Nonblocking Distributed State-Tree-Structures

Master of Science Thesis

Kristian Lyngbæk
Nonblocking Distributed State-Tree-Structures

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Kristian Lyngbæk
born in Mountain View, California, U.S.A.

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
Nonblocking Distributed State-Tree-Structures

Author: Kristian Lyngbæk
Student ID: 4033590
Email: K.Lyngbaek@student.tudelft.nl

Abstract

Discrete-event systems encompass a wide variety of today’s systems including manufacturing cells and networking protocols. To prevent a system from entering a forbidden state and to ensure nonblockingness, it is desirable to control, or restrict, the behavior of such a system. The supervisory control paradigm allows for the synthesis of a supervisor which produces the necessary feedback to appropriately restrict the behavior of the system. When discrete-event systems are described modularly, the number of states grows exponentially with the number of components; this is known as the state-space explosion problem and is one of the central challenges of supervisory control. State-tree-structures and modular approaches are strategies which address issues of complexity in supervisory control. However, both still suffer from heavy computations when ensuring nonblockingness.

In this thesis we introduce a novel procedure for a more efficient check for nonblockingness of state-tree-structures. This is accomplished using the structural properties of state-tree-structures to separate the shared events from the unshared events of a system. We build a recursive two-layer hierarchy in which the bottom level contains the parallel components, and the top level is an abstracted view containing the shared events between these components. To maintain reachability properties in the top level, the components of the system must be clustered such that in each cluster one can reach every outgoing transition from each incoming transition. This is formalized as a universally reachable cluster. We introduce an algorithm which optimally clusters system components in such a manner. This clustering allows us to reformulate the conditions of nonblockingness in a hierarchical manner; the system is nonblocking if and only if both the top level and each component in the bottom level are nonblocking. Given a small top level and that the bottom level has no shared events, the components can be analyzed independently and in parallel, thus drastically reducing computation times.

To verify performance, the universally reachable clustering algorithm was implemented and analyzed for a set of random automata and was found to reduce the number of states on average by 95%. Additionally, our entire procedure to verify nonblockingness was applied to a practical production cell example in which we achieved a 99.99% reduction in the number of states to be examined.
Thesis Committee:

<table>
<thead>
<tr>
<th>Role</th>
<th>Name</th>
<th>Faculty</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsible Professor</td>
<td>Prof. C. Witteveen</td>
<td>EEMCS</td>
<td>TU Delft</td>
</tr>
<tr>
<td>Daily Supervisor</td>
<td>Prof. J.H. van Schuppen</td>
<td>EEMCS</td>
<td>TU Delft</td>
</tr>
<tr>
<td>Committee Member</td>
<td>Dr. Z. Al-Ars</td>
<td>EEMCS</td>
<td>TU Delft</td>
</tr>
</tbody>
</table>
For my Master’s thesis, I was interested in theoretical work in the area of computer engineering. The work of professor Jan H. van Schuppen on control of discrete-event systems nicely couples concepts from computer engineering with control theory, a new and exciting topic for me.

My thesis work began in November 2010 with Professor Jan H. van Schuppen as my daily supervisory. A significant portion of the initial work involved reading and understanding Ma Chuan’s Ph.D. thesis on state-tree-structures. This was the foundation of my work, and from here methods of utilizing state-tree-structures in a distributed and hierarchical setting were investigated. Over the course of my work, I was involved with the European Union project for Distributed Supervisory Control of Complex Plants, INFSO-ICT-224498, (DISC). I attended the DISC project meeting in Amsterdam, held from March 14-16 2011. I also attended the DISC Ph.D. school held from June 6-10 2011 and participated in a poster session to present my work. I also attended a series of lectures at CWI (Center for Mathematics and Computer Science) in Amsterdam and gave a brief presentation of my work there as well. My work concluded in July 2011.
Acknowledgements

I thank my daily supervisor, Professor Jan H. van Schuppen, for his guidance with my thesis along with all the experiences and opportunities he exposed me to.

I would also like to thank my good friend and classmate, Nicola Pambakian, for the countless and lengthy discussions regarding my thesis work, often running late into the night.

Lastly, I thank Professor W.M. Wonham and his Ph.D. students, Ma Chuan and Chao Wujie, for their correspondence and guidance involving material on state-tree-structures.

Kristian Lyngbæk
Delft, the Netherlands
August 15, 2011
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1  Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Automata</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Control of Discrete-Event Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.3 State-Explosion Problem</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Modular Systems</td>
<td>5</td>
</tr>
<tr>
<td>1.5 State-Tree-Structures</td>
<td>5</td>
</tr>
<tr>
<td>1.6 Hierarchical State-Tree-Structure</td>
<td>6</td>
</tr>
<tr>
<td>1.7 Summary</td>
<td>6</td>
</tr>
<tr>
<td>2  Problem Statement</td>
<td>9</td>
</tr>
<tr>
<td>3  Supervisory Control</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Model</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Plant/Supervisor Interaction</td>
<td>11</td>
</tr>
<tr>
<td>3.4 Specification</td>
<td>13</td>
</tr>
<tr>
<td>3.5 Problem Formulation</td>
<td>13</td>
</tr>
<tr>
<td>3.6 Nonblocking Control</td>
<td>13</td>
</tr>
<tr>
<td>3.7 Conclusion</td>
<td>15</td>
</tr>
<tr>
<td>4  State-Tree-Structures (STS)</td>
<td>17</td>
</tr>
<tr>
<td>4.1 Motivation</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Introduction</td>
<td>17</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Closed-Loop Feedback System</td>
<td>3</td>
</tr>
<tr>
<td>4.1</td>
<td>STS Example - 1 Level</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>STS Example - 2 Levels</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>STS Example - 3 Levels</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>STS Example - 4 Levels</td>
<td>20</td>
</tr>
<tr>
<td>4.5</td>
<td>An example of a sub state-tree: How to represent a state in an STS</td>
<td>22</td>
</tr>
<tr>
<td>4.6</td>
<td>This is an example of a holon. This particular holon matches state $x_{121}$ of Figure 4.5</td>
<td>22</td>
</tr>
<tr>
<td>4.7</td>
<td>Local-coupling condition of wellformedness. The shared events must be at the top level (AND-adjacent). Note that event $c$ is shared and is at the top level. Therefore the local-coupling condition holds for this STS.</td>
<td>25</td>
</tr>
<tr>
<td>6.1</td>
<td>Superstate Type Pairings</td>
<td>35</td>
</tr>
<tr>
<td>6.2</td>
<td>An example of the flatten procedure</td>
<td>37</td>
</tr>
<tr>
<td>6.3</td>
<td>An example of the plug-in procedure</td>
<td>37</td>
</tr>
<tr>
<td>7.1</td>
<td>Sample STS</td>
<td>40</td>
</tr>
<tr>
<td>7.2</td>
<td>(Flattened) STS</td>
<td>41</td>
</tr>
<tr>
<td>7.3</td>
<td>Head of STS</td>
<td>42</td>
</tr>
<tr>
<td>7.4</td>
<td>Example of a universally reachable cluster</td>
<td>43</td>
</tr>
<tr>
<td>7.5</td>
<td>Initial second level OR-superstate. (A sub generator.)</td>
<td>45</td>
</tr>
<tr>
<td>7.6</td>
<td>Example of a strongly connected component</td>
<td>46</td>
</tr>
<tr>
<td>7.7</td>
<td>Generator with no Strongly Connected Components</td>
<td>47</td>
</tr>
<tr>
<td>7.8</td>
<td>Example of a transient edge (in bold)</td>
<td>47</td>
</tr>
<tr>
<td>7.9</td>
<td>Generator with no transient edges</td>
<td>49</td>
</tr>
<tr>
<td>7.10</td>
<td>Iteration 1</td>
<td>51</td>
</tr>
<tr>
<td>7.11</td>
<td>Iteration 2</td>
<td>52</td>
</tr>
<tr>
<td>7.12</td>
<td>Iteration 3</td>
<td>53</td>
</tr>
<tr>
<td>7.13</td>
<td>Iteration 4</td>
<td>54</td>
</tr>
<tr>
<td>7.14</td>
<td>Iteration 5</td>
<td>55</td>
</tr>
</tbody>
</table>
List of Figures

7.15 Iteration 6 ................................................................. 56
7.16 Generator maximally clustered into universally reachable components .... 57
7.17 Updated Parallel Composition .......................................... 59
7.18 Updated Abstracted Parallel Composition ............................. 60
7.19 Optimized Computation .................................................. 62

8.1 URC Algorithm Performance on Random 50 Node generators with Regard to
Number of Edge Density ..................................................... 65
8.2 URC Algorithm Performance for Different Generator Sizes .................. 66
8.3 URC Algorithm Performance on Random 50 Node generators with Regard to
Number of Marked States ................................................... 67
8.4 STS of the Production Cell ................................................. 69
8.5 Updated Abstracted Parallel Composition ................................ 70
Chapter 1

Introduction

This chapter introduces some basic concepts and summarizes the work of this thesis. This work covers several domains of computer science, control theory, and algorithms. It is assumed that the reader has a basic understanding of these topics. We begin by introducing the notion of discrete-event systems and automata. The next section describes why and how they are controlled. This is followed by a discussion of modular and structural approaches to the control of discrete-event systems. We then introduce our contribution and conclude with an outline of the remaining chapters.

1.1 Automata

Discrete-event systems (DESs) are systems in which the dynamics are described by the triggering of events rather than the passage of time. In the analysis of DESs, one is primarily concerned with the sequence of visited states and the events which trigger these transitions rather than the time spent in each state. DESs describe many real-life systems or processes including manufacturing plants, network protocols, and database management systems. A thorough introduction to discrete-event systems can be found in [4, 5].

There are various formalisms by which to model a DES; automata, Petri-nets, process algebras, and (max,+)-automata are all common. Automata represent a formal language using a transition structure. In other words, it specifies which events can occur at every state. This formalism allows for certain composition operations which makes it convenient for building complex models modularly. In this discussion we consider automata as the basis of our modeling formalisms. Automata are a state transition structure, where the state set $X$ and the event set $E$ are the two most significant components. An automaton is formally defined below.

**Definition 1.1.1** An Automaton $G$ is defined as the 5-tuple, $G := (X, E, \delta, x_0, X_m)$ where
1.1. Automata Introduction

• $X$ is the state set.
• $E$ is the event set.
• $\delta : X \times E \to X$ is the transition function.
• $x_o$ is the initial state.
• $X_m$ is the set of marked states.

Consider an automata $G$ with a state set $X$ and event set $E$. The initial state, $x_o$, is the state in which $G$ starts. From this state, the system can reach other states by following transitions triggered by certain events. The set of all states which can be reached from $x_o$ is called the reachable set of $G$, denoted by $R(G) \subseteq X$. A marked state is essentially the opposite; it is a state in which one would like to have $G$ finish in. The coreachable set $CR(G) \subseteq X$ is the set of all states from which one can reach a marked state. The trimmed automata of $G$ is a sub automata of $G$ which includes only the states which fall in both the reachable and coreachable sets of $G$, denoted as $\text{trim}(G)$. A given system is nonblocking if from every reachable state one can reach a marked state. If there is a reachable state, or set of states, from which $G$ cannot reach a marked state, $G$ is considered blocking, and this simply means that the system can get “stuck”. Blocking can be categorized into dead-lock and live-lock as described below.

• **Dead-lock** in an automaton occurs when one reaches an unmarked state from which no feasible event can occur. The automata is stuck in a state which is not designated as an end state.

• **Live-lock** in an automaton occurs when one reaches an unmarked connected component with no feasible transition away from this set. The automata is stuck in an infinite loop.

Automata are often thought of in terms of formal languages. Consider again an automata $G$. The set of all possible events which can occur can be considered to be the alphabet of $G$, denoted $E = \mathcal{E}(G)$, since it forms the basis of the possible behavior. The set of all possible sequences of events from $E$ is denoted by $E^*$. A sequence of events from the alphabet can be considered a string, and the set of all strings which adhere to the transitions in $G$ make up the generated language of $G$, denoted $\mathcal{L}(G)$. The marked language is the set of sequences which end in a marked state. It is denoted as $\mathcal{L}_m(G)$. The prefix-closure of a language $L$ is the language which consists of all the prefixes of all the strings in $L$. It is denoted as $\overline{L}$. A language $L$ is considered prefix-closed if $L = \overline{L}$. These terms can be used to formally define nonblockingness in an automaton in terms of languages.

An automaton $G$ is nonblocking if $\mathcal{L}(G) \subseteq \overline{\mathcal{L}_m(G)}$. 

