Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Kronecker Algebra Based Analysis of Shared Memory Concurrent Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Robert Mittermayr
Registration Number 9825671

to the Faculty of Informatics

at the TU Wien

Advisor: Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Johann Blieberger

The dissertation has been reviewed by:

Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Johann Blieberger	Associate Prof. Dipl.-Ing. Dr. techn. Bernd Burgstaller

Vienna, 17th August, 2016

Dipl.-Ing. Robert Mittermayr

Kronecker Algebra Based Analysis of Shared Memory Concurrent Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Robert Mittermayr
Matrikelnummer 9825671

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Johann Blieberger

Diese Dissertation haben begutachtet:

Ao. Univ.-Prof. Dipl.-Ing.
Dr. techn. Johann Blieberger

Associate Prof. Dipl.-Ing.
Dr. techn. Bernd Burgstaller

Wien, 17 August, 2016

Dipl.-Ing. Robert Mittermayr

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Robert Mittermayr
Johannesstr. 6
3304 St. Georgen am Ybbsfelde

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 17 August, 2016

Dipl.-Ing. Robert Mittermayr

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis.

I sincerely thank Prof. Bernd Burgstaller for his valuable hints and tips for further improvements. He often brought new aspects into my dissertation.

I am deeply indebted to my supervisor Prof. Johann Blieberger. His guidance, help, and stimulating suggestions helped me in all the time of research for this thesis.

Especially, I would like to thank my lovely wife Kathrin for all the support that she gave me during the years I have been working on this thesis. This work would not have been possible without her patient love.

Für meine Eltern Erna und Josef Mittermayr

Abstract

Program analysis of multi-threaded software is still a challenge. Beside other reasons, this comes from the fact that synchronization has to be taken into account. In particular, a suitable graph based model has been missing. In this dissertation, we introduce a novel graph based approach. It is based on the idea that thread interleavings can be studied with a matrix calculus. This is a novel approach in this research area. Our sparse matrix representations of a program are manipulated using Kronecker algebra. In the course of this dissertation we extend it and prove some important properties. The underlying graph – which we call concurrent program graph (CPG) – represents a multi-threaded program and plays a similar role for concurrent systems as control flow graphs do for sequential programs. Thus a suitable graph model for program analysis of multi-threaded software is set up. Due to synchronization, e.g., via synchronization primitives like semaphores and barriers, it turns out that often only very small parts of the resulting graph are actually needed, whereas the rest is unreachable. A lazy implementation of matrix operations ensures that unreachable parts are never calculated. This speeds up processing significantly and shows that our approach is very promising. We parallelized this lazy algorithm and thus gain additional speedup by exploiting the power of modern multi-core architectures.

In this dissertation, we will show how we use CPGs to detect deadlocks statically in concurrent programs.

We present a new synchronization construct modeling barriers. By applying this, we are able to statically analyze Ada multi-tasking programs that employ barriers for synchronization issues. It turns out that our Kronecker algebra implementation can be used out-of-the-box for CPGs using such barrier synchronization primitives.

In addition, we show that CPGs can be used as a basis for a worst-case execution time (WCET) analysis of multi-threaded programs. We employ a generating functions based approach for setting up data flow equations which are then solved by well-known elimination based data flow analysis methods. With this approach, we are able to calculate the WCET (including stalling times) of multi-threaded programs with a non-linear function solver. Non-linearity turns out to be inherent to the multi-threaded WCET problem.

Finally, we show how our Kronecker algebra based approach can be adopted in the field of railway systems. For multiple trains in a railway disposition system and based on given routes, it can be used to suggest deadlock-free movements only. Additional

constraints, such as overtaking and train connections, may be taken into account. Our railway approach was already extended by others, e.g., to save energy by minimizing stop and go of trains.

Kurzfassung

Programmanalyse von nebenläufigen Programmen ist immer noch eine Herausforderung. Das kommt unter anderem daher, dass Synchronisation berücksichtigt werden muss. Im Speziellen fehlte bisher ein geeignetes Graphenmodell. Mit dieser Dissertation führen wir einen graphenbasierten Ansatz ein. Dieser fußt auf der Idee, dass zeitlich verschränkte Abarbeitung (*thread interleavings*) durch einen Matrizenkalkül betrachtet werden kann. Dies ist neu in diesem Forschungsgebiet. Die dünnbesetzten Matrizen eines Programmes werden mittels Kronecker Algebra-Operationen manipuliert. Im Zuge dieser Dissertation wird diese erweitert und manch wichtige Eigenschaft bewiesen. Der zugrundeliegende Graph, den wir *concurrent program graph* (CPG) nennen, repräsentiert ein nebenläufiges Programm und spielt eine ähnliche Rolle wie Kontrollflussgraphen für sequentielle Programme. Daher ist er ein geeignetes Modell für die Programmanalyse von nebenläufigen Programmen. Es stellt sich heraus, dass durch Synchronisation (z.B. durch Synchronisationsprimitive wie Semaphoren oder Barrieren) oft nur ein sehr kleiner Teil des resultierenden Graphen nötig und der Rest unerreichbar ist. Eine verzögerte (im Englischen oft als "lazy" bezeichnete) Auswertung der Matrixoperationen stellt sicher, dass unerreichbare Teile nie berechnet werden. Das beschleunigt die Berechnung signifikant und zeigt, dass unser Ansatz vielversprechend ist. Die Parallelisierung dieses Algorithmus für Mehrkern-Architekturen ermöglichte eine signifikante Laufzeit-Reduktion.

In dieser Dissertation zeigen wir, wie CPGs zum statischen Finden von Deadlocks in nebenläufigen Programmen verwendet werden können.

