sigma^{User_1}, \Sigma^{User_2}\right\}$ *we get* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *as shown in Figure 3.12. The*

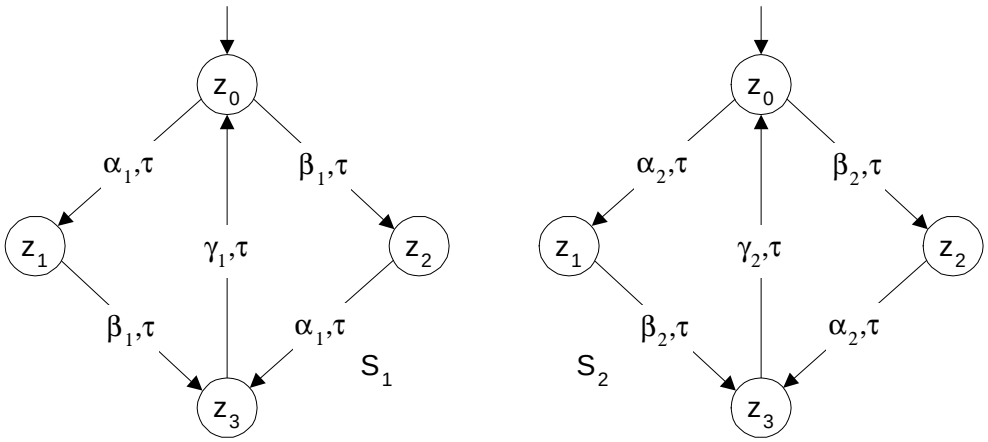


Figure 3.12: Distribution of Resource Sharing Specification

synchronous products $\mathbf{User}_1 \| \mathbf{S}_1$ *and* $\mathbf{User}_2 \| \mathbf{S}_2$ *are shown in Figure 3.13. The primary bad states are shown shaded. Since only controllable events are eligible at the initial states, it follows that the global initial state* (x_0, y_0, z_0) *is not a bad state and therefore we can use our symbolic supervisor. The set of global primary bad states is*

$$\{(x_3, \_, z_1), (x_3, \_, z_2), (\_, y_3, z_1), (\_, y_3, z_2)\}$$

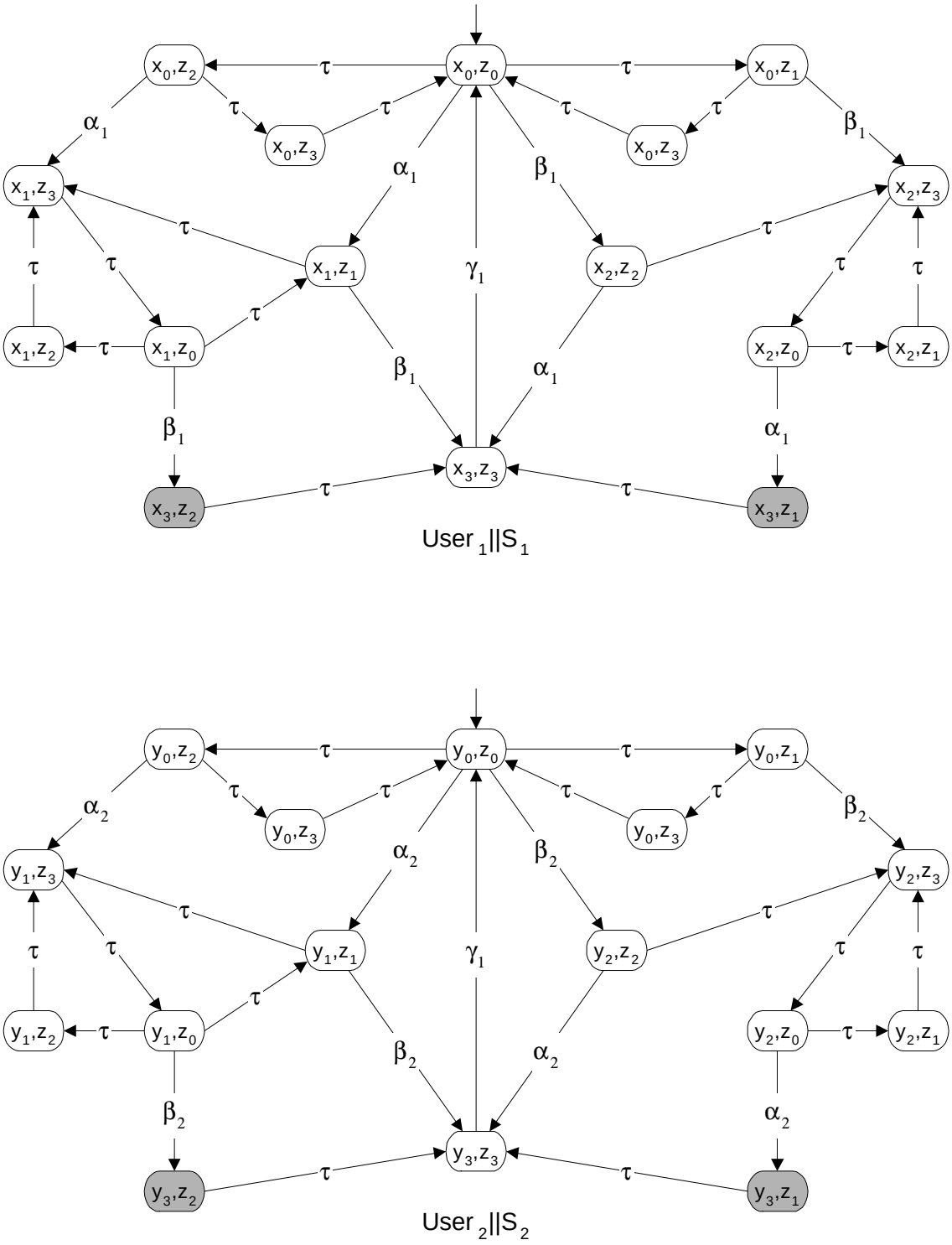


Figure 3.13: $User_1 \| S_1$ and $User_2 \| S_2$

where _ represents a "don't care" component. So, for instance, (x_3, y_0, z_1) *would be a global bad state because it matches* $(x_3, \_, z_1)$*. The table below shows the control action taken by the supervisor as the system executes the string* $\alpha_1\beta_2$*.*

String	*Global State*	*Events to Disable*
ϵ	(x_0, y_0, z_0)	$\emptyset$
α_1	(x_1, y_0, z_1)	$\{\alpha_2\}$
β_2	(x_1, y_2, z_3)	$\{\alpha_2, \beta_1\}$

The system does not violate the specification when the first user acquires the first resource while the second user acquires the second resource. However the system is now deadlocked as no event is eligible to occur at either (x_1, z_3) *in* $\mathbf{User}_1 \| \mathbf{S}_1$ *or at* (y_2, z_3) *in* $\mathbf{User}_2 \| \mathbf{S}_2$*. This is a direct result of the way we have specified the sharing of the two resources: nothing prevents the second user from acquiring the available resource after the first user has already acquired the other resource. Once this happens, each user keeps waiting for the other user to relinquish her resource and no action is possible.* □

Example 59 *(Material Feedback) [Won01, Page 115] Let us consider another seemingly innocuous example where deadlock occurs: a Transfer Line. The setup is shown in Figure 3.14. It consists of two machines (*$\mathbf{M}_1$ *and* $\mathbf{M}_2$*), a test unit (*$\mathbf{TU}$*), and two buffers (*$\mathbf{B}_1$ *and* $\mathbf{B}_2$*). Let the synchronous behaviour of the three machines be* $\mathbf{TL} = \mathbf{M}_1 \| \mathbf{M}_2 \| \mathbf{TU}$*. For this example we assume that the capacities of the buffers are 1 each. Machine* $\mathbf{M}_1$ *takes a piece from an inexhaustible source of raw material and processes it. Upon finishing, it deposits the semi-finished product into the buffer* $\mathbf{B}_1$*. Machine* $\mathbf{M}_2$ *takes its input from buffer* $\mathbf{B}_1$*,*

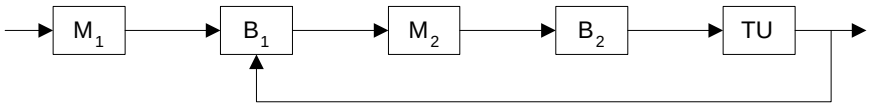


Figure 3.14: A Transfer Line

applies finishing touches to it and deposits the final product into the buffer $\mathbf{B}_2$. *Finally, the test unit* **TU** *examines this finished product and either approves and deposits it into an unfillable store or puts it back in the buffer* $\mathbf{B}_2$ *for reprocessing. The automata models of* $\mathbf{M}_1$, $\mathbf{M}_2$ *and* **TU** *are shown in Figure 3.15. Here the starting and finishing of work by* $\mathbf{M}_i$

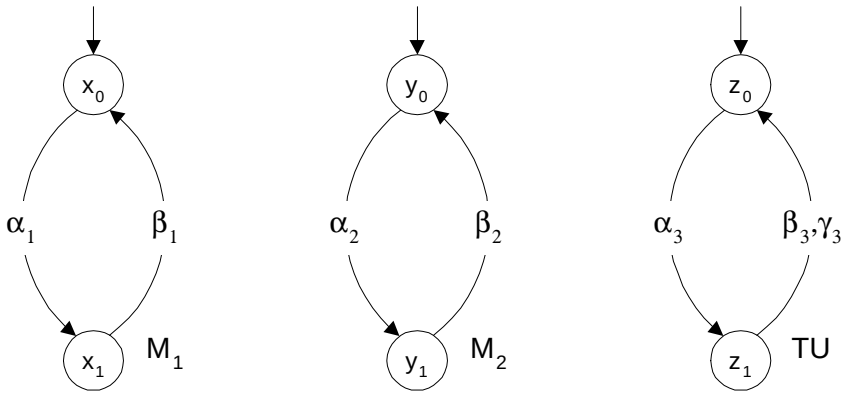


Figure 3.15: Transfer Line Components

is represented by the events α_i *and* β_i *respectively. Similarly, the starting of the testing by* **TU** *is represented by* α_3. *The event* γ_3 *represents the fact that the workpiece has passed the inspection while the event* β_3 *represents the redeposition of the workpiece in* $\mathbf{B}_1$. *We assume* α_i *to be controllable and all the other events to be uncontrollable. As usual, we require that the buffers should neither underflow nor overflow. The under/overflow specification* **S** *is shown in Figure 3.16. The various alphabets are*

$$\begin{aligned} \Sigma^{M_1} &= \{\alpha_1, \beta_1\} \\ \Sigma^{M_2} &= \{\alpha_2, \beta_2\} \\ \Sigma^{TU} &= \{\alpha_3, \beta_3, \gamma_3\} \end{aligned}$$

and the distribution of **S** *over these alphabets is shown in Figure 3.17. The synchronous*

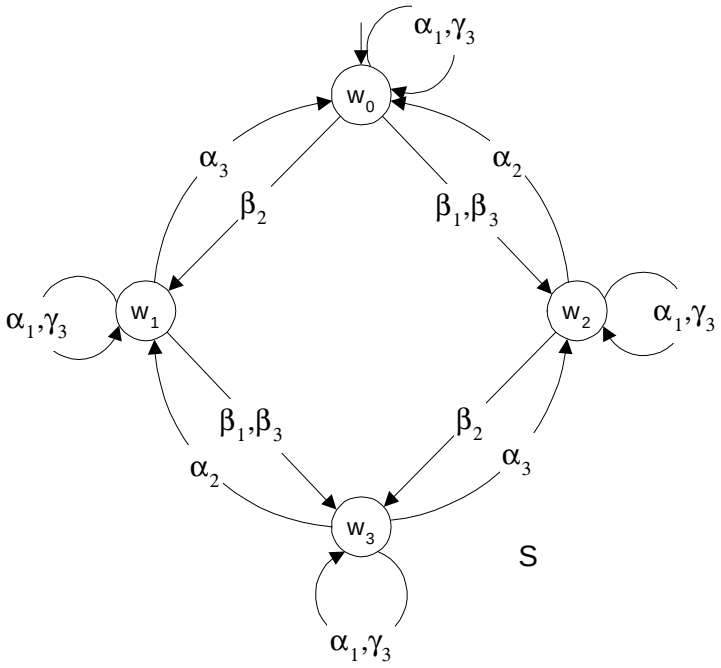


Figure 3.16: Transfer Line Under/Overflow Specification

compositions $\mathbf{M}_1\|\mathbf{S}_1$, $\mathbf{M}_2\|\mathbf{S}_2$ *and* $\mathbf{TU}\|\mathbf{S}_3$ *are shown in Figures 3.18, 3.19 and 3.20 respectively. The primary bad states in each of these synchronous compositions are shown shaded. The set of global primary bad states is*

$$\{(x_1, \_, \_, w_2), (x_1, \_, \_, w_3), (\_, y_1, \_, w_1), (\_, y_1, \_, w_3), (\_, \_, z_1, w_2), (\_, \_, z_1, w_3)\}$$

where _ *represents a "don't care" component. Again, all the events eligible at the initial state are controllable so the initial state is not a bad state. Therefore we can apply our symbolic supervision scheme. The table below shows the control action taken by the symbolic supervisor as the transfer line executes the string* $\alpha_1\beta_1\alpha_2\beta_2\alpha_1\beta_1$.

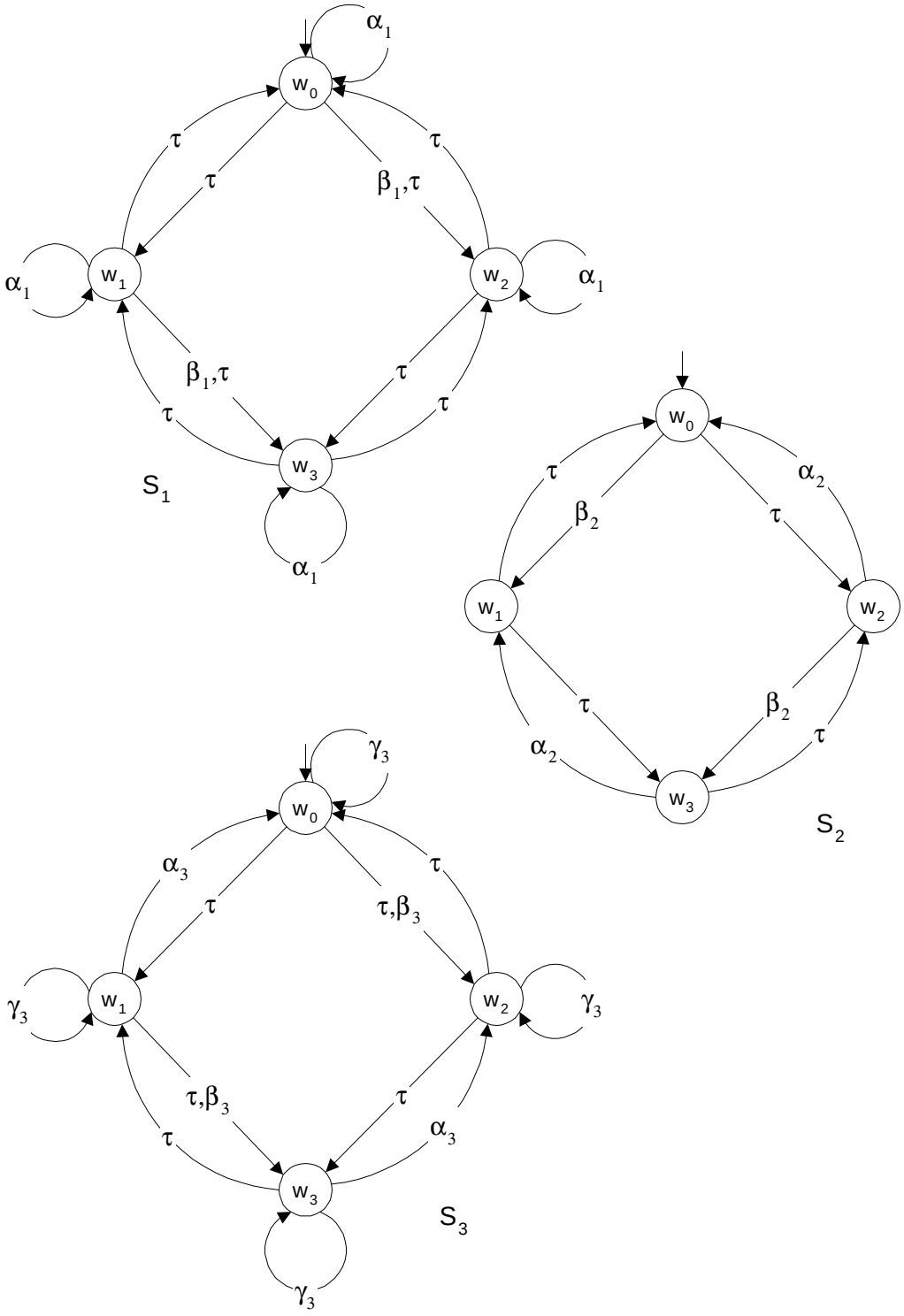


Figure 3.17: Distribution of Transfer Line Specification

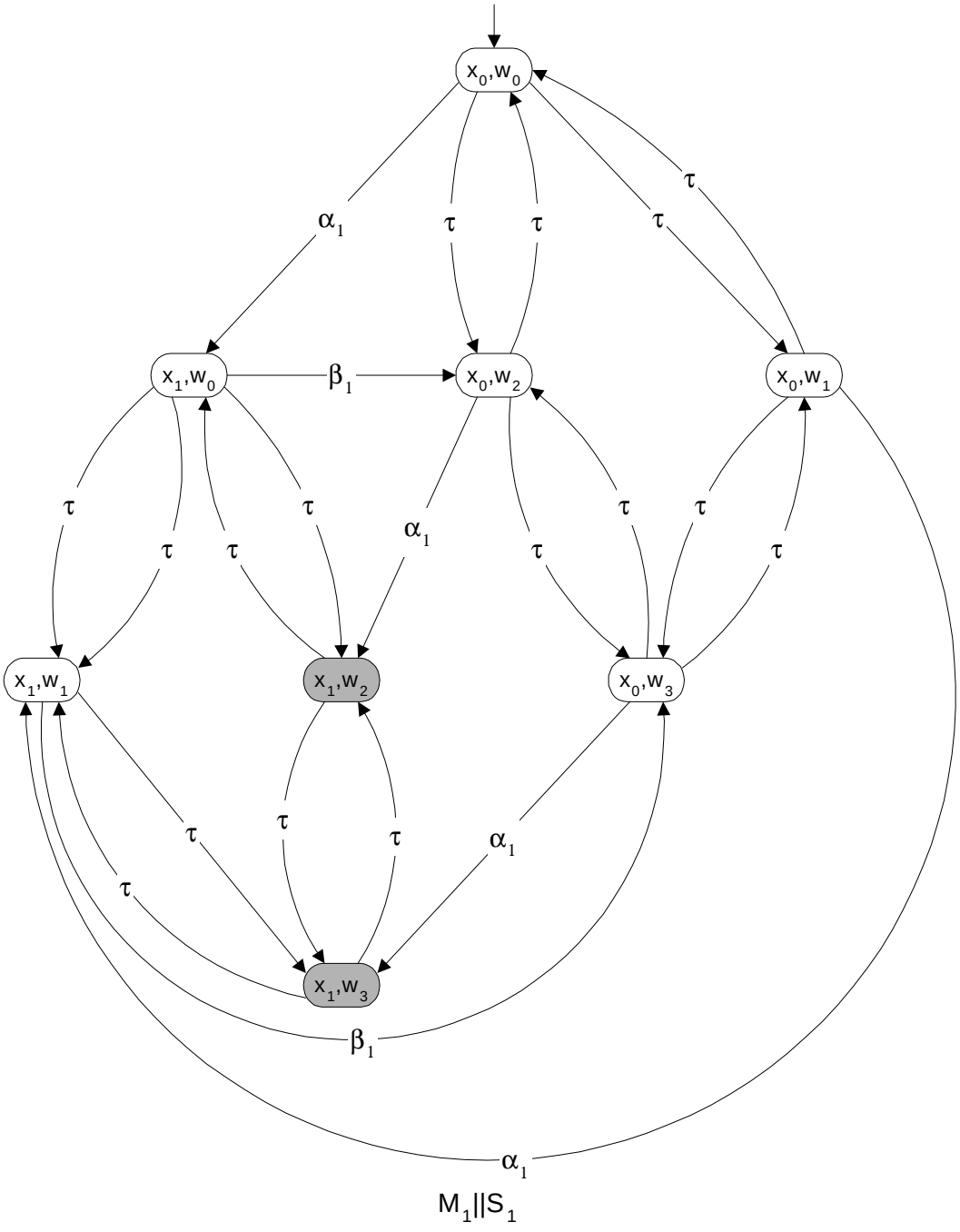


Figure 3.18: $M_1 \| S_1$

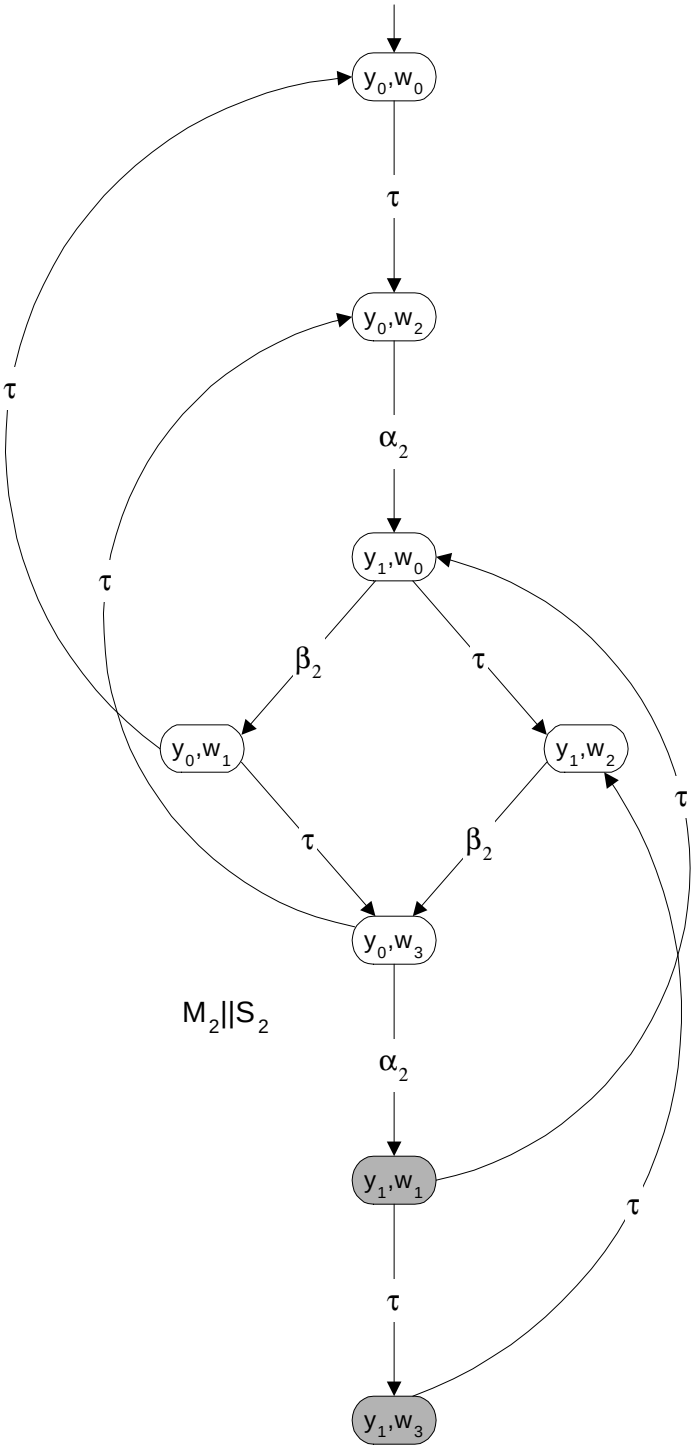


Figure 3.19: $M_2 \| S_2$

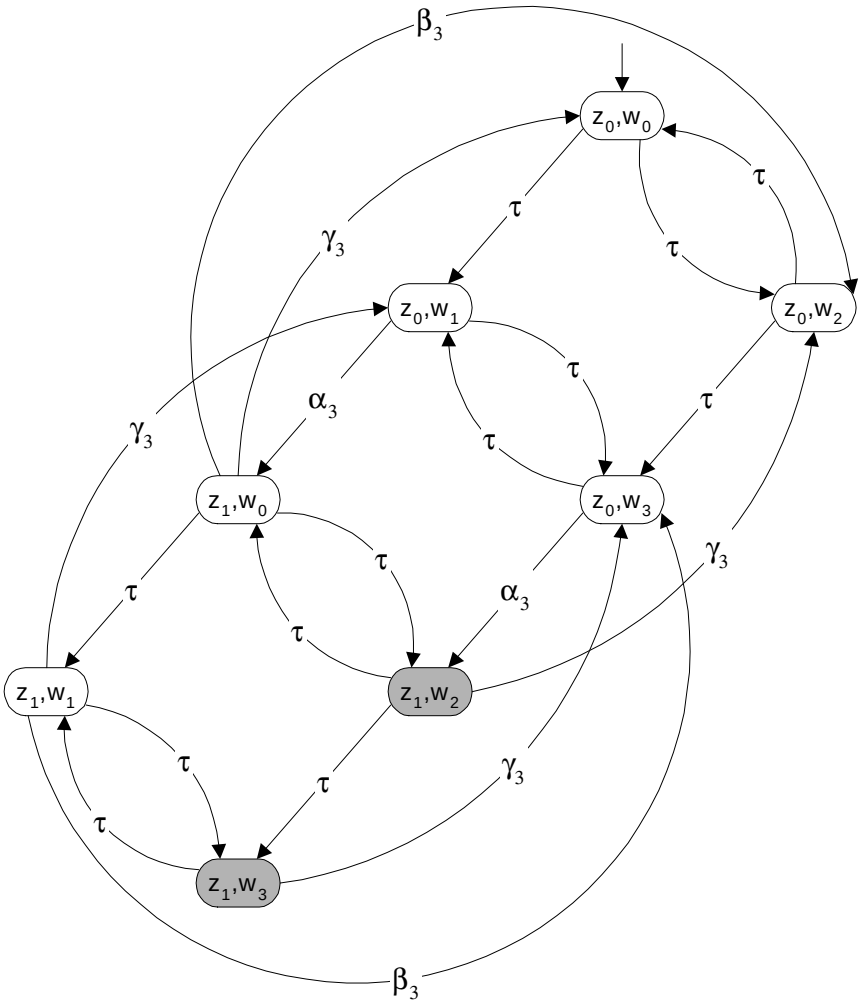


Figure 3.20: $M_3 \| S_3$

String	*Global State*	*Events to Disable*
ϵ	(x_0, y_0, z_0, w_0)	$\{\alpha_2, \alpha_3\}$
α_1	(x_1, y_0, z_0, w_0)	$\{\alpha_2, \alpha_3\}$
β_1	(x_0, y_0, z_0, w_2)	$\{\alpha_1, \alpha_3\}$
α_2	(x_0, y_1, z_0, w_0)	$\{\alpha_3\}$
β_2	(x_0, y_0, z_0, w_1)	$\{\alpha_2\}$
α_1	(x_1, y_0, z_0, w_1)	$\{\alpha_2, \alpha_3\}$
β_1	(x_0, y_0, z_0, w_3)	$\{\alpha_1, \alpha_2, \alpha_3\}$

After the occurrence of the string $\alpha_1\beta_1\alpha_2\beta_2\alpha_1\beta_1$ *the transfer line is in state* (x_0, y_0, z_0, w_3). *In this state, both the buffers are full and all the machines are disabled. This is because* $\mathbf{M}_1$ *and* $\mathbf{TU}$ *cannot begin operation while* $\mathbf{B}_1$ *is full and* $\mathbf{M}_2$ *cannot begin operation while* $\mathbf{B}_2$ *is full. Again our symbolic supervisor enforces the given specification but cannot prevent deadlock.* □

The state (x_1, y_2, z_3) in Example 58 is a deadlock state. The two users reach this state when the first user has acquired the first resource and the second user has acquired the second resource. The only eligible event at (x_1, z_3) in $\mathbf{User}_1 \| \mathbf{S}_1$ is the foreign event τ. Similarly, τ is the only event eligible at (y_2, z_3) in $\mathbf{User}_2 \| \mathbf{S}_2$. In Example 59, (x_0, y_0, z_0, w_3) is the deadlock state. The corresponding states in $\mathbf{M}_1 \| \mathbf{S}_1$, $\mathbf{M}_2 \| \mathbf{S}_2$ and $\mathbf{TU} \| \mathbf{S}_3$ are (x_0, w_3), (y_0, w_3) and (z_0, w_3) respectively. The only events eligible at these states are τ and α_1, τ and α_2, and τ and α_3. No uncontrollable event is eligible at these deadlock states. In fact, as mentioned earlier, this is a necessary condition for a deadlock state. Since we are dealing with controllable automata, a control action taken by a supervisor can never disable an uncontrollable event. So a state where an uncontrollable event is eligible can never be a deadlock state. This observation narrows down the number of states that may potentially

deadlock. We can evaluate our symbolic supervisor at these potentially deadlockable states and identify the actual deadlock states. Finally, we can extend our symbolic supervisor to ensure that these deadlock states are rendered unreachable. However we have to be a bit careful because a control action that prevents a system from reaching a deadlock state may convert a hitherto deadlock-free state into a deadlock state! We illustrate this scenario with a simple example.

Example 60 *Let* $\mathbf{G}$ *and* $\mathbf{S}$ *represent a plant and a specification respectively (Figure 3.21). Assume* α *and* β *to be controllable. Then* $\mathbf{G}||\mathbf{S}$ *is controllable with respect to* $\mathbf{G}$; *it is not*

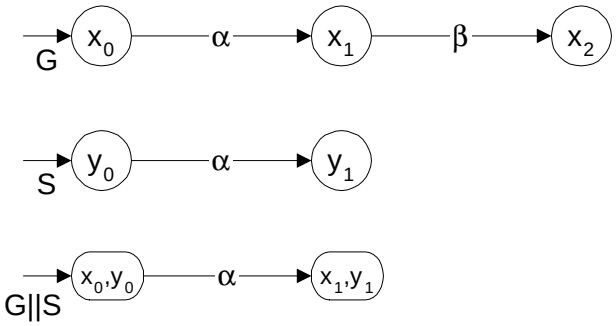


Figure 3.21: Deadlock removing control action may cause deadlock

deadlock free as (x_1, y_1) *is a deadlock state. In fact, it is the only deadlock state. We can expand our control action to disable* α *at* (x_0, y_0) *and thus prevent the system from reaching the deadlock state* (x_1, y_1) *. However, this control action turns* (x_0, y_0) *into a deadlock state. It turns out that, according to Definition 56, there does not exist any nonempty controllable subautomaton of* $\mathbf{G}||\mathbf{S}$ *that is also deadlock-free.* □

We now show how we may extend our symbolic supervisor to be deadlock-free.

Definition 61 *Let* $\mathcal{D}(\mathbf{G}, \mathbf{S}) \subseteq \mathcal{C}(\mathbf{G}, \mathbf{S})$ *be the set of all controllable subautomata of* $\mathbf{G}||\mathbf{S}$

that are deadlock-free, i.e.

$$\begin{aligned}\mathcal{D}(\mathbf{G},\mathbf{S}) &= \{\mathbf{S}' \in \mathcal{C}(\mathbf{G},\mathbf{S}) \,|\mathbf{S}' \textit{ is deadlock-free}\} \\ &= \{\mathbf{S}' \leq \mathbf{G}\|\mathbf{S} | \mathbf{S}' \textit{ is deadlock-free and controllable wrt } \mathbf{G}\}.\end{aligned}$$

Proposition 62 *The set $\mathcal{D}(\mathbf{G},\mathbf{S})$ is nonempty and its join is controllable with respect to $\mathbf{G}$ and is deadlock-free.*

Proof. The empty automaton $\mathbf{\Phi}$ is controllable with respect to $\mathbf{G}$ and is deadlock-free according to Definition 56. Thus $\mathcal{D}(\mathbf{G},\mathbf{S})$ is nonempty and from Lemma 32 we can conclude that $\vee\mathcal{D}(\mathbf{G},\mathbf{S})$ is controllable with respect to $\mathbf{G}$. So we only need to show that it is deadlock-free. Assume that $\mathbf{C} := \vee\mathcal{D}(\mathbf{G},\mathbf{S})$ is not deadlock-free. Then there exists some $x \in X^C \subseteq X^{G\|S}$ that is a deadlock state. We now use the method of contradiction to show that this cannot be the case. Let D be the set defined as follows:

$$D = \left\{\mathbf{H} \in \mathcal{D}(\mathbf{G},\mathbf{S}) \,|x \in X^H\right\}.$$

Then, by the definition of a join automaton, we must have

$$(\forall \mathbf{H} \in D)\left(Elig(\mathbf{C},x) \supseteq Elig(\mathbf{H},x)\right).$$

Since x is a deadlock state in $\mathbf{C}$ it follows that $Elig(\mathbf{C},x) = \emptyset$. Thus from the above equation we can conclude that $Elig(\mathbf{H},x) = \emptyset$ for all $\mathbf{H} \in D$. But this would imply that the elements of $D \subseteq \mathcal{D}(\mathbf{G},\mathbf{S})$ are not deadlock-free which is contradictory to our assumption. This leaves us with the conclusion that $D = \emptyset$ which implies that $x \notin X^C$. ■

Thus there exists a supremal deadlock-free controllable subautomaton of $\mathbf{G}\|\mathbf{S}$.

Definition 63 *A state* $(x, y) \in X^{G\|S}$ *is a* possible deadlock state *if it is not a bad state of* $\mathbf{G}\|\mathbf{S}$ *and does not have any eligible uncontrollable events but some events are eligible at* x *in* $\mathbf{G}$. *Let*

$$PD^{G\|S} := \left\{ \begin{array}{c} (x, y) \in X^{G\|S} | \\ (x, y) \notin B^{G\|S} \wedge Elig\left(\mathbf{G}, x\right) \neq \emptyset \wedge \\ Elig\left(\mathbf{G}\|\mathbf{S}, (x, y)\right) \cap \Sigma_u = \emptyset \end{array} \right\}$$

be the set of all possible deadlock states.

Definition 64 *Let* $A \supseteq PB^{G\|S}$ *be a subset of the states of* $\mathbf{G}\|\mathbf{S}$ *that contains all its primary bad states. Then we define* $V_C^A : X^{G\|S} \rightarrow 2^{\Sigma_c}$ *to be a* disablement map that disables A *where*

$$\left(\forall x \in X^{G\|S}\right) \left(\forall \sigma \in \Sigma_c\right) \left[\sigma \in V_C^A\left(x\right) \Leftrightarrow \left(\exists x_a \in U\left(x\right)\right) \left(x_a \in A\right)\right].$$

The disablement map V_C^A induces a controllable subautomaton $\mathbf{C}$ of $\mathbf{G}\|\mathbf{S}$ which does not contain any states belonging to A. If A equals $PB^{G\|S}$ then $\mathbf{C}$ is the supremal controllable subautomaton of $\mathbf{G}\|\mathbf{S}$. If A also includes all the deadlock states of $\mathbf{G}\|\mathbf{S}$ then the induced subautomaton will be controllable as well as deadlock-free.

Definition 65 *Let $A^{G\|S} \supseteq PB^{G\|S}$ be a set defined iteratively as follows:*

$$A^{G\|S} := PB^{G\|S}$$

do

$$temp := \left\{ \begin{array}{c} (x,y) \in PD^{G\|S} - A^{G\|S}| \\ V_C^{A^{G\|S}}((x,y)) = Elig(\mathbf{G}\|\mathbf{S},(x,y)) \wedge \\ Elig(\mathbf{G},x) \neq \emptyset \end{array} \right\}$$

$$A^{G\|S} := A^{G\|S} \cup temp$$

while $temp \neq \emptyset$

Then $A^{G\|S}$ is called the set of Augmented Primary Bad States*indexbad states!augmented primary of* $\mathbf{G}\|\mathbf{S}$.

The set $A^{G\|S}$ contains all the states of $\mathbf{G}\|\mathbf{S}$ where controllability is violated and where deadlock may result. The multiple passes of the do-while loop are necessary to ensure that removal of a deadlock state does not give rise to other deadlock states (as in Example 60). Before the first pass of the do-while loop, the set $A^{G\|S}$ contains the primary bad states of $\mathbf{G}\|\mathbf{S}$. After the first pass, it additionally contains all the deadlock states that may arise due to the control action that is needed to prevent access to the primary bad states. If any such deadlock states are found then the set *temp* is nonempty. This causes the do-while loop to be repeated. The next pass discovers any deadlock states that may have resulted due to the modified control action; and so on. This process can only be repeated a finite number of times since the set $X^{G\|S}$ is finite. The loop terminates when *temp* is empty; the set $A^{G\|S}$ contains all the primary bad states and all the deadlock states that may arise due to supervisory control action. In most systems, the removal of a deadlock state will not give rise to any other deadlock states so only one or two passes through the do-while loop

will be necessary. The complexity of computing $A^{G\|S}$ is directly proportional to the size of the set of possible deadlock states $PD^{G\|S}$. Thus the computation of $A^{G\|S}$ will be more efficient for systems that have fewer controllable transitions: *a more controllable system is more prone to deadlock*!

Lemma 66 *Let* $V_C^{A^{G\|S}} : X^{G\|S} \to 2^{\Sigma_c}$ *be a disablement map that disables* $A^{G\|S}$. *Let* $\mathbf{C}$ *be the subautomaton of* $\mathbf{G}\|\mathbf{S}$ *induced under* $V_C^{A^{G\|S}}$. *If* $U\left(x_0^{G\|S}\right) \cap A^{G\|S} = \emptyset$ *then* $\mathbf{C}$ *is the supremal controllable deadlock-free subautomaton of* $\mathbf{G}\|\mathbf{S}$ *with respect to* $\mathbf{G}$, *i.e.* $\mathbf{C} = \vee\mathcal{D}(\mathbf{G}, \mathbf{S})$.

Proof. We prove this in two parts. First we show that $\mathbf{C}$ is controllable and deadlock-free. Then we show that if there exists any deadlock-free $\mathbf{C}' \leq \mathbf{G}\|\mathbf{S}$ that is controllable with respect to $\mathbf{G}$ then $\mathbf{C}' \leq \mathbf{C}$.

Part 1: Since $A^{G\|S} \supseteq PB^{G\|S}$ it follows that $\mathbf{C}$ is controllable with respect to $\mathbf{G}$ if the initial state is not a bad state. By the definition of $V_C^{A^{G\|S}}$, it does not contain any deadlock states since $X^C \cap A^{G\|S} = \emptyset$. Finally, since no deadlock state is uncontrollably reachable from the initial state, it follows that it is controllable as well as deadlock-free.

Part 2: Let $\mathbf{C}'$ be any controllable and deadlock-free subautomaton of $\mathbf{G}\|\mathbf{S}$ and let $x \in X^{C'}$. Since $\mathbf{C}'$ is controllable and deadlock-free, it follows that $x \in X^{G\|S} - A^{G\|S}$ and $U(x) \cap A^{G\|S} = \emptyset$. Then $x \in X^C$ by the definition of $V_C^{A^{G\|S}}$. Thus $X^{C'} \subseteq X^C$. A similar argument shows that $E^{C'} \subseteq E^C$ which proves the claim that $\mathbf{C}' \leq \mathbf{C}$. ■

We now show how we may use this modified supervisor to get deadlock-free behaviour in the systems of Examples 58 and 59.