2
1.2 Control of Discrete-Event Systems

Sometimes the behavior of a given DES is considered undesirable and we would therefore like to modify, or control, it. In this discussion, controlling a DES is understood to mean restricting its behavior to a subset of its original behavior. As with traditional control theory, in order to control a system, a controller is introduced to form a closed-loop system with the given DES. The controller observes the DES and provides a feedback to alter the behavior of the given DES, such that the behavior of the closed-loop system is desirable. A general schematic of a closed-loop system can be seen in Figure 1.1. A DES can be either completely observable, where all events can be detected by an external observer (the controller), or partially observable where only a strict subset of events can be detected. In this discussion we consider only completely observable systems.

![Figure 1.1: Closed-Loop Feedback System](image)

There can be various properties of a closed-loop system; these are called control objectives. For example, perhaps one would like to ensure that all processes in a computer have equal processing time, or perhaps one would like to avoid certain dangerous states in the operation of a manufacturing plant. The most common control objectives for a system are summarized below [29, 16].

- **Safety**: Holds if unsafe strings do not belong to the closed-loop behavior.
- **Nonblockingness**: Holds if from any reachable state the system can reach a marked state.
- **Required behavior**: Holds if the closed-loop language contains the required sublanguage.
- **Non-starvation**: Holds if no subsystem is denied access to a shared resource forever.
- **Fairness**: Holds if each subsystem regularly has access to shared resources.
1.3 State-Explosion Problem

In this discussion, we focus on the two most important objectives: safety and nonblocking. Their importance can be illustrated through a simple example. Imagine a complex train network described as a discrete-event system. Multiple trains are in the network simultaneously and one would like to ensure that all trains eventually reach their destinations without collisions. Ensuring no collisions is a safety objective in that there are a certain set of event sequences (that lead to a collision) which we would like to avoid. At the same time, we want to ensure that the system is nonblocking and does not jam such that the trains will always eventually be able to reach their destination.

With these two goals in mind, one would like to synthesize a controller which interacts with the original system such that the behavior of the closed-loop system is both safe and nonblocking. There are two main paradigms for controlling discrete-event-systems: the input/output paradigm and the supervisory control paradigm introduced by W.M. Wonham. In the input/output paradigm, a given system is seen to have both inputs and outputs, and the controller has the ability to generate (input) events as a means to affect the behavior of the system [20, 21]. In supervisory control, the set of possible events is partitioned into two disjoint sets: controllable events and uncontrollable events. In supervisory control, the controller enables/disables controllable events at certain states as a means to affect the behavior of the system. If a controllable event is disabled at a state, it means that this event can no longer occur at that considered state, thus restricting the behavior of the system. In this discussion, we use supervisory control as our paradigm for the control of discrete-event systems.

1.3 State-Explosion Problem

When modeling discrete-event systems, it is common to describe them modularly as different components. It is often most straightforward to model the different components individually, and combine them as necessary. When using automata, one combines individual components through the parallel composition (||) operation. This operation constructs an automaton which results in all the possible interleavings of the events of both processes. Shared events constrain, or synchronize, the interleaving since these events must occur simultaneously in both processes. Given two automaton of sizes \( m \) and \( n \), their parallel composition will be of size in the order \( O(m \cdot n) \). Therefore the number of states in a model with many components can grow extremely quickly. More specifically, the number of total states grows exponentially with the number of components; this is known as the state explosion problem, and is one of the central challenges in the realm of supervisory control. It is infeasible for a computer to store the entire state space with limited time and memory, which means we can quickly reach the limit of which systems can effectively be controlled. In fact, it has been shown in [8] that this state-space explosion is unavoidable in general and that it is in fact an NP-hard problem. It is therefore suggested that efforts should be directed towards more specific solutions which take advantage of structural properties.
1.4 Modular Systems

Modular approaches have been the main strategy to tackle the state explosion problem. By representing a system in a modular fashion, one avoids having to compose an explicit representation of all the possible states. A brief survey of common modular control approaches is presented below.

A decentralized system is a monolithic system with two or more observation streams. In the control of decentralized systems, multiple supervisors each restrict part of the behavior of the system. The control laws of the supervisors can interact in either a disjunctive or conjunctive fashion as introduced by S. Lafortune. There have been approaches both with and without communication between supervisors. More information can be found in [32, 17, 1, 24, 23, 25].

A distributed system is an interconnection of two or more subsystems, each having its own observation stream. In the control of distributed systems, local supervisors, each receiving their own observation stream(s), locally restrict the behavior of their corresponding subsystems. The global system is defined as the parallel composition of all the subsystems, and the corresponding global supervisor is similarly defined as the parallel composition of all the local supervisors.

A coordinated system is one where there is a global supervisor which takes care of the coordination between two or more subsystems. The mathematical concept of a coordinator is based on the notion of conditional independence from probability theory. The coordinator, also an automaton, makes the subsystems conditionally independent. That is, every event shared between subsystems must also be shared with the coordinator. A formal definition can be found in [15, 13].

A hierarchical system is one where two or more layers vertical layers are distinguished. Each layer can have several subsystems, similar to the approach of coordinated systems. In the control of hierarchical systems, a supervisor is synthesized at every layer.

Modular systems are good at keeping the number of states in the model from growing too quickly, but as the theory is not fully developed, there are other issues such as increased supervisor synthesize computation times and more difficult checks for nonblockingness.

1.5 State-Tree-Structures

Keeping a system modular is not the only way of tackling the state explosion problem. State-tree-structures (STSs) are a framework first proposed by Ma Chuan in 2005 in [18] as a structured approach to tackle the state explosion problem. The framework is based on the model of state-trees introduced by David Harel in 1987 [10]. The state-tree-structure framework extends supervisory control to state-trees (structured modeling). State-tree-structures is a framework which structures the state space for an efficient representation of states and
transitions. The states and transitions are represented as predicates which, in turn are represented as binary decision diagrams.

1.5.1 Binary Decision Diagrams

Binary decision diagrams (BDDs) are a data structure first introduced by R.E. Bryant in 1986 in [2]. A binary decision diagram is a directed acyclic graph (DAG) used to represent a decision tree for a boolean function \( f : \{0, 1\} \rightarrow \{0, 1\} \). In many cases a BDD representation is much smaller than the corresponding explicit truth table representation of the boolean function. This is why it is so appealing to use a structure to represent the state space in a computer. After representing the predicates as BDDs, the computation complexity for supervisor synthesis using STSs is no longer polynomial in the number of states, but in the number of BDD nodes. In the worst case, the number of BDD nodes can grow at the same rate as the number of states, so the state-explosion problem is not fundamentally solved. However, for a wide array of practical examples, the number of BDD nodes is much less than the number of states, which is where the efficiency gain comes into play.

1.5.2 Nonblocking

The concept of controllability for supervisory control in terms of automata becomes the concept of weak controllability for supervisory control of state-tree-structures. This has the advantage that the computations for weak controllability in terms of state-tree-structures do not require the computation of the reachable set nor that of the coreachable set. However, to ensure nonblockingness using state-tree-structures, the coreachable predicate needs to be computed, which is an expensive operation.

1.6 Hierarchical State-Tree-Structure

In this thesis, a novel way to decompose an STS is presented in order to make the check for nonblockingness of complex modular systems more efficient. As most modular systems have very few shared events in comparison to the total number of events, we propose a way to decompose the system into a two-layer hierarchy where the top layer contains information about the shared events, and the bottom layer contains the remainder of the model. Because the bottom layer contains no shared events, nonblockingness checks can be performed locally on the modules without having to perform any parallel composition operations. Since the number of shared events is comparatively small, the top layer, the only layer where a parallel composition is necessary, will be much smaller than the explicit monolithic representation. This drastically reduces computation times.

1.7 Summary

The thesis is organized as follows. The next chapter formally introduces the problem. Chapter 3 provides a detailed introduction to supervisory control. Chapter 4 provides a detailed introduction to state-tree-structures. Chapter 5 provides an analysis of the computational
efficiency of state-tree-structures and abstraction of automata. Chapter 6 provides an explanation of how to combine two or more state-tree-structures describing the same system. Chapter 7 introduces the procedure in which such a complex STS can be analyzed for nonblockingness. Chapter 8 presents both computational results and a practical example. Chapter 9 concludes this thesis with a summary and suggestions for future work.
Chapter 2

Problem Statement

Currently, one of the most serious challenges facing supervisory control is the state explosion problem. Complex systems are often built in a modular way and can contain many parallel components. The total number of states in a system grows exponentially with the number of components, so a complex system can quickly grow too large to control. It is infeasible for a computer to store the entire state space with limited time and memory so new methods have to be developed.

State-tree-structures and modular approaches are two strategies for mitigating the state explosion problem for supervisory control. State-tree-structures offer a monolithic approach which takes advantage of structural properties of a system to make supervisor synthesis more efficient. This approach does not solve the fundamental state explosion problem, but provides a good heuristic based on the structure of the system. Unfortunately, expensive coreachability computations have to be performed to ensure nonblockingness. As for modular approaches, the idea is to represent the components of a system separately in order to minimize the number of states in the model. By representing a system modularly, no parallel compositions have to be performed and the state explosion is avoided. There have been several variations of distributed and hierarchical control, but they generally have the same pitfall in that computation times tend to be a little longer and ensuring nonblockingness is more difficult.

It is therefore interesting to investigate whether there is a more efficient way to check the nonblockingness of a system taking advantage of both modular and state-tree-structure strategies. If such a check does exist, it is ideal to provide a formal procedure for it. More specifically, the goal of this master’s thesis is to develop an efficient check for nonblockingness of distributed state-tree-structures. This is done by building a recursive two layer hierarchy where the top layer contains information regarding shared events, and the bottom layer contains the remaining asynchronous system information. Not only are components represented separately, but the components themselves can be represented as state-tree-structures, making the approach even more efficient.

The plan of this investigation is
1. To study state-tree-structures and their control based on the book [18].
2. To formulate a model for a hierarchical-distributed system using state-tree-structures.
3. To develop an efficient nonblockingness check for distributed systems.
Chapter 3

Supervisory Control

This chapter introduces the topic of supervisory control of discrete-event systems in more detail. The reader is expected to be familiar with automata theory. Section 3.2 introduces the model and Section 3.4 introduces the notion of a specification. Section 3.6 explains how to synthesize a controller and Section 3.7 concludes this chapter.

3.1 Introduction

Supervisory control is a popular paradigm used for the control of discrete-event systems. It is formulated as the Ramadge-Wonham (RW) Framework in [22]. The idea is that an automaton $G$ models the “uncontrolled behavior” of the system. The behavior of this original system may be unsatisfactory and must be restricted. In order to do so, a supervisor $S$ is introduced. Simply put, $S$ tells $G$ which events are allowed next. In supervisory control, the goal is to synthesize this supervisor $S$ such that the resulting closed-loop system exhibits the desired behavior and is nonblocking. A thorough background can be found in [28, 31, 4].

3.2 Model

Supervisory control, as introduced by W.M. Wonham, uses a derivate of automata to model discrete-event systems. A standard automaton has a complete transition function. That is to say, at every state, a transition for every event must be defined. Supervisory control allows for incomplete transitions functions. We call such a model a generator, and it differs from an automaton in that the transition function is only a partial function.

3.3 Plant/Supervisor Interaction

The interaction between a supervisor and a plant is structured by the notion of controllable and uncontrollable events. In supervisory control, the event set $E$ is partitioned into two disjoint sets, denoted as follows.
\[ E = E_c \cup E_{uc} \]

\( E_c \) is the set of controllable events and \( E_{uc} \) is the set of uncontrollable events. They are explained below.

**Controllable events** These are the events that a supervisor has control over. The supervisor can allow them to happen by enabling them, or prevent them from happening by disabling them. Common examples include the toggling of a switch, or the enabling of a certain process.

**Uncontrollable events** These are the events that cannot be prevented from happening, or disabled, by a supervisor. In other words, uncontrollable events can always occur. Common examples include system faults and updates in sensors readings.

Consider a generator \( G \), the event set \( E \) can always be broken down into the set of controllable events \( E_c \) and the set of uncontrollable events \( E_{uc} \). In order to modify the behavior of \( G \), a supervisor (i.e. controller) \( S \) is introduced. \( S \) observes \( G \), and for every given string returns a subset of enabled events. In this discussion, we will assume that \( S \) observes all events of \( G \).