Des Weiteren führen wir ein neues Synchronisationskonstrukt, das Barrieren modelliert, ein. Dieses Konstrukt ermöglicht die statische Analyse von nebenläufigen Ada-Programmen, die Barrieren zur Synchronisation verwenden. Programme, die auf diese Synchronisationsprimitive bauen, können mit unserer völlig unveränderten Kronecker Algebra-Implementierung analysiert werden.

Zusätzlich zeigen wir, dass CPGs als Basis für eine Analyse der maximalen Laufzeit – im Englischen oft als *worst-case execution time* (WCET) bezeichnet – verwendet werden können. Wir verwenden einen Ansatz, der auf erzeugende Funktionen basiert. Datenflussgleichungen werden durch etablierte (sogenannte *elimination based*) Datenflussanalysemethoden gelöst. Mit diesem Ansatz wird es möglich, die WCET von nebenläufigen Programmen zu berechnen. Diese WCET inkludiert auch Zeiten, in denen Threads blockiert sind. Mittels eines nicht-linearen Funktionslöser werden die Gleichungen gelöst. Es zeigt sich, dass nebenläufigen WCET-Problemen Nicht-Linearität inhärent ist.

Schließlich zeigen wir, wie unser Kronecker Algebra-Ansatz im Bereich der Eisenbahn verwendet werden kann. Bei gegebenen Fahrstraßen wird es dadurch Dispositionssystemen möglich, für mehrere Züge automatisch deadlockfreie Zugbewegungen vorzuschlagen. Zusätzliche Bedingungen, wie z.B. Überholungen und Zugverbindungen, können dabei beachtet werden. Dieser Ansatz wurde bereits von anderen erweitert, um z.B. durch Minimierung von Brems-/Beschleunigungszyklen energiesparend zu fahren.

Contents

Abstract xi

Kurzfassung xiii

List of Figures xvii

List of Tables xix

1 Introduction 1
1.1 Motivation . . . 1
1.2 Thesis Overview and Contributions . . . 2
1.3 Outline . . . 4

2 Preliminaries 7
2.1 Overview . . . 7
2.2 Semiring . . . 8
2.3 Control Flow Graphs . . . 8
2.4 Semaphores . . . 9
2.5 Edge Splitting . . . 11
2.6 Basic Matrix Notations and Operations . . . 14
2.7 Correspondence between Matrices and Directed Graphs . . . 15
2.8 Limitations . . . 16

3 Kronecker Algebra – A Matrix Calculus 17
3.1 Kronecker Product / Zehfuss Product . . . 17
3.2 Kronecker Sum . . . 20

4 Concurrent Program Graphs 27
4.1 Generating a Concurrent Program's Matrix . . . 29
4.2 Determining Entry and Final Nodes of CPGs . . . 30
4.3 ◦-Operation and Synchronization . . . 31
4.4 Unreachable Parts Caused by Synchronization . . . 32
4.5 Properties of CPGs . . . 33
4.6 Efficient Implementation of the ◦-Operation . . . 36

4.7 Lazy Implementation of Kronecker Algebra 39
4.8 CPGs are Irreducible 40

5 Deadlocks **43**
5.1 Deadlock Example 44

6 Examples and Empirical Data **49**
6.1 Deadlock Example 49
6.2 Modeling Static Fork Join 50
6.3 Client-Server Example 52
6.4 A Data Race Example 55
6.5 Empirical Data 58

7 Static Analysis of Barriers **61**
7.1 Barriers 62

8 Worst-Case Execution Time Analysis **71**
8.1 Worst-Case Execution Time Analysis on RCPGs 74
8.2 Example 81

9 Deadlock Avoidance for Railway Systems **97**
9.1 Deadlocks in Computers vs. Deadlocks in Railway Systems 99
9.2 Railway System Model 100
9.3 A Simple Example 102
9.4 Avoiding Deadlocks under Additional Constraints 106
9.5 Extensions of the Standard Railway Model 106
9.6 Concluding Remarks 108

10 Related Work **111**
10.1 Kronecker Algebra 111
10.2 Data Structures Modeling Concurrent Systems 112
10.3 Static Analysis of Barriers 112
10.4 Worst-Case Execution Time Analysis 112
10.5 Railway Systems 114

11 Conclusion **117**
11.1 Summary 117
11.2 Outlook 119

A The State Explosion Problem **125**

Bibliography **129**

List of Figures

2.1 Pseudocode Factorial 9
2.2 CFG of Factorial 10
2.3 Semaphores 10
2.4 Semaphores with Non-blocking v-operations 11
2.5 Edge Splitting for k Synchronization Primitive Calls 13
2.6 Edge Splitting for k Shared Variable Accesses 14

3.1 Simultaneous Executions via Kronecker Product $C \otimes D$ 19
3.2 Simultaneous Executions via Kronecker product $E \otimes F$ 21
3.3 The Graphs of C, D, and $C \oplus D$ 23
3.4 A Simple Cartesian Product Graph ($C \oplus D$) 25

4.1 Overview 28
4.2 Mutual Exclusion Example 34
4.3 Unreachable Parts of the Mutual Exclusion Example 35
4.4 Example Program with Self-Deadlock 37
4.5 CPGs are Irreducible 40

5.1 Deadlock Example 46
5.2 Some Unreachable Parts of the Deadlock Example 47

6.1 A Second Deadlock Example 51
6.2 RCFGs for Example Consisting of T_1 and T_2 52
6.3 Adapted RCFGs for Example Consisting of T_1 and T_2 53
6.4 RCPG of the T_1-T_2-System 53
6.5 Client-Server Example 54
6.6 Data Race Example 55
6.7 RCFGs for Data Race Example 56
6.8 CPG of Data Race Example 58
6.9 Improvements via a Parallel Version 60