Example 67 *(Transfer Line revisited) In Example 59 we saw that a deadlock occurs when both the buffers are full. We now use Lemma 66 to synthesize a deadlock-free behaviour of the Transfer Line that implements the under/overflow specification. From Example 59 we know that the set of primary bad states of the Transfer Line is*

$$PB^{TL\|S} = \{(x_1, \_, \_, w_2), (x_1, \_, \_, w_3), (\_, y_1, \_, w_1), (\_, y_1, \_, w_3), (\_, \_, z_1, w_2), (\_, \_, z_1, w_3)\}.$$

The set of possible deadlock states of the Transfer Line can be computed as follows. From Figure 3.18 we can see that the set of states of $\mathbf{M}_1\|\mathbf{S}_1$ *where no uncontrollable event is eligible is*

$$\{(x_0, w_0), (x_0, w_1), (x_0, w_2), (x_0, w_3)\}.$$

Similarly, from Figures 3.19 and 3.20, the sets of states of $\mathbf{M}_2\|\mathbf{S}_2$ *and* $\mathbf{TU}\|\mathbf{S}_3$ *where no uncontrollable events are eligible are*

$$\{(y_0, w_0), (y_0, w_1), (y_0, w_2), (y_0, w_3)\}$$

and

$$\{(z_0, w_0), (z_0, w_1), (z_0, w_2), (z_0, w_3)\}$$

respectively. Thus the set of all possible deadlock states of the Transfer Line is

$$PD^{TL\|S} = \{(x_0, y_0, z_0, w_0), (x_0, y_0, z_{0,} w_1), (x_0, y_0, z_0, w_2), (x_0, y_0, z_{0,} w_3)\}.$$

We now show the various steps involved in the computation of $A^{TL\|S}$ *(the set of augmented primary bad states).*

Initialization:

$$\begin{aligned} A^{TL\|S} &:= PB^{TL\|S} \\ &= \{(x_1, \_, \_, w_2), (x_1, \_, \_, w_3), (\_, y_1, \_, w_1), (\_, y_1, \_, w_3), (\_, \_, z_1, w_2), (\_, \_, z_1, w_3)\} \end{aligned}$$

First Pass of Do-While:

$$\begin{aligned} temp &:= \{(x_0, y_0, z_0, w_3)\} \\ A^{TL\|S} &:= A^{TL\|S} \cup temp \\ &= PB^{TL\|S} \cup temp \\ &= PB^{TL\|S} \cup \{(x_0, y_0, z_0, w_3)\} \end{aligned}$$

Second Pass of Do-While:

$$\begin{aligned} temp &:= \emptyset \\ A^{TL\|S} &:= A^{TL\|S} \cup temp \\ &= PB^{TL\|S} \cup \{(x_0, y_0, z_0, w_3)\} \cup \emptyset \\ &= PB^{TL\|S} \cup \{(x_0, y_0, z_0, w_3)\} \end{aligned}$$

We initialize $A^{TL\|S}$ to be the set of all primary bad states. In the first pass of the do-while loop, we evaluate $V_C^{A^{TL\|S}}$ (the disablement map that disables $A^{TL\|S}$) at all the states in $PD^{TL\|S}$, i.e. at a total of four states. We discover one of these states, (x_0, y_0, z_0, w_3), to be a deadlock state. This state corresponds to the scenario where both the buffers are full. Then we update the set of augmented bad states, $A^{TL\|S}$, by adding this deadlock state to it. In the second pass of the do-while loop, we re-evaluate $V_C^{A^{TL\|S}}$ at the remaining three states of $PD^{TL\|S}$. However this time we use the updated $A^{TL\|S}$. No new deadlock states are discovered and the do-while loop terminates. It can be easily checked that no state of

$A^{TL\|S}$ is uncontrollably accessible from the initial state of $\mathbf{TU}\|\mathbf{S}$. *Thus* $V_C^{A^{TL\|S}}$ *induces a supremal controllable and deadlock-free subautomaton of* $\mathbf{TU}\|\mathbf{S}$. □

Example 68 *(Deadly Embrace revisited) In Example 58 a scenario can arise where both the users have acquired a resource apiece. However no more progress is possible since they both need both the resources in order to continue. We now show how Lemma 66 can help these two users avoid such a deadlock. From Example 58 we know that the set of primary bad states is*

$$PB^{User\|S} = \{(x_3, \_, z_1), (x_3, \_, z_2), (\_, y_3, z_1), (\_, y_3, z_2)\}.$$

The set of possible deadlock states can be inferred from Figure 3.13 to be the entire non-bad state space of $\mathbf{User}\|\mathbf{S}$ *except* (x_3, y_3, z_3). *Thus the computation of* $A^{User\|S}$ *requires the evaluation of the disablement map over almost the entire state space! As mentioned earlier, such a scenario arises when the controllable transitions far outnumber the uncontrollable transitions. In* $\mathbf{User}_i\|\mathbf{S}_i$ *there are four controllable transitions and only one uncontrollable transition. However if we proceed ahead nonetheless and compute* $A^{User\|S}$ *then* $V_C^{User\|S}$ *ensures that once a user acquires a resource, the other user is prevented from acquiring any resource until the first user has relinquished all her resources.* □

3.4.1. Some Comments

In Example 68, the number of possible deadlock states is comparable to the size of the global state space. In such a case, the identification of the deadlock states may be very inefficient. In Example 67, the number of possible deadlock states comprises a much smaller fraction of the global state space. However even a small fraction of a very large number may be impractical to deal with. A designer's intuition about the physical system can be very useful in such cases. For instance, in a system such as a Transfer Line, it may be clear to a

designer that a deadlock can only occur if the buffers are full. The designer can then use this intuition to guide the construction of the set of augmented primary bad states. However, in general, there may be systems for which this modular approach is just as inefficient as a centralized design approach. This is to be expected given the NP-completeness of the deadlock avoidance problem.

Sometimes it may be possible to infer the nonblocking nature of a supervisor based on its deadlock-free nature. Let us consider the Transfer Line again and assume that the only marked state is the initial state. This corresponds to the scenario where both the buffers are empty. Nonblocking in this case corresponds to the guarantee that it is always possible to empty the buffers. If it is known that the system is free of deadlocks then it is always possible to empty the buffers. In this case deadlock-freeness implies nonblocking. However this is not a systematic approach and the inability to handle nonblocking directly remains the biggest drawback of the proposed scheme. Given the hard nature of nonblocking, it may perhaps be useful to impose structural restrictions on the plant as well as the specification.

3.5. Complexity Analysis

In this section we analyse the computational complexity of implementing the proposed schemes. We assume that the plant comprises n subsystems and is modelled as $\mathbf{G} = \mathbf{G}_1 \| \cdots \| \mathbf{G}_n$. We also assume that $\mathbf{S}$ represents a specification for this plant. Let $\mathbf{C}$ be the supremal controllable subautomaton of $\mathbf{G} \| \mathbf{S}$. From [Rud88] we know that the time complexity of synthesizing $\mathbf{C}$ is $O\left(|\Sigma| \left|X^{G_i}\right|^n \left|X^S\right| + \left|E_u^{G_i}\right|^n \left|E_u^S\right|\right)$; the space needed to store $\mathbf{C}$ is $O\left(\left|X^{G_i}\right|^n \left|X^S\right| + \left|E^{G_i}\right|^n \left|E^S\right|\right)$. We now compute the computational complexity of the proposed schemes.

3.5.1. Complexity without Deadlock Avoidance

The time complexity involved in the evaluation of a disablement map can be broken down into two halves: offline and online. The offline complexity results because we need to identify the primary bad states. The online complexity results because we need to evaluate the disablement map V_C at each state. Using Corollary 54 we can estimate the time needed for the identification of the primary bad states to be of the order of $O\left(n\left|\Sigma^i\right|\left|X^{G_i}\right|\left|X^{S_i}\right|\right) = O\left(n\left|\Sigma^i\right|\left|X^{G_i}\right|\left|X^{S}\right|\right)$. The online complexity is directly proportional to the complexity of computing the iterative uncontrollable span $IU(\cdot)$. From Definition 48 we can estimate this time to be of the order of $O\left(n\left|E_u^{G_i}\right|\left|E_u^{S_i}\right|\right) = O\left(n\left|E_u^{G_i}\right|\left|E_u^{S}\right|\right)$.

The space complexity of the proposed scheme can be broken down into two halves: the space needed for the storage of $\mathbf{G}_i\|\mathbf{S}_i$ and the space needed for the temporary storage of IU. The space required to store $\mathbf{G}_i\|\mathbf{S}_i$, $1 \leq i \leq n$, is of the order of $O\left(n\left|X^{G_i}\right|\left|X^{S_i}\right| + n\left|E^{G_i}\right|\left|E^{S}\right|\right)$. The temporary storage needed for IU can grow exponentially if it is stored in the same form as it is given in Definition 48. However, as shown below, we can use a much more efficient data structure for storing IU.

Definition 69 *Let* $x = (x^1, \ldots, x^n, s) \in X^{G\|S}$. *For* $1 \leq i \leq n$, *define the* local iterative uncontrollable spans *of* x *as follows.*

$$IU^1(x) := \left\{\left(x^1, s\right)\right\} \cup \left\{ \begin{array}{l} (x_1^1, s_1) \in X^{G_1\|S_1} \,|\, (\exists w \in \Sigma_u^{1*}) \\ \left[\eta^{G_1\|S_1}\left(\left(x^1, s\right), w\right) = (x_1^1, s_1)\right] \end{array} \right\}$$

$$\begin{array}{l} \textit{for} \quad i := 2 \ \textit{to} \ n \\ \qquad IU^i\left(x\right) := \left\{\left(x^i, s\right)\right\} \cup \left\{ \begin{array}{c} \left(x_1^i, s_1\right) \in X^{G_i \| S_i} \left| \left(\exists j < i\right) \right. \\ \left(\exists \left(x^j, s_2\right) \in IU^j\left(x\right)\right)\left(\exists w \in \Sigma_u^{i*}\right) \\ \left[\eta^{G_i \| S_i}\left(\left(x^i, s_2\right), w\right) = \left(x_1^i, s_1\right)\right] \end{array} \right\} \\ \textit{end} \quad \textit{for} \end{array}$$

The various combinations of IU^i can then be used to compute IU as shown below:

$$IU\left(x\right) = \left\{\left(x_1^1, \ldots, x_1^n, s\right) \mid \left(\forall i \in \{1, \ldots, n\}\right)\left[\left(x_1^i, s_1\right) \in IU^i\left(x\right)\right]\right\}.$$

However, due to Corollary 54, we never actually need to compute IU explicitly; the various IU^i are sufficient to compute V_C. In the worst case, the space needed to store IU^i, $1 \leq i \leq n$, is of the order of $O\left(n\left|X^{G_i}\right|\left|X^{S_i}\right|\right)$. Thus the overall space complexity of the proposed scheme is

$$\begin{array}{rl} & O\left(n\left|X^{G_i}\right|\left|X^{S_i}\right| + n\left|E^{G_i}\right|\left|E^S\right|\right) + O\left(n\left|X^{G_i}\right|\left|X^{S_i}\right|\right) \\ = & O\left(n\left|X^{G_i}\right|\left|X^{S_i}\right| + n\left|E^{G_i}\right|\left|E^S\right|\right). \end{array}$$

3.5.2. Complexity with Deadlock Avoidance

If we want the disablement map V_C to be deadlock-free then we have the additional task of identifying deadlock states. To do this, we need to evaluate V_C at all the possible deadlock states. We know that the time needed for the evaluation of V_C at any given state is $O\left(n\left|E_u^{G_i}\right|\left|E_u^S\right|\right)$. Therefore the time needed to identify the deadlock states is $O\left(n\left|E_u^{G_i}\right|\left|E_u^S\right|\left|PD^{G\|S}\right|\right)$. As seen in Example 58, the magnitude of $PD^{G\|S}$ can be exponential in n. This is inherent in the deadlock avoidance problem. Thus the offline time

complexity increases to

$$\begin{aligned} & O\left(n\left|E_u^{G_i}\right|\left|E_u^S\right|\right) + O\left(n\left|E_u^{G_i}\right|\left|E_u^S\right|\left|PD^{G\|S}\right|\right) \\ = \; & O\left(n\left|E_u^{G_i}\right|\left|E_u^S\right|\left|PD^{G\|S}\right|\right). \end{aligned}$$

The online time complexity and the space complexity remain unchanged.

3.6. Summary

In this chapter we focused our attention on plants that comprise n disjoint components. For such systems, we presented a modular and efficient mechanism to verify controllability of a given specification. We also proposed a symbolic scheme for the synthesis of supremal controllable subautomata. This scheme is symbolic because it produces a disablement map rather than a extensive look-up table. This disablement map is efficiently computable. The proposed scheme has a drawback in that it only handles safety specifications and is prone to blocking. Experience has shown that a number of blocking problems can be solved by ensuring deadlock-freeness. Deadlock avoidance turns out to be an easier problem to solve in our framework. So we extended our proposed scheme by making it deadlock-free.

4. Supervisor Reduction

4.1. An Overview

In the previous chapter we have presented a symbolic method for the design of a supervisor. We have shown that the proposed design can provide great savings in the space and time required to synthesize a supervisor. However the symbolic supervisor needs to be evaluated in real-time and is prone to blocking. So there may be scenarios where it may make more sense to work with a standard RW supervisor [RW82][Won01]. In such a case, the supervisor is typically an automaton that generates a sublanguage of the plant and is controllable with respect to it. The plant is synchronized with this automaton: an event is enabled in the plant if and only if it is enabled in this automaton. It is never required to disable an uncontrollable event because the automaton is controllable. This is analogous to generating a lookup table that associates with each state of the closed loop system the events that need to be disabled there. It is often the case that such a supervisor contains a lot of redundant information that can be easily inferred from the structure of the plant. Therefore it makes sense to ask whether a given supervisor is minimal (having the least possible number of states). If not, then is it possible to construct a minimal supervisor. These questions were first explored by Vaz and Wonham in [VW86]. They showed that the construction of a minimal supervisor is time-exponential with respect to the size of the state of the given supervisor. Su and Wonham have recently shown in [SW00] that in

fact this problem is NP-hard. However Su and Wonham [SW00],[SW01] have presented a heuristic polynomial-time algorithm that provides good reduction in a lot of cases. They have also given a conservative lower bound for the number of states in a minimal supervisor. The supervisor reduction problem seems to be indeed tractable for numerous systems even though finding the minimal supervisor is NP-hard.

In this chapter we present another heuristic polynomial-time algorithm for supervisor reduction. This algorithm provides reduction that is comparable to the reduction provided by the algorithm proposed in [SW01]. We also present an estimate for the number of states in a minimal supervisor. This estimate, used in conjunction with the one given in [SW01], can be used to judge the effectiveness of a reduction algorithm.

The rest of this chapter is structured as follows. We first give a brief overview of the implementation details of a supervisor. Then we briefly discuss the algorithms given in [VW86] and [SW01]. Finally we present the algorithms for supervisor reduction and estimation of the state size of a minimal supervisor.

4.2. Implementation of a Supervisor

In this chapter we assume that $\mathbf{G} = (X, \Sigma, E, x_0, X_m)$ is a plant automaton, $\mathbf{S} = (Y, \Sigma, F, y_0, Y_m)$ is a specification automaton. As in [Won01], we define

$$\Gamma = \left\{\gamma \in 2^{\Sigma} | \gamma \supseteq \Sigma_u\right\}$$

to be the set of all *control patterns* and define a *supervisory control* for $\mathbf{G}$ as follows.

Definition 70 *A* supervisory control *for* $\mathbf{G}$ *is any map* $W : L(\mathbf{G}) \to \Gamma$. *The closed loop system is written as* $W/\mathbf{G}$. *The* closed behaviour *of* $W/\mathbf{G}$ *is defined to be the language*

$L(W/\mathbf{G}) \subseteq L(\mathbf{G})$ *described as follows.*

1. $\epsilon \in L(W/\mathbf{G})$
2. *If* $s \in L(W/\mathbf{G})$, $\sigma \in W(s)$, *and* $s\sigma \in L(\mathbf{G})$ *then* $s\sigma \in L(W/\mathbf{G})$
3. *No other strings belong to* $L(W/\mathbf{G})$.

The marked behaviour of $L(W/\mathbf{G})$ *is* $L_m(W/\mathbf{G}) = L(W/\mathbf{G}) \cap L_m(\mathbf{G})$. *A supervisory control* W *is said to be* nonblocking *for* $\mathbf{G}$ *if* $\overline{L}_m(W/\mathbf{G}) = L(W/\mathbf{G})$.

A supervisory control for $\mathbf{G}$ may be implemented by synchronizing it with an automaton (supervisor) that *allows* precisely those strings of $\mathbf{G}$ that satisfy the given specification. If $\mathbf{C} \leq \mathbf{G}\|\mathbf{S}$ is a (perhaps supremal) controllable subautomaton then the corresponding supervisory control can be defined as

$$W_C(s) = Elig(\mathbf{C}, x) \cup \Sigma_u$$

where x is the state in $\mathbf{C}$ corresponding to s. In other words, we can implement a supervisory control for $\mathbf{G}$ that implements $\mathbf{S}$ by synchronizing it with $\mathbf{C}$ as shown in Figure 4.1. In such

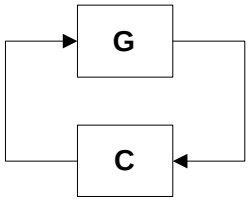


Figure 4.1: Supervisory Control Setup

a case we say that $\mathbf{C}$ is a supervisor for $\mathbf{G}$. If $V_C : X^{G\|S} \rightarrow 2^{\Sigma_c}$ represents a disablement map that induces $\mathbf{C}$ then we get the alternate setup shown in Figure 4.2. Due to the fact that $\mathbf{C}$ is a subautomaton of $\mathbf{G}\|\mathbf{S}$ we will have $L(W_C/\mathbf{G}) = L(\mathbf{C}) \subseteq L(\mathbf{G})$ as desired.

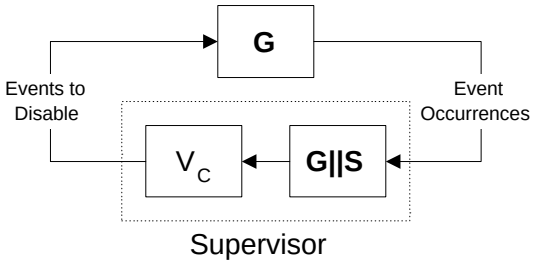


Figure 4.2: Supervisory Control Setup for the Proposed Scheme

The disablement map V_C is usually implemented as a static look-up table - there is a row in the look-up table corresponding to each state in $\mathbf{G}\|\mathbf{S}$.

Let $\mathbf{C}'$ be another supervisor for $\mathbf{G}$ and let $W_{C'}$ be the corresponding supervisory control. If we have $L(W_{C'}/\mathbf{G}) = L(\mathbf{C}) = L(W_C/\mathbf{G})$ then $W_{C'}$ is equivalent to W_C. In such a case we shall say that $\mathbf{C}'$ is *control equivalent* to $\mathbf{C}$. If $\mathbf{C}$ is the supremal controllable subautomaton of $\mathbf{G}\|\mathbf{S}$ then any supervisor that is control equivalent to it is said to be *normal*. In particular, the supremal controllable subautomaton of $\mathbf{G}\|\mathbf{S}$ is normal. We may sometimes write $\mathbf{C}/\mathbf{G}$ rather than $W_C/\mathbf{G}$ to represent the closed loop system. This will emphasize that $\mathbf{C}$ is the automaton *supervising* $\mathbf{G}$.

4.3. Control Covers and Supervisor Reduction

Given a supervisor $\mathbf{C}$ for $\mathbf{G}$ we want to compute a smaller control equivalent automaton $\mathbf{C}'$, i.e.

$$\begin{aligned} L(\mathbf{C}') \cap L(\mathbf{G}) &= L(\mathbf{C}) \cap L(\mathbf{G}), \\ L_m(\mathbf{C}') \cap L_m(\mathbf{G}) &= L_m(\mathbf{C}) \cap L_m(\mathbf{G}), \text{ and} \\ \left|X^{C'}\right| &< \left|X^{C}\right|. \end{aligned}$$

Such an automaton $\mathbf{C}'$ is called a reduced supervisor. In this section we present the generic approach taken for supervisor reduction. The following definitions are taken from [VW86] with minor modifications.

Definition 71 *A* cover *of* $\mathbf{C}$ *is defined to be a family* $\left\{X_i \subseteq X^C : i \in I\right\}$ *of the subsets of the state set of* $\mathbf{C}$ *with the following properties:*

1. $(\forall i \in I)\, X_i \neq \emptyset$
2. $(\forall i, j \in I)\, i \neq j \Rightarrow X_i \nsubseteq X_j$
3. *for a subset* $I_m \subseteq I$,

$$X_m^C = \cup_{i \in I_m} X_i,$$
$$X^C - X_m^C = \cup_{i \in I - I_m} X_i.$$

Here I *is some arbitrary index set.*

The elements of a cover of $\mathbf{C}$ inherit the marking structure of $\mathbf{C}$.

Definition 72 *A cover of* $\mathbf{C}$ *is defined to be* deterministic *if*

$$(\forall i \in I)\,(\forall \sigma \in \Sigma)\,(\exists y \in X_i)\,\eta^C(y, \sigma)!$$
$$\Rightarrow \left[(\exists j \in I)\,(\forall x \in X_i)\,\eta^C(x, \sigma)! \Rightarrow \eta^C(x, \sigma) \in X_j\right].$$

A cover is deterministic if no two states of any element of the cover make transitions to different elements of the cover under the same event. In other words, if a transition occurs at any state belonging to an element of the cover then we can uniquely identify the element of the cover that contains the target state.

Definition 73 *Let $x_1 \in X^C$ be any state of* **C**. *Let $X'(x_1) \subseteq X$ be a subset of the plant states defined as follows:*

$$X'(x_1) = \left\{x \in X : (x, x_1) \in X^{G\|C}\right\}.$$

Then the set of disabled events at x_1 is defined as

$$Disabled(x_1) := \{\sigma \in \Sigma_c : (\exists x \in X'(x_1))\,[\sigma \in Elig(\mathbf{G}, x) - Elig(\mathbf{C}, x_1)]\}.$$

The set $Disabled(x_1)$ comprises the events that are *disabled* in the plant at a state $x \in X$ such that $(x, x_1) \in X^{G\|C}$.

Definition 74 *Let $x_1, x_2 \in X^C$ be any two states of* **C**. *They are defined to be* control consistent *if $Disabled(x_1) = Disabled(x_2)$. A cover of* **C** *is defined to be* control consistent *if*

$$(\forall i \in I)\,(\forall x_1, x_2 \in X_i)\,[x_1 \text{ and } x_2 \text{ are control consistent}].$$

If a cover $\mathfrak{C}$ of **C** is deterministic and control consistent then we will simply call it a *control cover*. A control cover can be used to derive a new supervisor that is control equivalent to **C**. If $|\mathfrak{C}| = |I| < \left|X^C\right|$ then the new supervisor will be a reduced supervisor.

Definition 75 *Assume that $\mathfrak{C} = \left\{X_i \subseteq X^C : i \in I\right\}$ is a control cover of* **C**. *Then*

$$\mathbf{C}' := \{I, \Sigma, E', i_0, I_m\}$$

is an automaton induced by $\mathfrak{C}$ *where $i_0 \in I$ is such that $x_0^C \in X_{i_0}$ and*

$$E' := \left\{(i, \sigma, j) : \left(\exists\,(x, \sigma, y) \in E^C\right)\,[x \in X_i \wedge y \in X_j]\right\}.$$

Theorem 76 *[VW86]Let* $\mathbf{C}$ *be a standard supervisor for* $\mathbf{G}$, *let* $\mathfrak{C}$ *be a control cover for* $\mathbf{C}$, *and let* $\mathbf{C}'$ *be an automaton induced by* $\mathfrak{C}$. *Then*

(i) $L(\mathbf{C}'/\mathbf{G}) = L(\mathbf{C}/\mathbf{G})$, *and*

(ii) $L_m(\mathbf{C}'/\mathbf{G}) = L_m(\mathbf{C}/\mathbf{G})$.

In general, to find a cover with the least number of elements we may have to compute all possible control covers of $\mathbf{C}$. This is the reason behind the NP-hardness of the problem. However, it is possible to intuitively come up with a *good enough* control cover. Su and Wonham [SW01] give a polynomial algorithm for finding one such control cover; in fact, their cover is a partition. They define a control congruence on the state set of the given supervisor. This allows them to derive a reduced supervisor in polynomial time even though there is no guarantee that the reduced supervisor is minimal (or even close to it). In order to gauge how far they are from a minimal supervisor, they propose an algorithm to find a lower bound on the state set of a minimum supervisor. They find the biggest subset of the states of $\mathbf{C}$ such that all the states in this set are mutually control inconsistent. No two states in this subset can be in the same element of a control consistent cover. So it follows that the number of states in a minimal supervisor can be no fewer than the size of this subset. We will show later that this is very conservative and that it is possible to come up with a better estimate.

4.4. An Algorithm for Finding a Control Cover

Given a supervisor $\mathbf{C}$ for a specification $\mathbf{S}$ on a plant $\mathbf{G}$ we want to find a supervisor $\mathbf{C}'$ that is control equivalent to $\mathbf{C}$. For the time being we set aside the requirement that

$\left|X^{C'}\right| < \left|X^{C}\right|$. This may be expressed linguistically as follows:

$$\begin{aligned} L(\mathbf{C}') \cap L(\mathbf{G}) &= L(\mathbf{C}) \cap L(\mathbf{G}), \\ L_m(\mathbf{C}') \cap L_m(\mathbf{G}) &= L_m(\mathbf{C}) \cap L_m(\mathbf{G}). \end{aligned}$$

Lemma 77 *Let* $\mathbf{C}'$ *be any automaton such that*

$$L(\mathbf{C}') = L(\mathbf{C}) \cup L' \tag{4.1}$$

$$L_m(\mathbf{C}') = L_m(\mathbf{C}) \cup L'_m \tag{4.2}$$

where $L' \subseteq \Sigma^* - L(\mathbf{G})$ *and* $L'_m \subseteq \Sigma^* - L_m(\mathbf{G})$. *Then* $\mathbf{C}'$ *is control equivalent to* $\mathbf{C}$. *Additionally, if* $\mathbf{C} \leq \mathbf{G} \| \mathbf{S}$ *then 4.1 and 4.2 are necessary as well.*

Proof. We have

$$\begin{aligned} L(\mathbf{C}') \cap L(\mathbf{G}) &= (L(\mathbf{C}) \cup L') \cap L(\mathbf{G}) \\ &= (L(\mathbf{C}) \cap L(\mathbf{G})) \cup (L' \cap L(\mathbf{G})) \\ &= (L(\mathbf{C}) \cap L(\mathbf{G})) \cup \emptyset \\ &= L(\mathbf{C}) \cap L(\mathbf{G}) \end{aligned}$$

since $L' \subseteq \Sigma^* - L(\mathbf{G})$. Similarly we get

$$\begin{aligned} L_m(\mathbf{C}') \cap L_m(\mathbf{G}) &= (L_m(\mathbf{C}) \cup L'_m) \cap L_m(\mathbf{G}) \\ &= (L_m(\mathbf{C}) \cap L_m(\mathbf{G})) \cup (L'_m \cap L_m(\mathbf{G})) \\ &= L_m(\mathbf{C}) \cap L_m(\mathbf{G}). \end{aligned}$$

If $\mathbf{C} \leq \mathbf{G} \| \mathbf{S}$ then $L(\mathbf{C}) \subseteq L(\mathbf{G})$ so

$$\begin{aligned} L(\mathbf{C}) &= L(\mathbf{C}) \cap L(\mathbf{G}) \\ &= L(\mathbf{C}') \cap L(\mathbf{G}) \\ &\subseteq L(\mathbf{C}'). \end{aligned}$$

Since $L(\mathbf{C}) \subseteq L(\mathbf{C}')$ we get

$$\begin{aligned} L(\mathbf{C}') \cap L(\mathbf{G}) &= \Big[L(\mathbf{C}) \dot{\cup} (L(\mathbf{C}') - L(\mathbf{C}))\Big] \cap L(\mathbf{G}) \\ &= [L(\mathbf{C}) \cap L(\mathbf{G})] \dot{\cup} [(L(\mathbf{C}') - L(\mathbf{C})) \cap L(\mathbf{G})] \end{aligned}$$

which implies that

$$(L(\mathbf{C}') - L(\mathbf{C})) \cap L(\mathbf{G}) \subseteq L(\mathbf{C}) \cap L(\mathbf{G}).$$

But this is only possible if $(L(\mathbf{C}') - L(\mathbf{C})) \cap L(\mathbf{G}) = \emptyset$. In other words, $L(\mathbf{C}') - L(\mathbf{C}) \subseteq \Sigma^* - L(\mathbf{G})$. A similar argument gives $L_m(\mathbf{C}') - L_m(\mathbf{C}) \subseteq \Sigma^* - L_m(\mathbf{G})$. ■

This result can be used to formulate a greedy algorithm for the construction of a control equivalent supervisor. This greedy algorithm can be informally described as follows. Let $\mathfrak{C} = \left\{X_i \subseteq X^C : i \in I\right\}$ be a control cover. Then there must exist $i_0 \in I$ such that $x_0^C \in X_{i_0}$. So we begin our process at the initial state and define X_{i_0} to be a maximal set containing x_0^C such that all its elements are pairwise consistent from the point of view of supervision. Additionally any states reachable from the states of X_{i_0} under the action of the same event should also be consistent. Then for each event σ eligible at any state in X_{i_0} we find the maximal set, say X_1, containing $\eta^C(X_{i_0}, \sigma)$ such that all its elements are pairwise consistent. Again any states reachable from the states of X_1 under the action of the same

event should also be consistent. If $X_1 \nsubseteq X_{i_0}$ then we consider it to be an element of $\mathfrak{C}$. We continue this process until we have found a cover. This cover will be a control cover because it is deterministic and control consistent by design. It is a greedy algorithm because at each step we try to find a maximal element of $\mathfrak{C}$ that respects the transition structure of $\mathbf{C}$ and is control consistent. We will later show that an automaton induced by $\mathfrak{C}$ satisfies 4.1, 4.2 and is therefore an equivalent supervisor; however there is no guarantee that it is a reduced supervisor. The intuitive reason for expecting a reduced supervisor lies in the fact that a recognizer for a *less specific* language often has far fewer states than a recognizer for a *more specific* language. By the specificity of a language we roughly mean the detail needed for its description. For example, Σ^* is a less specific language than $H = \{\alpha^*\beta\}$ because it can be described by simply saying "anything over Σ." On the other hand a description for H would be something such as "any number of α's followed by a β." A recognizer for Σ^* needs just one state while a recognizer for H needs 2 states. Let us assume that $\Sigma = \{\alpha, \beta\}$. Then Σ^* and H are shown in Figure 4.3. Now if we merge states x_0 and x_1 in the recognizer

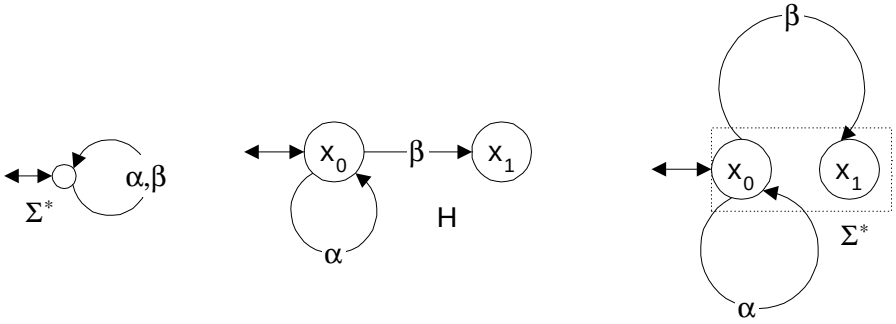


Figure 4.3: Reduction in Specificity by Merging States

for H then we get an automaton that generates a superlanguage of H (in fact it generates Σ^*) and has fewer states than the recognizer for H. In the proposed greedy algorithm we judiciously merge as many states as possible while forming the elements of the cover. If $\mathbf{C}'$

is the induced automaton then $L(\mathbf{C}') - L(\mathbf{C}) \subseteq \Sigma^* - L(\mathbf{G})$ so we merge as many states as possible in the hope that $L(\mathbf{C}') - L(\mathbf{C})$ will be approximately equal to $\Sigma^* - L(\mathbf{G})$.

There is another reason for using the greedy heuristic. It has been shown in [Chv79],[Hoc82] that a greedy heuristic offers very good approximations to the solution of the set covering problem. We will say more about the set covering problem in the next section. Here we just mention that in the numerous examples we have tried, the greedy heuristic seems to work quite well. Before we present the algorithm we give some definitions in the same spirit as [VW86].

Definition 78 *Let $x \in X^C$. Then the set of ineligible events at x is defined as*

$$Ineligible(x) := \Sigma_c - (Elig(\mathbf{C}, x) \cup Disabled(x)).$$

The set $Ineligible(x)$ comprises those events that are not physically possible at x. A distinction is thus made from those events that are disabled at x to impose some specification on $\mathbf{G}$.

Definition 79 *Any two states $x_1, x_2 \in X^C$ are defined to be* control compatible *if*

$$Disabled(x_1) \subseteq Disabled(x_2) \cup Ineligible(x_2), \text{ and}$$
$$Disabled(x_2) \subseteq Disabled(x_1) \cup Ineligible(x_1).$$

Definition 80 *Any two states $x_1, x_2 \in X^C$ are defined to be* marking compatible *if*

$$\left(\forall x \in X_m^G\right)\left[\left((x, x_1), (x, x_2) \in X^{G\|C}\right) \Rightarrow \left(x_1 \in X_m^C \leftrightarrow x_2 \in X_m^C\right)\right].$$

This requirement on marking is less restrictive than [VW86] where x_1 and x_2 are said

to be marking compatible only if $x_1 \in X_m^C \leftrightarrow x_2 \in X_m^C$. However any state $(x, y) \in X^{G\|C}$ can be a marked state only if $x \in X_m^G$ so we are able to relax the marking requirement.

Definition 81 *Any two states $x_1, x_2 \in X^C$ are defined to be* compatible *if they are control and marking compatible. For any $x \in X^C$ the sets of compatible and incompatible states are defined as*

$$Compatible\,(x) := \left\{y \in X^C : x \text{ and } y \text{ are compatible}\right\}$$

$$Incompatible\,(x) := \left\{y \in X^C : x \text{ and } y \text{ are not compatible}\right\}.$$

Definition 82 *Any two states $x_1, x_2 \in X^C$ are defined to be* mergeable *if they are compatible and*

$$(\forall \sigma \in \Sigma)\left[\eta^C(x_1, \sigma) \text{ and } \eta^C(x_2, \sigma) \text{ are compatible}\right].$$

For any $x \in X^C$ the set of its mergeable states is defined as

$$Mergeable\,(x) := \left\{y \in X^C : x \text{ and } y \text{ are mergeable}\right\}.$$

Thus two states are mergeable if they are compatible and their descendants under the action of the same events are compatible. Any cover whose elements comprise mutually mergeable states will be deterministic if the transition structure of the supervisor is respected. To compute the mergeable sets, we have adapted the algorithm given in [HU79, page 68] for the computation of a minimal state automaton. In [HU79] the algorithm distinguishes between marked and unmarked states while our adaptation distinguishes between compatible and incompatible states. The adapted algorithm is used in the procedure *findMergeableStateSets.*

proc *findMergeableStateSets*

input: $\mathbf{G}, \mathbf{C}$

output: $Mergeable(x)$ for each $x \in X^C$

begin

for all $x \in X^C$ and each $y \in Incompatible(x)$ **do**

cross (x, y);

endfor

for each pair of distinct states (x, y) in $X^C \times X^C$ **do**

if for some $\alpha \in \Sigma$, $(\eta^C(x, \alpha), \eta^C(y, \alpha))$ is crossed **then**

begin

cross (x, y);

recursively cross all uncrossed pairs on the list for (x, y) and on the list of other pairs that are crossed at this step;

end

else /* no pair $(\eta^C(x, \alpha), \eta^C(y, \alpha))$ is crossed */

for all $\alpha \in \Sigma$ **do**

put (x, y) on the list for $(\eta^C(x, \alpha), \eta^C(y, \alpha))$ unless $\eta^C(x, \alpha) = \eta^C(y, \alpha)$;

endfor

endif

endfor

for all $x \in X^C$ **do**

$Mergeable(x) := \{y \in X^C : (x, y)$ is not crossed$\}$;

endfor

end

From the analysis given in [HU79] we can infer the complexity of *findMergeableStateSets* to be $O\left(|\Sigma|\left|X^C\right|^2\right)$. Now that we know the mergeable state sets, we can describe the procedure for computing the elements of our desired control cover. As mentioned earlier, we begin the process by computing an element X_{i_0} that contains the initial state of **C**. Let α be an event that is eligible at any state belonging to X_{i_0}. Then to ensure that the resultant cover is deterministic we have to ensure that $\eta^C(X_{i_0}, \alpha)$ is a subset of some element of the cover. If $\eta^C(X_{i_0}, \alpha)$ is not a subset of an existing element then we need to compute an element that contains it. So now we describe a procedure that does exactly that.

proc *findMaximalMutuallyMergeableSet*

input: $S \subseteq X^C$ - all its elements are pairwise mergeable

output: $X_i \supseteq S$ - a maximal set containing S such that all its elements are pairwise mergeable

begin

1. $X_i := S$;
2. $tempMergeable := \cap_{x \in X_i} Mergeable(x)$;
3. **while** $|tempMergeable| > |X_i|$ **do**
4. pick a $y \in tempMergeable - X_i$ such that $tempMergeable \cap Mergeable(y)$ is maximal, i.e. $(\forall z \in tempMergeable - X_i)(|tempMergeable \cap Mergeable(y)|$ $> |tempMergeable \cap Mergeable(z)|)$;
5. $X_i := X_i \cup \{y\}$;
6. $tempMergeable := tempMergeable \cap Mergeable(y)$;
7. **endwhile**

end

The procedure *findMaximalMutuallyMergeableSet* takes as input a set S whose elements

are pairwise mergeable. It outputs a set $X_i \supseteq S$ such that all its elements are pairwise mergeable. Additionally X_i is maximal in the sense that if there is any other set $X_j \supseteq S$ all of whose elements are also pairwise mergeable then $|X_j| \leq |X_i|$. This set X_i will form an element of our desired cover if it is not a subset of any existing element. The procedure makes a nondeterministic choice at line 4 when it picks a state that maximizes the size of the temporary set $tempMergeable$. It is possible that more than one element satisfies the criterion. The choice made at line 4 does not affect the correctness of the procedure; however it may affect the size of the overall cover. This choice can be fine-tuned based on heuristic arguments. For instance, we can pick an element that has not been picked in any previous calls to *findMaximalMutuallyMergeableSet*.

The complexity of *findMaximalMutuallyMergeableSet* can be inferred as follows. The *while* loop block of lines 3 through 7 is executed at most $\left|X^C\right|$ times. Each time the *while* loop executes, there are at most $\left|X^C\right|$ comparisons done at line 4. So the overall complexity of the procedure is $O\left(\left|X^C\right|^2\right)$.

The main procedure can now be described very easily.

proc *findControlCover*

input: $\mathbf{G}, \mathbf{C}$

output: $\mathfrak{C}$ - a control cover for $\mathbf{C}$

begin

1. $findMergeableStates$;
   $X_{i_0} := findMaximalMutuallyMergeableSet(\{x_0^C\})$;
   $\mathfrak{C} := \{X_{i_0}\}$;
   $unprocessed := \{X_{i_0}\}$;
2. **while** $unprocessed \neq \emptyset$ **do**

$current :=$ first element of $unprocessed$;

$unprocessed := unprocessed - current$;

3. **for** each $\alpha \in \Sigma$ s.t. $\alpha \in Elig(\mathbf{C}, x)$ for some $x \in current$ **do**

4. $X_j := findMaximalMutuallyMergeableSet(\eta^C(current, \alpha))$;

if $X_j \not\subseteq X_i$ for any $X_i \in C$ **then**

$\mathfrak{C} := \mathfrak{C} \cup \{X_j\}$;

$unprocessed := unprocessed \cup \{X_j\}$;

endif

5. **endfor**

6. **endwhile**

end

The complexity of *findControlCover* may be inferred as follows. We know that the complexity of line 1 is $O\left(|\Sigma|\left|X^C\right|^2\right)$. The *for* loop block of lines 3 through 5 is executed $|\Sigma|$ times in the worst case. The complexity of line 4 is $O\left(\left|X^C\right|^2\right)$ so the overall complexity of the *for* loop block is $O\left(|\Sigma|\left|X^C\right|^2\right)$. Now let us consider the *while* loop block of lines 2 through 6. If *findControlCover* does not terminate before $\left|X^C\right|$ iterations of the *while* loop then the cover $\mathfrak{C}$ will have more than $\left|X^C\right|$ elements; the induced automaton will not be a reduced supervisor. So we can force the algorithm to terminate after $\left|X^C\right|$ passes of the *while* loop. Thus the complexity of *findControlCover* is $O\left(|\Sigma|\left|X^C\right|^3\right)$. Whenever it terminates normally we will be able to get a reduced supervisor. Before showing the correctness of *findControlCover*, we illustrate its operation with the help of a simple example.

Example 83 *Let us reconsider the Small Factory setup of Example 39. For simplicity, let us assume that there is only one machine feeding the buffer and only one machine emptying the buffer. The Smaller Factory setup is shown in Figure 4.4. As before, the*

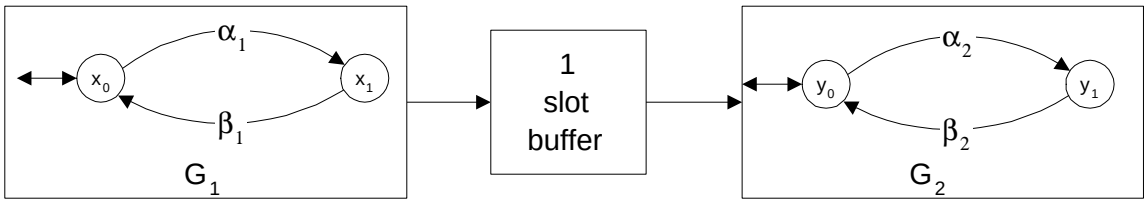


Figure 4.4: A Smaller Factory

imposed specification is the prevention of over/underflow of the buffer (as shown in Figure 4.5). This specification can be imposed by the supervisor **C** *shown in Figure 4.6. In fact,*

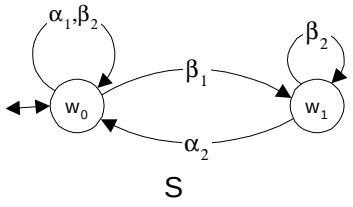


Figure 4.5: Buffer Under/Overflow Spec for the Smaller Factory

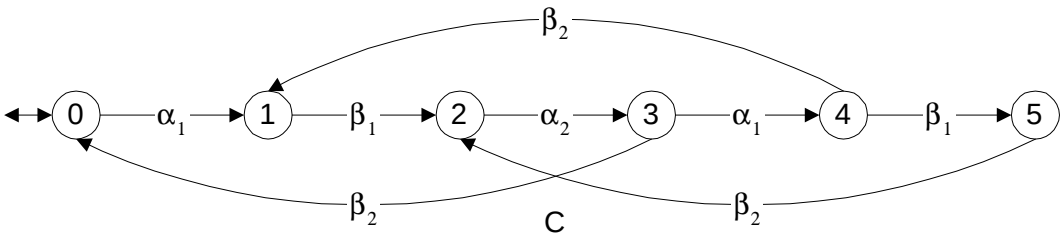


Figure 4.6: A Supervisor for the Smaller Factory

C *is the output of the CTCT design software [Won01]. We now desire to find a reduced supervisor for the Smaller Factory that is control equivalent to* **C**. *The table below shows*

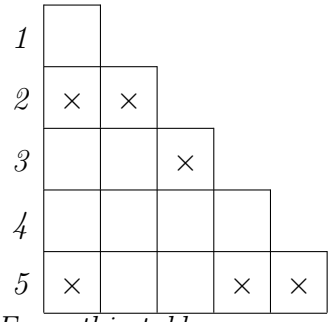
the disabled and the ineligible state sets for all the states of $\mathbf{C}$.

State	*Disabled Events*	*Ineligible Events*
0	$\{\alpha_2\}$	$\emptyset$
1	$\{\alpha_2\}$	$\{\alpha_1\}$
2	$\{\alpha_1\}$	$\emptyset$
3	$\emptyset$	$\{\alpha_2\}$
4	$\emptyset$	$\{\alpha_1, \alpha_2\}$
5	$\{\alpha_1\}$	$\{\alpha_2\}$

From this we can compute the compatible state sets shown below.