### 3.3.1 Supervisor

Consider a generator \( G \). A supervisor \( S \) plays the same role as a controller in traditional control theory and forms a closed-loop system with \( G \). The closed-loop system is denoted as \( S/G \), read as \( S \) controlling \( G \). It’s behavior is \( L(S/G) \) and must be a subset of \( L(G) \). \( S/G \) is a DES and can therefore also be characterized by its generated and marked languages.

The supervisor \( S \) affects the behavior of the closed-loop system by restricting the behavior of the original generator \( G \). It can do so by disabling specific controllable events after observing certain string of events. Formally, a supervisor is a function from the language generated by \( G \) to the power set of \( E \):

\[ S : L(G) \rightarrow 2^E \]

\( S \) is considered the control law while \( S(s) \) is considered the control action. That is given a string \( s \), a supervisor returns \( S(s) \), the subset of all events to be enabled. As a supervisor has no control over uncontrollable events, it can never disable an uncontrollable event. That is to say, given a current state, all uncontrollable events from that state are enabled. A supervisor which follows these rules is considered *admissible*. The goal of supervisory control is to synthesize a supervisor such that it disables those controllable events such that the resulting behavior is safe and nonblocking in regard to the specification. The complete
set of feasible events given a certain state is the union of all uncontrollable events and of the enabled controllable events.

3.4 Specification

It is desirable that a given generator $G$ exhibits the right behavior. The specification for the synthesis of a supervisor can be considered as a formal representation of the desired behavior of the closed-loop system. The specification can be formulated in a variety of ways: a set of allowable states, a set of illegal states, a necessary sequence of events, and more. For the remainder of this chapter we consider the specification for the behavior of a system to be a desired allowable marked language $L_a \subset L_m(G)$. (It should be a strict subset of the $L_m(G)$, because otherwise there would be no need to control the system in the first place.)

3.5 Problem Formulation

In the supervisory control paradigm as introduced by W.M. Wonham, it is desired to synthesize a controller to make the closed-loop system of plant and supervisor safe and non-blocking. Here we define it formally.

Problem 3.5.1

Consider:

1. A DES modeled by an automaton $G$ with event set $E$ with a generated language $L$ and marked language $L_m$

2. A specification in the form of an allowable marked language $L_a \subset L_m(G) \subset E^*$

Find:

1. $L(S/G) = L_c$ such that $L_c \subseteq L_a$ and $L_c = \overline{L_c}$.

2. The control law for the supervisor $S$ to achieve this maximum controllable and non-blocking sublanguage.

In other words, given generator and a specification, find the maximum sublanguage which forces this behavior and is nonblocking, and determine the corresponding supervisor.

3.6 Nonblocking Control

We have introduced the model and the problem. The question now is how to compute the supervisor.
3.6. Existence

Given a DES and a set of specifications, it is useful to determine whether or not there does indeed exist a supervisor which can make the controlled behavior safe and nonblocking. For this to happen a system must be both controllable as well as nonblocking. The conditions for controllability and nonblocking are explained below.

Controllability  The behavior of a closed-loop system is considered safe if it is a subset of the specification language $L_a$. For this to happen, the system must be controllable with respect to the specification. Formally, there exists a supervisor $S$ such that $L(S/G) = \overline{L_a}$ if and only if

$$\overline{L_a}E_{uc} \cap L(G) \subseteq \overline{L_a}$$

Intuitively, consider an admissible behavior $\overline{L_a}$. If a string of $\overline{L_a}$ is followed by an uncontrollable event and if the extended string is also part of the language of the plant, then the resulting string must also be part of $\overline{L_a}$. This is called the controllability condition and the proof can be read in [4, 31]. If this condition holds we call the sublanguage $L_a$ controllable with respect to $L(G)$.

Nonblockingness  Often times, safety is not enough. The controlled closed-loop system must also be nonblocking. A DES is blocking if a situation arises from which it cannot reach a marked state. A system is nonblocking if its reachable behavior is a subset of the prefix-closure of its marked language $L_m$. Formally, there exists a nonblocking supervisor $S$ for $G$ such that $L_m(S/G) = L_a$ and $L(S/G) \subseteq \overline{L_a}$ if and only if

1. $L_a$ is controllable (from above).
2. $L_m(G)$-closure: $L_a = \overline{L_a} \cap L_m(G)$ (The closure condition)

The proof of this can also be seen in [4, 31].

If the existence conditions are met, it means that a supervisor does exist to induce the specified behavior; the control law is then straightforward to determine. The behavior of the controlled system in this case is simply the same as the specification $\overline{L_a}$. The control law is rather simple as well. Simply disable any events which would lead to a string outside of the specification $\overline{L_a}$. The control law is formally defined below.

$$S(s) = [E_{uc} \cap \Gamma(x_s, s)] \cup \{\sigma \in E_c : s\sigma \in \overline{L_a}\}$$

Where $\Gamma(x)$ returns the set of feasible events from state $x$, and $\delta$ is the transition function. Intuitively, the control law is the set of feasible uncontrollable events along with the set of controllable events which do not lead to a string outside of the specification language $\overline{L_a}$. 

14
3.6.2 Supremal Sublanguage

Given a certain model of a DES and a specification, sometimes a safe nonblocking supervisor does not exist. It is then desired to find the “largest” sublanguage of $L_a$ that is controllable and nonblocking, where “largest” is in terms of set inclusion. In order to solve this problem we first define a class of languages, $C_{i_n}$, as the set of sublanguages of $L_a$, which are controllable and nonblocking. This is formally defined below.

$$C_{i_n} := \{ L \subseteq L_a : \overline{TE_{uc}} \cap \mathcal{L}(G) \subseteq \overline{T} \}$$

Let us then take the union of all the elements in the class $C_{i_n}(L_a)$ and denote the result by:

$$L_a^{\uparrow C} := \bigcup_{L \in C_{i_n}(L_a)} L$$

$L_a^{\uparrow C}$ has certain properties delineated below:

1. A well-defined sublanguage of $L_a$
2. $L_a^{\uparrow C} \in C_{i_n}(L_a)$
3. There is no strictly larger sublanguage also satisfying the first two conditions.

That is, $L_a^{\uparrow C}$ is the largest possible behavior given a model of a DES $G$ and is the optimal controlled behavior. The control law associated with $L_a^{\uparrow C}$ may be determined from the sublanguage as described before. Simply disable any events at a given state that would lead to a string outside of $L_a^{\uparrow C}$.

3.6.3 Synthesis Algorithm

In summary, the maximum controlled nonblocking behavior of a system with specification $L_a$ is given by $L_a^{\uparrow C}$, and the associated control law follows directly. A standard algorithm for computing this given in [4, 31]. The actual algorithm is not presented here as it is not the main topic of this thesis. However, it is important to note that it’s running time is $O(n^2 m^2 |E|)$, where $n$ is the number of states in the specification automaton, $m$ is the number of states in the automaton of the system model and $|E|$ is the size of the event set.

3.7 Conclusion

Supervisory control is a useful paradigm for controlling discrete-event systems. However, as the running time of supervisor synthesis depends on the number of states in the model, it is susceptible to the state explosion problem. This is addressed in the following chapters.
Lastly, it is good to note that supervisory control can be used for more than the common applications of ensuring safety of industrial plants, or ensuring that network protocols avoid certain states. There is an interesting computer engineering application presented in [12], where computer code is analyzed and concurrency bugs are eliminated through supervisory control. This example serves as a reminder that there are more applications for this theory than there may at first seem.
Chapter 4

State-Tree-Structures (STS)

This chapter introduces an efficient extension of the supervisory control paradigm introduced by W.M. Wonham. State-tree-structures are used as a means to structure the state-space such that binary-decision-diagrams can be used to efficiently represent the states and transitions of a system. The reader is expected to have a basic understanding of supervisory control.

4.1 Motivation

Current algorithms for supervisory control synthesis run in polynomial time with the number of states. As the state-space grows exponentially with the number of components which make up a system, systems can quickly become too complex for these algorithms to handle. It is therefore interesting to look at a representation which makes the computation more efficient.

4.2 Introduction

Supervisory control was extended to structured models based on state-charts with the introduction of state-tree-structures by W.M. Wonham and Ma Chuan in [18, 19]. State-tree-structures (STS) are framework to tackle the complexity problem of supervisor control. Rather than store all the information of a DES explicitly in a generator, STSs store the state information in a structured hierarchy and store only local transition information. The structured state-space is called a state-tree and a set of local transition information is called a holon. Together, a state-tree and all of its holons make up a state-tree-structure. States in the STS are represented with predicates, which are in turn represented by BDDs. With these advancements, the control synthesis algorithm is no longer polynomial in the number of states but polynomial in the number of BDDs. Very structured state-spaces tend to yield very efficient BDD representation with few nodes, which is why this model is efficient. In the remainder of this chapter follows the formal definition of a state-tree-structure illustrated with an example.
4.2. Introduction

State-Tree-Structures (STS)

4.2.1 State-Tree

A state-tree assigns structure to the state-space of a DES. Because a complex DES can have many components, it may have billions of states, which is too many to store explicitly in a computer. Therefore the state-tree-structure is used to efficiently represent the state-space through both vertical and horizontal concurrency. The notion of state-trees was derived from the paper by David Harel for state charts.

State-trees can be very efficient because they are able to easily represent many concurrent components in a system. State-trees take advantage of the natural structure of a system and the inherent hierarchy involved in modeling a system. The structure of the system is encoded in the structure of the state-tree. In order to explain the components of a state-tree we use a running example of modeling a car.

Let us begin with the motor component. Let us say that it has two states: on and off. The model for this motor component can be seen in Figure 4.1. In a state-tree-structure, each of these two states is considered a simple-state. It is also known that these two states are exclusive. That is, the system cannot be at more than one of these states at a time. This is denoted in a state-tree by grouping these two simple-states under an OR-superstate.

Let us say that there is also a brake component which also has two states: on and off. The generator representing this component can be seen in Figure 4.2. Similarly to the motor, the brake component has two simple states which are grouped into an OR-superstate. At this point in our state-tree, we have two components, each represented by an OR-superstate. Together, these two components, the motor and the brake, form part of the car model. Rather than performing the parallel composition of the two components, it is simply denoted that they are concurrent. In a state-tree, this is done by grouping the components under an AND-superstate. An AND-superstate denotes that these two states are concurrent. The system must be somewhere in the engine state AND somewhere in the brake state at all times.

At this point we have an AND-superstate representing our assembly. Let us say that the
4.2. Introduction

entire system starts off disabled, and must warm up before it can enter any of the states of the two components. Let us add a disabled state and a warm-up state. These two states are exclusive with respect to the AND-superstate, because it is not possible to be in more than one of the states at the same time, so they are all grouped under an OR-superstate. The STS diagram representing this can be seen in Figure 4.3.

In this manner it is possible to build up a straightforward model for a complex DES.
The beauty of this framework is that it is possible to visualize a complex DES which would otherwise occupy thousands of states, as a state-tree-structure diagram which fits onto a single page. A schematic of a complex system can be seen in Figure 4.4.

In summary, a state-tree is composed of three kinds of states:

1. **Simple state** The foundation of a state-tree are simple-states, represented as leaf nodes in the state-tree. From here states can be layered and grouped under parent superstates in one of two ways.

2. **AND-superstate** Concurrent states can be grouped into an AND-superstate. The system must be at all states simultaneously.

3. **OR-superstate** Disjoint states can be grouped into an OR-superstate. The system must be at exactly one state.

The type function \( T : X \rightarrow \{ \text{and, or, simple} \} \) is used to determine the types of the states in a state-tree. Given a state, it returns the type of that state. The expansion function \( E : X \rightarrow 2^X \) is used to determine which states are grouped under (or expand) a particular superstate. Given a superstate, it returns the set of states in the level below it in the state-tree. Putting this all together, a state-tree is formally defined below.

**Definition 4.2.1** A State-Tree ST is defined as the 4-tuple, 
\[
ST := (X, x_0, T, E)
\]
where

- \( X \) is a finite structure state set.
• \( x_0 \) is the root state.
• \( T : X \rightarrow \{ \text{and, or, simple} \} \) is the type function.
• \( E : X \rightarrow 2^X \) is the expansion function.

Remark 4.2.2 This representation is efficient because the synchronous product of concurrent modules is never computed and explicitly represented. Concurrent automata are simply represented as child nodes of the same AND-superstate.

4.2.2 States
A state-tree structures the state-space. A basic state in an STS is analogous to a state in a generator, and is represented as sub tree of the state-tree. In order to be a proper state, an OR-superstate can only be in one of its children at any given time, whereas each AND-superstate must be in all of its children simultaneously. An example of this is given in Figure 4.5.