7.1 RCFGs of Tasks T1, T2, and T3 Using a Barrier 63
7.2 CPG for Program Consisting of T1 and T2 64
7.3 CPG for Program Consisting of T1, T2, and T3 66

7.4 CFGs for Example Consisting of TL1 and TL2 67
7.5 CPG for Program Consisting of TL1 and TL2 68
7.6 CPG of Reusable Barrier Solution using Semaphores 69
7.7 CPG of Non-Reusable Barrier Solution using Semaphores 70

8.1 RCFGs of Threads A and B 75
8.2 Kronecker Sum $A \oplus B$ of Threads A and B 76
8.3 RCPG 82
8.4 Adapted RCPG 85

9.1 Graph $(C \oplus D) \otimes S$ 101
9.2 A Simple Example 102
9.3 Possible Train Movements 105
9.4 A More Elaborate Example 107
9.5 Resulting NDLS Graph 109

A.1 Interleavings for Fixed Number of Statements per Thread 128
A.2 Interleavings for a Variable Number of Threads and Statements per Thread 128

List of Tables

6.1 Empirical Data 59

8.1 WCET for $\ell = 2$ and Multiple Values of τ_c 94

9.1 Differences between the Deadlock Problem in Computer and Railway Systems 99
9.2 Routes of Trains L_1, L_2, and L_3 103
9.3 Four Possible Outcomes of an Analyzing Algorithm 104

A.1 Computation Results of Program $\mathcal{P}$ 126

CHAPTER 1

Introduction

"The hardest thing is to go to sleep at night, when there are so many urgent things needing to be done. A huge gap exists between what we know is possible with today's machines and what we have so far been able to finish."

– DONALD ERVIN KNUTH, U.S. American computer scientist
and ACM Turing Award winner 1974, 1938-

This dissertation is about analysis of concurrent programs including worst-case execution time analysis and deadlock detection. In order to generate graphs representing concurrent programs, we adopt the so-called Kronecker algebra which we explore and extend in the course of this thesis. In addition, we apply Kronecker algebra to support deadlock free train disposition in railway systems.

Organization of this chapter is as follows. We begin by motivating the topic in Section 1.1. In Section 1.2 we give a brief overview and highlight the major contributions of this doctoral thesis. Finally, we present the overall organization of this dissertation in Section 1.3.

1.1 Motivation

For safety-critical systems, dependable systems and robust embedded systems, software has to be provably correct. In particular, this is a very important issue, e.g., in the fields of medical systems, aviation, rail, and automotive industries. In general, deadlock freedom is a desired property. In addition, for real-time systems also the maximum computation time is an important issue, i.e., to have a computation result within a certain time. It is widely agreed that the problem of determining upper bounds on execution times for sequential programs has been more or less solved [WEE^{+}08]. With the advent of multi-core processors scientific and industrial interest focuses on analyzing and verifying multi-threaded applications. Analysis of multi-threaded software is still a challenge. Beside other reasons, this comes from the fact that synchronization has to be taken into account. For sequential programs, control flow graphs are often used as a basis for static program analysis. An equivalent model for multi-threaded programs taking synchronization between threads into account has been missing.

1.2 Thesis Overview and Contributions

The idea that thread interleavings of concurrent programs can be studied with a matrix calculus is novel in this research area. We are immediately able to support conditionals, loops, and synchronization. Our sparse matrix representations of the program are manipulated using a lazy implementation of Kronecker algebra. We use synchronization primitives (e.g semaphores and barriers) for thread synchronization. One goal is generating a data structure called *concurrent program graph* (CPG) which describes all possible interleavings and incorporates synchronization while preserving completeness. CPGs play a similar role for concurrent systems as control flow graphs (CFGs) do for sequential programs.

We prove that CPGs in general can be represented by sparse adjacency matrices. Thus the number of entries in the matrices is linear in their number of lines. Hence efficient algorithms can be applied to CPGs.

In the worst-case, the number of lines increases exponentially in the number of threads. In general, however, a CPG contains many nodes and edges unreachable from the entry node. We propose two major optimizations. First, if the program contains a lot of synchronization, only a very small part of the CPG is reachable. Thanks to a lazy implementation of the matrix operations, only the reachable part is computed. As a second optimization we have parallelized the CPG generation in order to exploit modern many-core hardware architectures. Both optimizations speed up processing significantly and show that our approach is very promising.

In the first place, we use CPGs as a vehicle for detecting deadlocks of multi-threaded programs. Deadlocks show up as a natural property of a concurrent program's adjacency matrix, namely as zero lines.

CPGs can also be used to analyze multi-threaded programs using barriers as a synchronization aid. We propose a novel barrier synchronization construct and compare it to a semaphore-based barrier implementation. As a byproduct, we show how our CPG-based approach can be used as a basis for proving semaphore-based barrier implementations and their usage scenarios correct.

In this dissertation, CPGs are used to show how to calculate the WCET of the underlying concurrent system. We adopt the generating functions-based approach presented in [Bli02, Section 4]. Since concurrent programs may contain blocking because of synchronization between threads, the terms execution time and worst-case execution time (WCET) do not apply directly to concurrent systems. However, we stick to the term WCET for concurrent systems. The reader, however, has to be aware of the fact that, in general, the WCET includes stalling time.

CPGs together with several techniques form a framework for analyses of various properties of multi-threaded shared memory programs. We will see that additional techniques including data flow analysis, symbolic evaluation, and (automata based) model checking can be applied to CPGs.

We also adapt our Kronecker algebra based deadlock detecting approach to railway systems. For multiple trains and a given track topology, all possible train movements can be calculated. States which will probably, certainly and certainly not lead to a deadlock of the involved trains are distinguished. The approach may take into account additional constraints, e.g., overtaking and train connections.