State	*Compatible States*
0	$\{0, 1, 3, 4, 5\}$
1	$\{0, 1, 3, 4, 5\}$
2	$\{2, 4, 5\}$
3	$\{0, 1, 3, 4\}$
4	$\{0, 1, 2, 3, 4, 5\}$
5	$\{1, 2, 4, 5\}$

With the compatible state sets in hand, we can execute findMergeableStateSets. *The table shows the intermediate output of* findMergeableStateSets. *A cross* ($\times$) *in entry* (i, j) *of the table signifies that the pair* (i, j) *is crossed.*



From this table we can compute the mergeable state sets shown below.

State	*Mergeable States*
0	$\{0,1,3,4\}$
1	$\{0,1,3,4,5\}$
2	$\{2,4,5\}$
3	$\{0,1,3,4\}$
4	$\{0,1,2,3,4\}$
5	$\{1,2,5\}$

Now we compute $X_{i_0} :=$ findMaximalMutuallyMergeableSet($\{0\}$). *Since all elements of* $Mergeable\,(0)$ *are pairwise mergeable we get*

$$\begin{aligned} X_{i_0} &= Mergeable\,(0) \\ &= \{0,1,3,4\}\,. \end{aligned}$$

The while-loop now begins with $\mathfrak{C} = unprocessed = \{\{0,1,3,4\}\}$. *In the first pass of the while-loop, current is defined as* X_{i_0} *and unprocessed gets updated to the empty set. The events eligible at* X_{i_0} *are* α_1, β_1 *and* β_2. *The for-loop now begins.*

First Iteration of for-loop: *For* α_1 *we have* $\eta^C\,(X_{i_0}, \alpha_1) = \{1,4\}$. *The output of the*

procedure findMaximalMutuallyMergeableSet($\{1,4\}$) *is equal to* X_{i_0} *so nothing else needs to be done.*

Second Iteration of for-loop: *For* β_1 *we have* $\eta^C(X_{i_0},\beta_1) = \{2,5\}$. *The output of* findMaximalMutuallyMergeableSet($\{2,5\}$) *is* $\{2,5\}$. *Since there is currently no element of* $\mathfrak{C}$ *that contains* $\{2,5\}$ *we add it to* $\mathfrak{C}$ *and unprocessed. So at this stage* $\mathfrak{C} = \{\{0,1,3,4\},\{2,5\}\}$ *while unprocessed* $= \{\{2,5\}\}$.

Third Iteration of for-loop: *For* β_2 *we have* $\eta^C(X_{i_0},\beta_2) = \{0,1\}$. *The output of* findMaximalMutual *is equal to* X_{i_0} *so nothing else needs to be done.*

This finishes the first iteration of the while-loop. In the second pass of the while-loop we process $\{2,5\}$ *and discover that no new sets are generated. So the procedure* findControlCover *terminates and returns* $\mathfrak{C} = \{\{0,1,3,4\},\{2,5\}\}$. *The automaton induced from* $\mathfrak{C}$ *is shown in Figure 4.7. It has only 2 states compared to the 6 states in* $\mathbf{C}$. *It turns out that*

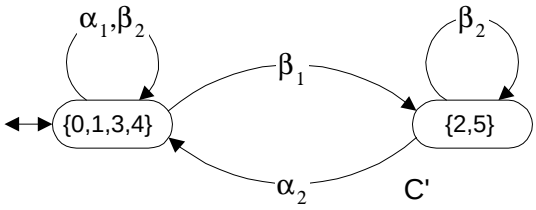


Figure 4.7: A Reduced Supervisor for the Smaller Factory

for this example our algorithm produces a minimal supervisor.

This example also provides an idea of the intuition behind supervisor reduction. For instance, α_1 *is eligible to occur at state* 0 *of* $\mathbf{C}$ *but ineligible to occur at state* 1 *of* $\mathbf{C}$. *However, from the point of view of implementing the over/underflow specification, both the states provide equivalent control action because* α_1 *cannot occur in the Small Factory while*

the supervisor $\mathbf{C}$ *is in state* 1. *It is such knowledge of the dynamics of a plant that allows us to merge the states of a given supervisor.* □

We have implemented this algorithm in the Java programming language. The program has not failed to terminate normally (i.e. without reduction) for any of the numerous examples we have tried. Table 4.1 shows reductions achieved by our program on a set of examples taken from [SW00] and [SW01]. For comparison we have also included the output of the algorithm proposed in [SW00] and [SW01]. The pair (n, m) means that the given automaton has n states and m transitions. The data for the given plant and supervisor automata is listed in the columns labelled **Plant** and **Super** respectively. The data related to the output generated by our program is listed in the column labelled **RSuper** while the data for the output generated by the algorithm of [SW00],[SW01] is given in the column labelled **SWSuper**.

Plant	**Super**	**RSuper**	**SWSuper**
(3,7)	(3,6)	(2,7)	(3,6)
(44,87)	(33,56)	(7,19)	(9,19)
(128,1288)	(776,3826)	(95,730)	(186,1130)
(390,1089)	(295,701)	(5,85)	(5,64)

Table 4.1: Typical Results of the Supervisor Reduction Algorithm

The plant and supervisor whose data are given in the first row of Table 4.1 are shown in Figure 4.8. This example has been taken from [SW00]. All the events are assumed to be controllable. The authors have shown in [SW00] that a reduced supervisor cannot be achieved using a control partition. Our algorithm is able to find a reduced supervisor in this case. In fact, the reduced supervisor is minimal and is shown in Figure 4.9. This example (and the others listed in Table 4.1) indicate the usefulness of the proposed algorithm despite the lack of a guarantee of reduction.

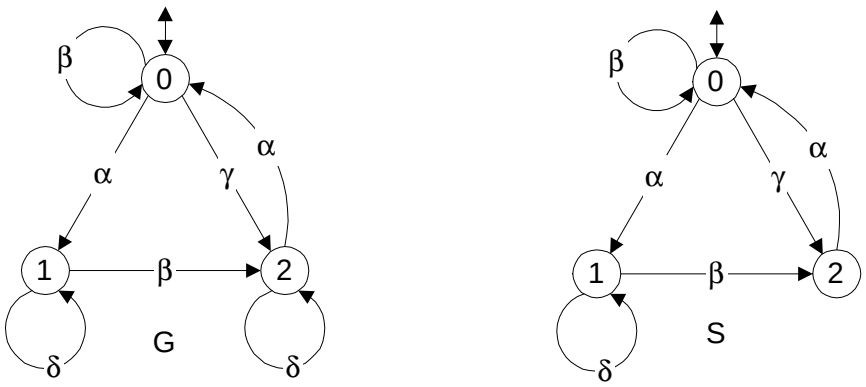


Figure 4.8: Supervisor Cannot be Reduced by a Partition

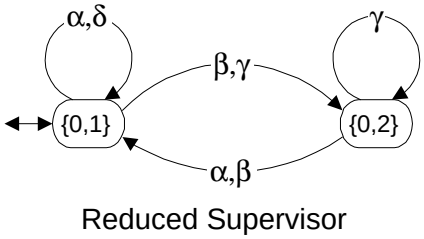


Figure 4.9: Supervisor Reducible using a Cover

Theorem 84 *Let* $\mathbf{G}$ *and* $\mathbf{C}$ *be automata representing a plant and its supervisor. Assume that* $\mathfrak{C} := findControlCover(\mathbf{G}, \mathbf{C})$ *and let* $\mathbf{C}'$ *be the automaton induced by it. Then* $\mathbf{C}'$ *is control equivalent to* $\mathbf{C}$*, i.e.*

$$L(\mathbf{G}) \cap L(\mathbf{C}') = L(\mathbf{G}) \cap L(\mathbf{C})$$
$$L_m(\mathbf{G}) \cap L_m(\mathbf{C}') = L_m(\mathbf{G}) \cap L_m(\mathbf{C}).$$

Proof. To prove this result it is sufficient to show that

$$L(\mathbf{C}') \quad = \quad L(\mathbf{C}) \cup L'$$

$$L_m(\mathbf{C}') = L_m(\mathbf{C}) \cup L'_m$$

where $L' \subseteq \Sigma^* - L(\mathbf{G})$ and $L'_m \subseteq \Sigma^* - L_m(\mathbf{G})$. The desired result will then follow from Lemma 77. Since $\mathbf{C}$ is trim, the set $\mathfrak{C}$ returned by procedure $findControlCover$ is at least a cover for the states of $\mathbf{C}$. This implies that $L(\mathbf{C}') \supseteq L(\mathbf{C})$ and $L_m(\mathbf{C}') \supseteq L_m(\mathbf{C})$. Thus in order to prove the result we just need to show that $L(\mathbf{C}') - L(\mathbf{C}) \subseteq \Sigma^* - L(\mathbf{G})$ and $L_m(\mathbf{C}') - L_m(\mathbf{C}) \subseteq \Sigma^* - L_m(\mathbf{G})$. We do this in two parts.

Part 1: Let $s \in L(\mathbf{C}') \cap L(\mathbf{C})$ and let $\sigma \in \Sigma$ be such that $s\sigma \in L(\mathbf{C}')$ but $s\sigma \notin L(\mathbf{C})$. We need to show that $s\sigma \notin L(\mathbf{G})$. If $s \notin L(\mathbf{G})$ then $s\sigma \notin L(\mathbf{G})$ so let us assume that $s \in L(\mathbf{G})$. Let $x_1 \in X^C$ be the state in $\mathbf{C}$ corresponding to s. Since $s\sigma \notin L(\mathbf{C})$ it follows that $\sigma \in Elig(\mathbf{C}, x)$. Let $i \in X^{C'}$ be the state in $\mathbf{C}'$ corresponding to s and let $X_i \subseteq X^C$ be the corresponding element of the cover $\mathfrak{C}$. Since $\sigma \in Elig(\mathbf{C}', i)$ there must exist some state $x_2 \in X_i \subseteq X^C$ such that $\sigma \in Elig(\mathbf{C}, y)$. Since x_1 and x_2 belong to the same element $X_i \in \mathfrak{C}$, they must be mergeable. So, we must have

$$Disabled(x_1) \subseteq Disabled(x_2) \cup Ineligible(x_2).$$

Now $\sigma \in Elig(\mathbf{C}, x_2)$ so $\sigma \notin Disabled(x_2) \cup Ineligible(x_2)$ which implies that $\sigma \notin Disabled(x_1)$. This means that there does not exist any state $z \in X^G$ such that $(z, x_1) \in X^{G\|C}$ and $\sigma \in Elig(\mathbf{G}, z)$. In particular, $\sigma \in Elig(\mathbf{G}, z_s)$ where z_s is the state in $\mathbf{G}$ corresponding to s. Thus $s\sigma \notin L(\mathbf{G})$. This shows that $(L(\mathbf{C}') - L(\mathbf{C})) \cap L(\mathbf{G}) = \emptyset$ which implies that $L(\mathbf{C}') - L(\mathbf{C}) \subseteq \Sigma^* - L(\mathbf{G})$.

Part 2: Let $s \in L_m(\mathbf{C}') - L_m(\mathbf{C})$. We need to show that $s \notin L_m(\mathbf{G})$. If $s \notin L(\mathbf{G})$ then $s \notin L_m(\mathbf{G})$ so let us assume that $s \in L(\mathbf{G})$. Thus $s \in L(\mathbf{C}') \cap L(\mathbf{G})$ which implies that $s \in L(\mathbf{C}) \cap L(\mathbf{G}) \subseteq L(\mathbf{C})$ (from Part 1). Let $x_1 \in X^C$ be the state

in $\mathbf{C}$ corresponding to s. Since $s \notin L_m(\mathbf{C})$ it follows that $x_1 \notin X_m^C$. Let $i \in X^{C'}$ be the state in $\mathbf{C}'$ corresponding to s and let $X_i \subseteq X^C$ be the corresponding element of the cover $\mathfrak{C}$. Since $s \in L_m(\mathbf{C}')$ it follows that there exists a state $x_2 \in X_i$ such that $x_2 \in X_m^C$. Now x_1 and x_2 are mergeable so we must have

$$\left(\forall z \in X_m^G\right)\left[\left((z,x_1),(z,x_2) \in X^{G\|C}\right) \Rightarrow \left(x_1 \in X_m^C \leftrightarrow x_2 \in X_m^C\right)\right]$$

which is equivalent to

$$\left[\left(\exists z \in X_m^G\right)\left((z,x_1),(z,x_2) \in X^{G\|C}\right)\right] \Rightarrow \left(x_1 \in X_m^C \leftrightarrow x_2 \in X_m^C\right).$$

Since $x_1 \notin X_m^C$ and $x_2 \in X_m^C$ it follows that the consequence in the above implication is false. However the implication itself must be true since x_1 and x_2 are mergeable. Thus the antecedent must be false, i.e.

$$\begin{gathered}\neg\left(\exists z \in X_m^G\right)\left((z,x_1),(z,x_2) \in X^{G\|C}\right) \\ \Updownarrow \\ \left(\forall z \in X_m^G\right)\left((z,x_1) \notin X^{G\|C} \vee (z,x_2) \notin X^{G\|C}\right) \qquad (4.3)\end{gathered}$$

is a tautology. Let $z_s \in X^G$ be the state in $\mathbf{G}$ corresponding to s. Since $s \in L(\mathbf{C}) \cap L(\mathbf{G}) = L(\mathbf{C}') \cap L(\mathbf{G})$ it follows that $(z_s,x_1),(z_s,x_2) \in X^{G\|C}$. Now we can conclude from (4.3) that $z_s \notin X_m^G$ which implies that $s \notin L_m(\mathbf{G})$. This shows that $(L_m(\mathbf{C}') - L_m(\mathbf{C})) \cap L_m(\mathbf{G}) = \emptyset$ which implies that $L_m(\mathbf{C}') - L_m(\mathbf{C}) \subseteq \Sigma^* - L_m(\mathbf{G})$. ■

4.5. Estimation of the Size of a Minimal Supervisor

In this section we present an estimate of the size of a minimal supervisor. In general it may be very hard to find a minimal supervisor but an almost minimal supervisor may be quite acceptable. Thus an estimate can be very useful in evaluating the output of a supervisor reduction algorithm. In [SW01] the authors have presented such an estimate. Their idea is to find a maximal set of states of the given supervisor such that all these states are pairwise unmergeable. Any control equivalent supervisor must have at least as many states as the size of this unmergeable set. This is a reasonable estimate; however it can be conservative as the following example demonstrates.

Example 85 *Let* **G** *and* **C** *represent a plant and its supervisor as shown in Figure 4.10. It is assumed that all the events are controllable. The table below shows the mergeable sets*

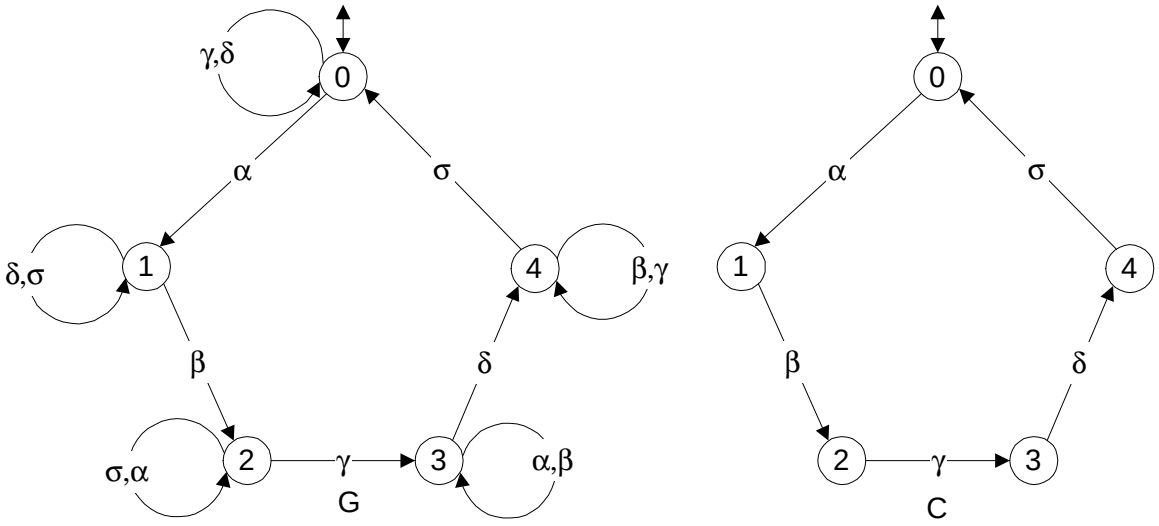


Figure 4.10: Conservative Lower Bound Estimation

for all the states.

State	*Mergeable States*
0	$\{0, 1, 4\}$
1	$\{0, 1, 2\}$
2	$\{1, 2, 3\}$
3	$\{2, 3, 4\}$
4	$\{0, 3, 4\}$

A state is mergeable with only those states that are adjacent to it; it is unmergeable with the two states that are not adjacent to it (but which are adjacent to each other). There are 5 maximal sets such that all the elements are pairwise unmergeable: $\{0, 2\}$, $\{0, 3\}$, $\{1, 3\}$, $\{1, 4\}$, *and* $\{2, 4\}$. *So, according to [SW01], a lower bound on the number of states of a minimal supervisor is 2. A minimal supervisor is shown in Figure 4.11 and has*

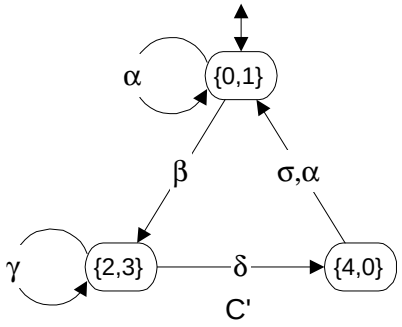


Figure 4.11: A Minimal Supervisor whose Estimate is Conservative

3 states. The reason for the conservative estimate of [SW01] is as follows. Consider a maximal unmergeable set such as $\{0, 2\}$. *It comprises only those states that are all pairwise unmergeable. There may exist states that are mergeable with some of the states of a maximal unmergeable set but it still may not be possible to merge them. For instance, states* 1 *and* 3 *are mergeable with* 2. *However we cannot merge all these three states because states* 1 *and*

3 *are unmergeable. In this example, it is not possible to form a control cover such that any element has more than* 2 *states.* □

The system in the above example exhibits a characteristic that can be used to derive a better estimate. Before explaining how that may be done, we need to give a bit of background regarding two classical NP-hard problems involving set covers [Joh74][GJ79].

Definition 86 *Let* $\mathfrak{F} := \{S_1, S_2, \ldots, S_p\}$ *be a finite family of finite sets. The* first set covering problem *(denoted* SC_1*) is to find a subcover* $\mathfrak{F}' \subseteq \mathfrak{F}$*, i.e. a subset such that*

$$\bigcup_{S \in \mathfrak{F}'} S = \bigcup_{S \in \mathfrak{F}} S.$$

In addition, if $\mathfrak{F}'' \subseteq \mathfrak{F}$ *is any other subcover then*

$$|\mathfrak{F}'| \leq |\mathfrak{F}''|.$$

In other words, given a cover $\mathfrak{F}$*, the first set covering problem is to find a minimal cardinality subcover* $\mathfrak{F}'$*.*

Definition 87 *Let* $\mathfrak{F} := \{S_1, S_2, \ldots, S_p\}$ *be a finite family of finite sets. The* second set covering problem *(denoted* SC_2*) is to find a subcover* $\mathfrak{F}' \subseteq \mathfrak{F}$*, i.e. a subset such that*

$$\bigcup_{S \in \mathfrak{F}'} S = \bigcup_{S \in \mathfrak{F}} S.$$

In addition, if $\mathfrak{F}'' \subseteq \mathfrak{F}$ *is any other subcover then*

$$\sum_{S \in \mathfrak{F}'} |S| \leq \sum_{S \in \mathfrak{F}''} |S|.$$

In other words, given a cover $\mathfrak{F}$, the second set covering problem is to a subcover $\mathfrak{F}'$ with the least overlap.

There exist good approximate solutions for SC_1 [Chv79],[Hoc82] and SC_2 [Joh74]. We present below a greedy approximation algorithm [Chv79] for SC_1.

proc $findApproximateSolutionSC_1$

input: $\mathfrak{F} := \{S_1, S_2, \ldots, S_p\}$ — a finite family of finite sets

output: $\mathfrak{F}_1' \subseteq \mathfrak{F}$ s.t. $\bigcup_{S\in\mathfrak{F}_1'} S = \bigcup_{S\in\mathfrak{F}} S$

begin

$T_i := S_i$ for all $1 \leq i \leq p$;

$\mathfrak{F}_1' := \emptyset$;

while $T_i \neq \emptyset$ for all $1 \leq i \leq p$ **do**

find $1 \leq j \leq p$ such that $|T_j|$ is maximal;

$\mathfrak{F}_1' := \mathfrak{F}_1' \cup \{S_j\}$;

$T_i := T_i - T_j$ for all $1 \leq i \leq p$;

endwhile

end

Theorem 88 *[Chv79]Let $\mathfrak{F} := \{S_1, \ldots, S_p\}$ be a finite family of finite sets and let $\mathfrak{F}_1' := findApproximateSolutionSC_1(\mathfrak{F})$. If $\mathfrak{F}_1^*$ is the optimal solution to SC_1 for $\mathfrak{F}$ then*

$$|\mathfrak{F}_1'| \leq \sum_{i=1}^{d} \frac{1}{i} |\mathfrak{F}_1^*|$$

where $d := \max\{|S_1|, \ldots, |S_p|\}$.

Let $\mathfrak{F}_2^*$ be an optimal solution for SC_2. It has been shown in [Joh74] that, in general, $\mathfrak{F}_2^*$ may be very far from any optimal solution for SC_1. However it is also shown in [Joh74]

that if $|S_i| = |S_j|$ for all $1 \leq i, j, \leq p$ then $\mathfrak{F}_2^*$ is also an optimal solution for SC_1. Now let us assume that

$$\mathfrak{F} := \left\{ X_i \subseteq X^C : (\forall x, y \in X_i)\, x \in Mergeable\,(y) \right\}$$

is cover of X^C such that all its elements comprise mutually mergeable states. Let $\mathfrak{F}^* \subseteq \mathfrak{F}$ be a subcover of X^C that induces a minimal supervisor which is control equivalent to $\mathbf{C}$. Let $\mathfrak{F}_1^* \subseteq \mathfrak{F}$ be a subcover of X^C such that it is an optimal solution for SC_1. Since all the states in any element of $\mathfrak{F}_1^*$ are pairwise mergeable, it satisfies one of the main conditions for control equivalence. However the automaton induced by $\mathfrak{F}_1^*$ may be nondeterministic so, in general, we will have $|\mathfrak{F}^*| \leq |\mathfrak{F}_1^*|$. Nonetheless $|\mathfrak{F}_1^*|$ is usually sufficiently close to $|\mathfrak{F}^*|$. Since $|\mathfrak{F}_1'|$ is an estimate for $|\mathfrak{F}_1^*|$ we may use it as an estimate for $|\mathfrak{F}^*|$. If it turns out that $\mathfrak{F}_1'$ is a partition then clearly it is an optimal solution for SC_2. Additionally if the S_i are all equal in size then we may conclude that $\mathfrak{F}_1'$ is also an optimal solution for SC_1 [Joh74]. In such a case $|\mathfrak{F}_1'|$ will be a very good estimate for $|\mathfrak{F}^*|$ (the size of a minimal supervisor). In fact this is what happens in Example 85. For the system in that example we have

$$\mathfrak{F} = \{\{0,1\}, \{0,4\}, \{1,2\}, \{2,3\}, \{3,4\}\}$$

and

$$\begin{aligned} \mathfrak{F}_1' &= findApproximateSolutionForSC_1\,(\mathfrak{F}) \\ &= \{\{0,1\}, \{2,3\}, \{0,4\}\}. \end{aligned}$$

Since all the elements of $\mathfrak{F}$ are of the same size it follows that $|\mathfrak{F}_1'| = 3$ is a very good estimate for the size of a minimal supervisor.

Remark 1 *We do not to need to know* $\mathfrak{F}$ *completely in advance; we can use the procedure* $findMaximalMutuallyMergeable$ *to compute the various elements of* $\mathfrak{F}$ *as the algorithm progresses.*

4.6. Summary

In this chapter we presented a heuristic algorithm for finding a reduced supervisor. Given a supervisor $\mathbf{C}$ for a plant $\mathbf{G}$ this algorithm produces a supervisor $\mathbf{C}'$ that is control equivalent to $\mathbf{C}$. While there is no guarantee that $\left|X^{C'}\right| < \left|X^{C}\right|$, the algorithm seems to perform quite well in practice. The proposed heuristic is greedy in nature and its complexity is $O\left(|\Sigma|\left|X^{C}\right|^{3}\right)$. We also presented an estimate for the size of a minimal supervisor. The estimation algorithm is based on an existing approximation for the set-covering problem [Chv79].

Su and Wonham [SW01],[SW00] have also presented algorithms for supervisor reduction and minimal supervisor size estimation. The algorithms presented in this chapter can be used in conjunction with their algorithms to achieve better results. Finding a minimal supervisor is NP-hard and an approximation algorithm that works well for one system may not work well for another. So a designer should try all the available algorithms and pick the one that works for the particular system at hand.

5. More Notation

5.1. An Overview

The rest of the thesis deals with timed discrete event systems (TDES). So in this chapter we introduce some additional notation related to timed discrete event systems. A TDES differs from an untimed DES in that there is an explicit notion of time. In an untimed DES it only makes sense to talk about the order in which various events occur. So it may be reasonable to say about a machine that it needs to be calibrated before it can be put to work but it is not possible to specify the exact time at which the calibration takes place. Similarly we cannot specify how long the calibration takes. Events in a DES are supposed to take place instantaneously and nondeterministically. For some systems it may be advantageous or even necessary to introduce time explicitly. It would then be possible to specify when an event takes place. There are various ways to model timed discrete event systems. In this thesis we use and extend the framework proposed by Brandin and Wonham [Bra93],[BW94]. Their framework is an extension of the RW framework and we will refer to it as the BW framework. In BW the essential idea is the modelling of the passage of a unit of time by a special event called a *tick*. The *tick* event models the passage of time according to a global clock. This explicit notion of time allows the introduction of activities. An activity is a state of a system that requires a time duration. For example, the time taken by a machine to process a workpiece is the duration of its *working* activity.

A system may be modelled by a graph of its activities called an *Activity Transition Graph*. This graph does not model time explicitly using the *tick* event; it simply associates time bounds with various events. However it can be converted into a graph called a *Timed Transition Graph* which models time explicitly using the *tick* event. The existing tools of the RW framework can be applied to a timed transition graph with minor modifications.

5.2. Notation for Timed Discrete Event Systems

A lot of the notation and definitions presented here are taken from [Bra93],[BW94],[Won01] with minor modifications; please refer to them for more details. If Σ represents an alphabet of events then, as before, let Σ^+ represent the language consisting of all possible strings of finite (non-zero) length. Let $\epsilon \notin \Sigma$ be the empty string; let $\Sigma^* := \Sigma^+ \cup \{\epsilon\}$ be the language comprising all possible strings of finite (possibly zero) length. Let t represent a *tick* of a global clock and let $\Sigma_t := \Sigma \cup \{t\}$. For each $\sigma \in \Sigma$, let $l_\sigma \in \mathbf{N}$ and $u_\sigma \in \mathbf{N} \cup \{\infty\}$ represent the lower and upper time bounds of σ. The lower bound l_σ represents the number of *ticks* that must elapse once σ is enabled before it is eligible to occur; the upper bound u_σ represents the maximum number of *ticks* that may elapse before σ is forced to occur. If $u_\sigma = \infty$ then σ is never forced to occur.

Definition 89 *An automaton* $\mathbf{G}_A$ *represents an* Activity Transition Graph *(ATG) of a system if*

$$\mathbf{G}_A = \left(X^{G_A}, \Sigma^{G_A}, E^{G_A}, x_0^{G_A}, X_m^{G_A}\right)$$

where the set of states X^{G_A} *represents a finite set of activities and each event* $\sigma \in \Sigma^{G_A}$ *has a lower bound* $l_\sigma \in \mathbf{N}$ *and an upper bound* $u_\sigma \in \mathbf{N} \cup \{\infty\}$. *We will often use the triple* $(\sigma, l_\sigma, u_\sigma)$ *to represent the time bounds associated with* σ. *If* $u_\sigma \neq \infty$ *then the event* σ *is*

said to be prospective*; if* $u_\sigma = \infty$ *then* σ *is said to be* remote*. Let*

$$\Sigma_{spe}^{G_A} \quad := \left\{\sigma \in \Sigma^{G_A} : \sigma \text{ is prospective}\right\}$$
$$\Sigma_{rem}^{G_A} \quad := \left\{\sigma \in \Sigma^{G_A} : \sigma \text{ is remote}\right\}$$

represent the sets of prospective and remote events. Let $[j, k]$ *represent the set of integers from* j *to* k *and define*

$$T'_\sigma = \begin{cases} [0, u_\sigma] & \text{if } \sigma \in \Sigma_{spe}^{G_A} \\ [0, l_\sigma] & \text{if } \sigma \in \Sigma_{rem}^{G_A} \end{cases}$$

to be the timer interval *of* σ*. Let*

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

represent the default timer value *of* σ*.*

If $l_\sigma = 0$ and $u_\sigma = \infty$ for all events $\sigma \in \Sigma^{G_A}$ then an ATG is the same as the standard untimed automaton model of the RW framework.

Example 90 *Let us consider the simple machine shown in Figure 5.1. An ATG of this*

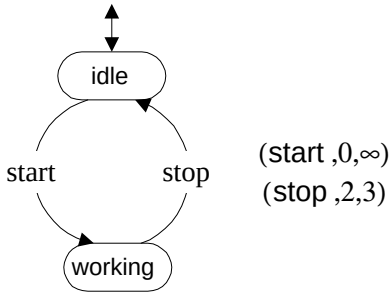


Figure 5.1: ATG of a Simple Machine

machine is

$$\left(\{idle, working\}, \{start, stop\}, \left\{ \begin{array}{c} (idle, start, working), \\ (working, stop, idle) \end{array} \right\}, idle, \{idle\} \right).$$

The time bounds are $(start, 0, \infty)$ *and* $(stop, 2, 3)$. *This may be interpreted as follows. The machine can be started as soon it is idle but does not need to started at all. When the machine is working, it cannot finish its operation before 2* ticks *but must finish before the* 4^{th} tick *(the upper bound requires that no more than* 3 ticks *take place). Here the timer intervals are*

$$\begin{aligned} T'_{start} &= [0,0] = \{0\} \\ T'_{stop} &= [0,3] = \{0,1,2,3\}. \end{aligned}$$

□

Definition 91 *Let* $s \in L \subseteq \Sigma^*$ *be a string belonging to a language* L. *A string* $w \in \Sigma^*$ *is a* prefix *of* s *in* L *(denoted* $w \leq s$*) if there exists a string* $v \in \Sigma^*$ *such that* $wv = s$. *Let*

$$Pre(L, s) := \{w \in \Sigma^* : w \leq s\}$$

be the set of all prefixes of s *in* L *and let*

$$Pre(L) := \{Pre(L, s) : s \in L\}$$

be the set of all prefixes of L. *We will often use* $\overline{L}$ *to denote* $Pre(L)$. *If* L *is obvious from the context then we may use* $Pre(s)$ *to denote* $Pre(L, s)$.

Definition 92 *Let $s \in L \subseteq \Sigma^*$ be a string belonging to a language L. A string $w \in \overline{L}$ is a* postfix *of s in L (denoted $w \geq s$) if s is prefix of w in $\overline{L}$, i.e. if there exists a string $v \in \Sigma^*$ such that $sv = w$. Let*

$$Pos\,(L, s) := \left\{w \in \overline{L} : w \geq s\right\}$$

be the set of all postfixes of s in L and let

$$Pos\,(L) := \{Pos\,(L, s) : s \in L\}$$

be the set of all postfixes of L. If L is obvious from the context then we may use $Pos\,(s)$ to denote $Pos\,(L, s)$.

Definition 93 *Given an ATG $\mathbf{G}_A$ for a system, the corresponding* Timed Transition Graph *(TTG) is defined to be an automaton*

$$\mathbf{G}_T = \left(X^{G_T}, \Sigma^{G_T}, E^{G_T}, x_0^{G_T}, X_m^{G_T}\right)$$

where

$$\begin{aligned}
X^{G_T} &= X^{G_A} \times \prod \left\{T'_\sigma : \sigma \in \Sigma^{G_A}\right\} \\
\Sigma^{G_T} &= \Sigma^{G_A} \dot{\cup} \{t\} \\
x_0^{G_T} &= \left(x_0^{G_A}, \left\{t_\sigma^0 : \sigma \in \Sigma^{G_A}\right\}\right) \\
X_m^{G_T} &\subseteq X_m^{G_A} \times \prod \left\{T'_\sigma : \sigma \in \Sigma^{G_A}\right\}
\end{aligned}$$

and the transition set E^{G_T} is defined as follows. Let $x_1 = \left(a_1, \left\{t_\sigma^1 : \sigma \in \Sigma^{G_A}\right\}\right)$ and $x_2 = \left(a_2, \left\{t_\sigma^2 : \sigma \in \Sigma^{G_A}\right\}\right)$ belong to X^{G_T}. Then $(x_1, \alpha, x_2) \in E^{G_T}$ if and only if

1. $\alpha = t$ *and* $\left(\forall \sigma \in \Sigma_{spe}^{G_A}\right) t_\sigma^1 > 0$, *or*

2. $\alpha \in \Sigma_{spe}^{G_A}$, $\eta^{G_A}(a_1, \alpha)!$, *and* $0 \leq t_\alpha^1 \leq u_\alpha - l_\alpha$, *or*

3. $\alpha \in \Sigma_{rem}^{G_A}$, $\eta^{G_A}(a_1, \alpha)!$, *and* $t_\alpha^1 = 0$.

If any of the above three conditions is satisfied then x_2 *is given as follows.*

1. *If* $\alpha = t$ *then* $a_2 = a_1$. *Additionally, if* σ *is prospective then*

$$t_\sigma^2 = \begin{cases} u_\sigma & \text{if } \neg\eta^{G_A}(a_1, \sigma)! \\ t_\sigma^2 - 1 & \text{if } \eta^{G_A}(a_1, \sigma)! \end{cases};$$

otherwise if σ *is remote then*

$$t_\sigma^2 = \begin{cases} l_\sigma & \text{if } \neg\eta^{G_A}(a_1, \sigma)! \\ t_\sigma^2 - 1 & \text{if } \eta^{G_A}(a_1, \sigma)! \text{ and } t_\sigma^2 > 0 \\ 0 & \text{if } \eta^{G_A}(a_1, \sigma)! \text{ and } t_\sigma^2 = 0 \end{cases}.$$

2. *If* $\alpha \in \Sigma^{G_A}$ *then* $a_2 = \eta^{G_A}(a_1, \alpha)$. *If* $\sigma \neq \alpha$ *and* σ *is prospective then*

$$t_\sigma^2 = \begin{cases} u_\sigma & \text{if } \neg\eta^{G_A}(a_2, \sigma)! \\ t_\sigma^2 & \text{if } \eta^{G_A}(a_2, \sigma)! \end{cases};$$

if $\sigma = \alpha$ *and* σ *is prospective then*

$$t_\sigma^2 = u_\sigma;$$

if $\sigma \neq \alpha$ *and* σ *is remote then*

$$t_\sigma^2 = \begin{cases} l_\sigma & \text{if } \neg\eta^{G_A}(a_2, \sigma)! \\ t_\sigma^2 & \text{if } \eta^{G_A}(a_2, \sigma)! \end{cases};$$

if $\sigma = \alpha$ *and* σ *is remote then*

$$t_\sigma^2 = l_\sigma.$$

The above equations define the derivation of an explicit timed automaton (TTG) from an implicit timed automaton (ATG). Condition 1 says that if a *tick* has just occurred then appropriately decrement the timer values of all the other enabled events. Condition 2 says that if a non-*tick* event has just occurred then reset its timers to their default values but leave unchanged the timers of other enabled events. We illustrate the conversion procedure with the help of an example.

Example 94 *Let us reconsider the simple machine of Example 90. The TTG for this machine is shown in Figure 5.2. At the* idle *state in the ATG, the only eligible event is* start *with time bounds* $(0, \infty)$*. Thus any number of* tick *events can occur (i.e. any amount of time can lapse) before, if at all,* start *occurs at the initial state in the TTG. Once* start *occurs, the* stop *event becomes enabled. The lower time bound of* stop *is* 2 *so two* ticks *must occur before* stop *becomes eligible to occur. The upper time bound of* stop *is* 3 *so no more than three* ticks *can occur before* stop *is forced to occur. This can be seen from Figure 5.2: no* tick *is possible at the state* $(working, 0, 0)$*. In such a scenario, we say that the event* stop *is* imminent*. The selfloop of the* tick *event at state* $(idle, 0, 3)$ *indicates that the* start *event is never imminent.* □

Definition 95 *Let* $\mathbf{G}_A$ *be an ATG and let* $\mathbf{G}_T$ *be the corresponding TTG. Let* $x = \left(a, \left\{t_\sigma : \sigma \in \Sigma^{G_A}\right\}\right)$

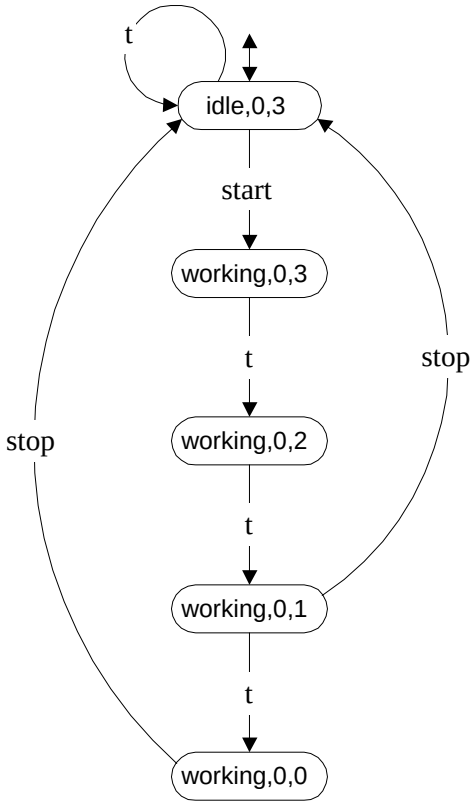


Figure 5.2: Timed Transition Graph of a Simple Machine

be a state in $\mathbf{G}_T$. *Then an event* $\sigma \in \Sigma^{G_A}$ *is* enabled *at* x *if there exists* $a_1 \in X^{G_A}$ *such that* $(a, \sigma, a_1) \in E^{G_A}$. *In other words, an event is enabled at a state in the TTG if the same event is eligible at the corresponding state in the ATG. Let*

$$Enab\left(\mathbf{G}_T, x\right) := \left\{\sigma \in \Sigma^{G_A} : \sigma \text{ is enabled at } x\right\}$$

be the set of all enabled events at x. *Sometimes it will be convenient to talk about the events enabled at a string. Let* $s \in L\left(\mathbf{G}_T\right)$ *and let* y *be the corresponding state in* $\mathbf{G}_T$. *Then*

$$Enab\left(L\left(\mathbf{G}_T\right), s\right) := Enab\left(\mathbf{G}_T, y\right).$$

Definition 96 *Let* $\mathbf{G}_T$ *be the corresponding TTG and let* $s \in L(\mathbf{G}_T)$. *Then an event* $\sigma \in \Sigma^{G_T}$ *is* eligible *at* s *if* $s\sigma \in L(\mathbf{G}_T)$. *Let*

$$Elig(L(\mathbf{G}_T), s) := \left\{\sigma \in \Sigma^{G_T} : s\sigma \in L(\mathbf{G}_T)\right\}$$

be the set of all eligible events *at* s. *Let* $x \in X^{G_T}$ *be the state corresponding to* s *in* $\mathbf{G}_T$. *Then clearly* $Elig(\mathbf{G}_T, x) = Elig(L(\mathbf{G}_T), s)$.

To simplify the already cumbersome notation, we adopt the following convention from now on. We assume that we are talking about a system whose ATG is represented by $\mathbf{A}$ and the corresponding TTG is represented by $\mathbf{G}$. We use Σ to represent Σ^A and use Σ_t to represent $\Sigma^G = \Sigma^A \dot{\cup} \{t\}$. We denote $L(\mathbf{G})$ by L, $L_m(\mathbf{G})$ by L_m, $L(\mathbf{A})$ by A and $L_m(\mathbf{A})$ by A_m. Let $P_t : \Sigma_t^* \to \{t\}^*$ represent the standard projection operator that erases all the occurrences of non-*tick* events:

$$\begin{aligned} P_t(\epsilon) &= \epsilon \\ P_t(\sigma) &= \begin{cases} t & \text{if } \sigma = t \\ \epsilon & \text{otherwise} \end{cases} \quad \text{for all } \sigma \in \Sigma_t \\ P_t(s\sigma) &= P_t(s)P_t(\sigma) \text{ for all } s \in \Sigma_t^*, \sigma \in \Sigma_t, \end{aligned}$$

and let $Q_t : \Sigma_t^* \to \Sigma^*$ represent the standard projection operator that erases all the occurrences of *ticks* :

$$\begin{aligned} Q_t(\epsilon) &= \epsilon \\ Q_t(\sigma) &= \begin{cases} \sigma & \text{if } \sigma \neq t \\ \epsilon & \text{otherwise} \end{cases} \quad \text{for all } \sigma \in \Sigma_t \\ Q_t(s\sigma) &= Q_t(s)Q_t(\sigma) \text{ for all } s \in \Sigma_t^*, \sigma \in \Sigma_t. \end{aligned}$$

For any fixed $n \in \mathbf{N}$, let $\alpha_1, \ldots, \alpha_n \in \Sigma$ represent some events (with the possibility that $\alpha_i = \alpha_j$ for $i \neq j$) such that $w = \alpha_1 \alpha_2 \cdots \alpha_n$ is a string in A. Define

$$S_w := \{s \in \Sigma_t^* \Sigma \cap L \mid Q_t(s) = w\}$$

to be the set of all strings in L that do not terminate with a *tick* and are equivalent under *tick*-projection. We shall often refer to the elements of $S_{\alpha_1 \cdots \alpha_n}$ as paths from α_1 to α_n. Assume $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ for some $m \in \mathbf{N}^+$ and let $s \in L$.