The graphical notation is quite straightforward to understand because it is simply a sub-tree. However, in order to manipulate the states in an algorithm on a computer, a symbolic representation is needed. Predicates are used to represent this information in the following manner. A sub-tree can be identified by a set of basic states and can be represented with a predicate. A predicate \( P : \text{ST} \rightarrow \{0,1\} \) is a function. Given a basic state, it returns a 1 if the basic state is satisfied by the predicate, and 0 otherwise. It can be identified by a set of basic states, namely the set of basic state which satisfy it. Since a predicate is simply a boolean function, it can be represented as a BDD.

Figure 4.5: An example of a sub state-tree: How to represent a state in an STS
4.2.3 Holon

A holon defines the local behavior of a specific OR-superstate to which it is matched. It defines the transitions within the OR superstate as well as the transitions into and out of that OR superstate. Holons are simply generators, so they are deterministic and can only be in one state at a time. A holon is formally defined below.

Definition 4.2.3 A Holon $H$ is defined as the 5-tuple, $H := (X, E, \delta, X_o, X_m)$ where

- $X$ is the nonempty state set.
- $E$ is the event set.
- $\delta : X \times E \to X$ is the partial transition structure.
- $X_o$ is the set of initial states.
- $X_m$ is the set of marked states.

4.2.4 STS

Together, the structured state-space along with a set of holons matching the OR super-states, make up a state-sree-structure. It is defined formally below.

Definition 4.2.4 A State-Tree-Structure STS is defined as the 6-tuple:

$STS := (ST, H, E, \Delta, ST_o, ST_m)$ where

- $ST$ is the state-tree representing the state-space
- $H$ is the set of holons matching the OR superstates.
- $E$ is the event set including all events appearing in $H$.  

Figure 4.6: This is an example of a holon. This particular holon matches state $x_{121}$ of Figure 4.5
4.2.5 Computer Representation

In order to input an STS into a computer, there must be a standard way to represent it. Ma Chuan has implemented the synthesis algorithm for STSs and has a corresponding format for describing STSs. The format is explained through an example. Below we represent the STS from Figure 4.4 in the proper computer format. We use the '%' symbol to denote comments to help clarify the format.

```plaintext
{ % this is the start of the state tree definition
root = x_0 % The root state is assigned a name 'x_0'.
{
% The expansion (type) functions of the state tree is defined in this set
x_0 = AND {x_1, x_2} % x_0 is an AND state expanded by x_1 and x_2
x_1 = OR {x_11, x_12, x_13} % x_1 is an OR state expanded by x_11, x_12, x_13
x_2 = OR {x_21, x_22, x_23} % x_2 is an OR state expanded by x_21, x_22, x_23
x_12 = AND {x_121, x_122}
x_121 = OR {x_1211, x_1212}
x_122 = OR {x_1221, x_1222}

% list of holons. There are 4 holons matched to the 4 OR states
% defined in the state tree.

x_1 % Holon x_1 is matched to the OR state x_1.
{a, c, f} % Set of controllable events in x_1. {} if no controllable events
{} % Set of uncontrollable events in x_1. {} if no uncontrollable events
{} % Set of internal transitions in x_1.
[x_11 a x_12]
[x_12 c x_13]
[x_13 f x_11]
}

x_2
{c, f, g}
{}
{}
[x_21 c x_22]
[x_22 f x_23]
```

---

- $\Delta : \text{ST} \times E \rightarrow \text{ST}$ is the transition function.
- $\text{ST}_0$ is the initial state tree. It is analogous to the initial state in an automata, but its representation is as a state-tree.
- $\text{ST}_m$ is the marker state tree set. This is analogous to the marked state set in an automata, but its representation is as a state-tree.
4.2. Introduction

State-Tree-Structures (STS)

4.2.6 Well Formedness

In order for a STS to be well formed, two conditions must hold:

1. **Boundary Consistency** Boundary consistency is defined within a holon. It simply means that the external states must belong to the holon directly above the current one in the state-tree hierarchy. That is, when layering holons, one should be able to plug one holon into the upper level.

2. **Local Coupling** Simply put it means that synchronous events must be AND-adjacent. That is for any two OR-superstates sharing a synchronous event, there can only be AND-superstates between themselves and their common ancestor. A schematic can be seen in Figure 4.7.

**Remark 4.2.5** *These two conditions are imposed by Ma Chuan and are necessary for his supervisor synthesis to work. It is significant for this work because it means that any events in the STS which are in the second OR-superstates or lower, cannot contain any synchronous events. We will see that the local-coupling condition becomes very important.*

4.2.7 Specification

In the STS paradigm, specifications are not represented as automata or languages, but rather as a sub state-tree, representing the set of illegal states. This in turn is represented as a
Figure 4.7: Local-coupling condition of wellformedness. The shared events must be at the top level (AND-adjacent). Note that event c is shared and is at the top level. Therefore the local-coupling condition holds for this STS.

predicate. For the rest of this chapter, the specification for supervisor synthesis of state-tree-structures will be denoted as a predicate of illegal states.

### 4.2.8 Supervisor

A supervisor for a state-tree-structure is analogous to the supervisor for a generator. Given a sub-state-tree, it too, returns a subset of controllable events to enable.

### 4.3 Problem Formulation

The control problem using the state-tree-structure framework is analogous to that for generators. Here we define it formally.

**Problem 4.3.1**

**Consider:**

2. A specification $P$ as a set of illegal states in the form of a predicate.

**Find:**

1. The optimal controlled behavior (a predicate representing all the legal states).
The control law for the supervisor. That is, determine the proper initial predicate, and determine which controllable events to be enabled at every state such that the controlled behavior is both safe and nonblocking.

4.4 Non-Blocking Control

Before we introduce non-blocking control, let us revisit the notions of reachability and coreachability. They are defined a little differently for state-tree-structures.

- \( R(G, P) \) is the set of reachable states of STS \( G \) contained within predicate \( P \).
- \( CR(G, P) \) is the set of coreachable states of STS \( G \) contained within predicate \( P \).

We have introduced the model and the problem for the STS framework, the question now is how to compute the supervisor.

4.4.1 Existence

Given a STS and a specification, it is useful to check if there does indeed exists a supervisor which can make the closed-loop behavior adhere to the specifications and be nonblocking. Given an STS \( G \), a supervisor exists which satisfies the specification \( P \) and is nonblocking if:

1. If \( P \) is weakly controllable w.r.t \( G \).
2. If additionally, \( P \) is coreachable w.r.t \( G \).

These conditions are explained in more detail below.

**Weak Controllability** The property of weak controllability for STSs is analogous to the property of controllability for automata. Let us first recall the transition function \( \Delta \) of a STS from Definition 4.2.4. This is used to define a predicate transformer \( M_\sigma \).

\[
\text{If } b \text{ satisfies } M_\sigma(P) \iff \Delta(b, \sigma) \text{ satisfies } P
\]

Intuitively, \( M_\sigma(P) \) is the predicate from which one can reach predicate \( P \) via transition \( \sigma \). Note that \( M_\sigma(P) \) is itself a function, not a function of \( P \).

Weak controllability is a condition which is slightly less stringent than the traditional definition of controllability as defined in [31]. It is proved in [18] that the condition of weak controllability is enough to guarantee the existence of a supervisor. The condition for weak controllability is defined below.

**Definition 4.4.1** A predicate \( P \) is weakly controllable w.r.t. \( G \) if \( (\forall \sigma \in \Sigma_u)P \leq M_\sigma(P) \)
Coreachability It is shown in [18] that ensuring that a predicate $P$ is coreachable with respect to $G$, is enough to ensure that it is also nonblocking.

If $P$ satisfies both of the conditions above, it means the exact behavior specified by $P$ can be achieved. The proof of this can be read in [18].

### 4.4.2 Supremal Predicate

If there does not exist a supervisor such that the closed loop-system adheres exactly to the specification, it is often desirable to find the best possible behavior. In this case, the goal is to find the supremal predicate which is controllable and nonblocking. In order to do so, we define a family of predicates, $C^2P$ as the family of weakly controllable and coreachable subpredicates that are stronger than $P$. This is formally defined below.

$$C^2P := \{ K \in \text{Pred}(ST) | K \leq P \land K \text{ is weakly controllable } \land K \text{ is coreachable} \}$$

Analogous to the RW framework, the idea is to then take the union of all the weakly controllable and nonblocking predicates. This results in the supremal subpredicate $\text{sup}C^2P$. This is defined below.

$$\text{sup}C^2P(P) = \bigvee \{ K | K \in C^2P(P) \}$$

$\text{sup}C^2P(P)$ is the optimal behavior possible given an STS and a specification. Once this is computed, the control law of the supervisor is given as follows.

- **The closed loop initial predicate** $P^f_o$
  $$P^f_o = P_o \land \text{sup}C^2P(P)$$
  where $P_o$ is the initial predicate of the STS and $P$ is the specification predicate.

- **The control law** $f_\sigma(b)$
  $$\langle \forall \sigma \in \Sigma_c \rangle f_\sigma(b) := \begin{cases} 0, & \text{if } \Delta(b, \sigma) \models \neg \text{sup}C^2P(P) \\ 1, & \text{otherwise} \end{cases}$$
  where $\Delta(b, \sigma)$ is the transition function; it returns the destination predicate of a transition from predicate $b$ due to event $\sigma$.

### 4.4.3 Synthesis Algorithm

The synthesis algorithm can be found in [18]. It is not presented in here because it is not the topic of this chapter. It is important to note that the running time is polynomial in the number of BDD nodes. This algorithm does not eliminate the state-explosion problem. However, due to the structure in the model, the number of BDD nodes is often much less than the number of states. The running time of this algorithm will be examined in more detail in the following chapter.
4.5 Conclusion

State-tree-structures offer many advantages and have been shown to work for a system with $10^{20}$ states. In general, the more structure there is to an STS, the more efficient it can be represented as a BDD. That means that a completely flat STS with only one AND-superstate, does not offer much of an efficiency gain. A balance in height and width of the state-tree gives the most structure and the most efficient representation when using BDDs. Note that to verify controllability of an STS, one does not need to calculate the reachability or the coreachability; the condition of weak controllability is enough to check for this. However, to verify nonblockingness, one must compute the coreachability, an expensive operation.
Chapter 5

Efficiency

Previous work on state-tree-structures and distributed systems involved methods in which to make supervisor synthesis for DESs more efficient, with the goal of ultimately being able to control larger and more complex systems. Below we take a deeper look into exactly why these methods are more efficient, and how much faster they really are.

5.1 Introduction

Let us examine a DES $G$ modeled by two generators, $G_x$ and $G_y$ in parallel with $x$ and $y$ number of states respectively. The monolithic representation (achieved through the parallel composition operation) has on the order of $x \cdot y$ states. In supervisory control, the standard algorithm to calculate the maximum supremal supervisor as described in [4], has a worst-case running time of:

$$O(n^2m^2|E|)$$

where $n$ is the number of states in the specification, $m$ is the number of states in model of the DES and $|E|$ is the number of events in the model.

Thus the running time of supervisor synthesis for $G$ composed of $G_x$ and $G_y$ with respect to the number of states would be $O((xy)^2(xy)^2|E|)$. This is clearly not very efficient with respect to the number of states $x$ and $y$. In the following two sections we analyze the running time gain when using state-tree-structures as well as when using a distributed approach.

5.2 State-Tree-Structures

State-tree-structures optimize the supervisor synthesis by introducing an efficient structural representation with BDD nodes. In his thesis, Ma Chuan claims that it is now polynomial in the number of BDD nodes rather than the number of states. This is certainly an improvement as the number of BDD nodes is usually much less than the number of states. However, in the worst case, the number of BDD nodes can reach the number of states, so the state explosion problem is not solved. An investigation into the algorithm was necessary to find the exact
5.2. State-Tree-Structures Efficiency

running time, which turned out to be a little different. I am trying to find the running time in \( O(\cdot) \) notation of the algorithm to compute the maximum supremal sub-tree. In other words, this is the algorithm to compute the controller, or supervisor, for a STS. For this we need to examine the actual algorithm from [18], which is presented in the following section.

5.2.1 The Algorithm

The basic algorithm for the computation of the supremal sub-language of a STS \( G \) given a specification predicate \( P \) is presented below.