The main contributions of this dissertation are:

1. A framework for analyzing concurrent systems. By using a matrix calculus which is often referred to as *Kronecker algebra*, we are able to model concurrent systems as matrices. We model thread interleavings and synchronization via semaphores by using Kronecker sum, Kronecker product, and a slightly modified Kronecker product operator which we call *selective Kronecker product.*

2. An approach for calculating the entries of the matrices lazily. In order to exploit multi-core architectures, we implemented a parallel version and thereby gained a very good speedup.

3. An approach for *detecting deadlocks* in concurrent systems.

4. We show how to model Ada's barriers such that Kronecker algebra can be employed for static analysis. This is done by introducing a novel synchronization primitive modeling the semantics of barriers. We compare our barrier synchronization primitive with a barrier implementation based on semaphores. As a byproduct, we show how our CPG-based approach can be used as a basis for proving semaphore-based implementations correct.

5. We propose a *worst-case execution time analysis* approach for concurrent programs and focus on automatically calculating and incorporating stalling times (e.g. caused by lock contention). We employ a generating functions-based approach for setting up data flow equations which are solved by well-known elimination based data flow analysis methods or an off-the-shelf equation solver. The WCET of multi-threaded programs can finally be calculated with a non-linear function solver. This novel approach is suitable for both, namely parallel and concurrent systems.

6. A *deadlock avoidance* approach for railway systems. For multiple trains and a given track topology, we are able to calculate all possible train movements. Some lead and some do not lead to deadlocks. From some points on no deadlocks are reachable and the trains can proceed with their movements in any order. From certain other points a deadlock is inevitable. Our approach, which may take into account additional constraints such as overtaking and train connections, can be used to avoid such situations and to suggest only deadlock-free train movements.

7. Proofs concerning properties of matrices representing concurrent programs and their operations. We proof a new property of the Kronecker sum of matrices which we call *Mixed Sum Rule*. In addition, we proof the *sparsity* of matrices representing concurrent systems, i.e., the number of entries in such a matrix is linear in its order.

Because we already have published parts of this thesis there already exist publications and projects building upon the matrix calculus established during the work for this dissertation. Our Kronecker algebra based approach is already used in the following papers. In [BB14] Kronecker algebra is applied in order to statically analyze Ada multi-tasking programs that employ protected objects for synchronization issues.

Our Kronecker algebra model is also used in the railway domain. The adaptations required for the railway domain were done during the work done for this doctoral thesis and appeared in [MBS12]. It was extended in several publications. For example, a Kronecker algebra based method for determining the travel time of trains in railway systems is presented in [VBS12]. Travel time analysis in railway systems may be compared to WCET analysis in computer systems. The approach of [MBS12] was also adopted in the *EcoRailNet* project. It was a joint project of ÖBB-Infrastruktur AG, ÖBB-Produktion GmbH, Thales Austria GmbH, and Vienna University of Technology (Institute of Computer Aided Automation) funded by the Austrian Ministry for Transport, Innovation and Technology (New Energy 2020, Project ID: 834586). The aim of the project was to save energy by minimizing stop and go of trains. Instead of only taking into account one single train, a railway system consisting of multiple trains turned out to be a solvable global optimization problem which can be solved fast and automatically [Vol14].

1.3 Outline

The outline of this doctoral thesis is as follows. In Chapter 2 some preliminaries such as our semiring, control flow graphs, edge splitting, and basic matrix notation and operations are introduced. The used matrix calculus, often referred to as Kronecker algebra, and some of its properties are introduced in Chapter 3. Our model of concurrency, its properties, and optimizations like our lazy approach are presented in Chapter 4. Chapter 5 demonstrates how we are able to detect deadlocks. In Chapter 6 we give examples and present empirical data. We introduce a novel barrier synchronization primitive and compare it to semaphore-based barrier implementations in Chapter 7. Chapter 8 is devoted to worst-case execution time analysis of concurrent programs and we present a detailed example. In Chapter 9 we show how our Kronecker algebra approach can be applied to railway systems in order to avoid deadlocks in train disposition systems. In Chapter 10 we survey related work. We draw our conclusion and outline possible future work in Chapter 11. Finally, we give an introduction to the state explosion problem and show how the number of interleavings can be calculated for an arbitrary number of threads and their corresponding statements in Appendix A.

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."

– ALAN TURING, English computer scientist, mathematician, logician, cryptanalyst
and theoretical biologist, 1912-1954

CHAPTER 2

Preliminaries

> *"Je mehr ich über die Sprache nachdenke,*
> *desto sonderbarer kommt es mir vor,*
> *dass sich die Leute jemals verstehen."*
>
> – KURT FRIEDRICH GÖDEL, Austrian, and later American, logician,
> mathematician, and philosopher, 1906-1978

In this chapter, we introduce preliminaries and basic notation required throughout this dissertation. We start with introducing our semiring and continue with *Control Flow Graphs* (CFGs). In addition, we show how we represent synchronization primitives, e.g., semaphores. Then we show how to split edges in order to get *Refined Control Flow Graphs* (RCFGs) from each thread's CFG. Edge splitting is done for synchronization primitive calls. For value-sensitive analysis, we propose an edge splitting for shared variables, too. This procedure ensures the granularity needed for manipulation with the matrix calculus as introduced below in Chapter 3. Because our models representing multi-threaded programs is generated out of matrices, we state basic matrix notations, terminology, operations, and discuss how matrices correspond to directed graphs. Finally, we mention some limitations which we admit throughout this dissertation.