Definition 97 *Define*

$$T_s := \begin{bmatrix} \bar{t}_1 & \bar{t}_2 & \cdots & \bar{t}_m \\ \underline{t}_1 & \underline{t}_2 & \cdots & \underline{t}_m \end{bmatrix}$$

to be the timer matrix *corresponding to s, i.e. T_s displays the state of each timer (for each event label in the alphabet Σ) at that node in the TTG of L which corresponds to the string s. Define*

$$\mathbf{T} := \{T_s \mid s \in L\}$$

to be the set of all possible timer matrices.

Example 98 *Consider the ATG shown in Figure 5.3 and assume that the following triples represent the time bounds associated with the events: $(\alpha, 1, 2), (\beta, 2, 4)$ and $(\gamma, 3, 4)$. The corresponding TTG is shown in Figure 5.4. For $w = \alpha\beta\gamma$, we have*

$$S_w = \left\{ \begin{array}{c} t\alpha t\beta t^2\gamma, t\alpha t\beta t^3\gamma, t\alpha t^2\beta t\gamma, t\alpha t^2\beta t^2\gamma, t\alpha t^3\beta\gamma, t\alpha t^3\beta t\gamma, \\ t^2\alpha\beta t^3\gamma, t^2\alpha\beta t^4\gamma, t^2\alpha t\beta t^2\gamma, t^2\alpha t\beta t^3\gamma, t^2\alpha t^2\beta t\gamma, t^2\alpha t^2\beta t^2\gamma \end{array} \right\}.$$

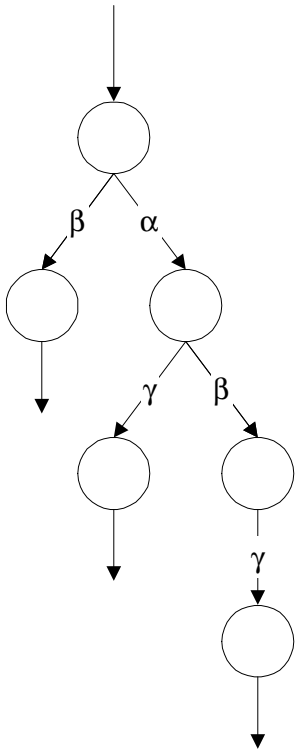


Figure 5.3: Activity Transition Graph of G

For $s = t\alpha$, *we have*

$$T_s = \begin{bmatrix} 2 & 3 & 4 \\ 1 & 1 & 3 \end{bmatrix},$$

while for $s = t\alpha t\beta$, *we have*

$$T_s = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 2 & 2 \end{bmatrix}.$$

In these timer matrices, the columns represent the upper and lower timer values associated with α, β, *and* γ *respectively.* □

In Chapter 7 we will present a compact model of TDES that is based on the idea of shortest and longest time paths in a TTG. We will pose and solve the shortest and longest

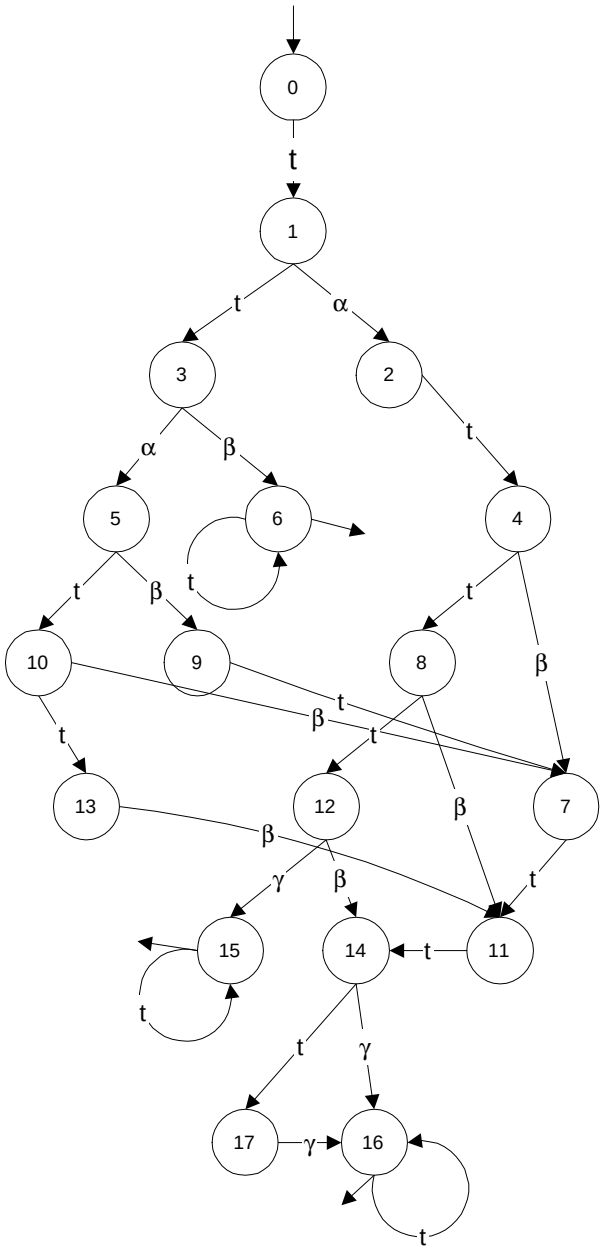


Figure 5.4: Timed Transition Graph of G

path problems in a dynamic programming framework. In order to formally define the shortest and longest path problems we need the following notation.

For $1 \leq j \leq k \leq n$, let $M_k[i_1, \ldots, i_{j-1}] \subseteq L$ be defined as

$$M_k\left[i_1, \ldots, i_{j-1}\right] := \left\{ t^{i_j}\alpha_j \cdots t^{i_k}\alpha_k \mid t^{i_1}\alpha_1 \cdots t^{i_{j-1}}\alpha_{j-1}t^{i_j}\alpha_j \cdots t^{i_k}\alpha_k \in L \right\}$$

where

$$M_k[] := \left\{ t^{i_1}\alpha_1 \cdots t^{i_k}\alpha_k \in L \right\}$$

denotes the set for $j = 1$. Similarly, for $1 \leq j \leq k \leq n$, let $M'_j[i_{k+1}, \ldots, i_n] \subseteq L$ be defined as

$$M'_j\left[i_{k+1}, \ldots, i_n\right] := \left\{ \begin{array}{c} t^{i_j}\alpha_j \cdots t^{i_k}\alpha_k \mid (\exists t^{i_1}\alpha_1 \cdots t^{i_{j-1}}\alpha_{j-1} \in L) \\ t^{i_1}\alpha_1 \cdots t^{i_{j-1}}\alpha_{j-1}t^{i_j}\alpha_j \cdots t^{i_k}\alpha_k t^{i_{k+1}}\alpha_{k+1} \cdots t^{i_n}\alpha_n \in L \end{array} \right\}$$

where

$$M'_j[] := \left\{ \begin{array}{c} t^{i_j}\alpha_j \cdots t^{i_n}\alpha_n \mid (\exists t^{i_1}\alpha_1 \cdots t^{i_{j-1}}\alpha_{j-1} \in L) \\ t^{i_1}\alpha_1 \cdots t^{i_{j-1}}\alpha_{j-1}t^{i_j}\alpha_j \cdots t^{i_n}\alpha_n \in L \end{array} \right\}$$

denotes the set for $k = n$.

Example 99 *Again consider a plant G whose ATG is given in Figure 5.3; the TTG of G is as shown in Figure 5.4. Assume that we are interested in the string $w = \alpha\beta\gamma$ in L_{act} . Then*

$$\begin{aligned} M_1[] &= \left\{ t\alpha, t^2\alpha \right\}, \\ M_2[] &= \left\{ t\alpha t\beta, t\alpha t^2\beta, t\alpha t^3\beta, t^2\alpha\beta, t^2\alpha t\beta, t^2\alpha t^2\beta \right\}, \\ M_2[0] &= \emptyset, \end{aligned}$$

$$
\begin{aligned}
M_2[1] &= \left\{t\beta, t^2\beta, t^3\beta\right\}, \\
M_2[2] &= \left\{\beta, t\beta, t^2\beta\right\}, \\
M_3[] &= S_w, \\
M_3[0] &= M_3[0,1] = M_3[0,2] = M_3[0,3] = \emptyset, \\
M_3[1] &= \left\{t\beta t^2\gamma, t\beta t^3\gamma, t^2\beta t\gamma, t^2\beta t^2\gamma, t^3\beta\gamma, t^3\beta t\gamma\right\}, \\
M_3[2] &= \left\{\beta t^3\gamma, \beta t^4\gamma, t\beta t^2\gamma, t\beta t^3\gamma, t^2\beta t\gamma, t^2\beta t^2\gamma\right\}, \\
M_3[1,1] &= \left\{t^2\gamma, t^3\gamma\right\}, \\
M_3[1,2] &= \left\{t\gamma, t^2\gamma\right\}, \\
M_3[1,3] &= \left\{\gamma, t\gamma\right\}, \\
M_3[2,0] &= \left\{t^3\gamma, t^4\gamma\right\}, \\
M_3[2,1] &= \left\{t^2\gamma, t^3\gamma\right\}, \\
M_3[2,2] &= \left\{t\gamma, t^2\gamma\right\};
\end{aligned}
$$

and

$$
\begin{aligned}
M_3'[] &= \left\{\gamma, t\gamma, t^2\gamma, t^3\gamma, t^4\gamma\right\}, \\
M_2'[] &= \left\{t\beta t^2\gamma, t\beta t^3\gamma, t^2\beta t\gamma, t^2\beta t^2\gamma, t^3\beta\gamma, t^3\beta t\gamma, \beta t^3\gamma, \beta t^4\gamma\right\}, \\
M_2'[0] &= \left\{t^3\beta\right\}, \\
M_2'[1] &= \left\{t^2\beta, t^3\beta\right\}, \\
M_2'[2] &= \left\{t\beta, t^2\beta\right\}, \\
M_2'[3] &= \left\{\beta, t\beta\right\}, \\
M_2'[4] &= \left\{\beta\right\}, \\
M_1'[] &= S_w,
\end{aligned}
$$

$$
\begin{aligned}
M_1'[0] &= \left\{t\alpha t^3\beta\right\}, \\
M_1'[1] &= \left\{t\alpha t^2\beta, t\alpha t^3\beta, t^2\alpha t^2\beta\right\}, \\
M_1'[2] &= \left\{t\alpha t\beta, t\alpha t^2\beta, t^2\alpha t\beta, t^2\alpha t^2\beta\right\}, \\
M_1'[3] &= \left\{t\alpha t\beta, t^2\alpha\beta, t^2\alpha t\beta\right\}, \\
M_1'[4] &= \left\{t^2\alpha\beta\right\}, \\
M_1'[0,0] &= M_1'[1,0] = M_1'[2,0] = \emptyset, \\
M_1'[3,0] &= M_1'[3,1] = \{t\alpha\}, \\
M_1'[0,1] &= M_1'[1,1] = \emptyset, \\
M_1'[2,1] &= \left\{t\alpha, t^2\alpha\right\}, \\
M_1'[0,2] &= \emptyset, \\
M_1'[1,2] &= M_1'[2,2] = \left\{t\alpha, t^2\alpha\right\}, \\
M_1'[3,2] &= \emptyset, \\
M_1'[0,3] &= \left\{t^2\alpha\right\}, \\
M_1'[1,3] &= \left\{t\alpha, t^2\alpha\right\}, \\
M_1'[0,4] &= \left\{t^2\alpha\right\}.
\end{aligned}
$$

□

Define $\#t : \Sigma_t^* \to \mathbf{N}$ such that $\#t(s) = |P_t(s)|$, for all $s \in \Sigma_t^*$, where $|\cdot|$ represents the length of a string. Clearly, for any string (or path) $s = t^{i_j}\alpha_j \cdots t^{i_k}\alpha_k$ we have $\#t\,(s) = \sum_{m=j}^{k} i_m$; we shall refer to $\#t(s)$ as the length of s. Let $\dot{-}$ denote asymmetric subtraction,

i.e. for any $x, y \in \mathbf{N} \cup \{\infty\}$

$$x \dot{-} y = \begin{cases} x - y & \text{if } x > y \\ 0 & \text{otherwise.} \end{cases}$$

Here we follow the convention that $a \dot{-} \infty = 0$ and $\infty \dot{-} b = \infty$ for any $a \in \mathbf{N} \cup \{\infty\}$, $b \in \mathbf{N}$.

For $1 \leq j \leq k \leq n$, we define

$$f_k(i_1, \ldots, i_{j-1}) = \min_{s \in M_k[i_1, \ldots, i_{j-1}]} \#t(s)$$

and

$$f'_j(i_{k+1}, \ldots, i_n) = \min_{s \in M'_j[i_{k+1}, \ldots, i_n]} \#t(s).$$

For convenience of representation, we will use

$$f_k(\varepsilon) = \min_{s \in M_k[]} \#t(s)$$

to denote the case where $j = 1$ and use

$$f'_j(\varepsilon) = \min_{s \in M'_j[]} \#t(s)$$

to denote the case where $k = n$. Similarly we define

$$\begin{aligned} g_k(i_1, \ldots, i_{j-1}) &= \max_{s \in M_k[i_1, \ldots, i_{j-1}]} \#t(s), \\ g'_j(i_{j+1}, \ldots, i_n) &= \max_{s \in M'_j[i_{k+1}, \ldots, i_n]} \#t(s) \end{aligned}$$

and use

$$\begin{aligned} g_k(\varepsilon) &= \max_{s \in M_k[]} \#t\,(s)\,, \\ g'_j(\varepsilon) &= \max_{s \in M'_j[]} \#t\,(s) \end{aligned}$$

to denote the cases where $j = 1$ and $k = n$ respectively. In the above, we follow the convention that if $M_k\,[i_1, \ldots, i_{j-1}] = \emptyset$ then $f_k\,(i_1, \ldots, i_{j-1}) := \infty$ and $g_k\,(i_1, \ldots, i_{j-1}) := -\infty$. Similarly if $M'_j\,[i_{k+1}, \ldots, i_n] = \emptyset$ then $f'_j\,(i_{k+1}, \ldots, i_n) := \infty$ and $g'_j\,(i_{k+1}, \ldots, i_n) := -\infty$.

Definition 100 *The problem of finding the* length of a shortest path *of the form* $t^{i_1}\alpha_1 \cdots t^{i_n}\alpha_n$ *in* L *is the problem of evaluating* $f_n(\varepsilon)$ *or* $f'_1(\varepsilon)$. *Similarly, the problem of finding* the length of a longest path *of the form* $t^{i_1}\alpha_1 \cdots t^{i_n}\alpha_n$ *in* L *is the problem of evaluating* $g_n(\varepsilon)$ *or* $g'_1(\varepsilon)$.

Informally the shortest and longest path problems may be explained as follows. Assume that $\alpha_1 \cdots \alpha_n$ is a string in A. Then the shortest path problem involves finding a string s in L that corresponds to $\alpha_1 \cdots \alpha_n$, i.e. $Q_t\,(s) = \alpha_1 \cdots \alpha_n$. Additionally, s has the fewest number of *ticks* possible, i.e. if s_1 is another string in L such that $Q_t\,(s_1) = \alpha_1 \cdots \alpha_n$ then $\#t\,(s) \leq \#t\,(s_1)$. The longest path problem is analogous. The aim of doing all this is to extract critical time information from an ATG. This time information can then be used to form a model of the system that is often much more compact than a TTG.

For ease of notation, we define

$$\begin{aligned}
i_1^* &= f_1(\varepsilon), & j_n^* &= f_n'(\varepsilon), \\
i_2^* &= f_2(i_1^*), & j_{n-1}^* &= f_{n-1}'\left(j_n^*\right), \\
i_3^* &= f_3(i_1^*, i_2^*), & j_{n-2}^* &= f_{n-2}'\left(j_{n-1}^*, j_n^*\right), \\
&\vdots & &\vdots \\
i_n^* &= f_n(i_1^*, \ldots, i_{n-1}^*), & j_1^* &= f_1'\left(j_2^*, \ldots, j_n^*\right),
\end{aligned}$$

$$\begin{aligned}
i_2' &= f_2(i_1), & j_{n-1}' &= f_{n-1}'\left(j_n\right), \\
i_3' &= f_3(i_1, i_2'), & j_{n-2}' &= f_{n-2}'\left(j_{n-1}', j_n\right), \\
&\vdots & &\vdots \\
i_n' &= f_n(i_1, i_2', \ldots, i_{n-1}'), & j_1' &= f_1'\left(j_2', \ldots, j_{n-1}', j_n\right),
\end{aligned}$$

and

$$\begin{aligned}
I_1^* &= g_1(\varepsilon), & J_n^* &= g_n'(\varepsilon), \\
I_2^* &= g_2(I_i^*), & J_{n-1}^* &= g_{n-1}'\left(J_n^*\right), \\
I_3^* &= g_3(I_1^*, I_2^*), & J_{n-2}^* &= g_{n-2}'\left(J_{n-1}^*, J_n^*\right), \\
&\vdots & &\vdots \\
I_n^* &= g_n(I_1^*, \ldots, I_{n-1}^*), & J_1^* &= g_1'\left(J_2^*, \ldots, J_n^*\right).
\end{aligned}$$

6. Shortest and Longest Paths in a TTG

6.1. An Overview

The BW framework extends RW by explicitly introducing the notion of time. The time is measured with respect to a global clock; the passage of a unit of time is modelled by a special event called *tick*. Time bounds are associated with events and a timed transition graph is used to model timed discrete event systems. The introduction of time provides a greater flexibility in the modelling and supervision of many systems. However, this comes at a price: greater complexity. A timed model of a system is usually much bigger in size than the corresponding untimed model. Let us assume that **A** represents an ATG of a system and **G** represents the corresponding TTG. If we are not interested in the timed behaviour of the system then the ATG can serve as an untimed model: we simply ignore the time bounds associated with the various events. Then $\left|X^G\right|$ can be as big as $\left|X^A\right| \cdot \prod_{\sigma \in \Sigma^A} t^0_\sigma$ where t^0_σ is the default timer value of any event $\sigma \in \Sigma^A$. If we assume that $\left|\Sigma^A\right| = m$ and $t^0_\sigma = u$ for all $\sigma \in \Sigma^A$ then $\left|X^G\right|$ can be bigger than $\left|X^A\right|$ by a factor of u^m. Clearly the size increases quite rapidly as m (the number of events) increases. The size also increases quite rapidly as u increases for a fixed m: a TTG does not scale well. In other words, a timed transition graph becomes unwieldy and impractical as the default timer values increase. The main reason for this additional complexity is the modelling of time using *ticks*. In a TTG there are a number of states that are target states for no events other than the *tick*

event. The number of such states increases as the default timer values increase.

It is possible to extract the time information from a TTG without using the *tick* event. The essential idea is the following. Suppose $w = \sigma_1 \cdots \sigma_n$ is a string generated by an ATG and S_w is the set of corresponding time strings in the associated TTG. Then S_w may be described by explicitly listing all its elements. This is essentially what a TTG does; it describes all the timed strings corresponding to a string in the ATG. However it may also be possible to give an alternative description of S_w. For instance, we may be able to find minimal and maximal elements of S_w such that they define an envelope for the elements of S_w. If this is possible then the description of S_w may be greatly simplified. Brandin [Bra97],[Bra98] makes such an attempt although his approach is quite different.

In this chapter we systematically propose the problem of finding minimal and maximal elements of S_w. This is done by casting it in a dynamic programming framework [Bel61],[Dre65]. We show that multiple solutions may exist and provide two greedy solutions. These greedy solutions are much simpler than the standard recursive solutions to dynamic programming problems. We go on to show that only one of the two solutions is suitable for the purposes of supervisory control. We also provide a counterexample to show why Brandin's solution [Bra97],[Bra98] is unsuitable for supervisory control.

We will assume that $\mathbf{A}$ represents an ATG model of the system of interest and that $\mathbf{G}$ represents the corresponding TTG. We additionally denote $L(\mathbf{A})$ by A and $L(\mathbf{G})$ by L.

6.2. Shortest Paths

From the *Principle of Optimality* [Bel61],[Dre65] we know that the past decisions should have no effect on an optimal decision. Thus the minimization problem may be represented

[Bel61],[Dre65] recursively as

$$f_n(\varepsilon) = \min_{\left\{i_1 | t^{i_1}\alpha_1 \in M_1[]\right\}} (i_1 + f_n(i_1)) \tag{6.1}$$

and

$$f'_1(\varepsilon) = \min_{\{j_n | t^{j_n}\alpha_n \in M'_n[]\}} (j_n + f'_1(j_n)). \tag{6.2}$$

To compute $f_n(\varepsilon)$, we have to recursively find the optimal paths in the sets of legal postfixes of the substrings of a given string; to compute $f'_1(\varepsilon)$, we have to recursively find the optimal paths in the sets of legal prefixes of the substrings of a given string. We illustrate these ideas with the help of concrete examples.

Example 101 *Consider the TTG shown in figure 5.4 and assume that we are interested in finding the length of a shortest path of the form $t^i\alpha t^j\beta t^k\gamma$, where $i, j, k \in \mathbf{N}$. Recurrence relation (6.1) states that this problem can be solved by*

1. *finding the lengths of the shortest paths in the sets of postfixes of $t\alpha$ and $t^2\alpha$ (let these lengths be denoted by x and y respectively), and then*
2. *choosing the minimum of $1 + x$ and $2 + y$.*

In the above, we also have the additional requirement that the postfixes be of the form $t^j\beta t^k\gamma$ since we are interested in the strings of the form $t^i\alpha t^j\beta t^k\gamma$. Now in order to compute x and y, we have to repeat this whole process again. So x can be computed by

1. *finding the length of the shortest paths in the sets of postfixes of $t\alpha t\beta$, $t\alpha t^2\beta$, and $t\alpha t^3\beta$ (let these lengths be denoted by x_1, x_2, and x_3 respectively), and then*
2. *choosing the minimum of $1 + x_1$, $2 + x_2$, and $3 + x_3$.*

However, we have now reached the end of the recursion for x. So x_1, x_2, and x_3 can be computed directly from the TTG as 2, 1, and 1 respectively. Therefore,

$$\begin{aligned} x &= \min(1 + x_1, 2 + x_2, 3 + x_3) \\ &= \min(1 + 2, 2 + 1, 3 + 1) \\ &= 3. \end{aligned}$$

Similarly, we have

$$\begin{aligned} y &= \min(0 + 3, 1 + 2, 2 + 1) \\ &= 3. \end{aligned}$$

Thus the overall shortest path of the form $t^i\alpha t^j\beta t^k\gamma$ has length $\min(1 + 3, 2 + 3) = 4$ ticks. *The string corresponding to this solution is $t\alpha t\beta t^2\gamma$. As we shall see in the next example, this string is not unique, i.e. there may be more than one shortest (or longest) string.* □

Example 102 *Again consider the TTG shown in figure 5.4 and assume that we are interested in finding the length of a shortest path of the form $t^i\alpha t^j\beta t^k\gamma$, where $i, j, k \in \mathbf{N}$. Recurrence relation (6.2) states that this problem can be solved by*

1. *finding the lengths of the shortest paths over s_1, s_2, s_3 and s_4 such that $s_1\gamma, s_2t\gamma, s_3t^2\gamma, s_4t^4\gamma \in L$ and s_1, s_2, s_3, s_4 are of the form $t^i\alpha t^j\beta$ (let these lengths be denoted by w, x, y, and z respectively), and then*
2. *choosing the minimum of $0 + w$, $1 + x$, $2 + y$, and $3 + z$.*

In order to compute w, x, y, and z, we have to recurse through the whole process again. For instance, x can be computed by

1. *finding the lengths of the shortest paths over* s_5 *and* s_6 *such that* $s_5 t^2 \beta t \gamma$, $s_6 t^3 \beta t \gamma \in L$ *and* s_5, s_6 *are of the form* $t^i \alpha$ *(let these lengths be denoted by* x_1 *and* x_2 *respectively), and then*

2. *choosing the minimum of* $2 + x_1$ *and* $3 + x_2$.

At this point we come to the end of the recursion for the computation of x. *We can directly compute*

$$\begin{aligned} x_1 &= \min(1, 2) \\ &= 1, \\ x_2 &= 1 \end{aligned}$$

to get

$$\begin{aligned} x &= \min(2 + 1, 3 + 1) \\ &= 3. \end{aligned}$$

Similarly, we can compute

$$\begin{aligned} w &= \min(1 + 3) \\ &= 4, \\ y &= \min(1 + 1, 2 + 1, 2 + 2) \\ &= 2, \\ z &= \min(1 + 1) \\ &= 2. \end{aligned}$$

So the overall shortest path of the form $t^i\alpha t^j\beta t^k\gamma$ *is of length* $\min(0+4, 1+3, 2+2, 3+2) = 4$ ticks. *The strings corresponding to this solution are* $t\alpha t\beta t^2\gamma$, $t\alpha t^2\beta t\gamma$, *and* $t\alpha t^3\beta\gamma$. □

As can be seen from the above examples, the solution to finding the length of a shortest path is highly recursive; the complexity of such a solution is proportional to n^2 [Dre65]. We now show that in a TTG, the recurrence relations (6.1) and (6.2) can also be solved in a stepwise and non-recursive manner. The complexity of our solutions is proportional to n.

6.2.1. Postfix Solution to the Shortest Path Problem

We show that it is possible to find the value of $f_n(\varepsilon)$ by simply finding the path which has shortest paths between successive events, i.e.

$$\begin{aligned} f_n(\varepsilon) &= i_1^* + i_2^* + \cdots + i_n^* \\ &= \sum_{k=1}^{n} i_k^*. \end{aligned}$$

In other words, the problem of finding the shortest overall path can be broken down into n intermediate problems of finding the shortest path to the next event in the legal postfixes of a string. So we begin by finding the shortest path to α_1 i.e. we first evaluate i_1^*; then from there we find the shortest path to α_2, i.e. we use the value of i_1^* to evaluate i_2^*; and so on until α_n is reached i.e. until we can evaluate i_n^*. The solution to these intermediate problems can be computed using the lower time bounds associated with each event.

Lemma 103 *For any* $2 \leq k \leq n$,

$$f_k\left(i_1, \ldots, i_{k-1}\right) = \begin{cases} l_{\alpha_k} & \text{if } \left(\alpha_1 \cdots \alpha_{k-2}\alpha_k \notin A\right) \\ & \quad \vee \left(\alpha_k = \alpha_{k-1}\right) \\ l_{\alpha_k} \dot{-} i_{k-1} & \text{if } \left(\begin{array}{c} \left(\alpha_1 \cdots \alpha_{k-3}\alpha_k \notin A\right) \\ \vee \left(\alpha_k = \alpha_{k-2}\right) \end{array} \right) \\ & \quad \wedge \left(\alpha_1 \cdots \alpha_{k-2}\alpha_k \in L_{act}\right) \\ \vdots & \\ l_{\alpha_k} \dot{-} \sum_{j=1}^{k-1} i_j & \text{if } \left(\alpha_k \in L_{act}\right) \wedge \left(\alpha_1\alpha_k \in A\right) \\ & \quad \wedge \ldots \wedge \left(\alpha_1 \cdots \alpha_{k-2}\alpha_k \in A\right), \end{cases} \tag{6.3}$$

and

$$f_1(\varepsilon) = l_{\alpha_1}. \tag{6.4}$$

In the above equation, the various conditional clauses correspond to all the different possible enablement instants of α_k. For example, if $\left(\alpha_1 \cdots \alpha_{k-2}\alpha_k \notin A\right) \vee \left(\alpha_k = \alpha_{k-1}\right)$ then α_k was either last enabled after the occurrence of α_{k-1} or the labels α_{k-1} and α_k both refer to the same event. Similarly, if $\left(\alpha_1 \cdots \alpha_{k-3}\alpha_k \notin A\right) \vee \left(\alpha_k = \alpha_{k-2}\right)$ then either α_k was last enabled after the occurrence of α_{k-2} or the labels α_{k-2} and α_k both refer to the same event.

Theorem 104 *The shortest path problem*

$$f_n(\varepsilon) = \min_{\left\{i_1 | t^{i_1}\alpha_1 \in M_1[]\right\}} \left(i_1 + f_n\left(i_1\right)\right) \tag{6.5}$$

can also be solved by computing

$$\sum_{k=1}^{n} i_k^*. \tag{6.6}$$

Proof. See Appendix A (page 212). ■

6.2.2. Prefix Solution to the Shortest Path Problem

Here we present a dual approach to the solution of the shortest path problem. In computing $f_n(\varepsilon)$, we begin at the initial state of $\mathbf{G}$ and successively find the shortest path to the next event in the set of legal postfixes of the path already traversed. To compute $f_1'(\varepsilon)$ we first find the shortest path between α_{n-1} and α_n in $\mathbf{G}$, i.e. we find the shortest string in $M_n'[]$. We then successively find the shortest path to the preceding event in the set of its legal prefixes. So in a sense, to compute $f_n(\varepsilon)$ we find the shortest path between successive events as we move forward in $\mathbf{G}$ while to compute $f_1'(\varepsilon)$ we find the shortest path between successive events as we move backwards in $\mathbf{G}$. Therefore, the problem of finding the shortest overall path can be broken down into n intermediate problems of finding the shortest path to the previous event in the legal prefixes of a string. We begin by finding the shortest path between α_{n-1} and α_n i.e. we first evaluate j_n^*; then we find the shortest path from α_{n-2} to that point, i.e. we use the value of j_n^* to evaluate j_{n-1}^*; and so on until ϵ is reached i.e. until we can evaluate j_1^*.

Theorem 105 *The shortest path problem*

$$f_1'(\varepsilon) = \min_{\{j_n | t^{j_n} \alpha_n \in M_n'[]\}} (j_n + f_1'(j_n)) \tag{6.7}$$

can also be solved by computing

$$\sum_{k=1}^{n} j_k^*. \tag{6.8}$$

Proof. See Appendix A (page 216). ■

6.3. Longest Paths

Before attempting to find the maximal possible number of *ticks* in a path (in L) of the form $t^{i_1}\alpha_1 \cdots t^{i_n}\alpha_n$, we need to redefine the upper time limits associated with each event. The reason for this is illustrated with the following example. Let $n = 2$, i.e. consider $\alpha_1\alpha_2 \in A$. Assume $Enab\,(A, \alpha_1) = \{\alpha_2, \sigma\}$ where $u_\sigma < u_{\alpha_2}$. Then if α_2 is to occur at all after α_1 then it must do so before u_σ *ticks* otherwise it will be preempted by σ which is forced to occur before u_σ *ticks*. We now generalize this reasoning.

Lemma 106 *For $\alpha_1 \cdots \alpha_n \in L_{act}$, the effective upper bounds for α_k, $2 \leq k \leq n$, are*

$$u'_{\alpha_k} = \min\{u_\sigma - t_\sigma \mid \sigma \in Enab\,(A, \alpha_1 \cdots \alpha_{k-1})\}$$

and

$$u'_{\alpha_1} = \min\{u_\sigma \mid \sigma \in Enab\,(A, \epsilon)\},$$

where σ has been enabled for t_σ ticks.

We can assume that u'_{α_k}, $1 \leq k \leq n$, are finite because otherwise the longest time-path problem has a trivial solution: $g_n(\varepsilon) = \infty$. Then the rest follows along the lines of the previous section.

6.3.1. Postfix Solution to the Longest Path Problem

Lemma 107 *For any* $2 \leq k \leq n$,

$$g_k\left(i_1, \ldots, i_{k-1}\right) = \begin{cases} u'_{\alpha_k} & \text{if } \left(\alpha_1 \cdots \alpha_{k-2}\alpha_k \notin A\right) \\ & \quad \vee \left(\alpha_k = \alpha_{k-1}\right) \\ u'_{\alpha_k} \dot{-} i_{k-1} & \text{if } \left(\begin{array}{c} \left(\alpha_1 \cdots \alpha_{k-3}\alpha_k \notin A\right) \\ \vee \left(\alpha_k = \alpha_{k-2}\right) \end{array} \right) \\ & \quad \wedge \left(\alpha_1 \cdots \alpha_{k-2}\alpha_k \in A\right) \\ \vdots & \\ u'_{\alpha_k} \dot{-} \sum_{j=1}^{k-1} i_j & \text{if } \left(\alpha_k \in A\right) \wedge \left(\alpha_1\alpha_k \in A\right) \\ & \quad \wedge \ldots \wedge \left(\alpha_1 \cdots \alpha_{k-2}\alpha_k \in A\right), \end{cases} \tag{6.9}$$

and

$$g_1(\varepsilon) = u'_{\alpha_1}. \tag{6.10}$$

Theorem 108 *The longest path problem*

$$g_n(\varepsilon) = \max_{s \in M_k[i_1, \ldots, i_{j-1}]} \left(i_1 + g_n\left(i_1\right)\right) \tag{6.11}$$

can also be solved by computing

$$\sum_{k=1}^{n} I_k^*. \tag{6.12}$$

Proof. The proof follows exactly along the lines of the proof for Theorem 104. ■

6.3.2. Prefix Solution to the Longest Path Problem

Theorem 109 *The longest path problem*

$$g_1'(\varepsilon) = \max_{\{j_n | t^{j_n} \alpha_n \in M_n'[]\}} (j_n + g_1'(j_n)) \tag{6.13}$$

can also be solved by computing

$$\sum_{k=1}^{n} J_k^*. \tag{6.14}$$

Proof. The proof follows exactly along the lines of the proof for Theorem 105. ■

Example 110 *Let us again consider the timed transition graph shown in Figure 5.4. Then, using Example 99, we can easily compute*

$$\begin{aligned}
i_1^* &= f_1(\varepsilon) \\
&= \min_{s \in M_1[]} \#t(s) \\
&= 1, \\
I_1^* &= g_1(\varepsilon) \\
&= \max_{s \in M_1[]} \#t(s) \\
&= 2, \\
i_2^* &= f_2(i_1^*) \\
&= f_2(1) \\
&= \min_{s \in M_2[1]} \#t(s) \\
&= 1,
\end{aligned}$$

$$\begin{aligned}
I_2^* &= g_2(I_1^*) \\
&= g_2(2) \\
&= \max_{s \in M_2[2]} \#t\,(s) \\
&= 2, \\
i_3^* &= f_3(i_1^*, i_2^*) \\
&= f_2(1,1) \\
&= \min_{s \in M_3[1,1]} \#t(s) \\
&= 2, \\
I_3^* &= g_3(I_1^*, I_2^*) \\
&= g_3(2,2) \\
&= \max_{s \in M_3[2,2]} \#t\,(s) \\
&= 2.
\end{aligned}$$

Similarly, we can compute

$$\begin{aligned}
j_3^* &= f_3'(\varepsilon) \\
&= \min_{s \in M_3'[]} \#t\,(s) \\
&= 0, \\
J_3^* &= g_3'(\varepsilon), \\
&= \max_{s \in M_3'[]} \#t\,(s) \\
&= 4, \\
j_2^* &= f_2'(j_3^*) \\
&= f_2'(0)
\end{aligned}$$

$$
\begin{aligned}
&= \min_{s \in M_2'[0]} \#t\,(s) \\
&= 3, \\
J_2^* &= g_2'(J_3^*) \\
&= g_2'(4) \\
&= \max_{s \in M_2'[4]} \#t\,(s) \\
&= 0, \\
j_1^* &= f_1'(j_2^*, j_3^*) \\
&= f_2'(3,0) \\
&= \min_{s \in M_1'[3,0]} \#t\,(s) \\
&= 1, \\
J_1^* &= g_1'(J_2^*, J_3^*) \\
&= g_1'(0,4) \\
&= \max_{s \in M_1'[0,4]} \#t\,(s) \\
&= 2.
\end{aligned}
$$

Thus, the shortest path over S_w has

$$
\begin{aligned}
\min_{s \in S_w} \#t(s) &= i_1^* + i_2^* + i_3^* \\
&= 1 + 1 + 2 \\
&= j_3^* + j_2^* + j_1^* \\
&= 0 + 3 + 1 \\
&= 4
\end{aligned}
$$

ticks. Similarly, the longest path over S_w *has*

$$\begin{aligned}\max_{s\in S_w} \#t(s) &= I_1^* + I_2^* + I_3^* \\ &= 2 + 2 + 2 \\ &= J_3^* + J_2^* + J_1^* \\ &= 4 + 0 + 2 \\ &= 6\end{aligned}$$

ticks. So $t\alpha t\beta t^2\gamma$ *and* $t\alpha t^3\beta\gamma$ *are two shortest paths over* S_w *while* $t^2\alpha t^2\beta t^2\gamma$ *and* $t^2\alpha\beta t^4\gamma$ *are two longest paths over* S_w. □

6.4. Ordering of Strings and Timer Matrices

In this section we define two linear orders on strings and a partial order on timer matrices. This will assist us in the comparison of the prefix and the postfix solutions to the shortest and longest path problems. Let $w = \sigma_1\sigma_2\cdots\sigma_n \in Q_t(L)$ and recall that

$$S_w := \{s \in \Sigma_t^*\Sigma_{act} \cap L \mid Q_t(s) = w\}.$$

Thus S_w is the set of all strings in L of the form $t^{i_1}\sigma_1\cdots t^{i_n}\sigma_n$ such that the image of these strings under the natural projection Q_t (which erases all occurrences of the *tick* event) is equal to w. For convenience we define

$$\mathbf{S}_w := \{s \in \Sigma_t^*\Sigma_{act} \mid Q_t(s) = w\}$$

Assume $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ and recall that T_s is defined to be the timer matrix corresponding to the string $s \in L$. Define

$$\mathbf{T}_w := \{T_s \mid s \in S_w\}$$

to be the set of timer matrices corresponding to the various strings in S_w.

Firstly, we define two linear orders among the member strings of $\mathbf{S}_w$. These induce linear orders on S_w since $S_w \subseteq \mathbf{S}_w$. Let $s^1_{\min}$ and $s^1_{\max}$ be the shortest and longest strings in S_w according to Theorems 104 and 108; let $s^2_{\min}$ and $s^2_{\max}$ be the shortest and longest strings in S_w according to Theorems 105 and 109. (Here it may be possible that $s_{\max}$ corresponds to an infinite time path.) Then $s^1_{\min}$, $s^1_{\max}$ and $s^2_{\min}$, $s^2_{\max}$ will be the same as the shortest and longest strings in S_w according to these two linear orders respectively.

Then we define a partial order among the timer matrices in $\mathbf{T}_w$ and show that the top and bottom elements exist in $\mathbf{T}_w$. In fact, these are the same as the timer matrices corresponding to $s^2_{\min}$ and $s^2_{\max}$.

6.4.1. A Postfix Total Order on Strings

Let $s^1_{\min} = t^{i^*_1}\sigma_1 \cdots t^{i^*_n}\sigma_n$ and $s^1_{\max} = t^{I^*_1}\sigma_1 \cdots t^{I^*_n}\sigma_n$ be the shortest and longest time paths as computed using Theorems 104,108. We now show that it is possible to define a total order on strings of the form $t^{i_1}\sigma_1 \cdots t^{i_n}\sigma_n$ according to which $s^1_{\min}$ and $s^1_{\max}$ are the extremal strings.

Proposition 111 *Let* $s_1 = t^{i_1}\sigma_1 \cdots t^{i_n}\sigma_n$ *and* $s_2 = t^{j_1}\sigma_1 \cdots t^{j_n}\sigma_n$ *be strings in* $\mathbf{S}_w$ *and define* $s_1 \leq_1 s_2$ *iff*

$$[i_1 < j_1] \vee [(i_1 = j_1) \wedge (i_2 < j_2)] \vee \cdots \vee [(i_1 = j_1) \wedge \cdots \wedge (i_{n-1} = j_{n-1}) \wedge (i_n \leq j_n)]\,.$$

Then $\leq_1$ defines a total order on $\mathbf{S}_w$.

Proof. In order to show that $\leq_1$ defines a total order on $\mathbf{S}_w$, we have to prove that $\leq_1$ is reflexive, antisymmetric, transitive and that for all $s_1, s_2 \in \mathbf{S}_w$ either $s_1 \leq_1 s_2$ or $s_1 \leq_1 s_2$.

Reflexive: We need to show that for all $s \in \mathbf{S}_w$, we must have $s \leq_1 s$. Let $s = t^{i_1}\sigma_1 \cdots t^{i_n}\sigma_n$. Then $s \leq_1 s$ because

$$(i_1 = i_1) \wedge \cdots \wedge (i_{n-1} = i_{n-1}) \wedge (i_n \leq i_n)\,.$$

Antisymmetric: We need to show that for all $s_1, s_2 \in \mathbf{S}_w$,

$$(s_1 \leq_1 s_2) \wedge (s_2 \leq_1 s_1) \Rightarrow s_1 = s_2.$$

Let $s_1 = t^{i_1}\sigma_1 \cdots t^{i_n}\sigma_n$ and $s_2 = t^{j_1}\sigma_1 \cdots t^{j_n}\sigma_n$. Since $s_1 \leq_1 s_2$ we must have

$$[i_1 < j_1] \vee [(i_1 = j_1) \wedge (i_2 < j_2)] \vee \cdots \vee [(i_1 = j_1) \wedge \cdots \wedge (i_{n-1} = j_{n-1}) \wedge (i_n \leq j_n)]\,. \tag{6.15}$$

Similarly, since $s_2 \leq_1 s_1$, we must have

$$[j_1 < i_1] \vee [(j_1 = i_1) \wedge (j_2 < i_2)] \vee \cdots \vee [(j_1 = i_1) \wedge \cdots \wedge (j_{n-1} = i_{n-1}) \wedge (j_n \leq i_n)]\,. \tag{6.16}$$

From (6.15) and (6.16) we can conclude that

$$[(i_1 = j_1) \wedge \cdots \wedge (i_{n-1} = j_{n-1}) \wedge (i_n \leq j_n)]$$

and

$$[(j_1 = i_1) \wedge \cdots \wedge (j_{n-1} = i_{n-1}) \wedge (j_n \leq i_n)]$$

which implies that

$$[(i_1 = j_1) \wedge \cdots \wedge (i_{n-1} = j_{n-1}) \wedge (i_n = j_n)]\,.$$

Transitive: We need to show that for all $s_1, s_2, s_3 \in \mathbf{S}_w$,

$$(s_1 \leq_1 s_2) \wedge (s_2 \leq_1 s_3) \Rightarrow s_1 \leq_1 s_3.$$

Let $s_1 = t^{i_1}\sigma_1 \cdots t^{i_n}\sigma_n$, $s_2 = t^{j_1}\sigma_1 \cdots t^{j_n}\sigma_n$ and $s_3 = t^{k_1}\sigma_1 \cdots t^{k_n}\sigma_n$. Since $s_1 \leq_1 s_2$ we must have

$$[i_1 < j_1] \vee [(i_1 = j_1) \wedge (i_2 < j_2)] \vee \cdots \vee [(i_1 = j_1) \wedge \cdots \wedge (i_{n-1} = j_{n-1}) \wedge (i_n \leq j_n)]\,. \tag{6.17}$$

Similarly, since $s_2 \leq_1 s_3$, we must have

$$[j_1 < k_1] \vee [(j_1 = k_1) \wedge (j_2 < k_2)] \vee \cdots \vee [(j_1 = k_1) \wedge \cdots \wedge (j_{n-1} = k_{n-1}) \wedge (j_n \leq k_n)]\,. \tag{6.18}$$

Since $<$ and $=$ are transitive relations on $\mathbf{N}$, we can conclude from (6.17) and (6.18) that

$$[i_1 < k_1] \vee [(i_1 = k_1) \wedge (i_2 < k_2)] \vee \cdots \vee [(i_1 = k_1) \wedge \cdots \wedge (i_{n-1} = k_{n-1}) \wedge (i_n \leq k_n)]$$

which implies that $s_1 \leq_1 s_3$.