**Algorithm 1** Supervisor Synthesis

1. \( K_o = P \)
2. \( K_{i+1} = \Omega_P (K_i) \)
3. If \( K_{n+1} \equiv K_n \), then \( \text{sup} C^2 P (P) := K_n \). Otherwise go back to step 2.

where,
\[ \Omega_P (K) = P \land CR(G, \text{sup} C^2 P (K)) = P \land CR(G, \neg \neg K) \]

and the basic algorithm for \( [\cdot] \) is:

**Algorithm 2** \( [\cdot] \) Computation

1. \( K_o := R \)
2. \( K_{i+1} := K_i \lor \bigvee_{\sigma_u \in \Sigma_u} \Gamma(K_i, \sigma_u) \)
3. If \( K_{n+1} \equiv K_n \), then \( [R] := K_n \). Otherwise go back to step 2.

where \( \Gamma(P, \sigma) \) returns the predicate that can reach \( P \) via transition \( \sigma \)

and the basic algorithm for \( CR(G, P) \) is:

**Algorithm 3** Coreachability Algorithm

1. \( K_o := P \)
2. \( K_{i+1} := K_i \lor \bigvee_{\sigma \in \Sigma} \Gamma(K_i, \sigma) \)
3. If \( K_{n+1} \equiv K_n \), then \( CR(G, P) := K_n \). Otherwise go back to step 2.

5.2.2 Running Time

In order to determine the running time, we work from the inside out. Let \( u_1 \) and \( u_2 \) be two BDDs let \( |u_1| \) and \( |u_2| \) be the number of nodes in \( u_1 \) and \( u_2 \), respectively. From [18] we see that the running time to compute \( u_1 \lor u_2 \) is \( O(|u_1| \times |u_2|) \).

The computation of \( \Gamma(K, \sigma) \) involves a single join \((\lor)\) of two predicates. Therefore the worst-case running time of \( \Gamma(K, \sigma) \) is in:

\[ O(n^2) \]

where \( n \) the maximum possible number of BDD nodes of the STS, and results in a predicate with worst-case \( n \) BDD nodes.
The operation $\bigvee_{\sigma \in \Sigma_u} (\cdot)$ results in $|\Sigma_u| - 1$ joins. Therefore the worst-case running time of $\bigvee_{\sigma \in \Sigma_u} \Gamma(K, \sigma)$ is in:

$$O \left( (|\Sigma| - 1) \cdot n^2 \right)$$

and results in a predicate with worst-case $n$ BDD nodes.

The final join with $K_i$ runs in worst case $O(n^2)$ as well since both operands of the join have at maximum $n$ BDD nodes. Therefore the worst-case running time of $K_{i+1}$ is in:

$$O \left( n^2 + (|\Sigma| - 1) \cdot n^2 \right)$$

$$= O \left( n^2 \cdot |\Sigma| \right)$$

For the basic algorithms of $[\cdot]$ and $CR(G, \cdot)$, in the worst-case, the predicate $K$ increases in cover by one basic-state-tree every iteration. This means that the fixed-point is not reached until all basic-state trees are added. There are count(STS) basic state trees in a given STS, so both of these basic algorithms run for count(STS) iterations in the worst case.

Therefore the worse-case running time of $[\cdot]$ is in:

$$O \left( \text{count}(STS) \cdot n^2 \cdot |\Sigma| \right)$$

The algorithm for $CR(G, P)$ is of the same form as $[\cdot]$ and thus has identical worst case running time. Therefore the worst-case running time of $\Omega_P$ is in:

$$O \left( \text{count}(STS) \cdot n^2 \cdot |\Sigma| + \text{count}(STS) \cdot n^2 \cdot |\Sigma| + n^2 \right)$$

$$= O \left( \text{count}(STS) \cdot n^2 \cdot |\Sigma| \right)$$

Applying the same logic to the basic algorithm $\sup^{C^2\mathcal{P}}(P)$, we can conclude that it too runs for $\text{count}(STS)$ iterations in the worst-case. Therefore the worst-case running of $\sup^{C^2\mathcal{P}}(P)$ is in:

$$O \left( \text{count}(STS) \cdot \text{count}(STS) \cdot n^2 \cdot |\Sigma| \right)$$

$$= O \left( \text{count}(STS)^2 \cdot n^2 \cdot |\Sigma| \right)$$

**Remark 5.2.1** The value $\text{count}(STS)$ is the number of basic states in the state-tree-structure STS. This is the number of states an equivalent representation using a generator would have. Therefore, this value is susceptible to an explosion; the value of $\text{count}(STS)$ increases exponentially with the number of components as well.

Consider two STSs $G_x$ and $G_y$ with $|G_x|$ and $|G_y|$ basic states respectively. Let us say that the BDD representation of STSs $G_x$ and $G_y$ have $|G_x|_{\text{bdd}}$ and $|G_y|_{\text{bdd}}$ nodes respectively. The corresponding algorithm for supervisor synthesis using state-tree-structure runs in time

Polynomial in the number of BDD nodes, or: $\left( |G_x|_{\text{bdd}} \right)^2 \cdot \left( |G_y|_{\text{bdd}} \right)^2$
Table 5.1: Summary of intermediate running times

<table>
<thead>
<tr>
<th>Computation</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma(K, \sigma)$</td>
<td>$O\left(n^2\right)$</td>
</tr>
<tr>
<td>$\bigvee_{\sigma_u \in \Sigma_u} \Gamma(K, \sigma)$</td>
<td>$O\left(n^2 \cdot</td>
</tr>
<tr>
<td>$CR(G, P)$</td>
<td>$O\left(count(\text{STS}) \cdot n^2 \cdot</td>
</tr>
<tr>
<td>$\Omega_P(K) = P \land CR(G, \neg[-K])$</td>
<td>$O\left(count(\text{STS}) \cdot n^2 \cdot</td>
</tr>
<tr>
<td>$\sup C^2 P (P)$</td>
<td>$O\left(count(\text{STS})^2 \cdot n^2 \cdot</td>
</tr>
</tbody>
</table>

where it is desirable that $|G_x|_{\text{bdd}} << |G_x|$ & $|G_y|_{\text{bdd}} << |G_y|$.

In the worst case scenario, it is obvious that state-tree-structures could provide no benefit if the number of BDD nodes is extremely high. As mentioned before, this occurs in state-tree-structures with almost no structure. For example, a flat state-tree, with only one AND-superstate and many components would result in a computation which is not much more efficient. However, if the state-tree had many levels, alternating between AND and OR-superstates, with a balanced number of substates, the BDD representation would be much more efficient.

5.3 Modular Approach

In modular control, the idea is to keep the components independent as much as possible. In general one can do the supervisor synthesis individually on each of the components with a little bit of overhead. This overhead can be from computing a supervisor for a coordinator, or perhaps for an abstracted hierarchical layer. Whatever the overhead is, the intuition behind this approach is that the overhead and computations for individual components should be less than the work of computing the supervisor for the monolithic system.

Consider a system with two subsystems with $n$ and $m$ states. Let us assume that the overhead computation is equivalent to $c$. If we were to compute the supervisor in a distributed fashion using STSs, the running time would be $O\left(n^2 + m^2 + c\right)$ as compared to $O\left(n^2 \cdot m^2\right)$ if it were done monolithically. The hope is that distributed approach is more efficient than the monolithic approach. In mathematical terms:

$$n^2 + m^2 + c < n^2 \cdot m^2$$

For this happen, the overhead $c$ must be constrained by the equation below.

$$c < n^2 \cdot m^2 - (n^2 + m^2)$$

In the approach we present in the following chapters, we analyze each of the modular components independently along with an additional overhead computation with the hope that the overhead is much less than the computation for the monolithic system as described here.
Chapter 6

Distributed State-Tree-Structures

6.1 Motivation

As distributed systems are often put together from independent components, it is desirable to be able to model a distributed system with a state-tree-structure by combining smaller state-tree-structures. In distributed systems, the component systems usually run in parallel with each other. This means that to create a STS out of component STSs in a distributed system, one must simply group all the components under an AND-superstate. This is exactly how the STS was built up in Chapter 4.

However, it is often not known a priori which events in the distributed system will be shared amongst components. Putting together two arbitrary STS under an AND-superstate may result in an STS which is not well-formed. Therefore we present an algorithm that, given any STS, restructures it as necessary to make it well-formed.

6.2 Introduction

Recall from Chapter 4 that the two conditions of well-formedness on an STS are boundary consistency and local coupling.

Since boundary consistency is only relevant to OR-superstates, combining two STSs under a new AND-superstate won’t change any parent-child Holon pairs, and will therefore not affect boundary consistency. If two STSs are boundary constant, so will their composition.

Maintaining the local-coupling property is not as straightforward. Consider a STS resulting from the combination of two component STSs. An event that was not shared in the individual components can all of a sudden become shared in the resulting STS if the two components have it in common. Even though the two component STSs may each individually satisfy the local coupling property, if this shared event is at a low level in the hierarchy, the resulting STS of the combined components will not be well formed.
This is summarized in the following proposition.

**Proposition 6.2.1** Consider a STS resulting from the combination of two well-formed component STSs. One only needs to ensure the local-coupling property holds in the resulting STS to ensure that it too is well-formed.

**Proof** This follows directly from the preceding discussion on local coupling and boundary consistency. □

In the following section we propose an algorithm to restructure a STS composed of two well-formed sub STSs to ensure that it too is well-formed.

### 6.3 Algorithm

The purpose of this algorithm is to restructure an STS if it is not well-formed such that it is well formed. From Proposition 6.2.1, this algorithm need only restructure the STS based on the local-coupling property (assuming the components of the given STS are well-formed). If a state-tree-structure does not satisfy local coupling, it is because a shared event is to low in the state-tree hierarchy. Recall that local-coupling means that states which share an event can only have AND-superstates between themselves and their common ancestor node in the state-tree. This algorithm needs to flatten the state-tree and remove OR-superstates between shared events until this condition holds.

Between any superstate and its parent superstate, there are only four possible combinations in the possible types of each state. These scenarios and appropriate actions to collapse them are described below.

- \( \times / \times \) The current layer is an OR-superstate and its parent is an OR-superstate. There is no need for anything to be collapsed. These two superstates are already AND-adjacent. An example of this can be seen in Figure 6.1a

- \( \times / \cup \) The current layer is an AND-superstate and its parent is an OR-superstate. There is no need for anything to be collapsed. This is the order we want the super states to be in. The parent AND-superstate is AND-adjacent with the common ancestor, and the child OR-superstate can be collapsed with the destination superstate. An example of this can be seen in Figure 6.1b

- \( \cup / \cup \) The current layer is an AND-superstate and its parent is an AND-superstate. Due to boundary consistency, the child superstates can be plugged into the parent superstate. An example of this can be seen in Figure 6.1c

- \( \cup / \times \) The current layer is an OR-superstate and its parent is an AND-superstate. This is the trickiest. The basic idea is to split the existing OR-superstate and move each copy down into the AND-superstate. An example of this can be seen in Figure 6.1d
The steps above are formally presented in the algorithm below.

**Algorithm 4 Flatten Algorithm**

Ensure: root > final

1: c ← root
2: while c ≠ final do
3: if \( T(C) = \text{or and } T(C) = \text{or} \) then plug-in
4: else if \( T(C) = \text{or and } T(C) = \text{and} \) then parallelize
5: end if
6: c ← root.next
7: end while

Where \( c \) denotes the current node being examined. The plug-in procedure merely collapses to OR-superstate and its parent OR-superstate. An example of this can be seen in Figure 6.2. The flatten procedure collapses an AND-superstate and its parent OR-superstate by duplicating the higher level OR-components and moving each copy into one of the AND-components. An example of this is shown in Figure 6.3.
6.3.1 Performance

**Proposition 6.3.1** The running time of Algorithm 4 is in $O(n)$, where $n$ is the length of the longest path between a state with shared events and its shared ancestor.

**Proof**

**Termination** With every iteration, the distance between the current node and the destination node decreases by one. As the path has a finite length, the algorithm is guaranteed to finish.

**Invariant** The current node is always AND-adjacent to the common ancestor.

**Initialization**
- $T(root) = and$, that is the root is the common ancestor and is therefore an AND-superstate.
- $C \in$ children of root
- Therefore, the current node $C$ starts off and adjacent to the common ancestor.

**Inductive Step**
1. if $T(C) = and & T(C) = and$: do nothing and current node is again an AND state and still and-adjacent.
2. if $T(C) = and & T(C) = or$: do nothing. current node becomes or state but is stall and-adjacent.
3. if $T(C) = or & T(C) = or$: current node stays OR state, but does not advance, so is still and-adjacent.
4. if $T(C) = or & T(C) = and$: current node becomes AND state, and does not advance so is still and adjacent.