2.1 Overview

We model shared memory concurrent systems by threads which use synchronization primitives, e.g., semaphores and barriers, for synchronization. Threads and synchronization primitives are represented by CFGs [Hec77, ASU86, AP02]. *Edge splitting* which is described in Subsection 2.5 has to be applied to the edges containing synchronization primitive calls. The resulting *Refined CFGs* (RCFGs) are represented by adjacency matrices. These matrices are then manipulated by Kronecker algebra. We assume that the edges of RCFGs are labeled by elements of a semiring defined in the following section.

2.2 Semiring

In this section, we define our semiring. Similar definitions and further properties of semirings can be found in [KS86].

Our semiring $\langle \mathcal{L}, +, \cdot, 0, 1 \rangle$ consists of a set of labels $\mathcal{L}$, two binary operations $+$ and $\cdot$, and two constants 0 and 1 such that

1. $\langle \mathcal{L}, +, 0 \rangle$ is a commutative monoid,
2. $\langle \mathcal{L}, \cdot, 1 \rangle$ is a monoid,
3. left and right distributivity of $\cdot$ over $+$:
    - $\forall l_1, l_2, l_3 \in \mathcal{L} : l_1 \cdot (l_2 + l_3) = l_1 \cdot l_2 + l_1 \cdot l_3$ and
    - $(l_1 + l_2) \cdot l_3 = l_1 \cdot l_3 + l_2 \cdot l_3$ hold and
4. constant 0 is an absorbing element concerning the semiring operation '$\cdot$': $\forall l \in \mathcal{L} : 0 \cdot l = l \cdot 0 = 0$.

Intuitively, our semiring is a unital ring without subtraction. For each $l \in \mathcal{L}$ the usual rules are valid, e.g., $l + 0 = 0 + l = l$ and $1 \cdot l = l \cdot 1 = l$. In general $\forall a, b \in \mathcal{L} : a \cdot b \neq b \cdot a$. In case of juxtaposition and if it is clear in the context, we often write $a\,b$ instead of $a \cdot b$. In addition, we equip our semiring with the unary star operation $*$. For each $l \in \mathcal{L}$, l^* is defined by

$$l^* = \sum\nolimits_{j \geq 0} l^j, \text{ where } l^0 = 1 \text{ and } l^{j+1} = l^j \cdot l = l \cdot l^j \text{ for } j \geq 0.$$

The set of labels $\mathcal{L}$ is defined by $\mathcal{L} = \mathcal{L}_{\mathrm{V}} \cup \mathcal{L}_{\mathrm{S}}$, where $\mathcal{L}_{\mathrm{V}}$ is the set of non-synchronization labels and $\mathcal{L}_{\mathrm{S}}$ is the set of labels representing synchronization primitive calls, e.g., p_i and v_i referring to the operation p and v of semaphore i. The sets $\mathcal{L}_{\mathrm{V}}$ and $\mathcal{L}_{\mathrm{S}}$ are disjoint.

A prominent example for semirings are regular expressions (cf. [Tar81]) which can be used for describing the behavior of finite state automata. A second example for semirings is performing data flow analysis (e.g. [BB98, BB03, BBS99, BBM06, SBF00]).

If it is not clear in the context to which thread a label l belongs, we write l^X to denote that l belongs to thread X.

2.3 Control Flow Graphs

We represent threads and synchronization primitives in form of control flow graphs. A *Control Flow Graph* (CFG) is a directed labeled graph defined by $G = \langle V, E, n_e, V_f \rangle$ with a set of nodes V, a set of directed edges $E \subseteq V \times V$, a so-called *entry* node $n_e \in V$, and a set of final nodes $V_f \subseteq V$. An entry node has an incoming edge which has no source node. A final node is depicted with a double circle. We require that each $n \in V$ is reachable through a sequence of edges from n_e.

```
FACTORIAL ()
1  fact ← 1
2  $n$ ← INPUT("Enter a number:")
3  while $n \geq 1$ do
4      fact ← fact $* n$
5      $n \leftarrow n - 1$
6  endwhile
7  PRINT(fact)
```

Figure 2.1: Pseudocode Factorial

Nodes can have at most two outgoing edges. Thus the maximum number of edges in CFGs is $2\,|V|$. We will use this property later.

Usually CFG nodes represent basic blocks (cf. [ASU86]). Because our matrix calculus manipulates the edges, we need to have basic blocks on the edges.[1] As usual the edges represent the transfer of control between the basic blocks. Each edge $e \in E$ is assigned a basic block b. In this dissertation, we refer to basic blocks by edge labels as defined in the previous section. Labels out of the set $\mathcal{L}_{\mathrm{S}}$ refer to synchronization primitive calls. The remaining labels refer to the elements of $\mathcal{L}_{\mathrm{V}}$ which model basic blocks consisting of ordinary program statements, i.e., non-synchronization statements. The operations on the basic blocks are $\cdot, +$, and $*$ from the semiring defined above (cf. [Tar81]). Intuitively, the semiring operations $\cdot, +$, and $*$ model consecutive program parts, conditionals, and loops, respectively.

As an example for a control flow graph consider the pseudocode implementation of factorial depicted in Figure 2.1. Line 1 and 2 form CFG node a. Node b refers to the while condition of line 3. The lines 4 and 5 build CFG node c. Finally, CFG node d is constructed out of line 7. The corresponding CFG having the basic blocks on the nodes is presented in Figure 2.2a. Node a and d are the entry and final node, respectively, of the original CFG. In contrast to that, Figure 2.2b depicts the same example with the basic blocks at the edges. Node 1 is the entry node whereas node 4 is the final node of the transformed CFG.