Linear: Finally, we need to show that for all $s_1, s_2 \in \mathbf{S}_w$, either $s_1 \leq_1 s_2$ or $s_2 \leq_1 s_1$. But this immediately follows from the fact that for any $i, j \in \mathbf{N}$ we must have $(i < j) \vee (i = j) \vee (i > j)$. ■

6.4.2. A Prefix Total Order on Strings

Let $s^2_{\min} = t^{j^*_1}\sigma_1 \cdots t^{j^*_n}\sigma_n$ and $s^2_{\max} = t^{J^*_1}\sigma_1 \cdots t^{J^*_n}\sigma_n$ be the shortest and longest time paths as computed using Theorems 105 and 109. We now show that it is possible to define a total order on strings of the form $t^{i_1}\sigma_1 \cdots t^{i_n}\sigma_n$ according to which $s^2_{\min}$ and $s^2_{\max}$ are the extremal strings.

Proposition 112 *Let* $s_1 = t^{i_1}\sigma_1 \cdots t^{i_n}\sigma_n$ *and* $s_2 = t^{j_1}\sigma_1 \cdots t^{j_n}\sigma_n$ *be strings in* $\mathbf{S}_w$ *and define* $s_1 \leq_2 s_2$ *iff*

$$[i_n < j_n] \vee [(i_n = j_n) \wedge (i_{n-1} < j_{n-1})] \vee \cdots \vee [(i_n = j_n) \wedge \cdots \wedge (i_2 = j_2) \wedge (i_1 \leq j_1)]\,.$$

Then $\leq_2$ *defines a total order on* $\mathbf{S}_w$.

Proof. In order to show that $\leq_2$ defines a total order on $\mathbf{S}_w$, we have to prove that $\leq_2$ is reflexive, antisymmetric, transitive and that for all $s_1, s_2 \in \mathbf{S}_w$ either $s_1 \leq_2 s_2$ or $s_1 \leq_2 s_2$. This can be done exactly along the lines of the proof of Propostion 111. ■

6.4.3. A Partial Order on Timer Matrices

Proposition 113 *Let*

$$T_{s_1} = \begin{bmatrix} \overline{x}_1 & \cdots & \overline{x}_m \\ \underline{x}_1 & \cdots & \underline{x}_m \end{bmatrix}$$

and

$$T_{s_2} = \begin{bmatrix} \overline{X}_1 & \cdots & \overline{X}_m \\ \underline{X}_1 & \cdots & \underline{X}_m \end{bmatrix}$$

belong to $\mathbf{T}_w$. *Define* $T_{s_1} \leq T_{s_2}$ *if*

$$\left(\overline{X}_i \leq \overline{x}_i\right) \wedge \left(\underline{X}_i \leq \underline{x}_i\right)$$

for all $1 \leq i \leq m$. *Then* $\leq$ *defines a partial order on* $\mathbf{T}_w$ *with* $T_{s_{\min}}$ *and* $T_{s_{\max}}$ *as the bottom and top elements respectively, where* $s_{\min}$ *and* $s_{\max}$ *are the shortest and longest strings according to the prefix order on* S_w.

Proof. For $\leq$ to be a partial order on $\mathbf{T}_w$, it must be reflexive, transitive and antisymmetric.

Reflexive: Let $s \in L$ and

$$T_s = \begin{bmatrix} \overline{x}_1 & \cdots & \overline{x}_m \\ \underline{x}_1 & \cdots & \underline{x}_m \end{bmatrix}.$$

Then we must have $T_s \leq T_s$ since $(\overline{x}_i \leq \overline{x}_i) \wedge (\underline{x}_i \leq \underline{x}_i)$ for all $1 \leq i \leq m$.

Transitive: Let $T_{s_1}, T_{s_2}, T_{s_3} \in \mathbf{T}_w$ such that $(T_{s_1} \leq T_{s_2}) \wedge (T_{s_2} \leq T_{s_3})$. We need to show that $T_{s_1} \leq T_{s_3}$, i.e. $(\overline{z}_i \leq \overline{x}_i) \wedge (\underline{z}_i \leq \underline{x}_i)$ for all $1 \leq i \leq m$, where

$$T_{s_1} = \begin{bmatrix} \overline{x}_1 & \cdots & \overline{x}_m \\ \underline{x}_1 & \cdots & \underline{x}_m \end{bmatrix},$$

$$T_{s_2} = \begin{bmatrix} \overline{y}_1 & \cdots & \overline{y}_m \\ \underline{y}_1 & \cdots & \underline{y}_m \end{bmatrix}, \text{ and}$$

$$T_{s_3} = \begin{bmatrix} \overline{z}_1 & \cdots & \overline{z}_m \\ \underline{z}_1 & \cdots & \underline{z}_m \end{bmatrix}.$$

Since $(\overline{y}_i \leq \overline{x}_i) \wedge \left(\underline{y}_i \leq \underline{x}_i\right) \wedge (\overline{z}_i \leq \overline{y}_i) \wedge \left(\underline{z}_i \leq \underline{y}_i\right)$, the transitivity of natural numbers implies that $(\overline{z}_i \leq \overline{x}_i) \wedge (\underline{z}_i \leq \underline{x}_i)$ for all $1 \leq i \leq m$.

Antisymmetric: Let $T_{s_1}, T_{s_2} \in \mathbf{T}_w$ such that $(T_{s_1} \leq T_{s_2}) \wedge (T_{s_2} \leq T_{s_1})$. We need to show that $T_{s_1} = T_{s_3}$, i.e. $(\overline{x}_i = \overline{y}_i) \wedge \left(\underline{x}_i = \underline{y}_i\right)$ for all $1 \leq i \leq m$, where

$$T_{s_1} = \begin{bmatrix} \overline{x}_1 & \cdots & \overline{x}_m \\ \underline{x}_1 & \cdots & \underline{x}_m \end{bmatrix}, \text{ and}$$

$$T_{s_2} = \begin{bmatrix} \overline{y}_1 & \cdots & \overline{y}_m \\ \underline{y}_1 & \cdots & \underline{y}_m \end{bmatrix}.$$

Again, the desired result follows from the antisymmetry of natural numbers.

Now we need to show that $T_{s_{\min}}$ and $T_{s_{\max}}$ are the bottom and top elements of $\mathbf{T}_w$. Assume $s_{\min} = t^{j_1^*}\alpha_1 \cdots t^{j_n^*}\alpha_n$ and let $s = t^{j_1}\alpha_1 \cdots t^{j_n}\alpha_n$ be an arbitrary element of L. Further assume that

$$T_{s_{\min}} = \begin{bmatrix} \overline{x}_{\sigma_1} & \cdots & \overline{x}_{\sigma_m} \\ \underline{x}_{\sigma_1} & \cdots & \underline{x}_{\sigma_m} \end{bmatrix}, \text{ and}$$

$$T_s = \begin{bmatrix} \overline{y}_{\sigma_1} & \cdots & \overline{y}_{\sigma_m} \\ \underline{y}_{\sigma_1} & \cdots & \underline{y}_{\sigma_m} \end{bmatrix}.$$

Let u_{σ_i} and l_{σ_i} be the upper and lower time bounds of an activity σ_i. If $\sigma_i = \alpha_n$ or $\alpha_1 \cdots \alpha_{n-1}\sigma_i \notin L_{act}$ then we must have $\overline{x}_{\sigma_i} = \overline{y}_{\sigma_i} = u_{\sigma_i}$ and $\underline{x}_{\sigma_i} = \underline{y}_{\sigma_i} = l_{\sigma_i}$. Otherwise, we

must have

$$\begin{aligned}
\overline{x}_{\sigma_i} &= u_{\sigma_i} - \sum_{v=k}^{n} j_v^*, \\
\underline{x}_{\sigma_i} &= l_{\sigma_i} \dot{-} \sum_{v=k}^{n} j_v^*, \\
\overline{y}_{\sigma_i} &= u_{\sigma_i} - \sum_{v=k}^{n} j_v, \text{ and} \\
\underline{y}_{\sigma_i} &= l_{\sigma_i} \dot{-} \sum_{v=k}^{n} j_v,
\end{aligned}$$

where σ_i was last enabled after the occurrence of α_{k-1}, for some $1 \leq k \leq n$. Here we assume that α_0 corresponds to the empty string. From Theorem 105 we know that $\sum_{v=k}^{n} j_v^* \leq \sum_{v=k}^{n} j_v$ for all $1 \leq k \leq n$. Therefore, it follows that $\left(\overline{y}_{\sigma_i} \leq \overline{x}_{\sigma_i}\right) \wedge \left(\underline{y}_{\sigma_i} \leq \underline{x}_{\sigma_i}\right)$ for all $1 \leq i \leq m$, which implies that $T_{s_{\min}} \leq T_s$. Since s is arbitrary, $T_{s_{\min}}$ must be the bottom element. A similar argument shows that $T_s \leq T_{s_{\max}}$ which implies that $T_{s_{\max}}$ must be the top element. ■

Meet and Join

Definition 114 *Let*

$$\begin{aligned}
T_{s_1} &= \begin{bmatrix} \overline{x}_1 & \cdots & \overline{x}_m \\ \underline{x}_1 & \cdots & \underline{x}_m \end{bmatrix}, \\
T_{s_2} &= \begin{bmatrix} \overline{y}_1 & \cdots & \overline{y}_m \\ \underline{y}_1 & \cdots & \underline{y}_m \end{bmatrix},
\end{aligned}$$

and let

$$T_{meet} = \begin{bmatrix} \overline{z}_1 & \cdots & \overline{z}_m \\ \underline{z}_1 & \cdots & \underline{z}_m \end{bmatrix}, \tag{6.19}$$

$$T_{join} = \begin{bmatrix} \overline{Z}_1 & \cdots & \overline{Z}_m \\ \underline{Z}_1 & \cdots & \underline{Z}_m \end{bmatrix}$$

be the meet *and* join *respectively. Then*

$$\begin{aligned} \overline{z}_k &= \max\left(\overline{x}_k, \overline{y}_k\right), \\ \underline{z}_k &= \max\left(\underline{x}_k, \underline{y}_k\right), \\ \overline{Z}_1 &= \min\left(\overline{x}_k, \overline{y}_k\right), \\ \underline{Z}_1 &= \min\left(\underline{x}_k, \underline{y}_k\right). \end{aligned}$$

for $1 \leq k \leq m$.

Since $(\overline{x}_k, \underline{x}_k)$ and $\left(\overline{y}_k, \underline{y}_k\right)$ both represent the upper and lower timer values of the same event, it follows that either

$$\begin{bmatrix} \overline{z}_k \\ \underline{z}_k \end{bmatrix} = \begin{bmatrix} \overline{x}_k \\ \underline{x}_k \end{bmatrix}$$

or

$$\begin{bmatrix} \overline{z}_k \\ \underline{z}_k \end{bmatrix} = \begin{bmatrix} \overline{y}_k \\ \underline{y}_k \end{bmatrix}.$$

Let us represent the *meet* and *join* by $\wedge$ and $\vee$ respectively. Clearly, if $T_{s_1} \leq T_{s_2}$ then $T_{s_1} \wedge T_{s_2} = T_{s_1}$ and $T_{s_1} \vee T_{s_2} = T_{s_2}$. We now show that *meet* (and *join)* of two elements in $\mathbf{T}_w$ always exists in $\mathbf{T}_w$.

Proposition 115 *Let $T_1, T_2 \in \mathbf{T}_w$; then the* meet *and* join *of T_1 and T_2 exist in $\mathbf{T}_w$.*

Proof. Let $s_1 = t^{i_1}\alpha_1 \cdots t^{i_n}\alpha_n$ and $s_2 = t^{j_1}\alpha_1 \cdots t^{j_n}\alpha_n$ be two strings in S_w such that T_1 and T_2 are the corresponding timer matrices. Let T_{meet} be defined as in (6.19). If $T_{meet} \in \mathbf{T}_w$ then there must exist at least one string in S_w such that T_{meet} is the corresponding timer matrix. We now define such a string and show that T_{meet} is the corresponding timer matrix. Let

$$s = t^{k_1}\alpha_1 \cdots t^{k_n}\alpha_n$$

be such that

$$\begin{aligned} k_n &= \min(i_n, j_n) \\ k_n + k_{n-1} &= \min(i_n + i_{n-1}, j_n + j_{n-1}) \\ &\vdots \\ k_n + \cdots + k_1 &= \min(i_n + \cdots + i_1, j_n + \cdots + j_1). \end{aligned}$$

Thus, given s_1 and s_2, we can compute s by successive subtraction. For example, $k_{n-1} = \min(i_n + i_{n-1}, j_n + j_{n-1}) - k_n$ and so on. By definition $s \in S_w$ so all we need to do now is show that T_{meet} corresponds to s. Let T be the timer matrix corresponding to s and let $\sigma \in \Sigma$ be an event that was last enabled after the occurrence of α_l for some $1 \leq l \leq n$. Then σ has been enabled for $k_{l+1} + \cdots + k_n$ ticks. However,

$$k_{l+1} + \cdots + k_n = \min(i_{l+1} + \cdots + i_n, j_{l+1} + \cdots + j_n).$$

Without loss of generality, let us assume that

$$k_{l+1} + \cdots + k_n = i_{l+1} + \cdots + i_n.$$

Then the timer values of σ in T are the same as the timer values of σ in T_1. However, by the definition of T_{meet}, the timer of values of σ in T_{meet} are the same as in T_1. Since σ is arbitrary, it follows that $T = T_{meet}$ which implies that the *meet* of T_1 and T_2 exists in $\mathbf{T}_w$. Similar arguments also hold for the *join* of T_1 and T_2. ■

6.5. Prefix versus Postfix Solution

In this section we compare the prefix and postfix solutions to the shortest and longest path problems. We will show that, given a TTG, the postfix solution is somewhat easier to compute than a prefix solution. However the prefix solution is more useful for supervisory control.

6.5.1. Ease of Computation

Let $w = \sigma_1\sigma_2\cdots\sigma_n \in Q_t(L)$ and let $s^1_{\min}$, $s^1_{\max}$ and $s^2_{\min}$, $s^2_{\max}$ be the shortest and longest strings in S_w according to the postfix and prefix order respectively. Now suppose that we are interested in finding shortest and longest paths in S_v where $v = w\sigma_{n+1}$. It would be nice if our knowledge about a shortest (longest) path in S_w could guide our search for a shortest (longest) path in S_v. Let us first consider the postfix solution. Clearly the starting point for a postfix solution in S_v is the same as the starting point for a postfix solution in S_w. This means that the shortest string in S_v according to the postfix order will be of the form $s^1_{\min}t^{i^*_{n+1}}\sigma_{n+1}$ where i^*_{n+1} is the least number of *ticks* that must elapse before σ_{n+1} can

occur after $s_{\min}$. Similarly, the longest string in S_v according to the postfix order will be of the form $s^1_{\max} t^{I^*_{n+1}} \sigma_{n+1}$ where I^*_{n+1} is the maximum number of *ticks* that may elapse before σ_{n+1} is forced to occur after $s_{\max}$. This is a very desirable feature as it allows the postfix solution over n events to be easily extended over $n+1$ events.

The scenario is quite different for prefix solutions. The starting point for a prefix solution in S_v is different than the starting point for a prefix solution in S_w. For instance, if we want to find the shortest string in S_w according to the prefix order then we begin with all the strings of the form $t^{i_1}\sigma_1 \cdots t^{i_{n-1}}\sigma_{n-1} t^{j^*_n}\sigma_n$ where j^*_n is the minimum number of *ticks* that may elapse between any occurrence of σ_{n-1} and σ_n. Then we work backwards until we have found the shortest string $s^2_{\min} = t^{j^*_1}\sigma_1 \cdots t^{j^*_n}\sigma_n$. Here j^*_k is the minimum number of *ticks* that may elapse between any occurrence of σ_{k-1} and σ_k. Now if we want to find the shortest string in S_v then we need to start with strings of the form $t^{i_1}\sigma_1 \cdots t^{i_n}\sigma_n t^{j^*_{n+1}}\sigma_{n+1}$ where j^*_{n+1} is the minimum number of *ticks* that may elapse between any occurrence of σ_n and σ_{n+1}. Then we need to work backwards until we find the shortest string (say $s_{\min}$). The knowledge of $s^2_{\min}$ does not guide the process of finding $s_{\min}$ as it does in the case of postfix solutions. In other words, a prefix solution over S_w does not extend into a prefix solution over S_v. Thus, in a sense, postfix solutions are easier to compute than prefix solutions. This is quite unfortunate because, as we will show later, prefix solutions are preferable to postfix solutions for the purpose of supervisory control.

6.5.2. Usefulness for Supervisory Control

Let $w = \sigma_1\sigma_2 \cdots \sigma_n \in Q_t(L)$. Let us assume that a string of the form $t^{i_1}\sigma_1 \cdots t^{i_n}\sigma_n$ has occurred in $\mathbf{G}$ and we do not know the values of i_k for $1 \leq k \leq n$. Such a scenario can arise if we are not keeping track of the total number of *tick* occurrences. The following two questions are now important from the point of view of supervisory control.

Q1. What is the earliest possible time at which an event σ_i can occur?

Q2. What is the latest possible time at which an event σ_i can occur?

We now show that the prefix solution is useful in answering these questions. Let $s_{\min}$ and $s_{\max}$ be the shortest and longest strings in S_w according to the prefix order. Let

$$T_{s_{\min}} = \begin{bmatrix} \overline{x}_1 & \cdots & \overline{x}_m \\ \underline{x}_1 & \cdots & \underline{x}_m \end{bmatrix}$$

and

$$T_{s_{\max}} = \begin{bmatrix} \overline{X}_1 & \cdots & \overline{X}_m \\ \underline{X}_1 & \cdots & \underline{X}_m \end{bmatrix}$$

be the corresponding timer matrices. Then $\begin{bmatrix} \overline{x}_i \\ \underline{x}_i \end{bmatrix}$ and $\begin{bmatrix} \overline{X}_i \\ \underline{X}_i \end{bmatrix}$ represent the status of the timer values of the event σ_i after the occurrence of $s_{\min}$ and $s_{\max}$ respectively. From Proposition 113 we know that $T_{s_{\min}}$ and $T_{s_{\max}}$ are the bottom and top elements of $\mathbf{T}_w$. Thus the event σ_i can occur no earlier than $\underline{X}_i$ *ticks* and no later than $\overline{x}_i$ *ticks*. So if we know the timer values corresponding to $s_{\min}$ and $s_{\max}$ then we can exercise supervisory control without knowing the total number of *tick* occurrences. In a sense, after the occurrence of any string s such that $Q_t(s) = w$, we can consider $\begin{bmatrix} \overline{x}_i \\ \underline{X}_i \end{bmatrix}$ to be the status of the timer associated with σ_i. Clearly any conclusions drawn from the timer matrices corresponding to the postfix solution have to be weaker and may lead to more conservative supervision. Thus the prefix solution is more desirable from the point of view of supervisory control. In the next chapter we present a model of timed discrete event systems that is based on the prefix solution.

6.6. A Counter-example

Let $w = \sigma_1\sigma_2\cdots\sigma_n \in Q_t(L)$ and let $s \in L$ be such that $Q_t(s) = w$. Let $\sigma \in \Sigma$ be an event that is enabled at s. As mentioned earlier, two important questions that need to be answered for effective supervisory control are: how soon can σ occur? and how long before σ is forced to occur? Brandin tries to answer these questions in [Bra97],[Bra98]. Here we present a counter-example to show that his solution can be incorrect.

Consider the ATG given in Figure 6.1 below and assume that the following triples define

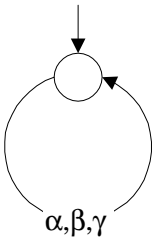


Figure 6.1: Counter-example: ATG

the timer values for the various events: $(\alpha, 2, 7)$, $(\beta, 1, 5)$, and $(\gamma, 5, 6)$. The corresponding TTG has 336 states and 838 transitions. A part of this TTG is shown in Figure 6.2. Assume that some string s (such that $Q_t(s) = \beta\alpha\gamma$) has occurred in the TTG. We are interested in finding how soon can α, β or γ occur now. Similarly, we are also interested in finding how long before α, β or γ is forced to occur. From 6.2 we may easily compute the prefix shortest path to be $t^5\beta\alpha\gamma$. Similarly the prefix longest path is $t\beta t\alpha t^4\gamma$. These paths are shown using *dashed* arrows in 6.2. The timer matrices corresponding to these paths are

$$\begin{bmatrix} 7 & 5 & 6 \\ 2 & 1 & 5 \end{bmatrix}$$

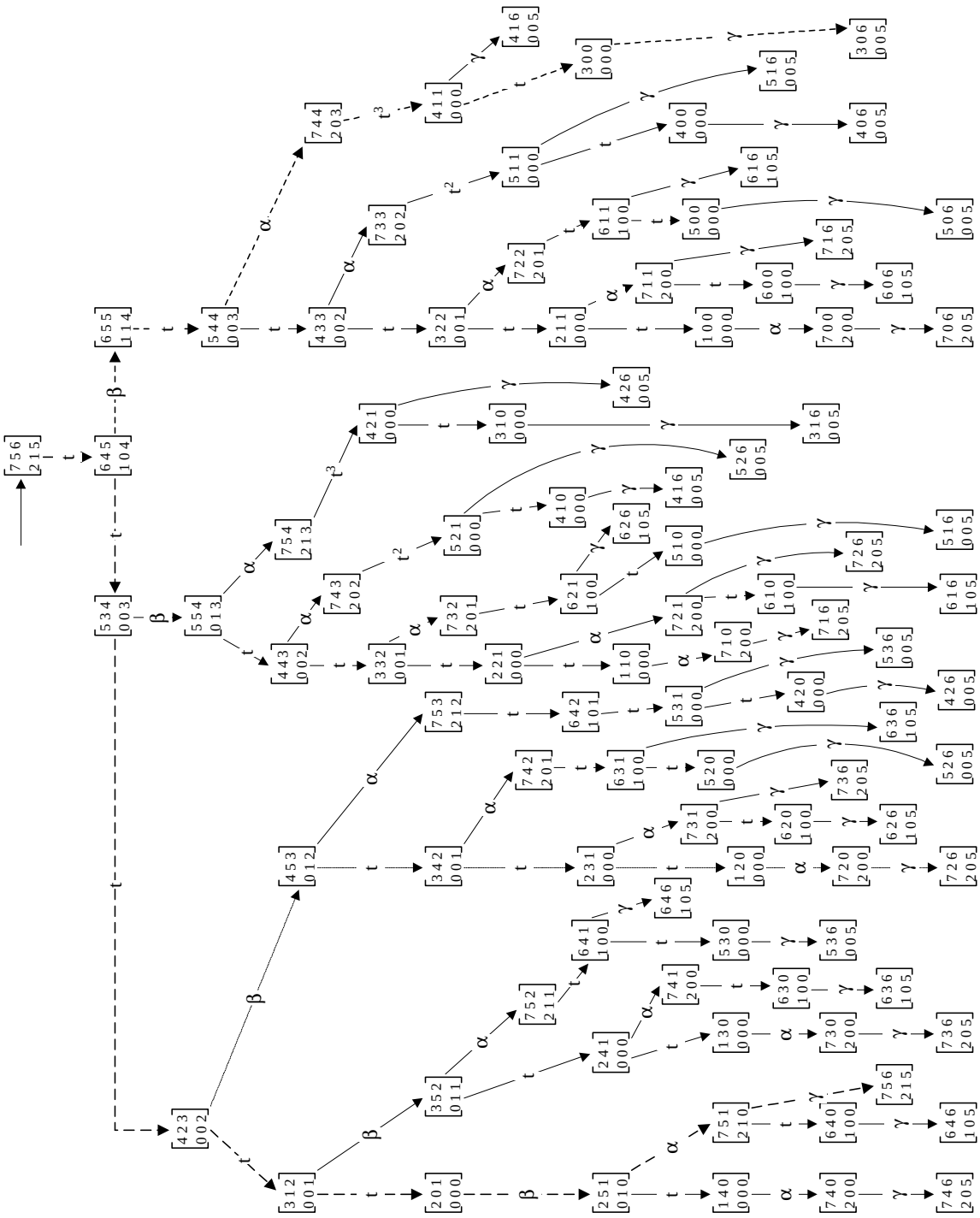


Figure 6.2: Counter-example: Partial TTG

and

$$\begin{bmatrix} 3 & 0 & 6 \\ 0 & 0 & 5 \end{bmatrix}$$

where the first column corresponds to α, the second column to β, and the third column to γ. From these timer matrices we can conclude that the effective timers after the occurrence of s are

$$\begin{bmatrix} 7 & 5 & 6 \\ 0 & 0 & 5 \end{bmatrix}.$$

In other words, α may occur right away but may take as long as 7 *ticks*. Similarly, β may occur right away but may take as long as 5 *ticks*.

We now use the algorithm proposed in [Bra97],[Bra98] to do the same. For the sake of fidelity, we use the notation used in [Bra97],[Bra98]. We will translate the result back into our notation for the purpose of comparision.

Brandin's Method: Let $f_{0,}f_1$, f_2, and f_3 represent the states corresponding to the strings ϵ, β, $\beta\alpha$, and $\beta\alpha\gamma$ respectively. The initial timed activity is

$$f_0 = (a_0, \{(2,2,7,7), (1,1,5,5), (5,5,6,6)\})$$

and initially

$$\begin{aligned} \underline{t} &= \min\{7,5,6\} \\ &= 5 \\ \overline{t} &= \min\{7,5,6\} \\ &= 5. \end{aligned}$$

Then the timers of α at f_1 are given by

$$\begin{aligned} & \left(\min\left(2 \dot{-} 5, 2 \dot{-} 5\right), \max\left(2 \dot{-} 1, 2 \dot{-} 1\right), \min\left(7 \dot{-} 5, 7 \dot{-} 5\right), \max\left(7 \dot{-} 2, 7 \dot{-} 2\right)\right) \\ = \; & (0, 1, 2, 5)\,; \end{aligned}$$

the timers of γ at f_1 are given by

$$\begin{aligned} & \left(\min\left(5 \dot{-} 5, 5 \dot{-} 5\right), \max\left(5 \dot{-} 1, 5 \dot{-} 1\right), \min\left(6 \dot{-} 5, 6 \dot{-} 5\right), \max\left(6 \dot{-} 1, 6 \dot{-} 1\right)\right) \\ = \; & (0, 4, 1, 5)\,; \end{aligned}$$

the timers of β are reset to their default values, i.e.

$$(1, 1, 5, 5)\,.$$

So we have

$$f_1 = (a_1, \{(0, 1, 2, 5)\,, (1, 1, 5, 5)\,, (0, 4, 1, 5)\})$$

which implies that now

$$\begin{aligned} \underline{t} &= \min\{2, 5, 1\} \\ &= 1 \\ \overline{t} &= \min\{5, 5, 5\} \\ &= 5. \end{aligned}$$

The timers of β at f_2 are given by

$$\begin{aligned} & \left(\min\left(1 \dot{-} 1, 1 \dot{-} 5\right), \max\left(1 \dot{-} 0, 1 \dot{-} 1\right), \min\left(5 \dot{-} 1, 5 \dot{-} 5\right), \max\left(5 \dot{-} 0, 5 \dot{-} 1\right)\right) \\ = \; & (0, 1, 0, 5)\,; \end{aligned}$$

the timers of γ at f_2 are given by

$$\begin{aligned} & \left(\min\left(0 \dot{-} 1, 4 \dot{-} 5\right), \max\left(0 \dot{-} 0, 4 \dot{-} 1\right), \min\left(1 \dot{-} 1, 5 \dot{-} 5\right), \max\left(1 \dot{-} 0, 5 \dot{-} 1\right)\right) \\ = \; & (0, 3, 0, 4)\,; \end{aligned}$$

the timers of α are reset to their default values, i.e.

$$(2, 2, 7, 7)\,.$$

This gives

$$f_2 = (a_2, \{(2, 2, 7, 7), (0, 1, 0, 5), (0, 3, 0, 4)\})$$

which now implies

$$\begin{aligned} \underline{t} &= \min\{7, 0, 0\} \\ &= 0 \\ \overline{t} &= \min\{7, 5, 4\} \\ &= 4. \end{aligned}$$

The timers of α at f_3 are given by

$$\begin{aligned} & \left(\min\left(2 \dot{-} 0, 2 \dot{-} 4\right), \max\left(2 \dot{-} 0, 2 \dot{-} 3\right), \min\left(7 \dot{-} 0, 7 \dot{-} 4\right), \max\left(7 \dot{-} 0, 7 \dot{-} 3\right)\right) \\ = \; & (0, 2, 3, 7)\,; \end{aligned}$$

the timers of β at f_3 are given by

$$\begin{aligned} & \left(\min\left(0 \dot{-} 0, 1 \dot{-} 4\right), \max\left(0 \dot{-} 0, 1 \dot{-} 3\right), \min\left(0 \dot{-} 0, 5 \dot{-} 4\right), \max\left(0 \dot{-} 0, 5 \dot{-} 3\right)\right) \\ = \; & (0, 0, 0, 2)\,; \end{aligned}$$

the timers of γ are reset to their default values, i.e.

$$(5, 5, 6, 6)\,.$$

Thus

$$f_3 = (a_3, \{(0, 2, 3, 7), (0, 0, 0, 2), (5, 5, 6, 6)\})\,.$$

This may be interpreted as follows. According to [Bra97],[Bra98] the effective timers after the occurrence of s are

$$\begin{bmatrix} 7 & 2 & 6 \\ 0 & 0 & 5 \end{bmatrix}.$$

This would mean that β is forced to occur after 2 *ticks*! This is clearly incorrect as can be verified from Figure 6.2. There may be a gap of as many as 5 *ticks* before β is forced to occur (which is the conclusion we arrive at using the prefix solution).

6.7. Summary

In this chapter we posed the problem of finding shortest and longest paths in a timed transition graph. The motivation for solving this problem is the hope that a model of TDES based on shortest and longest paths may be more compact than the corresponding timed transition graph. The problem is posed in a dynamic programming framework which guarantees that we can find a solution. However dynamic programming solutions are recursive in nature and therefore not suitable for our purpose. We proposed two greedy solutions to the problem and showed that the solution is not unique. We further showed that only one of the solutions is suitable for developing a model of timed discrete event systems.

7. A Model of Timed Discrete Event Systems

7.1. Motivation

In this chapter we present a new model of timed discrete event systems. This immediately raises the question: *Why is a new model needed?* So before going any further we would like to explain some of the reasons that make it necessary. The BW model of TDES presented by Brandin and Wonham [Bra93],[BW94] explicitly models the passage of time using a special event called *tick*. The BW model extends the untimed RW model and augments the class of controllable systems. However it suffers from a couple of major drawbacks.

Too Big in Size: It can be much bigger in size compared to the corresponding untimed model. We present a very simple example to illustrate this. Assume that a public parking spot is modelled as shown in Figure 7.1. It may be interpreted as follows. If the parking spot is *idle* then a car may be *park*ed in it. If the parking spot is *occupied* then the car may be *unpark*ed. However if the car remains parked for longer than a certain duration of time then a traffic cop may give it a *ticket* for traffic violation. Of course, the car may be *unpark*ed 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 7.1. It simply says "Avoid getting a ticket." Parking

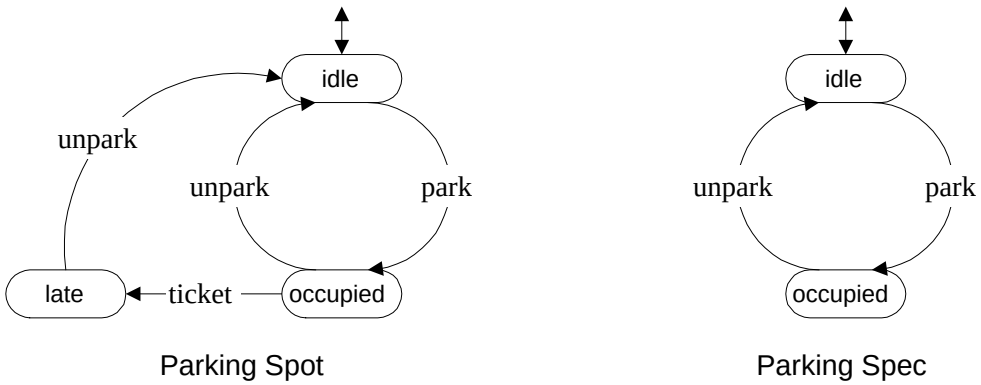


Figure 7.1: A Car Parking Setup and Specification

and unparking a car are events that can be controlled by a driver; however, getting a ticket is uncontrollable. This represents a scenario where a timed model is quite essential because the parking specification is uncontrollable if we use an untimed model. This is understandable because the parking setup inherently involves a notion of time: a parking ticket is issued only if the car has been parked for longer than a permissible duration. So let the following triples represent the time bounds associated with various events: $(park, 30, \infty)$, $(unpark, 7, \infty)$, $(ticket, 950, 999)$. This may be interpreted as follows. It takes at least 30 seconds to park a car; it takes at least 7 seconds to unpark; it is safe to park for 950 seconds but it is possible to not get ticketed for a further 49 seconds because the traffic cop may get delayed. We may use a timed transition graph to model the parking setup. We assume that *unpark* is a forcible event. The parking specification is now controllable; the resultant supervisor simply forces the car owner to *unpark* the car before 950 seconds. However the timed transition graph has 1052 states and 2115 transitions!! Clearly this is much more complicated than it needs to be. We will show later how the proposed model handles scenarios such as this one.

Not Closed Under Control: It is shown by Wong and Wonham in [WW96b] that the BW framework [BW94] is not closed under control. We borrow the counterexamples given in [WW96b] to illustrate this point.

Example 116 *Consider a language L whose ATG and TTG are given in Figure 7.2. Here α is a forcible event and has time bounds $(\alpha, 0, 1)$. Let H be the sublanguage*

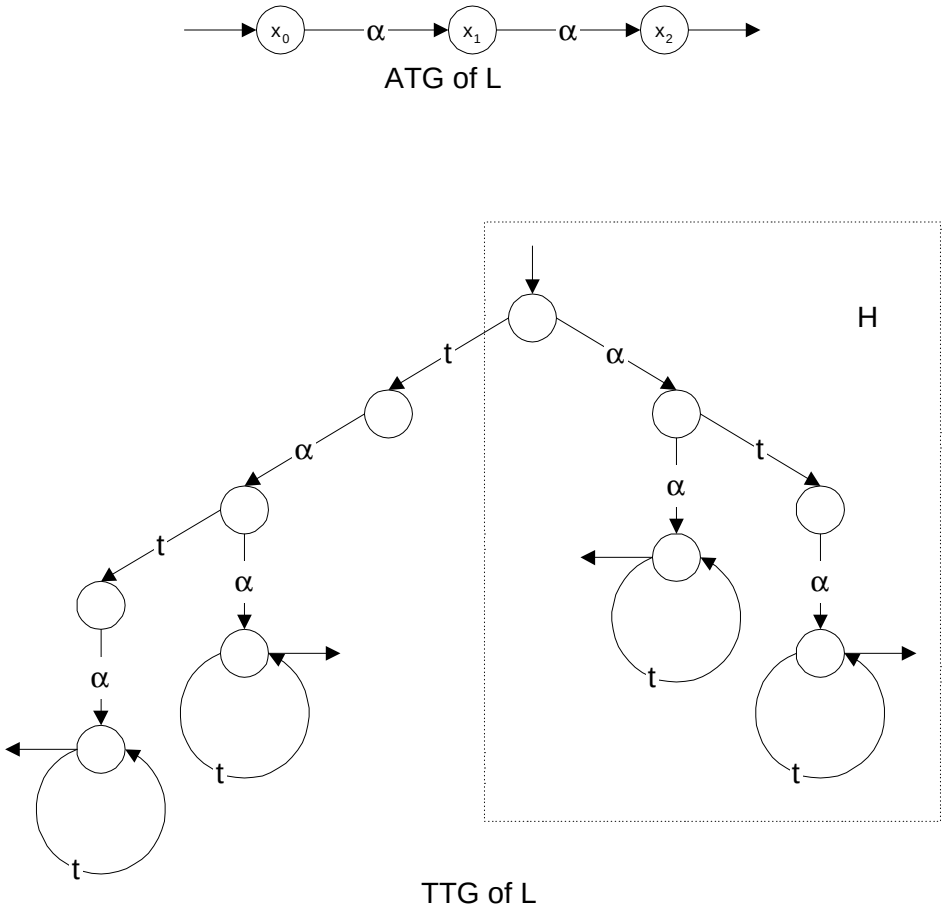


Figure 7.2: BW Not Closed Under Control: Scenario 1

of L represented by the subgraph in the dashed rectangle in Figure 7.2. Then H is controllable with respect to L since α can be forced to occur before the first tick.

However, there cannot exist an ATG such that the corresponding TTG generates H because the time bounds of the first α in H are $(\alpha, 0, 0)$ while the time bounds of the second α in H are $(\alpha, 0, 1)$. □

Example 117 *Consider a language L whose ATG and TTG are given in Figure 7.3. Here α is prohibitible and has time bounds $(\alpha, 0, \infty)$. Let H be the sublanguage of L generated by the subgraph inside the dashed polygon in Figure 7.3. Then H is controllable with respect to L since if α does not occur before the first tick then it can be disabled. Again, there can exist no ATG such that the corresponding TTG generates H because here α is an event that may occur before the first tick but never after that.* □

Both the above examples show that the BW framework is not closed under control because there may exist no ATG such that the corresponding TTG generates the controlled behavior. It is shown by Wong and Wonham in [WW96b] that even if an ATG does exist such that the corresponding TTG generates the controlled behavior, there may be other problems as the following examples show.