□
Figure 6.2: An example of the flatten procedure

Figure 6.3: An example of the plug-in procedure
Chapter 7

Nonblockingness of Distributed State-Tree-Structures

This chapter introduces the notion of nonblockingness with respect to distributed state-tree-structures. The reader is expected to be familiar with the content presented in previous chapters on supervisory control and state-tree-structures. We begin with a brief motivation and introduction. We then introduce the notion of universal reachability and how it applies to our approach. We conclude by applying our method to an example.

7.1 Introduction

Performing supervisor synthesis modularly is the predominant method used to deal with the state explosion problem as monolithic supervisor synthesis has been proven to be NP-Hard by Gohari and W.M. Wonham in [8] for the general case. Whether it is done hierarchically or in a distributed fashion, there have been many modular approaches as seen in [11, 26, 14, 27, 7, 3, 30, 32]. Although the modular approach helps with controllability, a common problem for most of these approaches is ensuring the nonblockingness of the system. It is therefore desirable to find a method which simplifies the nonblockingness calculation. In order to create an efficient check for nonblockingness, we propose a novel way to abstract the system into different levels of hierarchy and uses the benefits offered by state-tree-structures.

When analyzing a system modularly, an important task is to decide how to decompose the system to make the analysis most efficient. In hierarchical control, the system is broken up into layers, whereas in distributed control, the system is broken up into parallel modules. Here we introduce a vertical and horizontal decomposition based on the structure of the state-tree-structures. In [8] W.M. Wonham mentions that it is very unlikely that we can truly solve the state explosion problem and that we should focus our efforts on efficiency gains based on structural properties. This is exactly what we do.
7.2 Asynchronous Groups

We begin by introducing some new terminology. Consider the state-tree-structure depicted in Figure 7.1a. Note that it has one AND-superstate at the top level. This AND-superstate contains two OR-superstates. Each of these OR-superstates contains a further two more OR-superstates and a simple state. The synchronous events are shown in bold red. Note that the synchronous events are all at the top level, as they should be according to the well-formedness conditions. The corresponding state-tree is depicted in Figure 7.1b as well for clarity. Note that the shared events are in the top level of the state-tree and are AND-adjacent.

In traditional nonblocking supervisory control of state-tree-structures, the computation for the controller synthesis is done in a monolithic manner on the entire state-tree-structure. In that vain, Figure 7.2 depicts the parallel composition of the two top level OR-superstates which make up the state-tree-structure. One of the OR-superstates is shown horizontally on the top in the diagram, and the other OR-superstate is shown vertically on the left. The resulting parallel composition of the two top level OR-superstates is shown in the middle.
From this diagram we can see that the second level OR-superstates (enclosed in black rounded rectangles), within the top level OR-superstates, combine into distinct groups (as indicated by the blue dashed rectangles) in Figure 7.2. We call these *asynchronous groups* (ASs). Below we define an asynchronous group formally.

**Definition 7.2.1** Given two components $C_1$ and $C_2$ with a synchronous product $P$, $A \subseteq P$ is an asynchronous group if,

$$\exists S_1 \subseteq C_1, S_2 \subseteq C_2 \mid (S_1 || S_2 = A) \& (E(S_1) \cap E(S_2) = \emptyset)$$

That is, $A$ can be factored into two sub-automata which do not share any synchronous events.

**Remark 7.2.2** From the local-coupling condition of well-formedness on a state-tree-structure, we know that the second level OR-superstates which compose the asynchronous groups cannot contain any shared events. (These second level OR-superstates cannot be AND-adjacent to anything since their parent nodes are also OR-superstates). An asynchronous group is the...
parallel composition of two second level OR-superstates, and is therefore guaranteed not to contain any shared events either. This is the reason asynchronous groups are named the way they are. A source state or destination state of a synchronous (shared) transition may be contained in an asynchronous group, but an AS may not contain an entire synchronous transition.

7.3 Abstraction

Our goal is to separate the states involved with synchronous transitions from states not involved with synchronous transitions. Let us do so by abstracting away the asynchronous groups such that they are each represented by a single node. The resulting generator is shown in Figure 7.3. We call the result, the head of the STS. From this we can create a two layer hierarchy: the top level contains the head of the STS, and the bottom level contains the set of corresponding asynchronous groups. The top level contains the synchronous transitions whereas the bottom level does not. This method of abstraction is very straightforward. Unfortunately, the reachability properties of the original STS are not preserved in the head. For example, by examining Figure 7.3, it seems that asynchronous group [2,3] is reachable. However, by examining the original system in Figure 7.2 it can be seen that asynchronous group [2,3] is in fact not reachable at all. From this exercise one can see that this naive abstraction method can result in an incorrect set of reachable asynchronous groups.

The problem stems from the fact that when abstracting a clustering of states into a single state, information (namely the interconnections between the states within the cluster) is abstracted away. The interconnection information within a cluster may not always be trivial with respect to reachability, so the abstraction does not always hold because necessary reachability information is obscured. When abstracting a cluster of states into a single state, the resulting state only contains the incoming and outgoing transitions of the original cluster. In the abstraction, it seems that all incoming transitions can lead to all outgoing transitions,
since they all belong to the same state. The problem with the naive abstraction, is that this is not necessarily the case; incoming transitions may not all lead to every single outgoing transition. Recall the example in Figure 7.2, and you will see that in asynchronous group [1, 1] not all the outgoing transitions are reachable from every incoming transition. In order to solve this problem, we introduce the notion of universal reachability and cluster the asynchronous groups into universally reachable components such that the head of a system does preserve the reachability properties of the system.

7.4 Universal Reachability

7.4.1 Introduction

We would like to abstract information away from a given state-tree-structure, resulting in the head of the system, such that it maintains the reachability properties of the original system. The problem with abstracting asynchronous groups into a single state directly is that not necessary all incoming transitions of an asynchronous group can reach all outgoing transitions. We therefore split the asynchronous groups into universally reachable clusters, which do maintain this property.

Definition 7.4.1 A cluster of a state-tree-structure is called universally reachable if
\[ \forall \text{input-output tuples } (x_{in}, x_{out}) \in X_{in} \times X_{out} \exists \text{a path } x_{in} \rightarrow x_{out}, \]
where \( X_{in} \) is the set of initial states and incoming transition states and \( X_{out} \) is the set of marked states and outgoing transition states.

Figure 7.4 shows an example of a URC cluster.

Figure 7.4: Example of a universally reachable cluster

Remark 7.4.2 Note that this universally reachable cluster is part of a generator. Also note that the cluster is blocking at state 6. We consider the union of initial states and destination states of incoming transitions as incoming states. We consider the union of marked states and source states of outgoing transitions as outgoing states. The important thing is that every incoming state can reach every outgoing state.

By clustering the asynchronous groups into universally reachable components, each resulting cluster contains only redundant reachability information. If every incoming tran-
situation can reach every outgoing transition in a cluster, it may as well be represented as a single state. This means that when these reachability clusters are abstracted into a single node, we do not lose any reachability information. Since the asynchronous groups of an STS are not necessarily initially universally reachable, we need to cluster them such that they are.

Because asynchronous groups are guaranteed not to contain any synchronous events, many properties of an asynchronous group can be verified by simply examining the components which compose it. Universal reachability is one property which can be verified in that manner. That is to say, universal reachability of an asynchronous group can be verified by examining the second level OR-superstates which compose it. This is formalized in the following lemma.

**Lemma 7.4.3** Given two automata $A_1$ and $A_2$ where $E(A_1) \cap E(A_2) = \emptyset$, that is to say they share no synchronous events, $P = A_1 || A_2$ is universally reachable if and only if both $A_1$ and $A_2$ are universally reachable.

**Proof** This follows directly from the fact that for any input/output tuple in $P$ there must be a corresponding pair in both $A_1$ and $A_2$. □

Therefore, in order to split an asynchronous group into universally reachable clusters, one can simply split the second level OR-superstates which compose it into universally reachable clusters. In order to ensure that all the asynchronous groups of an STS are clustered into universally reachable clusters, one can simply split all the second level OR-superstates of the STS into universally reachable clusters. In the following section, the method for properly clustering the second level OR-superstates is presented.

### 7.5 URC Algorithm

**Introduction**

The goal of reachability clustering is to split a second level OR-superstate into as few clusters as possible while maintaining universal reachability for each cluster. This will allow us to abstract away as much information as possible and only keep the necessary information. Note that a top level OR-superstate is simply a generator, and a second level OR-superstate is simply a sub generator of the top level. Below are the general steps for reducing a generator into its universally reachability clusters. This algorithm has been split into distinct steps for the sake of clarity. Given an generator $G$, it will produce a simplified generator with fewer states, where each state represents a universally reachable cluster of $G$. We call this the Universally Reachability Clustering (URC) algorithm, and the basic steps are delineated below.
Algorithm 5 Universal Reachability Clustering

1: Trim
2: Cluster strongly connected components (SCCs)
3: Remove transient edges
4: Merge paths leading to the same state

The resulting generator can be used as the head of the state-tree-structure. Below, the steps are explained in more detail. For clarity, a running example will be used to demonstrate the steps on a concrete second level OR-superstate. The OR-superstate with which we start is shown in Figure 7.5.

Step 1: Trim

The first step is to trim the generator. This means removing any states which are not reachable or coreachable. As we are only concerned about the paths between incoming states and outgoing states, we have no use for states outside of the trim of the generator. In our case, the entire generator is both reachable and coreachable, so no action is required.
7.5. URC Algorithm

Nonblockingness of Distributed State-Tree-Structures

Step 2: Cluster SCCs

The first step is to cluster the strongly connected components. A strongly connected component (SCC) is a set of states where from any state, once can reach every other state. Figure 7.6 is an example of a strongly connected component. These strongly connected components can be abstracted away and clustered into a single node. In terms of reachability, the information in a strongly connected component is meaningless; if you can reach one state in a SCC, you can reach all of them. Figure 7.7 shows the original generator with the strongly connected components removed. States 12 and 20 in the resulting generator are the ones that represent the original strongly connected components. This procedure is well known and can be done in linear time [6]. Specifically, it runs in $O(|V| + |E|)$, where $|V|$ is the number of states and $|E|$ is the number of edges.

![Figure 7.6: Example of a strongly connected component](image)

The resulting graph will have certain properties, outlined below.

**Proposition 7.5.1**

1. The resulting generator will be a directed-acyclical-graph (DAG). This means there are no cycles in the generator.
2. If the generator is blocking, there must exist a unmarked reachable blocking (deadend with no outgoing events) state.

**Proof** This follows directly from the definition of a connected component. By removing all connected components, one removes all cycles. If there are no cycles in a generator, live-lock is not possible, so if the generator is blocking, it must deadlock. □

Step 3: Cluster Transient Edges

In next step, we reduce the generator once more. Now the reduction is based on eliminating transient edges. Some states in a generator only have one incoming edge. Such an edge is considered transient. A transient edge is formally defined below.

**Definition 7.5.2** An edge $e \in E$ from a source $s \in X$ to a destination $d \in X$ is considered transient if there are no other edges with destination $d$ in the generator. Equivalently, the transition $\delta(s, e) = d$ is transient if there does not exist a transition of the form $d = \delta(x, a)$ for any $x \in X$ and for any $a \in E$.

An example of a transient edge can be seen in Figure 7.8. These edges are meaningless in terms of reachability. As there is no other input into the destination, there is no way that merging the source and destination nodes would result in an incoherent situation. Figure
7.9 shows the original generator with both strongly connected components removed and transient edges removed. This is also a straightforward reduction which can be done in linear time with respect to the number of nodes. More specifically, it runs in time $O(|V|)$, where $|V|$ is the number of states.

The algorithm for merging transient edges is shown below. The basic idea is to visit every state. If the state has only one incoming transition, merge it with the source state of that transition.
Algorithm 6 Transient Edge Clustering

1: Push all marked states onto Queue
2: while Queue is not empty do
3: \( C \leftarrow \) removed element from Queue
4: if \( C \) only has one incoming node to node \( N \) then
5: Merge \( N \) into \( C \) and push \( C \) pack onto Queue
6: end if
7: end while

The resulting generator also has certain properties, outlined below.

**Proposition 7.5.3**

1. Every node non-initial cluster, will have more than one incoming event.

2. The generator will still be a DAG.

**Proof**

1. If a node did have only 1 incoming event, it would have been merged by the algorithm.

2. Given a DAG, merging nodes will never create a cycle.
Step 4: Merge Multiple Branches

Once both the strongly connected components and the transient states have been clustered, there is still one more step which reduces the generator. Consider the situation where a state has multiple outgoing transitions. At every reachable state, there may be even more outgoing transitions. We call every possible path from the original state a branch. If all the branches from a certain state end up meeting again at a second state, the reachability information contained in the intermediary states is redundant; if you reach the first state, you know for sure you must pass into the second state eventually. The basic idea for this reduction is outlined below.