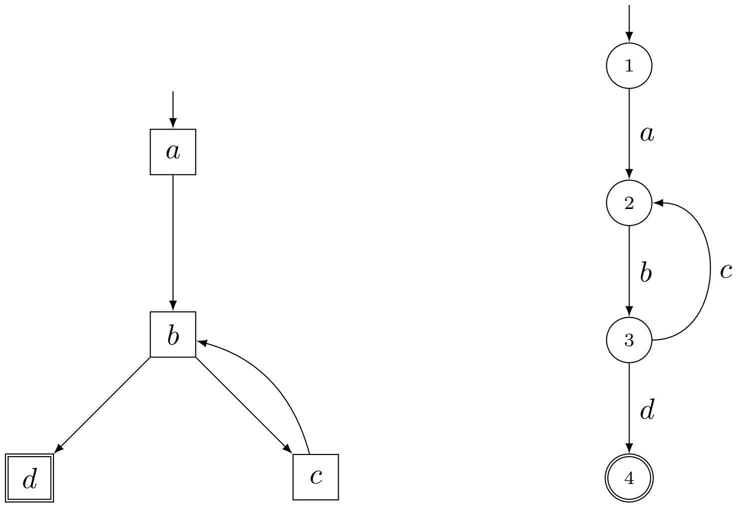
2.4 Semaphores

In order to model synchronization, we use synchronization primitives, e.g., semaphores which are presented in this section. During the course of this dissertation, we introduce additional synchronization primitives, namely barriers, in Chapter 7.

Semaphores [Dij65, Dijndb, Sta11] are a well-known vehicle for process synchronization, are available in all operating systems, and can be implemented efficiently. Semaphores typically implement two operations, namely p and v. Usually the operation p is used to acquire a resource, whereas v releases the resource. If the semaphore is already acquired by a thread, then the calling thread is being suspended and the thread is being added to the semaphore's *first-in, first-out* (FIFO) queue. After a thread releases the resource, the first thread from the queue is removed and resumes its execution. It has to be ensured

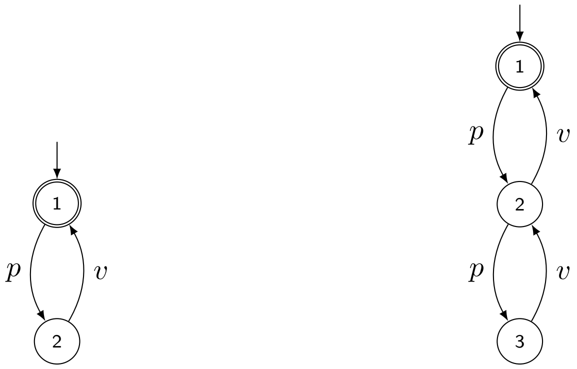
---

[1] We chose the incoming edges.



(a) Basic Blocks at the Nodes (b) Basic Blocks at the Edges

Figure 2.2: CFG of Factorial



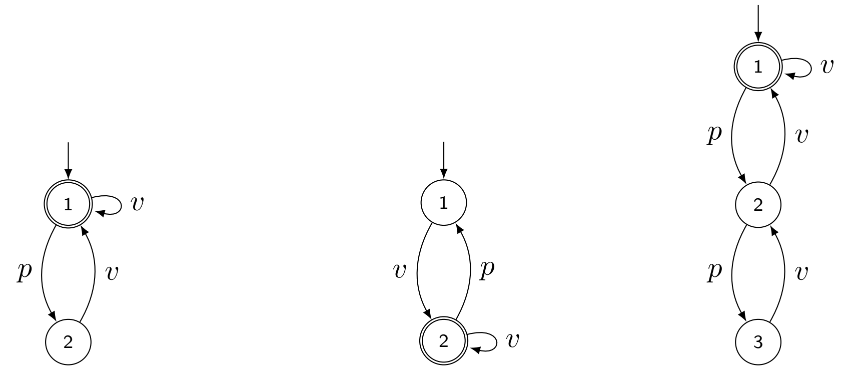
(a) Binary Semaphore (b) Counting Semaphore

Figure 2.3: Semaphores

that both operations, namely p and v, are executed atomically. Note that semaphores are similar to mutexes. In contrast to semaphores, mutexes have an owner. This means that a locked mutex can only be unlocked by the process that locked the mutex. In this dissertation, we sometimes use this kind of freedom which semaphores give.

Similar to threads, synchronization primitives like semaphores can be represented in form of CFGs. For semaphore i the corresponding edges typically have labels such as $p_i \in \mathcal{L}_\mathrm{S}$ and $v_i \in \mathcal{L}_\mathrm{S}$. Usually two or more distinct thread CFGs refer to the same semaphore to model synchronization.

In Figure 2.3a and 2.3b a binary and a counting semaphore are depicted. The latter



(a) Initially Unlocked Binary (b) Initially Locked Binary (c) Counting

Figure 2.4: Semaphores with Non-blocking v-operations

allows two threads to enter at the same time. In a similar way it is possible to construct semaphores allowing n non-blocking p-calls ($n \in \mathbb{N}, n \geq 1$). Node 1 in Figure 2.3a and 2.3b is both, entry and final node.

So, for example, a thread cannot do semaphore calls in the order v followed by p when the semaphore DFA only allows a p-call before a v-call (this is the case when using the semaphore without the self-loop at node 1 as depicted in Figure 2.3a). The graph of such an erroneous program will contain a node from which the final node of that graph cannot be reached. This node is the one preceding the v-call. Such nodes can easily be found by traversing the program's graph we introduce in the course of this dissertation. Thus deadlocks of concurrent systems can be detected with little effort.