Example 118 *Consider a language L whose ATG and TTG are given in Figure 7.4. Here, α is both prohibitible and forcible while the time bounds are $(\alpha, 0, \infty)$ and $(\beta, 0, \infty)$. Let H be the sublanguage of L generated by the subgraph inside the dashed rectangle in Figure 7.4. Then H is controllable with respect to L since α can be forced to occur before the first tick. Let K be the sublanguage of H and L as shown in Figure 7.4. Clearly, K is controllable with respect to H since α can be prohibited from occurring. However K is uncontrollable with respect to L since β cannot be guaranteed to occur before the first tick. This contradictory behavior results because*

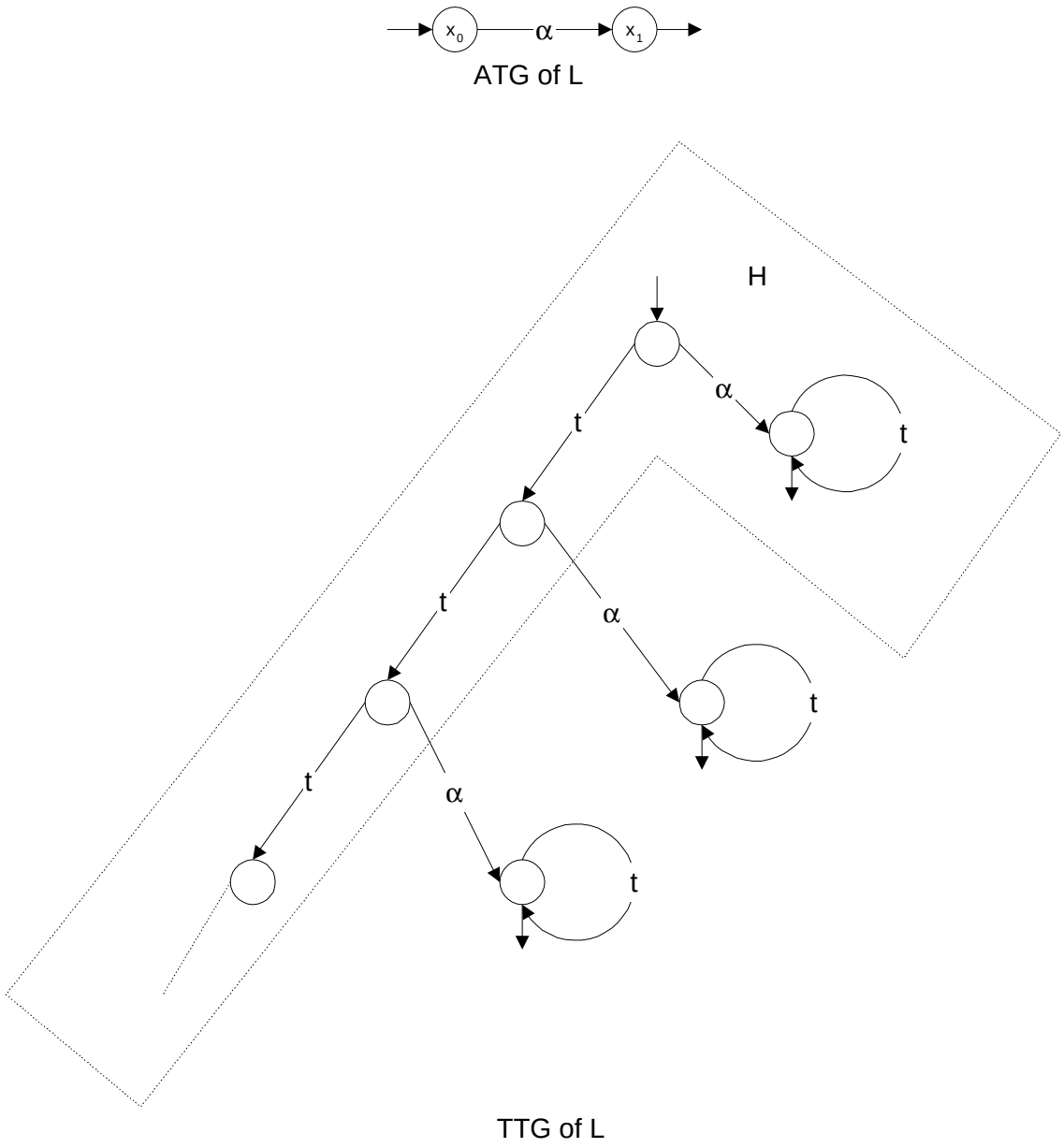


Figure 7.3: BW Not Closed Under Control: Scenario 2

we are trying to prohibit and force α simultaneously. □

In Examples 116 and 117, the problem arises because the basic building blocks in

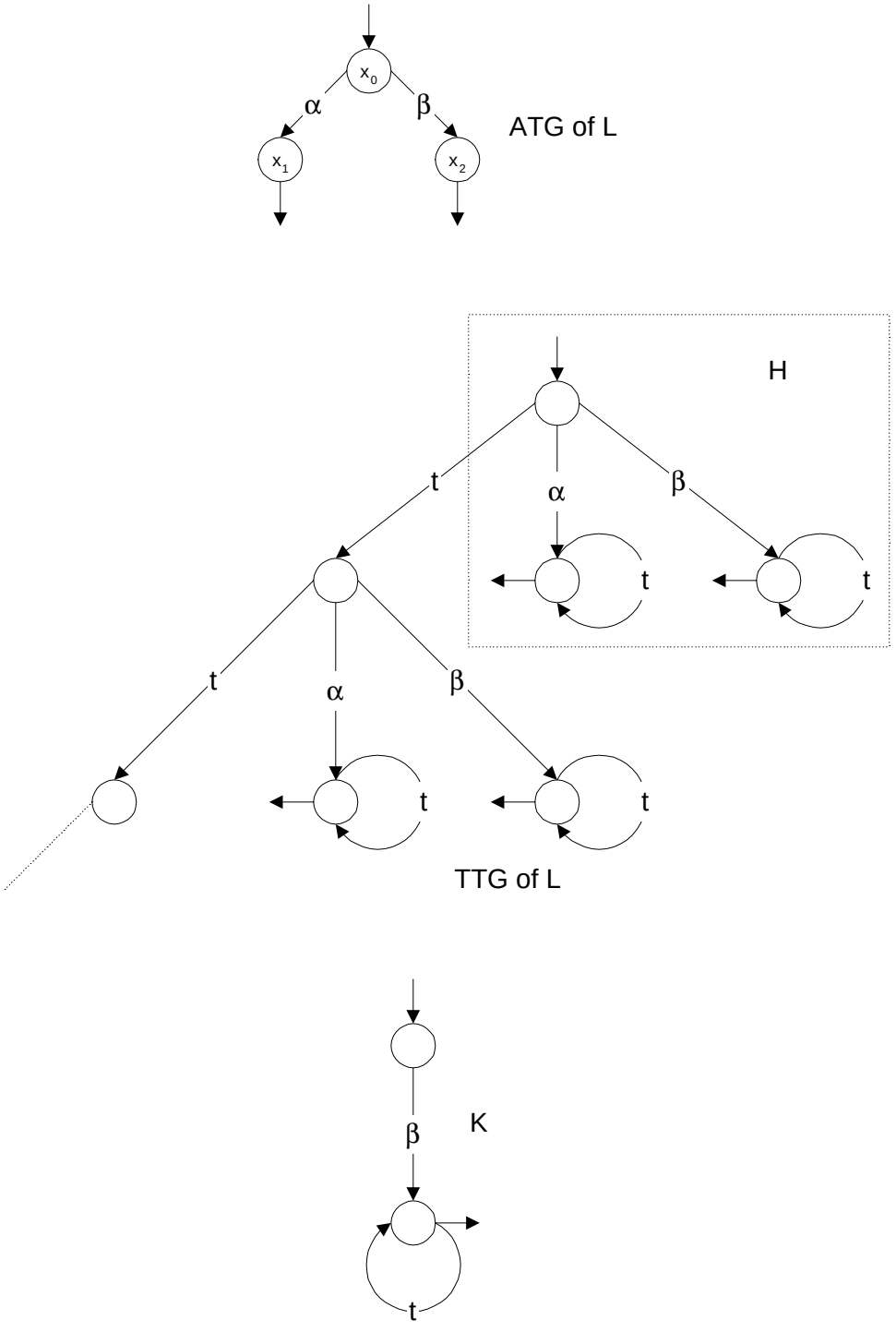


Figure 7.4: BW Not Closed Under Control: Scenario 3

BW are activity transition graphs while the control design is carried out on the corresponding timed transition graphs. Two timed transition graphs cannot be synchronized because that would require synchronizing the *tick* event (which can lead to a scenario where time *stops* [Won01, page 288]). So the corresponding activity transition graphs are composed and the resultant is converted to a timed transition graph. However, not all subautomata of a timed transition graph have corresponding activity transition graphs so there is no obvious way to compose a TTG under supervision, with another system. We will show that this problem does not arise in the proposed model because the control design is carried out on the building blocks directly.

The problem exposed by the Example 118 is a bit more subtle. In the standard RW framework, an event is either controllable or uncontrollable. However the notion of forcing arises naturally when dealing with timed systems. It may be possible to force an event to occur before a certain point in time. So now an event may be controllable as well as forcible. This is what causes the problem. In Example 118, the control design is carried out in two stages. The event α is forced to occur in the first stage; it is prohibited to occur in the second stage. So, in a sense, the memory of the control action taken in the first stage is lost during the second stage. Wong and Wonham [WW96b] get around this problem by postulating that an event cannot be both prohibitible and forcible. In the proposed model, we get around this problem by *remembering* the control actions from one stage to the next.

7.2. An Overview

The proposed model is inspired by the prefix solution to the shortest and longest path problem that was presented in Chapter 6. The main idea is to assign windows of opportunity for the various events at each state. In that respect, it is similar in spirit to the model presented in [Bra97],[Bra98]. However, as shown in Section 6.6, the analysis presented in [Bra97],[Bra98] is incorrect. The proposed model rectifies that mistake. Additionally, it improves upon it (as well as upon BW) in another important respect: it is closed under control.

We give the formal definition of the proposed model (which we call a *Timed Generator*) and define the notion of controllability. This notion of controllability is simply an extension of the RW notion of controllability. Whether or not a language is controllable depends on the controllability attributes of the events. However, in a timed generator, the controllability attribute of an event need not always be constant. The same event may be controllable at one state and uncontrollable at another. This allows a timed generator to *remember* control actions. We show that timed generators are closed under control: a timed generator subject to supervisory control action is still a timed generator.

7.3. Timed Generators

Before giving a formal definition of a timed generator we give its informal description here. A timed generator is a model of a timed discrete event system. As in the standard RW framework, events are thought of as instantaneous and occurring at quasi-random moments of real-time $\mathbf{R}^+ = \{t : 0 \leq t < \infty\}$. However, time is measured using a digital clock with

output $timercount : \mathbf{R}^+ \times \mathbf{R}^+ \rightarrow \mathbf{N}$ where

$$timercount\,(t_0, t) := n \text{ for } n \leq t - t_0 < n + 1.$$

Here $t_0 \leq t$ represents an initial time with respect to which the output of the digital clock is specified. This is akin to turning on a stopwatch at time t_0; the output of the stopwatch reflects the passage of integral units of time since t_0. Temporal conditions in a timed generator are specified in terms of this digital clock although *timercount* does not play a formal role in the development. It is assumed that upon entering a state, t_0 is set equal to the current real-time so that $timercount\,(t_0, t)$ defines the number of integral time units that have elapsed since entering that state. We associate lower and upper time bounds with the occurrence of each event at all the states. The lower bound indicates the number of time units (as measured by *timercount*) that must elapse before the event is eligible to occur. The upper bound indicates the number of time units that may elapse (as measured by *timercount*) before the event is guaranteed to occur. In graphical representations, it will be convenient to represent these time bounds using a matrix. Thus each state will have an associated timer matrix that represents the time bounds of all the events enabled at that state. The time bounds of a given event may differ from state to state. The eligibility of an event to occur at a state is defined in terms of its time bounds. This allows us to define the language generated by a timed generator. This language does not model time using any special events but it incorporates the temporal constraints imposed by the time bounds on events. The language generated by a timed generator is (defined to be) the open-loop behaviour of a timed system. As in the RW framework, we consider the role of a supervisor to restrict this behaviour to meet a given design specification. This restriction of the system behaviour may be carried out in two different ways. The first way is to

directly prevent an event (called a *prohibitible* event) from occurring at a state. This is the only means of control in the RW framework. The second way is to indirectly prevent an event from occurring at a state by ensuring that some other event (called a *forcible* event) is guaranteed to always occur before it. This control action is called *pre-empting* an event occurrence by *forcing* another event to occur before it. The concept of forcing is present in the BW framework in a slightly different guise. These two control actions allow us to define the notion of controllability and supremal controllable sublanguage along the lines of the standard RW framework.

We now give the formal definitions. Let $B \subseteq (\mathbf{N} \cup \{\infty\}) \times \mathbf{N}$ be such that if $(x, y) \in B$ then $x \geq y$. Here we assume that $\infty > z$ for all $z \in \mathbf{N}$.

Definition 119 *A* Timed Generator *(TG)* $\mathbf{G} = \left(X^G, \Sigma^G, E^G, x_0^G, X_m^G, \mathfrak{F}^G\right)$ *is a 6-tuple where* $\left(X^G, \Sigma^G, E^G, x_0^G, X_m^G\right)$ *is the underlying automaton and* $\mathfrak{F}^G = \left\{f_\sigma^G : \sigma \in \Sigma^G\right\}$ *is a family of timer functions. For any* $\sigma \in \Sigma^G$*, the* timer function $f_\sigma^G : X^G \to B$ *maps each state to the upper and lower time bounds of* σ *at that state. Let* $x \in X^G$ *and assume that* $f_\sigma^G(x) = (b^1, b^2)$*. Then* $u_\sigma^G(x) = b^1$ *and* $l_\sigma^G(x) = b^2$ *are the upper and lower time bounds of* σ *at* x*. It will often be convenient to refer to the underlying automaton directly; we will use* $\mathbf{U}^G$ *to refer to* $\left(X^G, \Sigma^G, E^G, x_0^G, X_m^G\right)$*. When* $\mathbf{G}$ *is clear from the context, we will drop the superscript.*

Remark 2 *As in [Bra93],[BW94], we also require that there should not exist a sequence of transitions* $(x_1, \sigma_1, x_2), \ldots, (x_n, \sigma_n, x_1) \in E^G$ *such that* $l_{\sigma_1}(x_1) = \cdots = l_{\sigma_n}(x_n) = 0$*. If this were allowed then it would represent a scenario where a chain of events could occur infinitely often in a finite amount of time. There are no other restrictions on the timer functions* f_σ*.*

Definition 120 *Let* $\mathbf{G}$ *be a timed generator. The set of* enabled events *at any state* $x \in X^G$ *is defined to be*

$$Enab\left(\mathbf{G}, x\right) := Enab\left(\mathbf{U}^G, x\right).$$

An event $\sigma \in \Sigma$ *is* enabled *at* x *if* $\sigma \in Enab\left(\mathbf{G}, x\right)$.

Definition 121 *Let* $\mathbf{G}$ *be a timed generator. The set of* eligible events *at any state* $x \in X^G$ *is defined to be*

$$Elig\left(\mathbf{G}, x\right) := \left\{\sigma \in Enab\left(\mathbf{G}, x\right) : \left(\forall \alpha \in Enab\left(\mathbf{G}, x\right)\right) l_\sigma\left(x\right) \leq u_\alpha\left(x\right)\right\}.$$

An event $\sigma \in Elig\left(\mathbf{G}, x\right)$ is *eligible to occur* at x if its lower bound is not greater than the upper bound of any enabled event. In other words, an event is eligible to occur at a state if no other event is guaranteed to occur before it. If there exists a transition $(x, \sigma, x_1) \in E^G$ such that $\sigma \notin Elig\left(\mathbf{G}, x\right)$ then σ can never occur at x even though it is enabled there.

Definition 122 *Let* $\mathbf{G}$ *be a timed generator. Then* $\mathbf{G}$ *is* proper *if* $Elig\left(\mathbf{G}, x\right) = Enab\left(\mathbf{G}, x\right)$ *for all* $x \in X^G$. *If* $\mathbf{G}$ *is a timed generator then the corresponding proper timed generator is defined to be* $\mathbf{P}^G = \left(X^P, \Sigma^P, E^P, x_0^P, X_m^P, \mathfrak{F}^P\right)$ *where*

$$\left(X^P, \Sigma^P, E^P, x_0^P, X_m^P\right) := \left(X^G, \Sigma^G, E^G - E, x_0^G, X_m^G\right)_{rch},$$

$$E := \left\{(x_1, \sigma, x_2) \in E^G : \sigma \in Enab\left(\mathbf{G}, x_1\right) - Elig\left(\mathbf{G}, x_1\right)\right\},$$

$$\mathfrak{F}^P := \left\{ f_\sigma^G\big|_{X^{G_p}} : \sigma \in \Sigma^P \right\}.$$

Here $f_\sigma^G\big|_{X^P}$ *implies the restriction of* f_σ^G *to* X^P*. We will drop the superscript when* $\mathbf{G}$ *is clear from the context*

Thus $\mathbf{P}^G$ is defined by removing all those transitions from $\mathbf{G}$ that are enabled at some state but not eligible to occur at that state. The timer functions of $\mathbf{P}^G$ are defined by simply restricting the timer functions of $\mathbf{G}$ to the state set of $\mathbf{P}^G$.

Definition 123 *Let* $\mathbf{G}$ *be a timed generator. The* closed *and* marked languages *generated by* $\mathbf{G}$ *are defined to be the closed and marked languages generated by the underlying automaton of the corresponding proper timed generator, i.e.*

$$L(\mathbf{G}) := L\left(\mathbf{U}^{P^G}\right),$$
$$L_m(\mathbf{G}) := L_m\left(\mathbf{U}^{P^G}\right).$$

Example 124 *Let* $\mathbf{G}$ *be a timed generator as shown in Figure 7.5. The timer matrices*

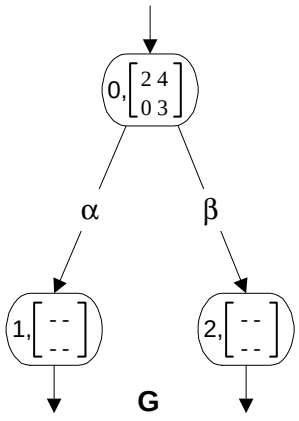


Figure 7.5: A timer graph that is not proper

shown in the nodes represent the time bounds of α *and* β*. Then we have*

$$Enab\left(\mathbf{G},0\right)=\left\{\alpha,\beta\right\},$$
$$Enab\left(\mathbf{G},1\right)=Enab\left(\mathbf{G},2\right)=\emptyset.$$

The lower time bound of β *at state* 0 *is* 3 *which is greater than the upper time bound,* 2*, of* α*. Thus* β *is not eligible to occur at state* 0*. So we have*

$$Elig\left(\mathbf{G},0\right)=\left\{\alpha\right\},$$
$$Elig\left(\mathbf{G},1\right)=Elig\left(\mathbf{G},2\right)=\emptyset.$$

From this we can compute the proper timed generator $\mathbf{P}^G$ *corresponding to* $\mathbf{G}$ *(as shown in Figure 7.6). Now the closed and marked languages generated by* $\mathbf{G}$ *are*

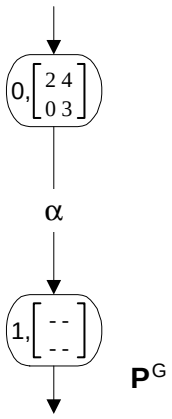


Figure 7.6: A proper timer graph

$$L\left(\mathbf{G}\right)=\left\{\epsilon,\alpha\right\},$$
$$L_m\left(\mathbf{G}\right)=\left\{\alpha\right\}.$$

□

Remark 3 *The language generated by a timed generator does not explicitly include time. However, if we want, we can give an interpretation that does explicitly include time in a manner similar to the language generated by a timed transition graph in the BW framework. Let us consider the timed generator* **G** *shown in Figure 7.7. At state* 0*, the time bounds of*

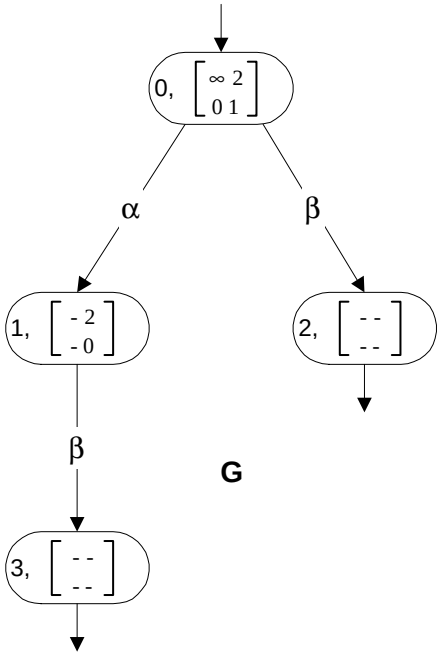


Figure 7.7: A timed generator

α and β are $(0, \infty)$ and $(1, 2)$ respectively; at state 1*, the time bounds of β are $(0, 2)$. So, at the initial state, the timer functions tell us that α can occur right away while β must wait for one time unit to elapse before it can occur. Let t represent the passage of one unit of time as measured by a digital clock. Now consider the graph* **H** *shown in Figure 7.8. The dashed boxes correspond to the states of the timed generator* **G**. *The box labelled* 0 *reflects the process going on at the initial state of* **G***: α can occur right away while β is not possible before the first occurrence of t. Now if α occurs then* **G** *tells us that the time bounds for β*

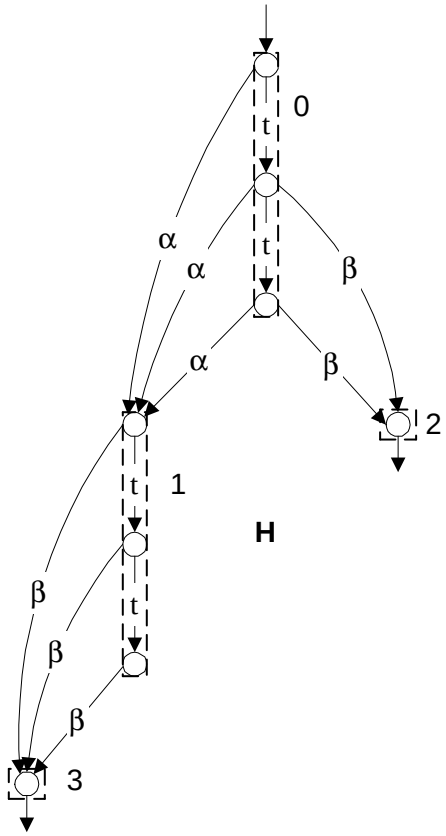


Figure 7.8: An explicit timed interpretation of a timed generator

are 0 *and* 2. *This is reflected in the box labelled* 1 *in* **H**. *So after the occurrence of* α *we may have any of the following strings:* β, $t\beta$, *or* $t^2\beta$. *The details of the various boxes in* **H** *are captured in* **G** *using timer functions. Given any timed generator* **G** *it is always possible to construct a corresponding timed transition graph similar to* **H**. *This is due to the fact that each state in* **G** *can be expanded into a corresponding box in* **H** *independently of all the other states. Each box has a unique entry point and has as many* t *events as required by the timer functions of* **G**. *We can think of* **G** *as being analogous to a timed activity transition graph (TATG) of* **H**. *A TATG is obtained by projecting out the tick transition from a TTG. Thus, in a sense, a timed generator is akin to a TATG equipped with timer*

functions.

7.4. Controllability and Supervision

In this section we assume that we are given a timed generator $\mathbf{G} = (X, \Sigma, E, x_0, X_m, \mathfrak{F})$ with $L = L(\mathbf{G})$ as its closed language. In order to use $\mathbf{G}$ as a model for supervisory control, we need to impose on it some sort of control technology. This control technology specifies ways in which the transitions $\mathbf{G}$ can be controlled by a supervisor. As in the RW framework, our control technology has events that can be disabled, namely *prohibitible events*. A supervisor may remove a prohibitible event from the list of eligible events at a state. An event $\sigma \in \Sigma$ can be prohibitible at a state $x \in X$ only if $u_\sigma(x) = \infty$. It is possible for an event to be prohibitible at one state and not prohibitible at another. Let $\Sigma_c^G(x) \subseteq \Sigma$ represent the set of prohibitible events at any state $x \in X$ in $\mathbf{G}$. Our control technology also has events, namely *forcible events*, that may be used to pre-empt the occurrence of other events. Let $\Sigma_f^G \subseteq \Sigma$ represent the set of forcible events in $\mathbf{G}$. With each forcible event we associate a partial map $m_\sigma : X \to \mathbf{N}$ such that $l_\sigma(x) \leq m_\sigma(x) \leq u_\sigma(x)$ for all $x \in X$. (We implicitly assume that any such statements about a partial map like m_σ apply to "all $x \in X$ at which $m_\sigma(x)$ is defined.") We shall assume that $m_\sigma(x) = l_\sigma(x)$ unless stated otherwise. Our control technology permits a supervisor to remove an event α from the list of eligible events at a state $x \in X$ if a forcible event σ is eligible there such that $m_\sigma(x) < l_\alpha(x)$. This supervisory action is called *pre-emption* of α by σ. Here $m_\sigma(x)$ represents the number of time units that must elapse before a supervisor is able to force σ to occur. This number can be used to model any forcing constraints that prevent σ from

being forced as soon as it is eligible. For any set $F \subseteq \Sigma$ let

$$\Sigma_p^G(x,F) := \left\{\alpha \in Elig(\mathbf{G},x) : \left(\exists \sigma \in \Sigma_f^G \cap Elig(\mathbf{G},x) \cap F\right) m_\sigma(x) < l_\alpha(x)\right\}$$

represent the set of events that may be pre-empted at a state x by some eligible forcible event in F. If $\Sigma_p^G(x, Elig(\mathbf{G},x)) \neq \emptyset$ then it necessarily means that $\Sigma_f^G \cap Elig(\mathbf{G},x) \neq \emptyset$ although more than one forcible event may be eligible at x. Let $\Sigma_u^G(x)$ represent the set of eligible events that are not prohibitible at x; this is called the set of *uncontrollable events* at x. Let $\mathbf{U}$ be the underlying automaton of the corresponding proper timed generator of $\mathbf{G}$. For any $s \in L(\mathbf{G})$, a state $x \in X^G$ is the *state corresponding to* s in $\mathbf{G}$ if it is the state corresponding to s in $\mathbf{U}$. Let $Elig(L,s) := Elig(\mathbf{G},x)$ where x is the state corresponding to s. Similarly, let $\Sigma_u^G(s) := \Sigma_u^G(x)$, $\Sigma_p^G(s,F) := \Sigma_p^G(x,F)$, $u_\sigma^G(s) := u_\sigma^G(x)$ and $l_\sigma^G(s) := l_\sigma^G(x)$. From this point on the treatment in this section follows along the lines of similar treatment in the RW framework [Won01]. In effect, we extend the RW notion of controllability and supervision to timed generators.

Definition 125 *A* supervisory control *for* $\mathbf{G}$ *is any map* $V : L \to 2^\Sigma$ *such that for all* $s \in L$

$$V(s) \supseteq \Sigma_u^G(s) - \Sigma_p^G(s, V(s)).$$

As before, we write $V/\mathbf{G}$ *to denote that* $\mathbf{G}$ *is under the supervision of* V. *The* closed behaviour *of* $V/\mathbf{G}$ *is the language* $L(V/\mathbf{G}) \subseteq L$ *defined inductively as given below.*

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

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

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

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

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

We say that V *is a* nonblocking supervisory control *(NSC) for* $\mathbf{G}$ *if* $\overline{L}_m(V/\mathbf{G}) = L(V/\mathbf{G})$.

A supervisory control must permit all those uncontrollable events that cannot be pre-empted. If $\Sigma_p^G(s, V(s)) = \emptyset$ then no forcible event is available at s so in that case a supervisory control must permit all the eligible uncontrollable events.

Definition 126 *A language* $K \subseteq L$ *is* controllable with respect to $\mathbf{G}$ *if for all* $s \in \overline{K}$

$$Elig(K, s) \supseteq \Sigma_u^G(s) - \Sigma_p^G(s, Elig(K, s)).$$

A language is controllable if it allows an eligible uncontrollable event whenever it cannot be pre-empted. As in RW, controllability of a language is a property only of its prefix closure. We now present a technical definition that will be needed later.

Definition 127 *Let* $H \subseteq K \subseteq \Sigma^*$. *Then* H *is* K-closed *if* $H = \overline{H} \cap K$. *Thus* H *is* K-closed *if it contains every one of its prefixes that belong to* K.

Theorem 128 *Let* $K \subseteq L_m(\mathbf{G})$, $K \neq \emptyset$. *Then there exists a nonblocking supervisory control* V *for* $\mathbf{G}$ *such that* $L_m(V/\mathbf{G}) = K$ *if and only if*

(i) K *is controllable with respect to* $\mathbf{G}$, *and*

(ii) K *is* $L_m(\mathbf{G})$*-closed.*

Proof. (**If**) Define $V : L \to 2^{\Sigma}$ as follows.

$$V(s) := \begin{cases} Elig(K, s) & \text{if } s \in \overline{K} \\ \Sigma & \text{if } s \in L - \overline{K} \end{cases}.$$

We first show that V is a supervisory control for $\mathbf{G}$, i.e.

$$V(s) \supseteq \Sigma_u^G(s) - \Sigma_p^G(s, V(s)).$$

If $s \in L - \overline{K}$ then

$$\begin{aligned} V(s) &= \Sigma \\ &\supseteq \Sigma_u^G(s) - \Sigma_p^G(s, V(s)) \end{aligned}$$

and if $s \in \overline{K}$ then

$$\begin{aligned} V(s) &= Elig(K, s) \\ &\supseteq \Sigma_u^G(s) - \Sigma_p^G(s, Elig(K, s)) \\ &= \Sigma_u^G(s) - \Sigma_p^G(s, V(s)) \end{aligned}$$

since K is controllable with respect to $\mathbf{G}$. Now $L(V/\mathbf{G}) = \overline{K}$ since $V(s) = Elig(K, s)$ so

$$\begin{aligned} L_m(V/\mathbf{G}) &= L(V/\mathbf{G}) \cap L_m(\mathbf{G}) \\ &= \overline{K} \cap L_m(\mathbf{G}) \\ &= K \end{aligned}$$

since K is $L_m(\mathbf{G})$-closed. Thus $\overline{L}_m(V/\mathbf{G}) = \overline{K} = L(V/\mathbf{G})$ which implies that V is a nonblocking supervisory control for $\mathbf{G}$.

(**Only If**) Let V be a nonblocking supervisory control for $\mathbf{G}$ with $L_m(V/\mathbf{G}) = K$. Since V is nonblocking, we have $L(V/\mathbf{G}) = \overline{K}$ so

$$\begin{aligned} K &= L(V/\mathbf{G}) \cap L_m(\mathbf{G}) \\ &= \overline{K} \cap L_m(\mathbf{G}) \end{aligned}$$

which shows that K is $L_m(\mathbf{G})$-closed. We now show that K is controllable with respect to $\mathbf{G}$. Let $s \in \overline{K}$. We need to show that

$$Elig(K,s) \supseteq \Sigma_u^G(s) - \Sigma_p^G(s, Elig(K,s)).$$

By the definition of $L(V/\mathbf{G})$ we have

$$\begin{aligned} Elig(K,s) &= V(s) \cap Elig(L,s) \\ &\supseteq \left(\Sigma_u^G(s) - \Sigma_p^G(s, V(s))\right) \cap Elig(L,s) \\ &= \Sigma_u^G(s) - \Sigma_p^G(s, V(s)) \\ &= \Sigma_u^G(s) - \Sigma_p^G(s, V(s) \cap Elig(L,s)) \\ &= \Sigma_u^G(s) - \Sigma_p^G(s, Elig(K,s)) \end{aligned}$$

which shows that K is controllable. ■

As in RW, we introduce a slight generalization of NSC in which the supervisory action includes marking as well as control.

Definition 129 *Let* $M \subseteq L_m(\mathbf{G})$. *A* marking nonblocking supervisory control *(MNSC)*

for the pair $(M, \mathbf{G})$ *is a supervisory control* V *such that the marked behaviour of* $V/\mathbf{G}$ *is defined as*

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

Theorem 130 *Let* $K \subseteq L_m(\mathbf{G})$, $K \neq \emptyset$. *Then there exists an MNSC for* $(K, \mathbf{G})$ *such that* $L_m(V/\mathbf{G}) = K$ *if and only if* K *is controllable with respect to* $\mathbf{G}$.

Proof. (**If**) Again let $V(s) := Elig(K, s)$. Then following along the lines of the proof of Theorem 128 we get $L(V/\mathbf{G}) = \overline{K}$. Thus

$$\begin{aligned} L_m(V/\mathbf{G}) &= L(V/\mathbf{G}) \cap K \\ &= \overline{K} \cap K \\ &= K \end{aligned}$$

so that $\overline{L}_m(V/\mathbf{G}) = \overline{K} = L(V/\mathbf{G})$. Thus V is nonblocking for $\mathbf{G}$.

(**Only If**) We have $\overline{K} = \overline{L}_m(V/\mathbf{G}) = L(V/\mathbf{G})$. The rest of the proof is the same as the proof for Theorem 128. ■

Example 131 *Let us continue with the parking setup. Let the timed generator shown in Figure 7.9 be called* $\mathbf{G}$. *We still want to avoid getting a ticket. Let this specification (shown in Figure 7.1) be called* K. *From Theorem 130 we know that an MNSC exists if* K *is controllable. We now show how its controllability may be verified. Let* $s \in \overline{K} \subseteq L(\mathbf{G})$.

Case 1: *Assume that* s *corresponds to the* idle *state in* $\mathbf{G}$. *There is no uncontrollable event eligible at* idle *so* $\Sigma_u^G(s) = \emptyset$. *Then*

$$Elig(K, s) = \{park\}$$

$$\begin{aligned} &\supseteq \Sigma_u^G(s) \\ &\supseteq \Sigma_u^G(s) - \Sigma_p^G(s, Elig(K,s)). \end{aligned}$$

Case 2: *Assume that s corresponds to the* occupied *state in* **G**. *From Figure 7.9 we can see that*

$$\begin{aligned} \Sigma_u^G(occupied) &= \{ticket\} \\ &= \Sigma_p^G(occupied, \{unpark\}) \end{aligned}$$

so $\Sigma_u^G(s) - \Sigma_p^G(s) = \emptyset$. *Again*

$$\begin{aligned} Elig(K,s) &= \{unpark\} \\ &\supseteq \emptyset \\ &= \Sigma_u^G(s) - \Sigma_p^G(s, \{unpark\}). \end{aligned}$$

These are the only two cases that we need to consider because no string belonging to $\overline{K}$ *contains the event* ticket. *Thus we see that K is controllable and we can find a marking nonblocking supervisory control to enforce it.* □

7.5. Supremal Controllable Sublanguages

The treatment in this section also follows along the lines of RW [Won01]. Let $\mathbf{G} = (X, \Sigma, E, x_0, X_m, \mathfrak{F})$ be a timed generator.

Definition 132 *Let $S \subseteq \Sigma^*$. Then the set of all sublanguages of S that are controllable*

with respect to $\mathbf{G}$ *is defined to be*

$$\mathcal{C}(\mathbf{G}, S) := \{K \subseteq S : K \text{ is controllable with respect to } \mathbf{G}\}.$$

Proposition 133 *Let* $S \subseteq \Sigma^*$. *Then* $\mathcal{C}(\mathbf{G}, S)$ *is nonempty and closed under arbitrary unions. In particular,* $\mathcal{C}(\mathbf{G}, S)$ *contains a unique supremal element (denoted by* $\sup \mathcal{C}(\mathbf{G}, S)$*).*

Proof. The empty language is trivially controllable so it belongs to $\mathcal{C}(\mathbf{G}, S)$ which implies that $\mathcal{C}(\mathbf{G}, S) \neq \emptyset$. Let K_i belong to $\mathcal{C}(\mathbf{G}, S)$ for all i in some index set I and let $K = \bigcup_{i \in I} K_i$. Then $K \subseteq S$ since $K_i \subseteq S$ for all $i \in I$. We need to show that K is controllable with respect to $\mathbf{G}$. Let $s \in \overline{K} \cap L(\mathbf{G})$. Then $Elig(K, s) = \bigcup_{i \in I} Elig(K_i, s)$ since $\overline{K} = \bigcup_{i \in I} \overline{K}_i$. Since K_i are controllable with respect to $\mathbf{G}$ it follows that

$$\begin{aligned}
Elig(K, s) &= \bigcup_{i \in I} Elig(K_i, s) \\
&\supseteq \bigcup_{i \in I} \left[\Sigma_u^G(s) - \Sigma_p^G(s, Elig(K_i, s))\right] \\
&\supseteq \Sigma_u^G(s) - \bigcup_{i \in I} \Sigma_p^G(s, Elig(K_i, s)) \\
&= \Sigma_u^G(s) - \Sigma_p^G\left(s, \bigcup_{i \in I} Elig(K_i, s)\right) \\
&= \Sigma_u^G(s) - \Sigma_p^G(s, Elig(K, s)).
\end{aligned}$$

If $s \in \overline{K} - L(\mathbf{G})$ then $Elig(L(\mathbf{G}), s) = \emptyset$ so

$$\Sigma_u^G(s) - \Sigma_p^G(s, Elig(K, s)) = \emptyset$$

$$\subseteq \quad Elig\left(K, s\right).$$

Thus K is controllable with respect to $\mathbf{G}$. Finally, for the supremal element we have

$$\sup \mathcal{C}\left(\mathbf{G}, S\right) = \bigcup \left\{K : K \in \mathcal{C}\left(\mathbf{G}, S\right)\right\}.$$

■

Definition 134 *Let $H, K \subseteq \Sigma^*$. We say that H is* K-marked *if $H \supseteq \overline{H} \cap K$. So H is* K-marked *if any prefix of H that belongs to K also belongs to H.*

Lemma 135 *[Won01, page 295] Let $S \subseteq \Sigma^*$ be $L_m\left(\mathbf{G}\right)$-marked. Then $\sup \mathcal{C}\left(\mathbf{G}, S \cap L_m\left(\mathbf{G}\right)\right)$ is $L_m\left(\mathbf{G}\right)$-closed.*

We now present the main results of this section.

Theorem 136 *Let $S \subseteq \Sigma^*$ be $L_m\left(\mathbf{G}\right)$-marked, and let $K = \sup \mathcal{C}\left(\mathbf{G}, S \cap L_m\left(\mathbf{G}\right)\right)$. If $K \neq \emptyset$ then there exists a nonblocking supervisory control V for $\mathbf{G}$ such that $L_m\left(V/\mathbf{G}\right) = K$.*

Proof. K is $L_m\left(\mathbf{G}\right)$-closed by Lemma 135; it is controllable by definition. The desired result now follows from Theorem 128. ■

This theorem may be interpreted as follows. The language S represents a specification that needs to be imposed on $\mathbf{G}$. If K is non-empty then it is the maximally permissive solution to the problem of supervising $\mathbf{G}$ such that its behaviour abides by the specification S. Additionally, the control is nonblocking. If K is empty then there exists no nonblocking supervisor that can impose S on $\mathbf{G}$.

As in Section 7.4, we now present the following result on the marking nonblocking supervisory control of $\mathbf{G}$.

Theorem 137 *Let* $S \subseteq \Sigma^*$ *and let* $K = \sup \mathcal{C}(\mathbf{G}, S \cap L_m(\mathbf{G}))$. *If* $K \neq \emptyset$ *then there exists a marking nonblocking supervisory control* V *for* $\mathbf{G}$ *such that* $L_m(V/\mathbf{G}) = K$.

7.6. Temporal Specifications

Let a timed generator $\mathbf{G}$ represent a physical system. Then a language $S \subseteq \Sigma^{G*}$ can be used to specify how the system should behave. We have seen that if $\sup \mathcal{C}(\mathbf{G}, S \cap L_m(\mathbf{G}))$ is nonempty then we can construct a nonblocking supervisor to achieve the desired system behaviour. The timing information inherent in $\mathbf{G}$ may allow the construction of a supervisor that would not have been possible otherwise. This is what happens in the parking scenario shown in Figure 7.1. The parking specification cannot be achieved if the timing information is absent. The parking specification itself contains no explicit temporal requirement but its achievement is not possible without temporal information. However there may exist scenarios where we may want to specify temporal requirements explicitly. In this section we show how that may be done in the framework of timed generators.

Definition 138 *Let* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *be two timed generators over the same alphabet* Σ. *Then the* meet *of* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *(denoted* $\mathbf{G}_1 \wedge \mathbf{G}_2$*)is a timed generator* $\mathbf{G} := \left(\mathbf{U}, \mathfrak{F}^G\right)$ *where* $\mathbf{U} = \mathbf{U}^{G_1} \wedge \mathbf{U}^{G_2}$ *and the timer functions are defined as follows. Let* $x = (x_1, x_2) \in X^G$ *and* $\sigma \in \Sigma$; *then*

$$f_\sigma^G(x) := \left(\min\left(u_\sigma^{G_1}(x_1), u_\sigma^{G_2}(x_2)\right), \max\left(l_\sigma^{G_1}(x_1), l_\sigma^{G_2}(x_2)\right)\right).$$

The meet of $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *is well-defined only if* $\min\left(u_\sigma^{G_1}(x_1), u_\sigma^{G_2}(x_2)\right) \geq \max\left(l_\sigma^{G_1}(x_1), l_\sigma^{G_2}(x_2)\right)$. *An event* $\sigma \in \Sigma$ *is prohibitible at* x *in* $\mathbf{G}$ *if and only if it is prohibitible at* x_1 *at* $\mathbf{G}_1$ *and at* x_2 *in* $\mathbf{G}_2$. *An event* $\sigma \in \Sigma$ *is forcible in* $\mathbf{G}$ *if and only if it is forcible in both* $\mathbf{G}_1$ *and* $\mathbf{G}_2$.

If σ *is forcible then* $m_{\sigma}^{G}(x) = \max\left(m_{\sigma}^{G_1}(x_1), m_{\sigma}^{G_2}(x_2)\right)$.

Remark 4 *If* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *have to cooperate on the execution of an event* σ *then it is not possible for* σ *to occur before it is eligible to occur in both* $\mathbf{G}_1$ *and* $\mathbf{G}_2$. *Similarly, it is not possible for* σ *to be delayed longer than it can be delayed in either* $\mathbf{G}_1$ *or* $\mathbf{G}_2$. *This requirement is the same as in [Bra93],[BW94].*

Example 139 *Let us reconsider the parking spot scenario shown in Figure 7.1. We can model the parking spot using a timed generator as shown in Figure 7.9. The various states*

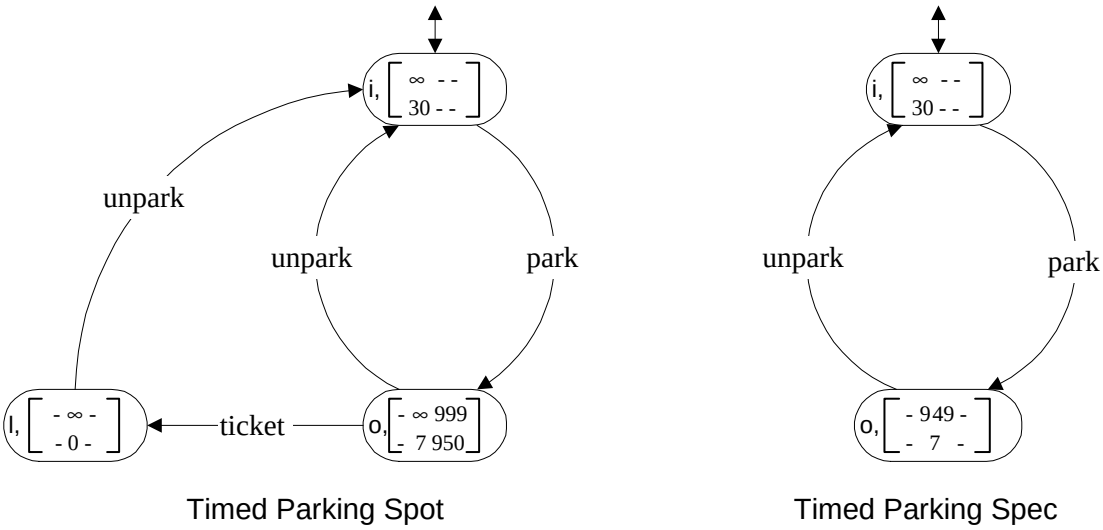


Figure 7.9: Timed Generator for the Parking Spot

are shown with timer matrices representing the output of timer functions. For brevity, we have labelled idle *with* i, occupied *with* o, *and* late *with* l. *The controllability attributes of*

various events are as follows.

$$\begin{aligned}
&\Sigma_f = \{unpark\} \\
&\Sigma_c(idle) = \{park, unpark\} && \Sigma_u(idle) = \{ticket\} \\
&\Sigma_c(occupied) = \{park, unpark\} && \Sigma_u(occupied) = \{ticket\} \\
&\Sigma_c(late) = \{park, unpark\} && \Sigma_u(late) = \{ticket\}
\end{aligned}$$

The only forcible event is unpark *and it can be used to preempt* ticket *at the* occupied *state. Thus*

$$\Sigma_p(occupied, \{unpark\}) = \{ticket\}.$$

□

Let $\mathbf{S}$ be a timed generator such that $L_m(\mathbf{S}) = S$. We can define its timer functions to reflect our temporal requirements. Then the timer functions of $\mathbf{G} \wedge \mathbf{S}$ will retain that temporal specification. Now we just need to check the controllability of S with respect to $\mathbf{G} \wedge \mathbf{S}$. If S is controllable then there exists an MNSC that implements the linguistic specification S as well as the temporal specification conveyed by $\mathbf{S}$. We present a couple of examples to illustrate this.