1. For every node, visit it only when all incoming edges have been visited, starting with the initial states.
2. Keep track of global state information indicating for every node, how many branches it has reached. Store this in the table Global Branch Counts.
3. Keep track of local information per node for every edge, indicated which nodes can reach it. We mark an edge with a node if it can be reached by that node.

Figure 7.9: Generator with no transient edges
4. When visiting the current node, if the number of incoming edges marked by a certain 
node, \( N \), equals the number of total branches \( N \) can reach, then merge node \( N \) with 
the current node.

This algorithm can run in linear time if the nodes are visited in the right order. Below is 
the pseudocode of the algorithm.

**Algorithm 7 Branch Merging**

1. \( nodesToVisit \leftarrow \) incoming states \{Initial set is union of initial states and incoming arrows\}
2. \( \textbf{while} \ nodesToVisit \) is not empty \( \textbf{do} \)
3. \( x \leftarrow nodesToVisit.pop \)
4. \( x.isVisited \leftarrow \text{true} \)
5. \( \textbf{for all} \) descendants \( d \) of \( x \) \( \textbf{do} \) \{Do not visit a node until all of it’s parents have been visited\}
6. \( \textbf{if} \) All parents of \( d \) are visited \( \textbf{then} \)
7. \( \quad \text{add } d \text{ to } nodesToVisit \)
8. \( \textbf{end if} \)
9. \( \textbf{end for} \)
10. \{When to merge\}
11. \( \textbf{for all} \) Nodes \( n \) \( \textbf{do} \)
12. \( \quad \textbf{if} \) (\# of incoming links marked with \( n \)) = (global branch count for \( n \)) \( \textbf{then} \)
13. \( \quad \text{merge } v \text{ into } n \)
14. \( \quad \text{add } n \text{ to } nodesToVisit \)
15. \( \textbf{end if} \)
16. \( \textbf{end for} \) \{Maintain proper labels and counts\}
17. \( \text{For every incoming label, decrement global branch count} \)
18. \( \text{For every outgoing label, increment global branch count} \)
19. \( \text{Mark every outgoing branch} \)
20. \( \textbf{end while} \)

**Example**

In this section we continue to reduce the sample generator by performing the steps of the 
Branch Merging algorithm. The algorithm is quite complex, so every step is explained 
thoroughly. We begin by visiting the initial state \([1,2,3]\) and mark it’s outgoing edges. We 
also update the global branch count for state \([1,2,3]\) to 2 because there are two outgoing 
paths from the node. Table 7.1 is the global branch count table and Figure 7.10 depicts the 
resulting generator after iteration 1.

<table>
<thead>
<tr>
<th>Node</th>
<th>[1,2,3]</th>
<th>4</th>
<th>5</th>
<th>[6,7,8,9,10]</th>
<th>11</th>
<th>[12,13,14]</th>
<th>[15,16,17]</th>
<th>[18,19,20,21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Count</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Global Branch Counts: Iteration 1
In the next iteration, we visit state 5. We update its global branch count to 1, since there is 1 outgoing edge. We also mark that edge. Table 7.2 is the global branch count table and Figure 7.11 depicts the resulting generator after iteration 2.

<table>
<thead>
<tr>
<th>Node</th>
<th>[1,2,3]</th>
<th>4</th>
<th>5</th>
<th>[6,7,8,9,10]</th>
<th>11</th>
<th>[12,13,14]</th>
<th>[15,16,17]</th>
<th>[18,19,20,21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Count</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: Global Branch Counts: Iteration 2
In iteration 3, we visit state 4, since all of its incoming edges have been visited. We set the global branch count for state 4 to 1, since there is one outgoing path, and mark the outgoing edge. Table 7.3 is the global branch count table and Figure 7.12 depicts the resulting generator after iteration 3.

<table>
<thead>
<tr>
<th>Node</th>
<th>1,2,3</th>
<th>4</th>
<th>5</th>
<th>6,7,8,9,10</th>
<th>11</th>
<th>12,13,14</th>
<th>15,16,17</th>
<th>18,19,20,21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Count</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.3: Global Branch Counts: Iteration 3
Iteration 4 is a big step. We visit state [6,7,8,9,10] because all of its incoming edges have been visited. There are 2 incoming edges marked with [1,2,3], 1 incoming edge marked with 4, and one incoming edge marked with 5. We compare this to the global branch counts. The branch count for [1,2,3] is 2, the branch count for state 4 is 1, and the branch count for state 5 is 1. The number of marked edges and global branch counts match exactly for every state. This means that we can merge the current state [6,7,8,9,10] with states [1,2,3], 4, and 5. After this, we mark the outgoing edges of the newly created state and update its branch count to 2, since there are 2 outgoing edges. Table 7.4 is the global branch count table and Figure 7.13 depicts the resulting generator after iteration 4.

<table>
<thead>
<tr>
<th>Node</th>
<th>[1,2,3,4,5,6,7,8,9,10]</th>
<th>11</th>
<th>[12,13,14]</th>
<th>[15,16,17]</th>
<th>[18,19,20,21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Count</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.4: Global Branch Counts: Iteration 4
In iteration 5 we visit state [15,16,17]. We mark its outgoing edges and set its global branch count to 2, since there are 2 outgoing edges. There is nothing to merge since the number of marked incoming edges does not match the global branch count for any state. Table 7.5 is the global branch count table and Figure 7.14 depicts the resulting generator after iteration 5.

<table>
<thead>
<tr>
<th>Node</th>
<th>[1,2,3,4,5,6,7,8,9,10]</th>
<th>11</th>
<th>[12,13,14]</th>
<th>[15,16,17]</th>
<th>[18,19,20,21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Count</td>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.5: Global Branch Counts: Iteration 5
In iteration 6 we visit state \([18,19,20,21]\). We mark its outgoing edges and set its global branch count to 2, since there are 2 outgoing edges. There is nothing to merge since the number of marked incoming edges does not match the global branch count for any state. Table 7.6 is the global branch count table and Figure 7.15 depicts the resulting generator after iteration 6.

![Figure 7.14: Iteration 5](image-url)

<table>
<thead>
<tr>
<th>Node</th>
<th>[1,2,3,4,5,6,7,8,9,10]</th>
<th>11</th>
<th>[12,13,14]</th>
<th>[15,16,17]</th>
<th>[18,19,20,21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Count</td>
<td>2</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.6: Global Branch Counts: Iteration 6
In iterations 7 and 8, we visit states 11 and [12, 13, 14]. We set there global counts to 1 since there is 1 outgoing edge. Table 7.7 is the global branch count table for iteration 8.

<table>
<thead>
<tr>
<th>Node</th>
<th>[1,2,3,4,5,6,7,8,9,10]</th>
<th>11</th>
<th>[12,13,14]</th>
<th>[15,16,17]</th>
<th>[18,19,20,21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Count</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.7: Global Branch Counts: Iteration 8

At this point we have visited all the states. It is not possible to merge anything else, because the number of marked incoming edges does not equal the global branch counts for any states. This means that we have merged as much as possible, and the algorithm is finished. The final generator is shown in Figure 7.16. The efficiency gain is quite good for this example. We started off with a second level OR-superstate with 21 states, and we were able to condense it into only 5 states. Each of these 5 clusters are universally reachable.
7.5. URC Algorithm

7.5.1 Remarks

This Universal Reachability Algorithm optimally clusters a generator; it creates as few clusters as possible, each one as large as possible. This is formalized below.

**Proposition 7.5.4** The URC clustering algorithm produces an optimally clustered generator. That is, if any more nodes were to be merged, a cluster would be created that is not universally reachable.

**Proof** This follows directly from the algorithm. If a node can still be merged into a universally reachable component, then it will. The algorithm only stops when no more merges are possible. □

Additionally, the URC algorithm produces a unique result. In fact, any optimally clustered graph is unique. This concept is also formalized below.

**Proposition 7.5.5** An optimally universally reachable clustered generator is unique. That is, given a generator, there is only one correspond optimal universally reachable clustered graph.

**Proof** Consider a graph $G$ with a node $x$ having the potential to merge with any of the states in the subset $M$. 

---

Figure 7.16: Generator maximally clustered into universally reachable components
7.6. Nonblocking Formulation

If the clustering solution were not unique, it would mean that state $x$ could only merge exclusively with one strict subset of $M$ for any given solution. Every different subset of $M$ with which $x$ can merge, would then yield a different clustering solution.

However, the conditions for merging a state rely solely on the incoming edges of the merging state as well as the global branch count of the node to be merged with. Merging two states does not change either of these two variables. It might only rearrange the source nodes of certain transitions to the newly created merged state. Therefore merging two nodes has absolutely no effect on the merge ability of any other nodes. Therefore if node $x$ has the potential to cluster with multiple states of $M$ exclusively, it will always be able to cluster with all of the states in $M$. This means there will never be a situation required for different solutions of clustering. □

Therefore, the URC algorithm results in a unique optimally clustered graph.

7.5.2 Result

Let us reconsider the example in Figure 7.2. Let us split the second level OR-superstates into universally reachable clusters. The result is depicted in Figure 7.17. Note that the asynchronous groups (denoted by blue dashed rectangles) are all universally reachable as well. Now consider the head of this system, shown in Figure 7.18. Each of the universally reachable components is abstracted away into a single node. Note that every reachable asynchronous group in the head is also reachable in the original STS; the URC algorithm has successfully clustered the asynchronous groups such that the head preserves the proper reachability information.

7.6 Nonblocking Formulation

With the notion of universal reachability, we can now reformulate the condition for nonblockingness. The universally reachable asynchronous groups of a STS can be used to define a different set of conditions for nonblockingness. Rather than comparing the reachability and coreachability of the monolithic system, we can instead check the nonblockingness of the reachable asynchronous groups. This concept is formalized in the Theorem 7.6.1.

**Theorem 7.6.1** A state-tree-structure $G$ is nonblocking if and only if the reachable asynchronous groups are locally nonblocking and the head is nonblocking.

**Proof** If $G$ is blocking, there must be at least one dead-lock state or live-lock cycle somewhere in the system. That is, a dead-lock state or live-lock cycle must either be within an asynchronous group or in the head. Therefore $G$ is blocking if and only if the head of an asynchronous group is blocking or one of the asynchronous groups is nonblocking. Therefore, if neither the head nor the asynchronous groups are blocking (i.e. they are both
This theorem has an important implication: we no longer have to perform a parallel composition on the asynchronous groups in order to check for nonblockingness.

**Remark 7.6.2** Note that when considering the nonblockingness of an asynchronous group, we consider the set of initial states to include the initial states of the system as well as the incoming transition states. We consider the set of marked states to include the marked states of the system as well as the outgoing transition states.

### 7.7 Putting It All Together

This reformulation of the nonblockingness check gives us a different perspective on how to check the nonblockingness of an STS. This is significant because of the following lemma.

nonblocking) $G$ cannot be blocking and vice versa. □
Lemma 7.7.1 Given two automata $A_1$ and $A_2$ where $E(A_1) \cap E(A_2) = \emptyset$, that is to say they share no synchronous events, $A_1 \parallel A_2$ is nonblocking if and only if both $A_1$ and $A_2$ are nonblocking.

Proof This follows directly from the fact that every transition in the individual components is also possible in the parallel composition and vice versa. □

Because the second level OR-superstates which compose an asynchronous group contain no synchronous events, it is possible to check the nonblockingness of an asynchronous group simply by checking the nonblockingness of the OR-superstates which compose it. This provides a big performance gain because it is not necessary to check all the states of the parallel composition explicitly. Checking the nonblockingness of the asynchronous groups can now be done in linear time with respect to the number of components.

Consider an asynchronous group composed of two second level OR-superstates. In order to verify nonblockingness of this asynchronous group, two things need to be verified with respect to the OR-superstates.

1. The reachability set must be a subset of the coreachability set for both OR-superstates.

2. For every combination of outgoing transitions in the two OR-superstates, there must be an actual synchronous event.

Remark 7.7.2 The second check only takes linear time with respect to number of edges in the OR-superstates as there is a bound on the number of synchronous events in a system; There can be no more synchronous events in an asynchronous group than the maximum number edges in one of the component OR-superstates.
7.8 The Procedure

We have now introduced the process of combining multiple STSs properly and how to cluster them into universally reachable components. Now we explain how to use this to efficiently check for nonblockingness. The basic procedure for checking the nonblockingness of a distributed system described with multiple state-tree-structures is described below. From henceforth, this will be referred to as the Universal Reachability Method (URM) method for checking nonblockingness of state-tree-structures.