In Figure 2.4 an initially unlocked (a) and locked (b), respectively, binary and a counting semaphore (c) are depicted. The latter (like the counting semaphore in Figure 2.3b) allows two threads to enter at the same time. In contrast to the semaphores depicted in Figure 2.3, the semaphores in Figure 2.4 support multiple subsequent non-blocking v-calls. Similar to the initially locked semaphore shown in Figure 2.4b, we can construct an initially locked semaphore with potentially blocking v-operations, i.e., Figure 2.3a with entry node 2.

2.5 Edge Splitting

As already mentioned above, we have the basic blocks on the incoming edges. A basic block consists of multiple consecutive statements without jumps. For our purpose we need a finer granularity which we achieve by splitting edges. We apply it to basic blocks containing synchronization calls and shared variables. For both, edge splitting results in a *Refined Control Flow Graph* (RCFG).

Edge splitting for calls to synchronization primitives, e.g., semaphore calls p_1 and v_1, is as follows. We require that a call – referred to as s_i – has to be the only statement on the corresponding edge. In Figure 2.5, edge splitting is shown for the edge e depicted in Figure 2.5a. The edge's basic block is assumed to contain k synchronization primitive calls. The resulting edges are presented in Figure 2.5b. The edge e is replaced by the edges $e_1 \dots e_{k+1}$ and the nodes $n_1 \dots n_{2k}$ are introduced. Roughly speaking, edge splitting maps a CFG edge e whose corresponding basic block contains k synchronization primitive calls to a subgraph $\circ \xrightarrow{e_1} \circ \xrightarrow{s_1} \circ \xrightarrow{e_2} \circ \xrightarrow{s_2} \circ \cdots \circ \xrightarrow{e_k} \circ \xrightarrow{s_k} \circ \xrightarrow{e_{k+1}} \circ$, such that each s_i represents a single synchronization primitive call, and e_i and e_{i+1} represent the consecutive parts before and after s_i, respectively $(1 \leq i \leq k)$. Note that each synchronization primitive's CFG contains only edges labeled by $l \in \mathcal{L}_\mathrm{S}$. Thus each synchronization primitive's CFG is a RCFG without any further modification.

For shared variables, edge splitting can be done in a similar fashion. In contrast to calls to synchronization primitives, we do not require that shared variable accesses are the only statements on the corresponding edge. The remaining consecutive parts of the basic block are situated on the previous and succeeding edges, respectively. Note that edges representing a call to a synchronization primitive are not considered to access shared variables.

Let $\mathcal{V}$ be the set of shared variables.[2] In addition, let each shared variable $v \in \mathcal{V}$ be a volatile variable located in the shared memory which is accessed by two or more threads. Splitting an edge depends on the number of statements accessing shared variables in the corresponding edge (i.e. basic block). For edge e labeled by basic block b this number is being referred to as $\mathrm{NSV}(b)$. If $\mathrm{NSV}(b) > 1$, then edge splitting has to be applied to edge e; the edge is used unchanged otherwise.

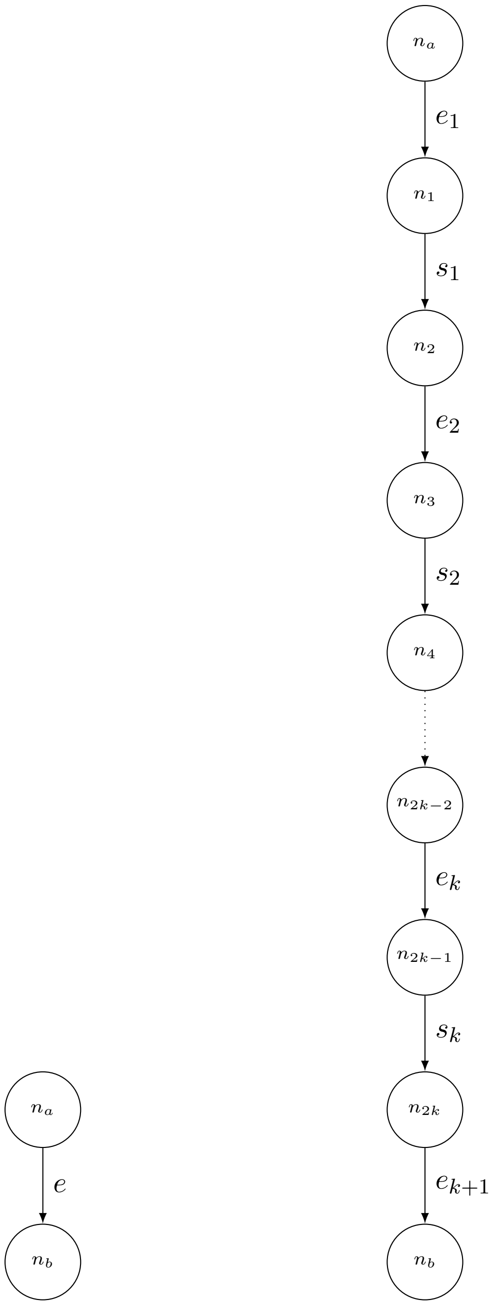
If edge splitting has to be applied to edge e which has basic block b assigned and $\mathrm{NSV}(b) = k$ then the basic blocks $b_1, \dots, b_k$ represent the subsequent parts of b in such a way that $\forall b_i : \mathrm{NSV}(b_i) = 1$, where $1 \leq i \leq k$. Edges e_j get assigned basic block b_j, where $1 \leq j \leq k$. In Figure 2.6 splitting of an edge with basic block b and $\mathrm{NSV}(b) = k$ is depicted.

Edge splitting for shared variables is relevant for a value-sensitive analysis (such as symbolic analysis) when taking shared variables into account. This approach ensures a representation in a manner exact enough in order to allow modeling all possible context switches, i.e., interleavings. We say "exact enough" because by using basic blocks together with edge splitting, we already have coarsened the granularity compared to the statement-level. This helps to keep the generated graph model (which we define in Chapter 4) small. With a granularity on statement-level the graph model would be unnecessarily big. In addition, we do not lose any information nor accuracy for our analysis purposes.