Example 140 *Let us consider our running example of the parking spot. Let the new specification be that the car should be unparked within 500 seconds of being parked. Consider the timed generator shown in Figure 7.10. The linguistic interpretation still remains the same: do not get a ticket. However the upper bound of* unpark *at the occupied state is now 500. This reflects the requirement that the car should be unparked within 500 seconds.* □

Example 141 *Let us consider the setup of Example 116. The specification for this system, H, is shown in Figure 7.2. It says that α should be forced to occur right away at the initial*

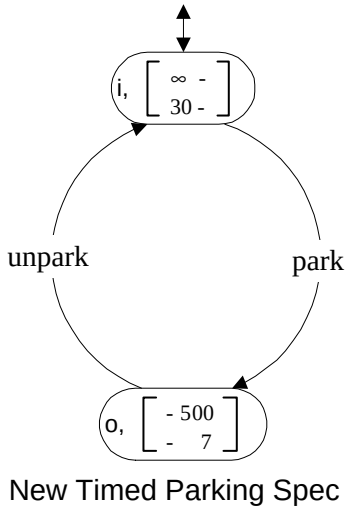


Figure 7.10: Temporal Specification for Parking Setup

state. A timed generator representing this specification is shown in Figure 7.11. The upper

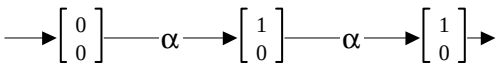


Figure 7.11: Timed Generator for the Scenario of Example 116

bound of α is set to zero at the initial state to reflect the temporal aspect of H. The bounds for the second occurrence of α are left unchanged. Therefore the second occurrence of α is not constrained. □

7.7. Closure Of Timed Generators Under Control

As mentioned earlier, one of the major drawbacks of the BW framework is that it is not closed under control. In BW, the design begins with a TTG that corresponds to an ATG. However, there may not exist an ATG corresponding to a TTG under control. In this section we show that timed generators are closed under control. Let $\mathbf{G}$ be a timed generator and $K \subseteq L(\mathbf{G})$ be a controllable sublanguage. Let V be an MNSC that restricts

G to K. Then, as we now show, there always exists a timed generator that generates K and respects the dynamics of **G**. For the rest of this section we assume that **G** and K are given.

Definition 142 *Let* **K** *be any timed generator that generates* K*. Then we say that* **K** *is* faithful *to* **G** *if for all* $s \in \overline{K}$*,* $\sigma \in \Sigma^G$

$$\begin{array}{l} l_\sigma^K(s) \geq l_\sigma^G(s) \\ u_\sigma^K(s) \leq u_\sigma^G(s) \end{array}.$$

If **K** is faithful to **G** then its time bounds reflect the status of the corresponding time bounds in $V/\mathbf{G}$. It is important that any construction of a timed generator for K be faithful to **G** because K is intended to represent the behaviour of the closed-loop system. The language K is synthesized under the supervisory control V and any timed generator for K should reflect that. We also want **K** to have the least restrictive time bounds while maintaining its faithfulness to **G**.

Let **K** be a proper timed generator with a minimal-state underlying automaton that generates K, i.e. $L_m(\mathbf{K}) = K$. For each $\sigma \in \Sigma$ let $f_\sigma^K : X^K \to B$ be a function defined as follows. Let $x \in X^K$ and let $s \in K \subseteq L(\mathbf{G})$ be a string corresponding to x. Let $y \in X^G$

be the state in $\mathbf{G}$ corresponding to s. Then

$$f_\sigma^K(x) := \begin{cases} f_\sigma^G(y) & \text{if } (\sigma \notin \Sigma_f) \vee \\ & Elig(\mathbf{K}, x) \supseteq \Sigma_u(s) \cap Elig(\mathbf{G}, y) \vee \\ & m_\sigma^G(y) > m \\ & \\ \left(m, l_\sigma^G(y)\right) & \text{if } (\sigma \in \Sigma_f) \wedge \\ & Elig(\mathbf{K}, x) \not\supseteq \Sigma_u(s) \cap Elig(\mathbf{G}, y) \wedge \\ & m_\sigma^G(y) \leq m \end{cases}$$

where

$$m = \left(\min_{\alpha \in \Sigma_p^G(y)} l_\alpha^G(y) \right) - 1.$$

Also, if $\sigma \in \Sigma_f$ then

$$m_\sigma^K(x) := m_\sigma^G(y).$$

Thus, for a non-forcible event σ, the timer function f_σ^K is defined to be the same as f_σ^G restricted to the strings in $\overline{K}$. This scheme is also followed for forcible events except when an uncontrollable event has been pre-empted. So if one or more uncontrollable events have been pre-empted at a state $x \in X^K$ then $u_\sigma^K(x)$ is defined to be one time unit less than the earliest time at which a pre-empted event is eligible. This is done only if the forcible event is actually capable of performing the pre-emption, i.e. if $m_\sigma^G(y)$ is smaller than the earliest time at which a pre-empted event is eligible. Note that this does not designate any particular forcible event to do the pre-emption. That choice is left to the designer.

Theorem 143 *Let* $\mathbf{K}$ *be a timed generator as defined above. Then* $\mathbf{K}$ *is faithful to* $\mathbf{G}$*. If* $\mathbf{K}'$ *is any other timed generator that is faithful to* $\mathbf{G}$ *and generates* K *then for all* $s \in \overline{K}$*,*

$\sigma \in \Sigma^G$ we must have $u_\sigma^{K'}(s) \leq u_\sigma^K(s)$ and $l_\sigma^{K'}(s) \geq l_\sigma^K(s)$.

Proof. **K** is faithful by definition. Its lower bounds are always equal to the lower bounds of the corresponding events in **G**. The upper bound of any non-forcible event also always equals the upper bound of the corresponding event in **G**. The upper bounds of its forcible events may be less, but never more, than the upper bounds of the corresponding events in **G**.

Now assume that $\mathbf{K'}$ is a faithful timed generator that generates K. Let $s \in \overline{K}$. Then for all $\sigma \in \Sigma^G - \Sigma_f^G$ we must have

$$\begin{aligned} u_\sigma^{K'}(s) &\leq u_\sigma^G(s) \\ &= u_\sigma^K(s). \end{aligned}$$

Now let $\sigma \in \Sigma_f$. If $\Sigma_p^G(s) = \emptyset$ then again we must have

$$\begin{aligned} u_\sigma^{K'}(s) &\leq u_\sigma^G(s) \\ &= u_\sigma^K(s). \end{aligned}$$

So let us assume that $\Sigma_p^G(s) \neq \emptyset$. If $u_\sigma^{K'}(s) > u_\sigma^K(s)$ then there must exist some uncontrollable event $\alpha \in \Sigma_p^G(s)$ eligible at s such that $u_\sigma^{K'}(s) \geq l_\alpha^G(s)$. This would mean that K is no longer controllable since $\alpha \notin Elig(K, s)$ and there is no forcible event that could have pre-empted it. However this is contradictory to our assumption that $\mathbf{K'}$ is faithful to **G**. Thus in this case also we must have $u_\sigma^{K'}(s) \leq u_\sigma^K(s)$. ■

In a sense, **K** is the most faithful timed generator for K. It reflects the least restrictive supervisory control needed to synthesize K. We now present a few examples to illustrate

the closure of timed generators under control.

Example 144 *Let us reconsider the parking spot scenario shown in Figures 7.1 and 7.9. We already know that the parking specification is controllable. The corresponding timed generator is shown in Figure 7.12.* □

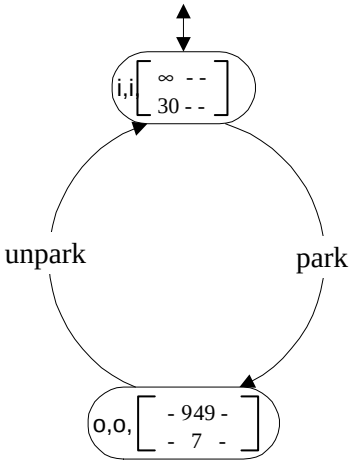


Figure 7.12: Timed Generator for the Parking Specification

Example 145 *Let us consider the setup of Example 116. Here the problem arises because α has two different upper bounds. This problem does not arise in the framework of timed generators because the time bounds are local to each state. The corresponding timed generator is shown in Figure 7.11.* □

Example 146 *Let us consider the setup of Example 118. A timed generator that generates the language L is shown in Figure 7.13. A timed generator that generates the language H is shown in Figure 7.14. Now it is easily seen that the language K is no longer controllable with respect to the timed generator of H. This is due to the fact that the timed generator of H contains information regarding the control action used to synthesize H. As a result, the event α is no longer prohibitible at the initial state (its upper bound is not ∞).* □

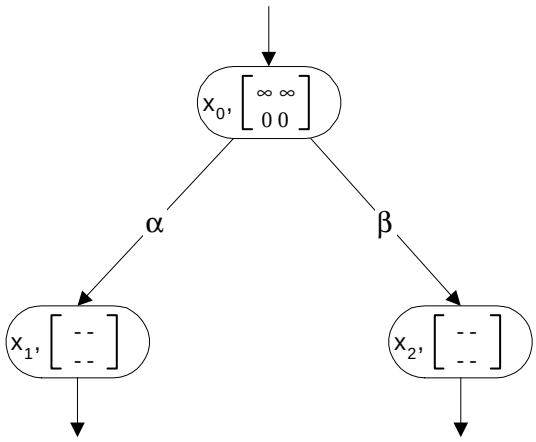


Figure 7.13: Timed Generator that generates L of Example 118

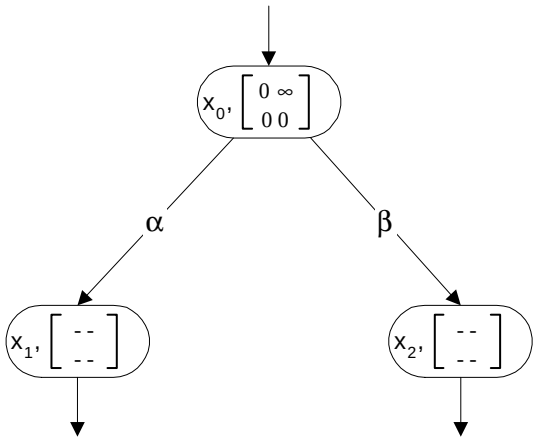


Figure 7.14: Timed Generator that generates H of Example 118

7.8. Derivation of a TG from an ATG

Thus far in this chapter we have specified timed generators directly. The timed generator framework has been laid down without explicitly mentioning activity or timed transition graphs. However it may sometimes be convenient to specify a physical system using an ATG. In this section we show how we may derive an equivalent timed generator representation. Let **A** be an ATG and assume that the time bounds for the various events

are given. We are interested in finding a timed generator model **G** for the system modelled by **A**. Let **TA** and **PA** represent the TTG and the TATG corresponding to **A**. Then $L(\mathbf{PA}) = Q_t(L(\mathbf{TA}))$ and $L_m(\mathbf{PA}) = Q_t(L_m(\mathbf{TA}))$ where Q_t is the natural projection that removes the occurrences of the *tick* event. The language $L(\mathbf{PA})$ reflects the various phase relationships between the events of the system modelled by **A**. Thus any timed generator modelling the same physical system should also generate $L(\mathbf{PA})$. In particular, we must have $L(\mathbf{G}) = L(\mathbf{PA})$ and $L_m(\mathbf{G}) = L_m(\mathbf{PA})$. This can be done by choosing **PA** as the underlying automaton for **G**, i.e. by setting $\mathbf{U}^G = \mathbf{PA}$. We also need to specify the timer functions for **G**. In order to do that, we make the following two observations.

1. The state set X^{PA} of **PA** can be induced from an appropriate cover of the state set X^{TA} of **TA** [HU79]. This process is called subset-construction. So with each state $x \in X^{PA}$ we can associate a subset $TA_x \subseteq X^{TA}$. With each state $q \in X^{TA}$ there is an associated timer matrix T_q which reflects the status of the various event timers at q. Let

$$B_x = \{T_q : q \in TA_x\}$$

be the set of timer matrices corresponding to the subset TA_x.

2. Let $w \in L(\mathbf{PA})$ be a string corresponding to a state $x \in X^{PA}$. Recall from Chapter 6 that

$$S_w = \left\{s \in L(\mathbf{TA}) \cap \Sigma^{TA*}\Sigma^{PA} : Q_t(s) = w\right\}$$

is the set of strings in $L(\mathbf{TA})$ that correspond to w. Let

$$B'_x = \{T_s : s \in S_w\}$$

be the set of timer matrices corresponding to the strings in S_w. These timer matrices

reflect the status of the event timers after the occurrence of a string s. From Section 6.4 we know that a partial order can be defined on B'_x under which the minimal and maximal elements correspond to the shortest and longest prefix path strings in S_w.

Now the timer functions can be defined in accordance with the minimal and maximal elements of B'_x. However, B'_x is not easy to compute directly. On the other hand, B_x can be easily constructed but, in general, it is bigger than B'_x , i.e. $B_x \supseteq B'_x$. This can be seen as follows. Every string $s \in S_w$ belongs to $L(\mathbf{TA})$ and $Q_t(s) = w$. Now w corresponds to $x \in X^{TA}$ so $T_s \in B_x$. Thus we want to consider only those elements of B_x which also belong to B'_x. In other words, we can represent the set B'_x as

$$B'_x = \left\{ T_q : q \in TA_x \text{ and } x \text{ corresonds to some } s \in L(\mathbf{TA}) \cap \Sigma^{TA*}\Sigma^{PA} \right\}.$$

Now B'_x can be easily constructed from the knowledge of **TA** and **PA**. Let $T_{\min}(x)$ and $T_{\max}(x)$ be the minimal and maximal elements of B'_x . Let $\sigma \in \Sigma^{PA} = \Sigma$. Then the timer function $f^G_\sigma : X \to B$ can be defined as

$$f^G_\sigma(x) = (u, l)$$

where u is the upper timer value of σ in $T_{\min}(x)$ and l is the lower timer value of σ in $T_{\max}(x)$. In addition, if $\sigma \in \Sigma_f$ then $m_\sigma(x)$ is set equal to the lower timer value of σ in $T_{\min}(x)$. Since $T_{\max}(x)$ is the prefix longest path, we can be sure that $l_\sigma(x)$ represents the smallest time (as measured by the digital clock) that must elapse before σ may occur at x. Similarly, we can be sure that $u_\sigma(x)$ represents the greatest time that may elapse before σ must occur at x. We have also defined $m_\sigma(x)$ to be equal to the lower timer value of σ in $T_{\min}(x)$. We present a small example to clarify the reason behind this choice.

Example 147 *Let* **A** *be an ATG as shown in Figure 7.15. Let us assume that the default*

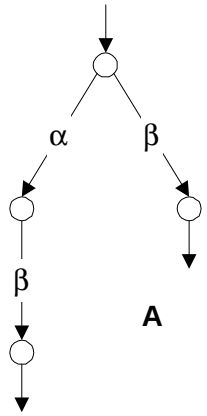


Figure 7.15: Reason for setting m_σ: ATG

timer values are given by the following triples: $(\alpha, 0, 1)$ *and* $(\beta, 1, 1)$. *The corresponding TTG,* **GA**, *is shown in Figure 7.16. Let us assume that* β *is a forcible event. Now consider the scenario where* α *has occurred at the initial state. The system could be in either of the states shown in the dashed rectangle. So the lower bound of* β *could be either* 0 *or* 1. *Clearly the value of* m_β *should not be less than the lower bound. From Chapter 6 we know that the maximal value of the lower bound corresponds to the shortest prefix string. So we set* m_β *equal to* 1 *in this case.* □

We now illustrate the overall derivation process with the help of a simple example.

Example 148 *Let* **A** *be an ATG as shown in Figure 7.17. Let us assume that the default timer values are given by the following triples:* $(\alpha, 0, 1)$ *and* $(\beta, 1, 1)$. *The corresponding TTG,* **GA**, *is shown in Figure 7.18. The timer matrix corresponding to each state is shown in the corresponding node of the graph. The timed activity transition graph* **PA** *is shown in Figure 7.19. Next to each state of the TATG, we have shown the corresponding*

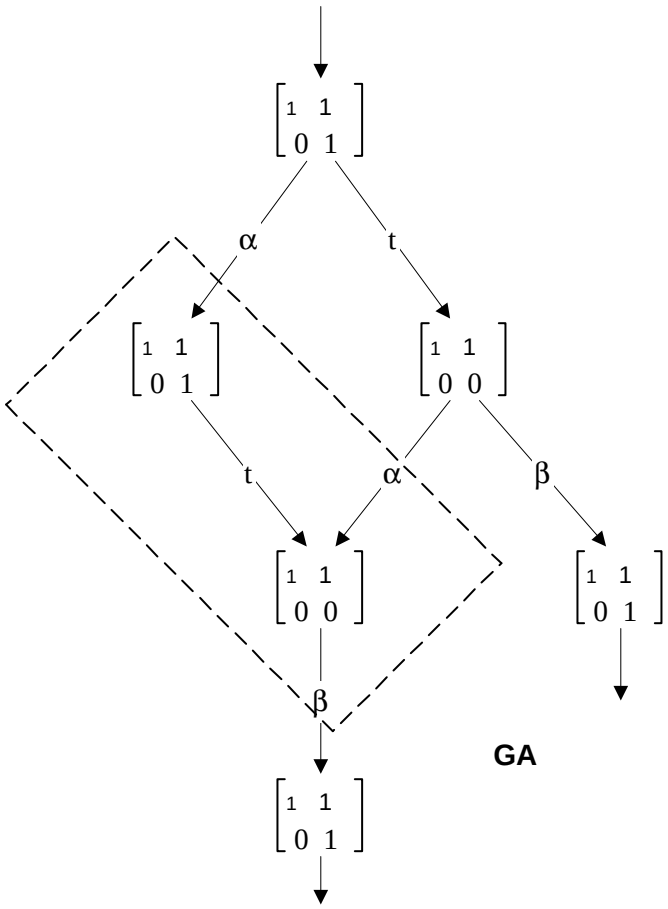


Figure 7.16: Reason for setting m_σ: TTG

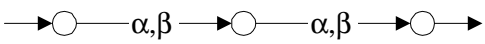


Figure 7.17: ATG **A** for which an equivalent TG is to be found

subset of the state set of **GA**. *From this we get*

$$B_0 = \{0, 1\} \quad B_1 = \{2, 3, 6\}$$
$$B_2 = \{4\} \qquad B_3 = \{5, 7, 8, 9, 10\}\,.$$

It can be seen from Figure 7.18 that state 1 *in* **GA** *corresponds to the string* t *and therefore*

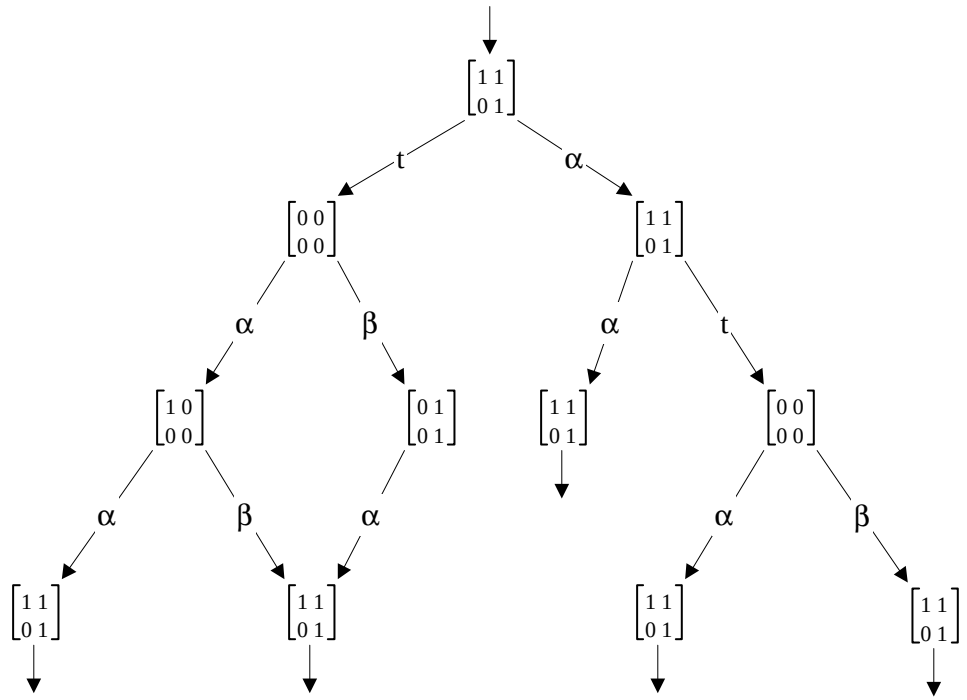


Figure 7.18: TTG **GA**

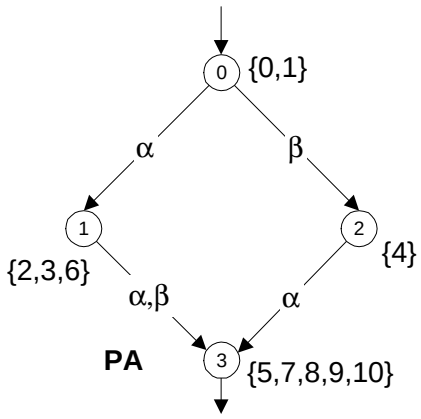


Figure 7.19: TATG **PA**

it must belong to $B_0 - B_0'$. *Similarly, state* 6 *in* **GA** *corresponds to the string* αt *and*

therefore it must belong to $B_1 - B_1'$. *This gives*

$$B_0' = \{0\} \quad B_1' = \{2, 3\}$$
$$B_2' = \{4\} \quad B_3' = \{5, 7, 8, 9, 10\}$$

and

$$T_{\min}(0) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad T_{\max}(0) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$
$$T_{\min}(1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad T_{\max}(1) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$
$$T_{\min}(2) = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \quad T_{\max}(2) = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$
$$T_{\min}(3) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad T_{\max}(3) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

The derived timed generator is shown in Figure 7.20. □

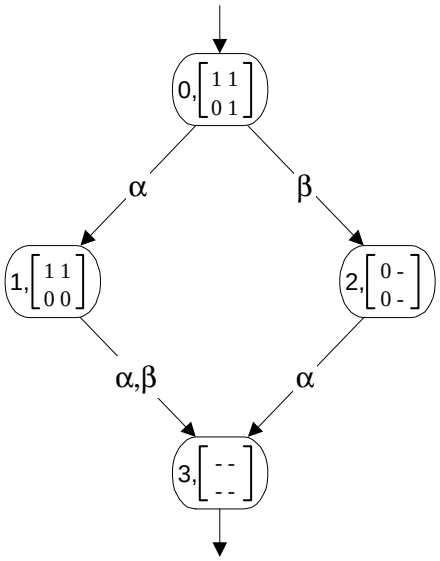


Figure 7.20: Derived Timed Generator **G**

Remark 5 *Each state in an ATG may give rise to* $\prod_{\sigma\in\Sigma} t_\sigma^0$ *states in the corresponding TTG where*

$$t_\sigma^0 := \begin{cases} u_\sigma & \text{if } u_\sigma \neq \infty \\ l_\sigma & \text{otherwise} \end{cases}$$

is the default timer value of an event σ*. This is the reason why a TTG may be much bigger in size than the ATG. We have shown a number of examples where a timed generator provides a much more compact representation than a TTG. However, if we are deriving a timed generator from an ATG then it is possible that the timed generator is even bigger in size than the corresponding TTG. This may be explained as follows. It is no longer sufficient to give one number, namely* t_σ^0*, to specify the timer values of* σ*. The lower and upper bounds of* σ *may vary from state to state. Thus each state in an ATG may give rise to* $\prod_{\sigma\in\Sigma} (l_\sigma \cdot u'_\sigma)$ *states in the timed generator, where*

$$u'_\sigma := \begin{cases} u_\sigma & \text{if } u_\sigma \neq \infty \\ 1 & \text{otherwise} \end{cases} .$$

Luckily, the number of states in the timed generator is usually far fewer. A typical scenario where the number of states in a timed generator is more than the number of states in a TTG is when the ATG has a lot of self-looped events with finite upper bounds.

7.9. Synchronous Composition of Timed Generators

In this section we show how to compose timed generators. Often a system comprises many subsystems and the overall system behaviour is due to the concurrent operation of the subsystems. The timed generator model of such a system may be obtained by forming the synchronous composition of the timed generators of the subsystems. When the subsystems

are defined over the same alphabet then the synchronous composition reduces to the *meet* defined in Definition 138. We present an algorithm for forming the synchronous composition of timed generators. The procedure is quite similar to the derivation of a timed generator from an ATG: we want to find the timer values corresponding to the shortest and longest paths in the concurrent operation of two timed generators. This is much simpler than the corresponding problem in a timed transition graph due to the fact that we already know the timer functions in the component timed generators. The timer functions for the composite system can be found by simple arithmetic manipulations; there is no need to compute the actual paths. For example, consider the concurrent operation of two machines and assume that an event α_1 is eligible in the first machine and an event α_2 is eligible in the second machine. Now presume that α_1 occurs in the first machine. The task now is to find the timer values corresponding to the various events. The timer values corresponding to all the events in the first machine can be easily deduced from its timer functions. The timer values corresponding to α_2 may be inferred as follows. Since α_1 could not have occurred before l_{α_1} time units it follows that α_2 has at most $u_{\alpha_2} \dot{-} l_{\alpha_1}$ time units before it is forced to occur. Similarly, since α_1 could not have taken longer than u_{α_1} time units to occur it follows that α_2 can occur no earlier than $l_{\alpha_2} \dot{-} u_{\alpha_1}$ time units. The timer values for the remaining eligible events in the second machine may be computed similarly. The complete algorithm is given in Appendix B.

7.10. Design Examples Using Timed Generators

We now present a couple of examples to illustrate control design with timed generators.

Example 149 *Let us consider a simple factory comprising two machines and a buffer connecting them. The setup is shown in Figure 7.21. Machine* $\mathbf{G}_1$ *takes its input from a*

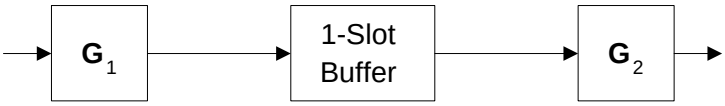


Figure 7.21: A Simple Factory Setup

never-ending source of raw materials and deposits a semi-finished product into a one slot buffer. Machine $\mathbf{G}_2$ *takes its input from the buffer and deposits the final product into an unfillable storage space. The timed generators for* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *are shown in Figure 7.22. Here* α_i *represents the machine* $\mathbf{G}_i$ *starting its operation while* β_i *represents the machine*

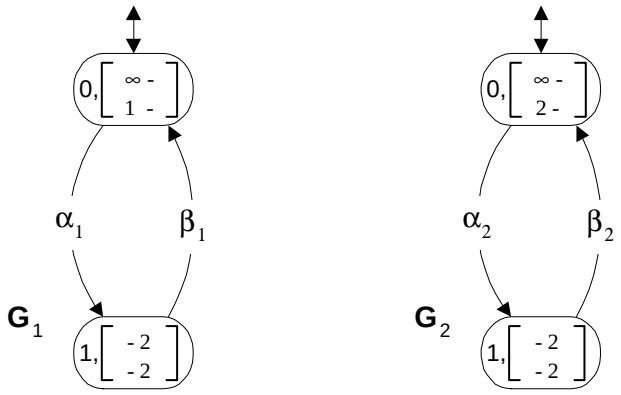


Figure 7.22: Component Machines of the Simple Factory

finishing its operation. Let us assume that α_i *are forcible and initially prohibitible while* β_i *are uncontrollable. It is assumed that* $m_{\alpha_i}^{G_i}(x) = l_{\alpha_i}^{G_i}(x)$ *for* $i = 1, 2$ *and* $x = 0, 1$*. The concurrent operation of the two machines is given by their synchronous composition* $\mathbf{G}$ *partly shown in Figure 7.23. The timed generator* $\mathbf{G}$ *has 16 states and 27 transitions. A TTG representation of this setup has 30 states and 42 transitions. Further note that an increase in the clock frequency would not affect the size of* $\mathbf{G}$*; the timer values would just have to be increased by the same factor.*

Let us impose two specifications on this setup. The first specification is a behavioural specification requiring that the buffer should neither overflow nor underflow. This can be

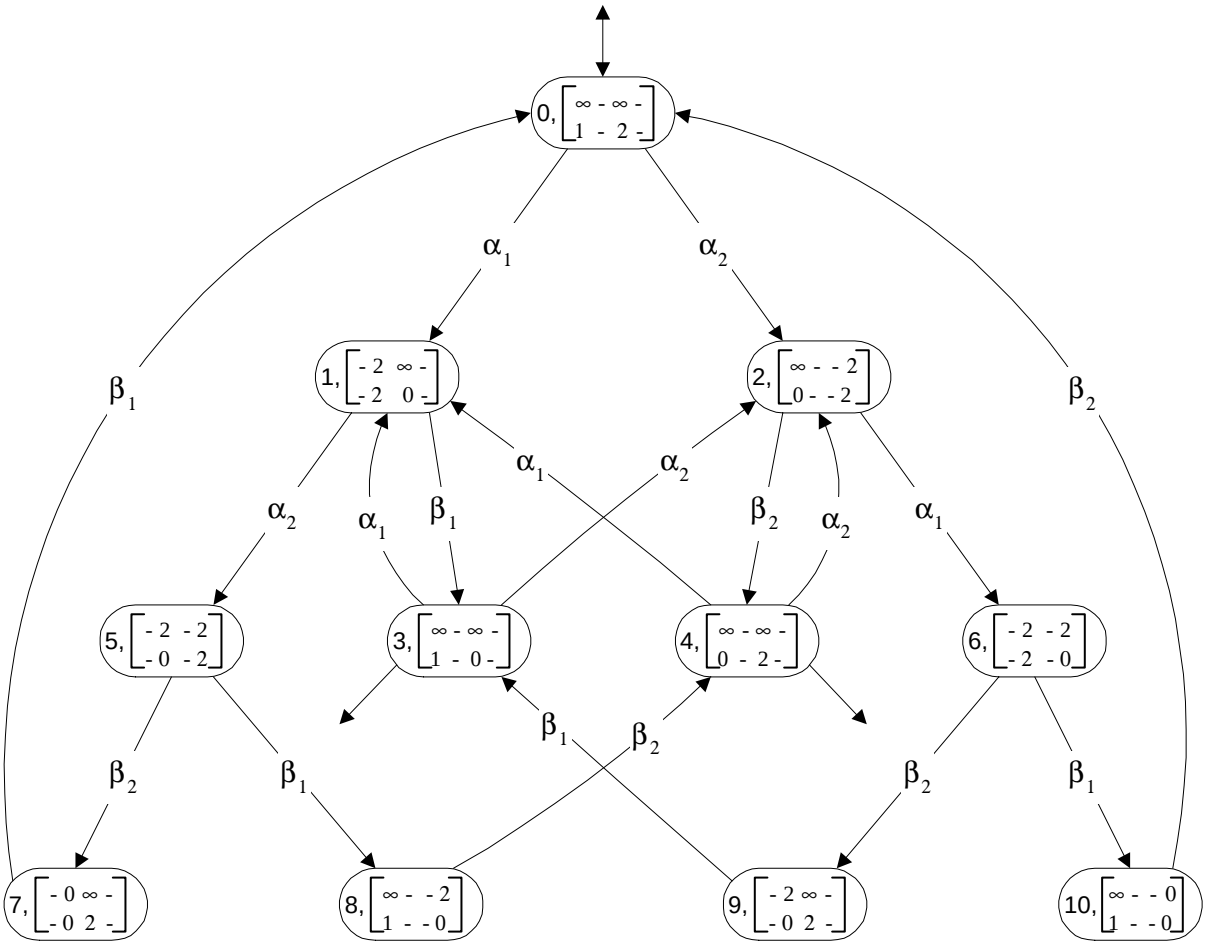


Figure 7.23: Synchronous Composition of $\mathbf{G}_1$ and $\mathbf{G}_2$

represented by the language shown in Figure 7.24. The second specification is a temporal

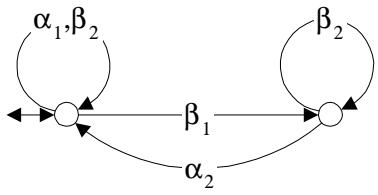


Figure 7.24: Under/Overflow Specfication

specification requiring that the machine $\mathbf{G}_2$ *should start its operation within two time units whenever it can do so safely. The timed generator shown in Figure 7.25 represents the maximal behaviour of the small factory subject to the buffer under/overflow specification. The temporal specification can now be imposed by the timed generator shown in Figure 7.26 because* $m^G_{\alpha_2}(2) = 0$. *This specification simply sets to* 1 *the upper time bound of* α_2. *The overall controlled behaviour of the factory is the same as Figure 7.26. The event* α_2 *is no longer prohibitible if the buffer is nonempty.* □

Example 150 *Here we consider a manufacturing cell presented in [Bra93, page 105]. The manufacturing cell is shown in Figure 7.27 and consists of two machines (*$\mathbf{M}_1$ *and* $\mathbf{M}_2$*), an input conveyor and an output conveyor. The input conveyor provides raw materials from an infinite source to the two machines while the output conveyor deposits the finished product into an infinite sink. Two types of parts,* p_1 *and* p_2 *are processed by both machines. Part* p_1 *takes* 3 *time units to be processed on* $\mathbf{M}_1$ *and* 1 *time unit on* $\mathbf{M}_2$. *Part* p_2 *takes* 2 *time units on* $\mathbf{M}_1$ *and* 4 *time units on* $\mathbf{M}_2$. *The closed loop behaviour of the cell must satisfy the following behavioural and temporal specifications.*

Behavioural Specs:

- *A part can only be processed by one machine at a time,*
- *A* p_1 *part must be processed first by* $\mathbf{M}_1$ *and then by* $\mathbf{M}_2$,

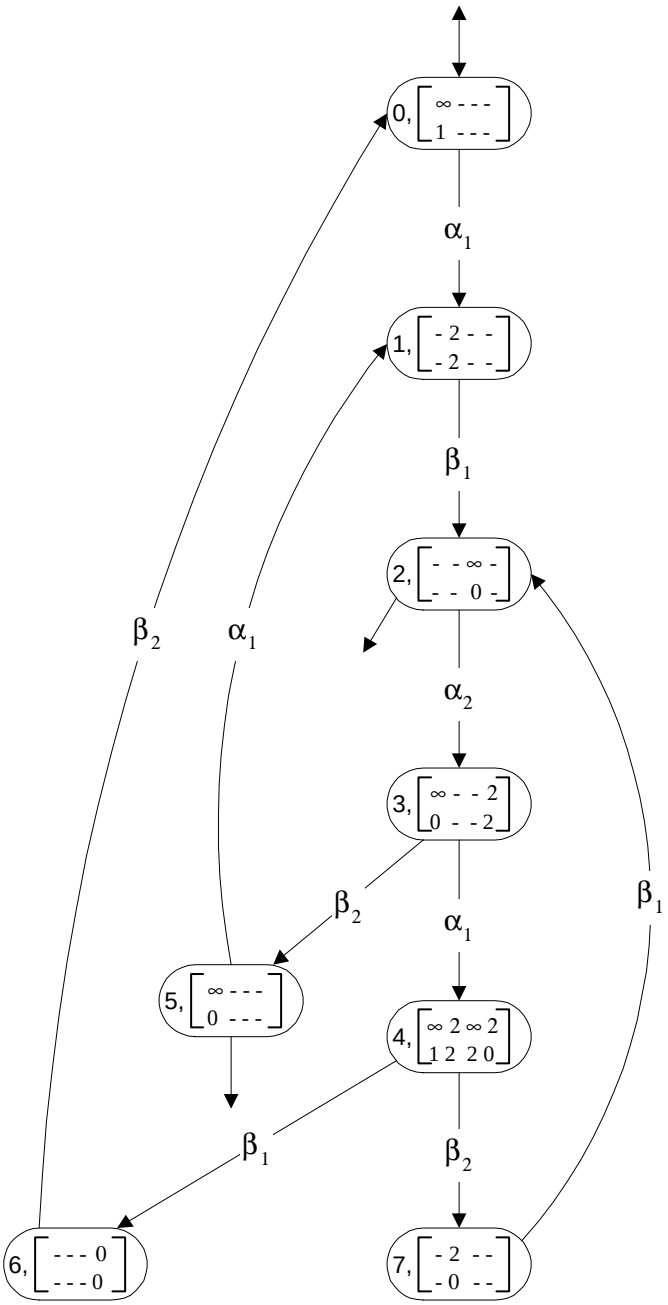


Figure 7.25: Maximal Behaviour of the Small Factory subject to the Under/Overflow Spec

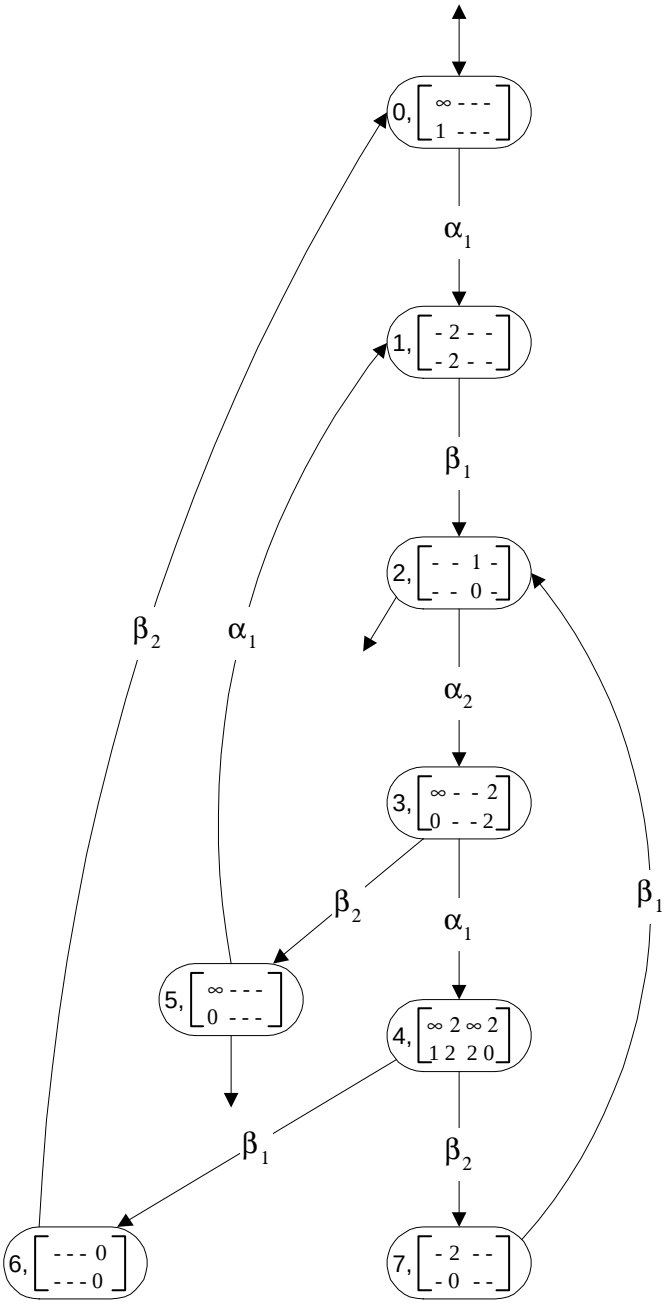


Figure 7.26: Temporal Specification for the Small Factory

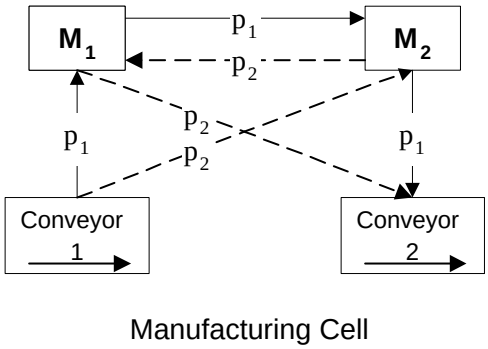


Figure 7.27: A Manufacturing Cell

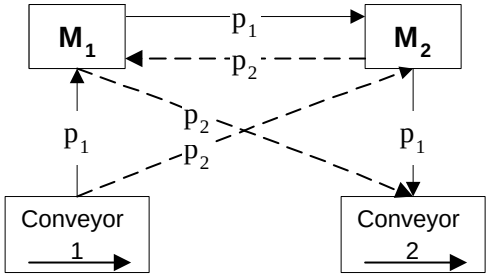
- *A p_2 part must be processed first by $\mathbf{M}_2$ and then by $\mathbf{M}_1$,*
- *One p_1 part and one p_2 part must be processed in each production cycle.*

Temporal Specs:

- *A production cycle must take at most 10 time units to complete,*
- *The production cycle time is to be minimized.*

The timed generators of $\mathbf{M}_1$ and $\mathbf{M}_2$ are shown in Figure 7.28. The events α_{ij} represent machine $\mathbf{M}_i$ starting work on part p_j while the events β_{ij} represent machine $\mathbf{M}_2$ finishing work on part p_j. The lower bounds of β_{ij} are assumed to be equal to 1. It is further assumed that α_{ij} are forcible and prohibitible everywhere while β_{ij} are uncontrollable. It is further assumed that $m_{\alpha_{ij}}^{M_i}$ is the same as $l_{\alpha_{ij}}^{M_i}$ at all the states. The synchronous composition of $\mathbf{M}_1$ and $\mathbf{M}_2$ has 56 states and 121 transitions and represents the open loop behaviour of the manufacturing cell. The TTG representation of the manufacturing cell has 81 states and 121 transitions. So the TTG representation has 44 percent more states than the timed generator representation.