Procedure 7.8.1

1. When given a distributed system of multiple modules, one must first combine it into a single model. If each component of the system is modeled as a STS, simply create a new STS by combining the components under an AND-superstate.

2. The newly created STS, the model of the entire system may not necessarily be well-formed, even though its components may be. Ensure well-formedness by rearranging the synchronous events using the flattening algorithm. This can be done in linear time in the number of state-tree nodes.

3. One must now cluster the second level OR-superstates of the STS into universally reachable components such that the head of the STS maintains the proper reachability information. This can be done in linear time in the number of states in the OR-superstates. The resulting asynchronous groups can be abstracted into a single state, forming the proper head of the STS.

4. Use the head in order to compute the reachable asynchronous groups. Also check if the head is nonblocking. This runs in linear time in the number of asynchronous groups. Therefore the efficiency gain here depends on how well the URC clustering works.

5. Compute whether or not each reachable asynchronous group is nonblocking. This can be done independently and in parallel for each asynchronous group as they have no common events. This can be done checking the nonblockingness of the universally reachable clusters which compose the asynchronous group. This entire step is done in time linear in the number of states in the clusters.

6. The entire system is nonblocking if and only if each of the reachable asynchronous groups is nonblocking and the head is nonblocking.

In the end, instead of doing the computation on the entire monolithic system, one must only analyze the components that make up the asynchronous groups in addition to the corresponding head of the STS. An example of what needs to be analyzed can be seen in Figure 7.19. Note that there is no longer a need to do the parallel composition of the individual components. In our method, it is only effective if the head is smaller than the parallel composition of the components.
Each of the steps is done in linear time, with the goal of reducing the input size to a problem which can currently be solve at best in exponential time. The performance of overall nonblockingness check depends on how much we small we can make the head (how much we can cluster the graph into universally reachable clusters). The performance of this method will be examined in the following chapter.
Chapter 8

Results

In the examples from previous chapters it has been seen that the universal reachability method for checking nonblockingness can be very efficient. In this chapter a more thorough examination of performance is presented. First a computational analysis of the universal reachability clustering algorithm is performed for a set of random graphs. This is followed by an application of the entire universal reachability method to a realistic practical problem. We conclude with some qualitative remarks on performance.

8.1 Computational Analysis

The universal reachability method (URM) for checking nonblockingness is a heuristic based on the structure of a given state-tree-structure. Recall that the universal reachability clustering (URC) algorithm is part of the URM. Because there is no $O(\cdot)$ running time gain, the URC algorithm was run on a set of random automata to estimate an average performance gain. The URC algorithm was implemented in Java such that it could be easily tested on a wide range of generators. The Java implementation takes a generator as an input and computes the resulting clustered generator for every step of the algorithm. These intermediary generators can be exported into a .dot format which can be used by the graph visualizer program GraphViz such that one can actually see the process. Consider an initial generator with $n_o$ nodes. The clustered generator resulting from the URC algorithm will have $n_c$ nodes where $n_c \leq n_o$. As a measure of performance, the number of nodes in the clustered generator is compared to the number of nodes in the original generator providing the percent reduction of the original graph. This is formally defined below.

Definition 8.1.1 Percent Reduction = \( \frac{N_o-N_c}{N_o} \)

For an in-depth analysis on the performance of the URC algorithm, it was run over a large set of random generators. In order to emulate realistic models, the random generators were restricted to have certain properties. Most importantly, the randomly generated generators were all reachable; this is quite common for models of real-life systems. Additionally, the number of initial and marked states in the generators were held to certain levels. Naturally not all system models in the world follow these properties, but these conditions are
an attempt to restrict the test-space based on reasonable assumptions. For every situation, 1,000 random graphs were generated and the resulting average percent reduction in generator size was then plotted.

There were several variables of interest to examine. Primarily, the edge density of the generators was thought to have a strong effect on how much it could be clustered. In this setting, edge density is understood to mean the ratio of edges in the generators to the maximum possible number edges given the number of nodes. In our case the maximum number of edges is the square of the number of states, as the structure of a generator is equivalent to that of a directed graph allowing self loops. Edge density is formally defined in [9].

**Definition 8.1.2** Consider an undirected graph $G$ with $E$ edges and $N$ nodes, allowing self loops. Define the Edge Density $D = \frac{|E|}{|N||N|}$.

Since our random test generators were guaranteed to be reachable, the lower bound on edge density was therefore $\frac{1}{|N|}$ for a given generator, where $|N|$ is the number of states. In Figure 8.1 the percent reduction of a generators with 50 nodes, 5 initial states, and 10 marked states is shown.

**Remark 8.1.3** With a graph size of 50, the sparsest (with fewest transitions) generator has an edge density of .02. The URC algorithm performs extremely well with an average percent reduction of 95%. However there is a slight drop in efficiency as the edge density increases. As the edge density continues to increase past a certain point, the percent reduction increases again dramatically. This is most likely due to the fact that as the edge density increases, the entire generator is likely to become a single large connected component.

We extended this test to generators of different sizes. Figure 8.2 shows the results in a 3D plot using generators with 10 through 100 states.

**Remark 8.1.4** The clustering performance follows the same pattern regardless of generator size. The average percent reduction in the number of states is about 95%. There there is a dip in percent reduction as edge density increases before it drastically shoots up for all sizes.

Another interesting variable to analyze is the number of marked states. Figure 8.3 shows how the URC algorithm performs with regard to the number of marked states. The edge density is set at the worst case of 0.3 and the number of initial states is held at 3. The number of marked states ranged from 5 to 50.

**Remark 8.1.5** Note that the clustering still works remarkable well when every single node is a marked state. The important thing to take away is that the original generators often get reduced by 90%. That is a big achievement as computing the reachable set of the reduced generator would be 10 times faster than on the original.
8.2 Production Cell Example

In this section we apply the entire universal reachability method on a real life practical example. In his Ph.D. thesis, which first introduced state-tree-structures, Ma Chuan analyzed a production cell as a practical example. The production cell is a system composed of multiple modules involving conveyers and robotic arms. A complete description of the system can be found in [18]. This example is of interest to formal methods researchers because it is complicated yet realistic and manageable. The state-tree-structure model of the production cell as modeled by Ma Chuan can be seen in Figure 8.4, copied from [18].

Remark 8.2.1 Note the compact representation offered by state-tree-structures. This is a complex example with many states, yet it can be fully represented on one page. For comparison, the equivalent generator representation has 222,363,648 states. Although the graphical representation is quite simple, there is quite some computation needed to find all the reachable and coreachable states. Below are some properties of this STS.

- 6 top level parallel components
- 16 top level OR-components
8.2. Production Cell Example

The structure offers some compression, but it can be done even better modularly. We first examine the top level OR-components, which must be Universally Reachably Clustered. Each resulting cluster can then be treated as an independent system and can be analyzed for nonblockingness in parallel. Only the reachable clusters must be examined, and to compute these we use the head of the STS. In Figure 8.5, the head of the original STS is shown.

**Remark 8.2.2** Note that the equivalent generator representation of the head only has 1152 states, which is no trouble for any modern computer to handle. Rather than examining the Production Cell as a monolithic system, we break it down into an upper layer with 1152 states and 22 independent lower level components, of which only the reachable ones need to be examined. Moreover, all 22 lower level states can be checked directly from their respective components, simplifying computations even more. In fact there are only 272 states in the lower level components which need to be checked. These lower components are still represented as state-tree-structures, ensuring that their checks for nonblockingness...
8.3 Qualitative Analysis

The Universal Reachability Method works by separating the shared events from the asynchronous events. Since the components with shared events have to be combined by the parallel composition operation while the states without shared events can be computed independently, it is desirable to have as few states as possible containing shared events and as many states as possible with no shared events. Usually a system with few synchronous events results in a much more efficient nonblockingness check than a system with many shared events. However, the number of shared events is not the only metric. If there were many synchronous transitions between the same sets of states, one could still group the remaining states into an asynchronous group. One must also look at how the transitions are distributed throughout the system. In general, the more the events are scattered throughout the system, the worse the Universal Reachability Method will perform. This boils down to a simple measure, the higher the number unique states involved with a synchronous transition, the worse the Universal Reachability Method will tend to perform. As with any

![UCR Algorithm Performance on a Random 50 Node Automaton](image)

Figure 8.3: URC Algorithm Performance on Random 50 Node generators with Regard to Number of Marked States

stay effective as well. The total number of states which need to be checked is merely 1424, which is a 99.99% reduction of the original system.
distributed system, the more information (events) you share, the more complex your system will be.

This method is most efficient when the states can be grouped into very large groups containing no synchronous events. The best case would obviously be when there are no shared events at all, and all checks can be done independently. In general, this approach would work very poorly in systems where there are many synchronous events spread out evenly throughout the state space. The worst case scenario would be a system with only shared events as nothing can be grouped and abstracted.

8.4 General Comments

It is important to note that this method does not solve the state-explosion problem. Similarly to state-tree-structures, it is an approach which simplifies the problem as much as possible based on structural properties. This approach works by bringing all the shared information to the top level and abstracting everything else away. Therefore, the more synchronous events there are between different state trees, the less effective this approach is. In fact, it is certainly possible that this approach may result in more work if there are too many synchronous events and the OR-superstates in the STS are not efficiently defined. STS gain efficiency based on the structure of the system, whereas this distributed approach gains efficiency by keeping non synchronous events independent from each other. In most real world distributed systems, the number of states involved with a shared event, is much less than the total number of states in the system, which is a property we take advantage of.
Figure 8.4: STS of the Production Cell
8.4. General Comments

Results

![Figure 8.5: Updated Abstracted Parallel Composition](image)

Figure 8.5: Updated Abstracted Parallel Composition
Chapter 9

Conclusion

9.1 Summary

An innovative approach has been presented for tackling the complexity problem plaguing supervisory control of discrete-event systems. By using the structural information contained within a state-tree-structure model, we are able to separate the states involved with synchronous transitions from states not involved with synchronous transitions. A two-level hierarchy is created where the top level contains an abstracted view, where each state represents a cluster of states containing no shared events and where every incoming transition can reach every outgoing transition. That is, every state in the top level represents a universally reachable asynchronous group. The bottom level contains the corresponding set of these universally reachable asynchronous groups. The top level is made in such a way that is preserves the reachability properties of the system. This allows us to reformulate the nonblockingness condition; the system is nonblocking if and only if both the top level is nonblocking and the components of the lower level are nonblocking. As the components in bottom level are guaranteed not to have any shared events, they can be checked for nonblockingness independently and in parallel, drastically reducing computation times.

The algorithms we introduced for decomposing a state-tree-structure into this hierarchy provide for an optimal clustering of the components resulting in an efficient top level. These decomposition operations all run in linear time with respect to the number of components in a modular system. With this computation it is now possible to avoid the parallel composition (and state-explosion problem) for a subset of states which are not involved with synchronous transitions. This method has been applied to a practical example and has been seen to reduce the number of states needed to be examined to verify nonblockingness by 99.99 percent.

9.2 Remarks

Although state-tree-structures are remarkably efficient, they do not directly solve that exponential complexity problem. It is a framework which allows the modeler to easily incorporate the structure of a system into the model. State-tree-structures merely use this
structure to condense the information into binary decision diagrams for efficient synthesis of a controller. In the same vain, our Universally Reachable Method also uses structural information of state-tree-structures to make supervisor synthesis more efficient. It is important to note that the effectiveness of this new approach depends on the structure of the state-tree-structure; the more synchronous events there are, the more information is shared amongst the components which make up a system, and the smaller the asynchronous groups will tend to be. Therefore, for optimal performance, one would strive to have a system with few synchronous events such that it is possible to cluster the states into large groups containing no synchronous events. Fortunately, in many real world systems, the number of shared events is considerably smaller with respect to the total number of events.

9.3 Future Work

This approach tackles the problem of verifying nonblockingness for distributed systems modeled with state-tree-structures. A natural extension would be to treat the aspect of controllability and combine this with the nonblockingness verification to create an efficient control synthesis algorithm. Our approach separates synchronous events from non synchronous events. State-tree-structures help with this as they have information encoded in their structure about which sets of states are involved with synchronous events. Perhaps an efficient algorithm exists to partition any system in such a manner, which would allow our approach to be used for other types of models as well.