We can impose the behavioural specifications exactly along the lines of [Bra93]. The various behavioural specifications are shown in Figure 7.29. Specifications $\mathbf{S}_1$ and $\mathbf{S}_2$ ensure



Manufacturing Cell

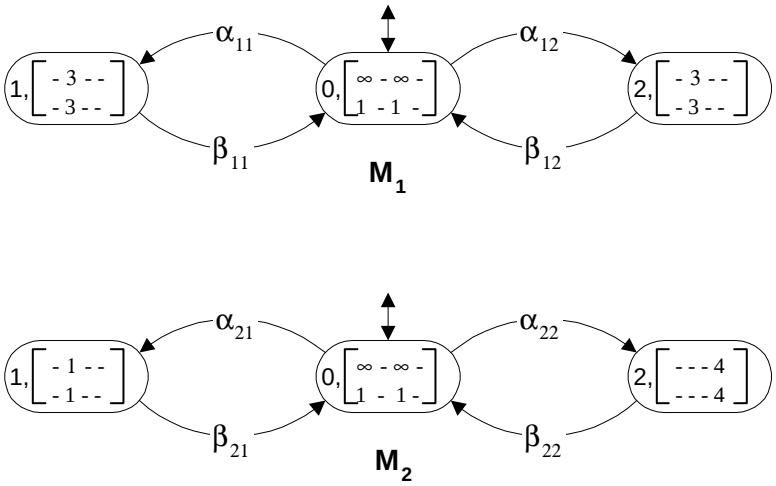


Figure 7.28: Manufacturing Cell: M_1 and M_2

that only one machine processes a part at a time; specifications $\mathbf{S}_3$ *and* $\mathbf{S}_4$ *impose the desired order of processing; and specification* $\mathbf{S}_5$ *ensures that a production cycle includes a* p_1 *as well as a* p_2 *part. The timed generator representing the optimal closed loop behaviour of the manufacturing cell subject to the behavioural specification has* 33 *states and* 42 *transitions. The TTG representation of this closed loop behaviour has* 108 *states and* 144 *transitions.*

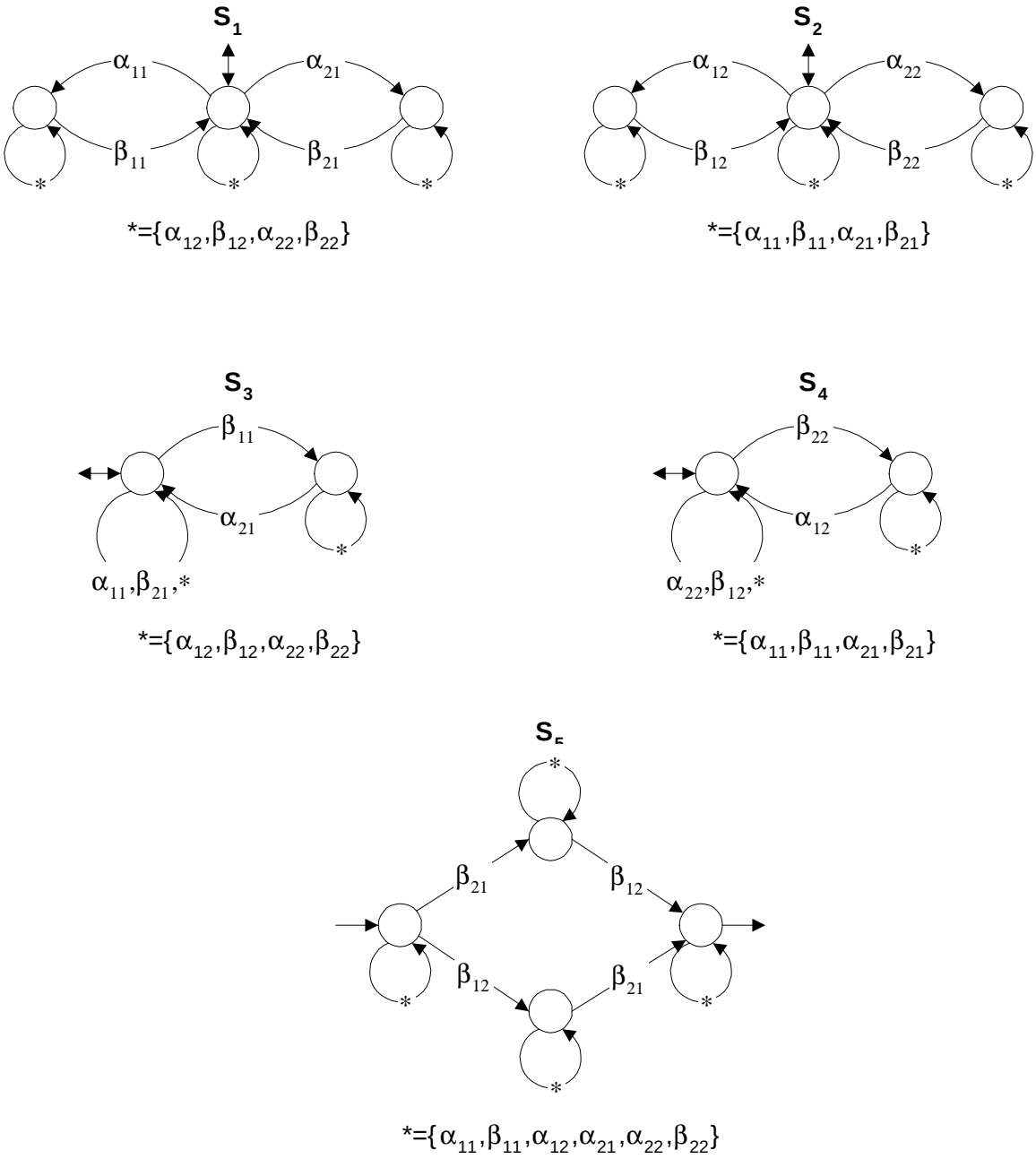


Figure 7.29: Manufacturing Cell: Behavioural Specifications

We now try to find out the minimum time it takes to complete a production cycle over all possible paths. This will allow us to verify that a production cycle takes no longer than 10 *time units as well as find the production cycle path that takes the least time. Since we have no control over the uncontrollable events, we can only use the forcible events to expedite the production cycle. We do this by forcing them to occur as soon as possible, i.e. by setting the upper bound of a forcible event to the least possible value that can be assigned to it. We can now compute the time it may take to finish a production cycle by summing the* upper *bounds of all the events in a production cycle. It turns out that the least time required for a production cycle is* 7 *units. One of the optimal production paths is* $\alpha_{22}\alpha_{11}\beta_{11}\beta_{22}\alpha_{21}\alpha_{12}\beta_{21}\beta_{12}$. □

7.11. Summary

In this chapter we presented a new model for timed discrete event systems, namely timed generators. We showed that timed generators are closed under control. This ensures that further control design may be carried out on a system already under control. Timer graphs scale well because time is modelled implicitly using timer functions. This feature is especially useful when a system has events with widely different time scales as in the case of the simple example of a parking setup where one event takes much longer to occur than the other events. We showed that a timed generator provides a much more economical representation of this setup than a timed transition graph. We showed that the RW notions of controllability and supremal controllable sublanguages carry over to the timed generator framework. We also showed that any system that can be modelled in the BW framework can be modelled using timed generators. Finally, we presented a couple of examples to illustrate the control design process using timed generators.

8. Conclusions

We have presented methods for complexity reduction in discrete event systems (DES). We began by exploring the *state explosion* problem. The size of a DES increases multiplicatively with respect to the size of its component systems. So as the number of components increases, the overall size of a DES can increase very rapidly even if the individual components are relatively small in size. A centralized approach to the supervisory control of such a DES may not be feasible. We presented a modular supervisory synthesis approach for such systems. Our approach is symbolic in nature: it produces a function that can be efficiently evaluated at each state to compute the control action. This is in contrast to the standard practice of producing a lookup table for the supervisory control action. A lookup table for a DES with a large number of states may be impractical to implement so some sort of symbolic methodology may in fact be necessary. The main drawback of our approach is that it does not handle nonblocking. The control action guarantees the satisfaction of a given safety specification but may cause the system to block. However we are able to extend our approach to implement deadlock-avoidance. Nonblockingness can be achieved in a large number of systems as a result of deadlock-freeness but it is not possible to know this apriori. Thus our symbolic supervision scheme is best suited for systems which comprise a large number of subsystems but where nonblockingness is not a big concern. If it is important to ensure nonblockingness then the standard RW approach may be more suitable.

In RW, a supervisor is implemented using an automaton. The supervision is carried out by synchronizing the plant with the supervisor automaton. An event is permitted in the plant only if it is also permitted in the supervisor automaton. Often the supervisor automaton contains a lot more information than is necessary for control. This redundant information often concerns the structure of the plant. Since the plant is synchronized with the supervisor, it may be possible to remove this structural information from the supervisor without affecting the control action. This may make it feasible to reduce the size of the supervisor automaton. We present a heuristic greedy algorithm for supervisor reduction. The algorithm constructs a new supervisor that provides the same control action as the original supervisor. This is done by constructing an appropriate cover of the state set of the original supervisor. There is no guarantee that the new supervisor is actually smaller in size than the original supervisor. However the algorithm seems to perform quite well in practice.

Finally we present a new and compact model for timed discrete event systems (TDES), namely, timed generators. A timed generator comprises an automaton and timer functions for each event in the alphabet of the automaton. A timer function for an event defines upper and lower timer bounds on the occurrence of the event at any given state of the automaton. We do not use any special events to model the passage of time; timers are used instead. There are two timers corresponding to each event: one for the lower bound and one for the upper bound. The timer functions are used to initialize these timers. Upon entering a state, the timers are initialized and start counting down to zero. An event is considered eligible to occur if its lower timer has counted down to zero. An event is forced to occur when its upper timer has counted down to zero. We show that the timed generators are closed under control. It is always possible to construct a timed generator that corresponds to another timed generator under supervisory control. This addresses a major drawback

of BW [Bra93],[BW94].

8.1. Limitations and Future Research

The symbolic supervisory scheme presented in this thesis does not handle nonblocking. Nonblocking is an NP-complete problem and it is highly unlikely that an efficient and general method can ever be found to achieve it. This clearly suggests that we should be looking for special, yet not too restrictive, traits in a system that make it amenable to the synthesis of nonblocking supervisory control. All natural complex systems are highly hierarchical in nature and perhaps that is the best way of handling complexity. So a next step in the symbolic supervision scheme could explore the decomposition of a system into different level. We could explore the conditions under which a control function at a high level would be implementable at a lower level. Similarly, we could look for conditions under which nonblockingness at a high level guarantees nonblockingness at a lower level.

The symbolic supervision scheme is presented for the scenario where the plant comprises various components but there is only one specification. If we have multiple specifications then we have to construct their *meet* and use that as the overall specification. It would be nice if the scheme could be extended for multiple specifications. Perhaps this can be done by finding control functions for all the specifications separately and then composing them somehow.

The framework of timed generators can be easily extended for modular decomposition and control. However the case of hierarchical decomposition and control is not so clear. A number of issues need to be resolved. For instance, does the clock have to remain the same across different levels or is it possible to have different clocks for different levels? It is easy to think of scenarios where the clock at a lower level has more resolution (i.e.

runs at a higher frequency) than the clock at a higher level. Since we use timers to model the passage of time, intuitively there should no reason that prevents the usage of different clocks at different levels. After all, there is no special event that models time and needs to be reported to a higher level. But we still need to define timer functions at the higher level. It seems inevitable that the output of these timer functions would depend on the timer functions at the lower level. Thus we need to look carefully at the case of hierarchical control in timed generators.

In Section 7.8 we have presented an algorithm for deriving a timed generator from an activity transition graph. During each pass of the main loop of this algorithm we have to compute the timer matrices corresponding to the prefix minimal and maximal paths. It would be nice if the solutions from a previous step could be used to generate the solutions at the current step. From Section 6.5.1 we know that the prefix minimal and maximal solutions (strings) at one step cannot be used to generate the solutions at the next stage. However it may still be possible to find a relation between their timer matrices.

Finally, it would be nice to extend to timed generators the symbolic supervision scheme of Chapter 3. This would require defining appropriate *distribution* and *join* operations over timed generators.

A. Some Proofs

A.1. Proof of Theorem 104

We first state and prove a result that will be useful in proving Theorem 104.

Lemma 151 *For* $2 \leq k \leq n$,

$$i_1 + \sum_{k=2}^{n} i'_k = K_1 + K_2 i_1 \tag{A.1}$$

where $K_1 \in \mathbf{N}$ *is a term which depends only on* l_{α_k}, *and* $K_2 \in \{0, 1\}$.

Proof. We prove this result by induction on n.

Base Step: 1. Let $n = 1$. Then (A.1) holds with $K_1 = 0$ and $K_2 = 1$.

2. Let $n = 2$. Using (6.3) we know that i'_2 is equal to either l_{α_2} or $l_{\alpha_2} - i_1$. Thus (A.1) holds with either $K_1 = l_{\alpha_2}$ and $K_2 = 1$ or with $K_1 = l_{\alpha_2}$ and $K_2 = 0$.

Assumptive Step: Let us assume that (A.1) holds for all values less than or equal to $n - 1$, i.e.

$$i_1 + \sum_{k=2}^{j} i'_k = K_1^j + K_2^j i_1, \tag{A.2}$$

where $K_1^j \in \mathbf{N}$ and $K_2^j \in \{0, 1\}$ for $2 \leq j \leq n - 1$.

Inductive Step: Assuming that (A.2) holds for $2 \leq j \leq n-1$, we need to show that (A.1) holds. From (6.3) we know that

$$i'_n = \begin{cases} l_{\alpha_n} & \text{if } (\alpha_1 \cdots \alpha_{n-2}\alpha_n \notin A) \\ & \vee (\alpha_n = \alpha_{n-1}) \\ l_{\alpha_n} \dot{-} i'_{n-1} & \text{if } \left(\begin{array}{c} (\alpha_1 \cdots \alpha_{n-3}\alpha_n \notin A) \\ \vee (\alpha_n = \alpha_{n-2}) \end{array} \right) \\ & \wedge (\alpha_1 \cdots \alpha_{n-2}\alpha_n \in A) \\ \vdots & \\ l_{\alpha_n} \dot{-} \left(\sum_{k=2}^{n-1} i'_k + i_1\right) & \text{if } (\alpha_n \in L_{act}) \wedge (\alpha_1\alpha_n \in A) \\ & \wedge \ldots \wedge (\alpha_1 \cdots \alpha_{n-2}\alpha_n \in A)\,. \end{cases}$$

If $i'_n = 0$ then (A.1) reduces to (A.2) which is assumed to be true. Therefore we only need to concern ourselves with the case when $i'_n \neq 0$. In such a case, the assymetric subtraction operator in the above equation can be replaced by the standard subtraction operator. So adding $i_1 + \sum_{k=2}^{n-1} i'_k$ to both sides of the above equation and using (A.2) we get

$$i_1 + \sum_{k=2}^{n} i'_k = \begin{cases} l_{\alpha_n} + i_1 + \sum_{k=2}^{n-1} i'_k & \text{if } (\alpha_1 \cdots \alpha_{n-2}\alpha_n \notin A) \\ & \vee (\alpha_n = \alpha_{n-1}) \\ l_{\alpha_n} - i'_{n-1} & \text{if } \left(\begin{array}{c} (\alpha_1 \cdots \alpha_{n-3}\alpha_n \notin A) \\ \vee (\alpha_n = \alpha_{n-2}) \end{array} \right) \\ +i_1 + \sum_{k=2}^{n-1} i'_k & \wedge (\alpha_1 \cdots \alpha_{n-2}\alpha_n \in A) \\ \vdots & \\ l_{\alpha_n} - \left(\sum_{k=2}^{n-1} i'_k + i_1\right) & \text{if } (\alpha_n \in L_{act}) \wedge (\alpha_1\alpha_n \in A) \\ +i_1 + \sum_{k=2}^{n-1} i'_k & \wedge \ldots \wedge (\alpha_1 \cdots \alpha_{n-2}\alpha_n \in A) \end{cases}$$

$$= \begin{cases} l_{\alpha_n} + i_1 + \sum_{k=2}^{n-1} i'_k & \text{if } (\alpha_1 \cdots \alpha_{n-2}\alpha_n \notin A) \\ & \quad \vee (\alpha_n = \alpha_{n-1}) \\ l_{\alpha_n} + i_1 + \sum_{k=2}^{n-2} i'_k & \text{if } \left(\begin{array}{c} (\alpha_1 \cdots \alpha_{n-3}\alpha_n \notin A) \\ \vee (\alpha_n = \alpha_{n-2}) \end{array} \right) \\ & \quad \wedge (\alpha_1 \cdots \alpha_{n-2}\alpha_n \in A) \\ \vdots & \\ l_{\alpha_n} & \text{if } (\alpha_n \in L_{act}) \wedge (\alpha_1\alpha_n \in A) \\ & \quad \wedge \ldots \wedge (\alpha_1 \cdots \alpha_{n-2}\alpha_n \in A) \end{cases}$$

$$= \begin{cases} \left(l_{\alpha_n} + K_1^{n-1}\right) + K_2^{n-1} i_1 & \text{if } (\alpha_1 \cdots \alpha_{n-2}\alpha_n \notin A) \\ & \quad \vee (\alpha_n = \alpha_{n-1}) \\ \left(l_{\alpha_n} + K_1^{n-2}\right) + K_2^{n-2} i_1 & \text{if } \left(\begin{array}{c} (\alpha_1 \cdots \alpha_{n-3}\alpha_n \notin A) \\ \vee (\alpha_n = \alpha_{n-2}) \end{array} \right) \\ & \quad \wedge (\alpha_1 \cdots \alpha_{n-2}\alpha_n \in A) \\ \vdots & \\ l_{\alpha_n} & \text{if } (\alpha_n \in A) \wedge (\alpha_1\alpha_n \in A) \\ & \quad \wedge \ldots \wedge (\alpha_1 \cdots \alpha_{n-2}\alpha_n \in A) \end{cases}.$$

It is clear from the above equation that we can find $K_1 \in \mathbf{N}$ and $K_2 \in \{0, 1\}$ to satisfy (A.1) irrespective of the value taken by i'_n. ■

We are now ready to prove Theorem 104. Assume that the solution to (6.5) is given by an integer x and the solution to (6.6) is given by some integer y. Clearly, we must have $y \geq x$ because x is the length of the shortest path. So all we need to do is show that $y \leq x$; we use induction on n to do that.

Base Step: Let $n = 1$. Then from (6.4) we know that $y = l_{\alpha_1} = i_1^*$. Since l_{α_1} is the lower

bound on α_1, we must have $l_{\alpha_1} \leq x$.

Assumptive Step: Let us assume that the result holds for $n-1$, i.e.

$$\begin{aligned} f_{n-1}() &= \min_{\{i_1 | t^{i_1}\alpha_1 \in M_1[]\}} (i_1 + f_{n-1}(i_1)) \\ &= \sum_{k=1}^{n-1} i_k^*. \end{aligned} \tag{A.3}$$

Inductive Step: Given that (A.3) holds, we now need to show that the result holds for n, i.e.

$$f_n() = \sum_{k=1}^{n} i_k^*.$$

From (6.5), we know that

$$f_n() = \min_{\{i_1 | t^{i_1}\alpha_1 \in M_1[]\}} (i_1 + f_n(i_1)).$$

In the above equation, $f_n(i_1)$ represents the number of *ticks* in the shortest string among strings of the form $t^{i_2}\alpha_2 \cdots t^{i_n}\alpha_n$ with the restriction that these strings must be suffixes of the string $t^{i_1}\alpha_1$. Consider the sublanguage of L that would be generated if the initial state of **G** were the state corresponding to $t^{i_1}\alpha_1$. In this sublanguage, we can use (A.3) to find the number of *ticks* in the shortest strings among strings of the form $t^{i_2}\alpha_2 \cdots t^{i_n}\alpha_n$. Then, adjusting for the subscripts, we have

$$f_n(i_1) = \sum_{k=2}^{n} i_k'.$$

We can now rewrite (6.5) as

$$\begin{aligned} f_n() &= \min_{\left\{i_1 | t^{i_1}\alpha_1 \in M_1[]\right\}} (i_1 + f_n(i_1)) \\ &= \min_{\left\{i_1 | t^{i_1}\alpha_1 \in M_1[]\right\}} \left(i_1 + \sum_{k=2}^{n} i'_k\right) \\ &= \min_{\left\{i_1 | t^{i_1}\alpha_1 \in M_1[]\right\}} (i_1 + i'_2 + \cdots + i'_n). \end{aligned} \tag{A.4}$$

Using Lemma 151 we can rewrite (A.4) as

$$\begin{aligned} f_n() &= \min_{\left\{i_1 | t^{i_1}\alpha_1 \in M_1[]\right\}} (i_1 + i'_2 + \cdots + i'_n) \\ &= \min_{\left\{i_1 | t^{i_1}\alpha_1 \in M_1[]\right\}} (K_1 + K_2 i_1) \end{aligned} \tag{A.5}$$

where $K_1 \in \mathbf{N}$ and $K_2 \in \{0, 1\}$. Since K_1 is independent of i_1, the minimization problem is now trivial: we can solve (A.5) by assigning the smallest possible value to i_1. However, we know that $i_1 \geq l_{\alpha_1}$ so $i_1 = l_{\alpha_1} = i_1^*$ is the solution to (A.5). Therefore

$$\begin{aligned} f_n() &= \min_{\left\{i_1 | t^{i_1}\alpha_1 \in M_1[]\right\}} (i_1 + i'_2 + \cdots + i'_n) \\ &= i_1^* + i_2^* + \cdots + i_n^* \\ &= \sum_{k=1}^{n} i_k^*. \end{aligned}$$

■

A.2. Proof of Theorem 105

We use induction on n to show that (6.8) is a solution to (6.7).

Base Step: If $n = 1$ then $j_1^* = l_{\alpha_1}$ which is the length of the shortest possible path.

Assumptive Step: Let us assume that the result holds for $n-1$, i.e.

$$\begin{aligned} f'_{n-1}() &= \min_{\{j_n | t^{j_n}\alpha_n \in M'_n[]\}} (j_n + f'_2(j_n)) \\ &= \sum_{k=2}^{n} j_k^*. \end{aligned} \tag{A.6}$$

Inductive Step: Now assuming that (A.6) holds, we need to show that (6.8) is the solution to (6.7). Let $t^{i_1^*}\alpha_1 \cdots t^{i_n^*}\alpha_n$ be the (postfix) piecewise shortest path; from Theorem 104 we know that it must be also be an overall shortest path. Let $t^{j'_2}\alpha_2 \cdots t^{j'_n}\alpha_n$ be the (postfix) piecewise shortest path starting at $t^{j_1^*}\alpha_1$. Again, from Theorem 104 we know that it must be a shortest path starting at $t^{j_1^*}\alpha_1$.So (A.6) implies that

$$j'_2 + \cdots + j'_n = j_2^* + \cdots + j_n^*.$$

Also, from Lemma 151, we have

$$j_1^* + j'_2 + \cdots + j'_n = K_1 + K_2 j_1^*,$$

where $K_2 \in \{0, 1\}$ and K_1 does not depend on j_1^*. This gives

$$j'_2 + \cdots + j'_n = K_1 + (K_2 - 1)\, j_1^*$$

Case 1: $\mathbf{K_2 = 0}$. In this case, we have

$$j_1^* + j_2^* + \cdots + j_n^* = j_1^* + j'_2 + \cdots + j'_n$$

$$\begin{aligned} &= j_1^* + K_1 - j_1^* \\ &= K_1. \end{aligned}$$

Similarly, we must have

$$\begin{aligned} i_1^* + i_2^* + \cdots + i_n^* &= i_1^* + K_1 - i_i^* \\ &= K_1 \\ &= j_1^* + j_2^* + \cdots + j_n^* \end{aligned}$$

which implies that $\sum_{k=1}^{n} j_k^*$ must be the length of a shortest path since $\sum_{k=1}^{n} i_k^*$ is the length of a shortest path.

Case 2: $\mathbf{K}_2$= 1. In this case, we have

$$\begin{aligned} j_2^* + \cdots + j_n^* &= j_2' + \cdots + j_n' \\ &= K_1 \\ &= i_2^* + \cdots + i_n^*. \end{aligned} \tag{A.7}$$

Thus we must have $t^{j_1^*}\alpha_1 t^{i_2^*}\alpha_2 \cdots t^{i_n^*}\alpha_n \in L$ as shown below. Let $k \in \{2, \ldots, n\}$ be the first index such that $j_k' \neq i_k^*$. From (6.3) we know that it is only possible if α_k has been enabled since ϵ; in that case we must have

$$i_k^* - j_k' = j_1^* - i_1^*.$$

If $t^{j_1^*}\alpha_1 t^{i_2^*}\alpha_2 \cdots t^{i_k^*}\alpha_k \notin L$ we must have

$$\begin{aligned} u_{\alpha_k} - l_{\alpha_k} &< i_k^* - j_k' \\ &= j_1^* - i_1^*. \end{aligned} \tag{A.8}$$

We also must have

$$j_1^* + j_2^* + \cdots + j_k^* \leq u_{\alpha_k}$$

because α_k has been enabled since ϵ. Therefore, using (A.8), we get

$$\begin{aligned} j_2^* + \cdots + j_k^* &\leq u_{\alpha_k} - j_1^* \\ &< (l_{\alpha_k} + j_1^* - i_1^*) - j_1^* \\ &= l_{\alpha_k} - i_1^* \\ &= i_2^* + \cdots + i_k^*. \end{aligned}$$

Since $j_2^* + \cdots + j_n^* = i_2^* + \cdots + i_n^*$ we must have $j_{k+1}^* + \cdots + j_n^* > i_{k+1}^* + \cdots + i_n^*$ which is in contradiction to our assumption that $j_{k+1}^* + \cdots + j_n^*$ is the length of a shortest path between α_k and α_n. Thus $t^{j_1^*}\alpha_1 t^{i_2^*}\alpha_2 \cdots t^{i_k^*}\alpha_k \in L$. The whole argument can now be repeated by taking $t^{j_1^*}\alpha_1 t^{i_2^*}\alpha_2 \cdots t^{i_k^*}\alpha_k$ as the starting point and so on to conclude that $t^{j_1^*}\alpha_1 t^{i_2^*}\alpha_2 \cdots t^{i_n^*}\alpha_n \in L$. Now, in turn, we must have that $t^{i_1^*}\alpha_1 t^{j_2^*}\alpha_2 \cdots t^{j_n^*}\alpha_n \in L$ as shown below. Assume that α_l, $l \in \{2, \ldots, n\}$, was last enabled after the occurrence of α_h, $h \in \{0, \cdots, n-1\}$, $h < l$, with the convention that $\alpha_0 = \epsilon$. Then, we must have $u_{\alpha_l} \geq \max(i_{h+1}^* + \cdots + i_l^*, j_{h+1}^* + \cdots + j_l)$. This, in conjunction with the fact that $i_1^* \leq j_1^*$, implies that $t^{i_1^*}\alpha_1 t^{j_2^*}\alpha_2 \cdots t^{j_n^*}\alpha_n \in L$. Since i_1^* is the shortest possible delay before the

occurrence of α_1, and $t^{i_1^*}\alpha_1 t^{j_2^*}\alpha_2 \cdots t^{j_n^*}\alpha_n \in L$, we must have $j_1^* = i_1^*$. This, along with (A.7), implies

$$\begin{aligned} j_1^* + j_2^* + \cdots + j_n^* &= i_1^* + j_2^* + \cdots + j_n^* \\ &= i_1^* + i_2^* + \cdots + i_n^* \end{aligned}$$

which is the desired result ■

B. Synchronous Composition of Timed Generators

proc *sync*

input: $\mathbf{G}_1, \mathbf{G}_2$

output: $\mathbf{G}$: the synchronous composition of $\mathbf{G}_1$ and $\mathbf{G}_2$

begin

$\mathbf{H}$:= synchronous product of the automata $\mathbf{U}^{G_1}$ and $\mathbf{U}^{G_2}$;

$counter = 0$;

$E := \emptyset$;

$unprocessed := \{0\}$;

$X := \{0\}$;

$stateInH[0] := x_0^H$;

for $\sigma \in \Sigma$ **do**

if $\sigma \in \Sigma^{G_1}$ **then**

$f_\sigma^G[0] := (u_\sigma^{G_1}(x_0^{G_1}), l_\sigma^{G_1}(x_0^{G_1}))$;

else

$f_\sigma^G[0] := (u_\sigma^{G_2}(x_0^{G_2}), l_\sigma^{G_2}(x_0^{G_2}))$;

endif

if $\sigma \in \Sigma_f^{G_1} \cap \Sigma_f^{G_2}$ **then**

$m_\sigma^G[0] := max(m_\sigma^{G_1}(x_0^{G_1}), m_\sigma^{G_2}(x_0^{G_2}))$;

endif

endfor

while $unprocessed \neq \emptyset$ **do**

let $source$ be an element of $unprocessed$

$unprocessed := unprocessed - \{source\}$;

$sourceH := stateInH[source]$;

$x_1 :=$ first component of $sourceH$;

$x_2 :=$ second component of $sourceH$;

for all $\sigma \in Elig(\mathbf{H}, sourceH)$ **do**

$targetH := \eta^H(sourceH, \sigma)$;

$y_1 :=$ first component of $targetH$;

$y_2 :=$ second component of $targetH$;

if $\sigma \in \Sigma^{G_1}$ **then**

for all $\alpha \in \Sigma$ **do**

if $\alpha \in \Sigma^{G_1} - \Sigma^{G_2}$ **then**

$U_\alpha := u_\alpha^{G_1}(y_1)$;

$L_\alpha := l_\alpha^{G_1}(y_1)$;

if $\alpha \in \Sigma_f^{G_1}$ **then**

$M_\alpha := m_\alpha^{G_1}(y_1)$;

endif

elseif $\alpha \in \Sigma^{G_1} \cap \Sigma^{G_2}$ **then**

$U_\alpha := \min(u_\alpha^{G_1}(y_1), u_\alpha^{G_2}(y_2))$;

$L_\alpha := \max(l_\alpha^{G_1}(y_1), l_\alpha^{G_2}(y_2))$;

if $\alpha \in \Sigma_f^{G_1} \cap \Sigma_f^{G_2}$ **then**

$M_\alpha := max(m_\alpha^{G_1}(y_1), m_\alpha^{G_2}(y_2));$

endif

else

$U_\alpha := u_\alpha^G(source) \dot{-} l_\alpha^{G_1}(x_1);$

$L_\alpha := l_\alpha^G(source) \dot{-} u_\alpha^{G_1}(x_1);$

if $\alpha \in \Sigma_f^{G_2}$ **then**

$M_\alpha := m_\alpha^G(source) \dot{-} l_\sigma^{G_1}(x_1);$

endif

endif

endfor

else

for all $\alpha \in \Sigma$ **do**

if $\alpha \in \Sigma^{G_2} - \Sigma^{G_1}$ **then**

$U_\alpha := u_\alpha^{G_2}(y_2);$

$L_\alpha := l_\alpha^{G_2}(y_2);$

if $\alpha \in \Sigma_f^{G_2}$ **then**

$M_\alpha := m_\alpha^{G_2}(y_2);$

endif

elseif $\alpha \in \Sigma^{G_1} \cap \Sigma^{G_2}$ **then**

$U_\alpha := \min(u_\alpha^{G_1}(y_1), u_\alpha^{G_2}(y_2));$

$L_\alpha := \max(l_\alpha^{G_1}(y_1), l_\alpha^{G_2}(y_2));$

if $\alpha \in \Sigma_f^{G_1} \cap \Sigma_f^{G_2}$ **then**

$M_\alpha := max(m_\alpha^{G_1}(y_1), m_\alpha^{G_2}(y_2));$

endif

else

$U_\alpha := u_\alpha^G(source) \dot{-} l_\alpha^{G_2}(x_2)$;

$L_\alpha := l_\alpha^G(source) \dot{-} u_\alpha^{G_2}(x_2)$;

if $\alpha \in \Sigma_f^{G_1}$ **then**

$M_\alpha := m_\alpha^G(source) \dot{-} l_\sigma^{G_2}(x_2)$;

endif

endif

endfor

endif

if $(\nexists x \in X)[stateInH[x] = targetH \wedge (\forall \alpha \in \Sigma)(f_\alpha[x] = (U_\alpha, L_\alpha) \wedge m_\alpha[x] = M_\alpha)]$ **then**

$counter := counter + 1$;

$target := counter$;

$stateInH[target] := targetH$;

$X := X \cup \{target\}$;

$unprocessed := unprocessed \cup \{target\}$;

for all $\alpha \in \Sigma$ **do**

$f_\alpha[target] := (U_\alpha, L_\alpha)$;

if $\alpha \in \Sigma_f^{G_1} \cap \Sigma_f^{G_2}$ **then**

$m_\alpha[target] := M_\alpha$;

endif

endfor

else

let $target := x$ s.t. $stateInH[x] = target$ and $(\forall \alpha \in \Sigma)(f_\alpha[x] = (U_\alpha, L_\alpha))$;

endif

$E := E \cup \{(source, \sigma, target)\}$;

endfor

endwhile

$X_m := X \cap \{x : stateInH[x] \in \mathbf{H}_m\}$;

$\mathfrak{F} := \{f_\sigma : \sigma \in \Sigma\}$;

$\mathbf{G} := (X, \Sigma^{G_1} \cup \Sigma^{G_2}, E, 0, X_m, \mathfrak{F})$;

end

Bibliography

[AD92] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1992.

[AHK97] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In 38^{th} *IEEE Symposium on Foundations of Computer Science*, pages 100–109, 1997.

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

[AMP95] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid System II*, volume 999 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1995.

[ASK77] T. Araki, Y. Sugiyama, and T. Kasami. Complexity of the deadlock avoidance problem. In 2^{nd} *IBM Symposium on Mathematical Foundations in Computer Science*, pages 229–257, Tokyo, Japan, 1977.

[AW99] Sherif Abdelwahed and W. M. Wonham. Interacting discrete event systems. In *Proceedings of Thirty-Seventh Annual Allerton Conference on Communication, Control and Computing*, pages 85–92, Allerton, IL, September 1999.

[BCM92] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[Bel61] Richard Bellman. *Adaptive Control Processes: a Guided Tour*. Princeton University Press, Princeton, NJ, 1961.

[Ber87] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice Hall, New Jersey, 1987.

[BH88] Y. Brave and M. Heymann. Formulation and control of real time discrete-event systems. In *Proceedings of the* 27^{th} *Conference on Decision and Control*, pages 1131–1132, Austin, December 1988.

[Bra93] Bertil A. Brandin. *Real-Time Supervisory Control of Automated Manufacturing Systems*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, January 1993.

[Bra97] Bertil A. Brandin. The modelling and supervisory control of timed discrete event systems. Siemens internal report, October 1997. Advance copy sent to Prof. W.M. Wonham.

[Bra98] Bertil A. Brandin. The modelling and supervisory control of timed discrete event systems. In *Proceedings of the International Workshop on Discrete Event Systems*, pages 8–13. IEE, August 1998.

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

[BW94] Bertil A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39(2):329–342, 1994.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the International Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CGP01] E.M. Clarke, Jr., O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2001.

[Che96] Shu-Lin Chen. *Control of Discrete-Event Systems of Vector and Mixed Structural Type*. PhD thesis, University of Toronto, Department of Electrical and Computer Engineering, September 1996.

[Chv79] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, August 1979.

[CLL92a] Sheng-Luen Chung, Stéphane Lafortune, and Feng Lin. Limited lookahead policies in supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, 37(12):1921–1935, December 1992.

[CLL92b] Sheng-Luen Chung, Stéphane Lafortune, and Feng Lin. Supervisory control with variable lookahead policies: Illustrative example. In S. Balemi, P. Kozák, and R. Smedinga, editors, *Proceedings of the Workshop on Discrete Event Systems*, pages 207–214. Birkhauser Verlag, 1992.

[Dre65] Stuart E. Dreyfus. *Dynamic Programming and the Calculus of Variations*. Academic Press, New York, 1965.

[EH86] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisted: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[EH91] F.S. Etessami and G.S. Hura. Rule based design methodology for solving control problems. *IEEE Transactions on Software Engineering*, 17:274–282, 1991.

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

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the* 24^{th} *ACM SIGPLAN-SIGACT symposium on principles of programming languages*, Paris, France, 1997. ACM Press.

[Gol78] E. M. Gold. Deadlock prediction: Easy and difficult cases. *SIAM Journal on Computing*, 7:320–336, 1978.

[Gun97] Johan Gunnarsson. *Symbolic Methods and Tools for Discrete Event Dynamic Systems*. PhD thesis, Linköping University, Linköping, Sweden, 1997. Also appears as Dissertation No. 477 in the Linköping Studies in Science and Technology.

[GW00] Peyman Gohari and W. M. Wonham. On the complexity of supervisory control design in the RW framework. *IEEE Transactions on Systems, Man and Cybernetics*, 30(5):643–652, October 2000. Special Issue on Discrete Systems and Control.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[Har97] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1997.

[Ho89] Yu-Chi Ho. Dynamics of discrete event systems. In *Proceedings of the IEEE*, January 1989.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoc82] Dorit S. Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal of Computing*, 11(3):555–556, August 1982.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HPSS87] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceeding of the* 2^{nd} *IEEE Symposium on Logic in Computer Science*, pages 54–64, Ithaca, NY, June 22-24 1987.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Don Mills, Ontario, 1979.

[HWT92a] G. Hoffman and H. Wong-Toi. Symbolic supervisory synthesis for the animal maze. In S. Balemi, P. Kozák, and R. Smedinga, editors, *Proceedings of the Workshop on Discrete Event Systems*, pages 189–197. Birkhauser Verlag, 1992.

[HWT92b] G. Hoffman and H. Wong-Toi. Symbolic synthesis of supervisory controllers. In *Proceedings of the 1992 American Control Conference*, pages 2789–2793, 1992.

[Joh74] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[Law97] Mark Lawford. *Model Reduction of Discrete Real-Time Systems*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, January 1997.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[LW88] Y. Li and W.M. Wonham. On supervisory control of real-time discrete event systems. *Information Sciences*, 44:199–224, 1988.

[Ma02] Chuan Ma. *An Architectural Approach to the Supervisory Control of DES*. PhD thesis, University of Toronto, Department of Electrical and Computer Engineering, 2002. In preparation.

[McM92] K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1992.

[Mol86] P. Moller. Introducing real time in the algebraic theory of finite automata. Working Paper WP-86-49, International Institute for Applied System Analysis, September 1986.

[MP96] Zohar Manna and Amir Pnueli. Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Stanford, 1996.

[MW99] Rajinderjeet Minhas and W. M. Wonham. Modelling of timed discrete event systems. In *Proceedings of Thirty-Seventh Annual Allerton Conference on Communication, Control and Computing*, pages 75–84, Allerton, IL, September 1999.

[Ost89] J.S. Ostroff. *Temporal logic for real-time systems*. Research Studies Press Ltd., 1989.

[Ost90] J.S. Ostroff. Deciding properties of timed transition models. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):170–183, April 1990.

[OW90] 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.

[O'Y91] S.D. O'Young. On the synthesis of supervisors for timed discrete event processes. Technical Report 9107, Systems Control Group, Department of Electrical and Computer Engineering, Univesity of Toronto, August 1991.

[Pet81] J. L. Peterson. *Operating System Concepts*. Addison-Wesley, Reading, MA, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the* 18^{th} *International Symposium on Foundation of Computer Science*, pages 348–363. IEEE Computer Society Press, 1977.

[Pu00] K.Q. Pu. Modeling and control of discrete-event systems with hierarchical abstraction. Master's thesis, University of Toronto, Department of Electrical and Computer Engineering, March 2000.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[RF96] Spiridon A. Reveliotis and Placid M. Ferreira. Deadlock avoidance policies for automated manufacturing cells. *IEEE Transactions on Robotics and Automation*, 12(6):845–857, 1996.

[Rud88] Karen Gail Rudie. Software for the control of discrete event systems. Master's thesis, Department of Electrical Engineering, University of Toronto, Toronto, Canada, September 1988.

[Rus95] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, March 1995.

[Rut] J.J.M.M. Rutten. Universal coalgebra: A theory of systems. Personal Communication.

[RW82] P.J. Ramadge and W. M. Wonham. Supervision of discrete event systems. In *Proceedings of IEEE, Special Issue on Discrete Event Dynamic Systems*, volume 37, pages 1692–1708. IEEE, 1982.

[RW87a] P.J. Ramadge and W.M. Wonham. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 1987.

[RW87b] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event systems. *SIAM Journal of Control and Optimization*, 25(1):206–230, January 1987.

[RW89] P.J. Ramadge and W.M. Wonham. The control of discrete-event systems. *IEEE Proceedings*, 77(1):81–98, January 1989.

[RW92] Karen Rudie and W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[Sim96] Herbert A. Simon. *The Sciences of the Artificial*. The MIT Press, Cambridge, Massachusetts, third edition, 1996.

[Str97] Karl Stroetmann. The constrained shortest path problem: A case study in using ASMs. Technical report, Siemens AG, Munich, Germany, 1997.

[SW00] Rong Su and W. M. Wonham. Supervisor reduction for discrete-event systems. In *Proceedings of the Conference on Information Sciences and Systems*, Princeton, March 2000.

[SW01] Rong Su and W. M. Wonham. Supervisor reduction for discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 2001. submitted.

[VW86] A. F. Vaz and W. M. Wonham. On supervisor reduction in discrete-event systems. *International Journal of Control*, 44(2):475–491, 1986.

[Won01] W. M. Wonham. Notes on control of discrete-event systems, 2001. Course Notes for ECE 1636F/1637S.

[WR88] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete event systems. *Mathematics of Control, Signal & Systems*, 1(1):13–30, 1988.

[WTH91] H. Wong-Toi and G. Hoffmann. The control of dense real-time discrete-event systems. In *Proceedings of the* 30^{th} *Conference on Decision and Control*, pages 1527–1528, Brighton, England, December 1991.

[WW96a] Kai C. Wong and W. M. Wonham. Hierarchical control of discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 6(3):241–273, July 1996.

[WW96b] 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.

[WW98] Kai C. Wong and W. M. Wonham. Modular control and coodination of discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 8(3):247–297, October 1998.

[Zha01] Zhonghua Zhang. Smart TCT: An efficient algorithm for supervisory control design. Master's thesis, University of Toronto, Department of Electrical and Computer Engineering, April 2001.

[Zho92] H. Zhong. *Hierarchical Control of Discrete-Event Systems.* PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, July 1992.

[ZW90] H. Zhong and W. M. Wonham. On the consistency of hierarchical supervision in discrete-event systems. *IEEE Transactions on Automatic Control*, 35(10):1125–1134, October 1990.

Index