From the used matrix calculus point of view, edge splitting ensures that it is possible to generate all necessary interleavings. Nevertheless, generating unnecessary interleavings should be prevented. When e.g. static scheduling is used, the dispatching points are usually known. If between two different shared variable accesses no dispatching is possible, then we relax the rule (i.e. $\forall b_i : \mathrm{NSV}(b_i) = 1$) above. This may lead to an coarser granularity on the RCFG-level.

---

[2] As an approach for finding the set of shared variables in Ada programs we suggest [BBM06].



(a) Edge Before Edge Splitting (b) Edge After Edge Splitting

Figure 2.5: Edge Splitting for k Synchronization Primitive Calls

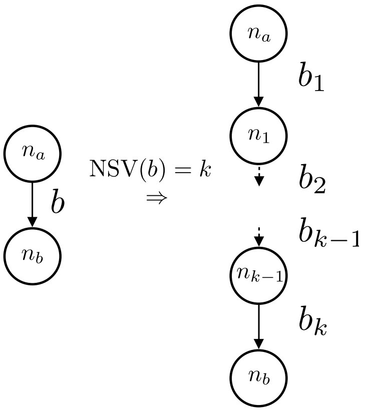


Figure 2.6: Edge Splitting for k Shared Variable Accesses

Without loss of generality, we assume that the statements in each basic block are atomic. Thus, while executing a statement, context switching is impossible. In RCFGs the finest possible granularity is at statement-level. If, according to the hardware architecture or program's semantic, atomic statements may access two or more shared variables, then we make an exception to the above rule, too, and allow two or more shared variable accesses on a single edge. Such edges have at most one of these atomic statements in their basic block and no additional statements accessing shared variables.

The effects of edge splitting for both, namely synchronization primitive calls and shared variable accesses, can be seen in the data race example given in Section 6.4. Each RCFG depicted in Figure 6.7 is constructed out of one basic block (cf. Figure 6.6).

2.6 Basic Matrix Notations and Operations

In this section, we introduce basic matrix terminology, notations, and operations used in the remainder of this thesis.

A p-by-q matrix

$$
\begin{aligned}
M &= (m_{i,j}) \\
&= \begin{array}{c c}
 & \begin{array}{cccc} 1 & 2 & \dots & q \end{array} \\
\begin{array}{c} 1 \\ 2 \\ \vdots \\ p \end{array} &
\begin{pmatrix}
m_{1,1} & m_{1,2} & \dots & m_{1,q} \\
m_{2,1} & m_{2,2} & \ddots & \vdots \\
\vdots & \ddots & \ddots & \vdots \\
m_{p,1} & m_{p,2} & \dots & m_{p,q}
\end{pmatrix}
\end{array}
\end{aligned}
$$

has p rows and q columns. Thus it consists of p times q entries $m_{i,j}$. Note that $m_{i,j}$'s i and j denote its row and column, respectively, within matrix M.

We define the set of matrices $\mathcal{M} = \{M = (m_{i,j}) \mid m_{i,j} \in \mathcal{L}\}$. In the remaining parts of this dissertation only matrices $M \in \mathcal{M}$ will be used. In addition let $o(M)$ refer to the order[3] of square matrix $M \in \mathcal{M}$. Furthermore, we will use zero matrices $Z_n = (z_{i,j})$, where $\forall i, j : z_{i,j} = 0$ and $o(Z_n) = n$.

The identity matrix I_n of order n has ones at the main diagonal and zeros elsewhere. Thus

$$I_n = (m_{i,j}), \text{ where } m_{i,j} = \begin{cases} 1 & i = j, \\ 0 & \text{otherwise.} \end{cases}$$

By the definition above we already have $I_1 = (1)$. We additionally define $I_0 = (1)$.

Let M, N, and R be p-by-q matrices. In addition, let a be a label out of $\mathcal{L}$. In the following, the used indices range between $1 \leq i \leq p$ and $1 \leq j \leq q$. We get $a \cdot M$ and $M + N$ as follows:

$$a \cdot M = a \cdot (m_{i,j}) = (r_{i,j}) = R, \text{ where } r_{i,j} = a \cdot m_{i,j} \text{ and}$$
$$M + N = (m_{i,j}) + (n_{i,j}) = (r_{i,j}) = R, \text{ where } r_{i,j} = m_{i,j} + n_{i,j}.$$

Sometimes, we also use the transpose M^T of a p-by-q matrix M. The result is a q-by-p matrix which is defined by

$$M^T = (m_{i,j})^T = (m_{j,i}).$$

Intuitively, columns are turned into rows and vice versa.

Definition 1 (Number of Nonzero Entries in a Matrix) *Let* $M = (m_{i,j}) \in \mathcal{M}$. *We denote the number of entries unequal to zero by* $||M|| = |\{m_{i,j} \mid m_{i,j} \neq 0\}|$.

Definition 2 (Sparse Matrix) *We call an* n*-by-*n *matrix* M *sparse if and only if* $||M|| \leq c * n$, *where* c *is a constant independent from* n.

2.7 Correspondence between Matrices and Directed Graphs

There is a correspondence between matrices and graphs. In general, a directed labeled graph $G\langle V, E, n_e\rangle$ consists of a set of labeled nodes V, a set of labeled directed edges $E = V \times V$, and an entry node n_e. Correspondence between directed graphs and matrices – in this context the latter are referred to as adjacency matrices – is as follows. In this dissertation, we label graph nodes simply by positive integers which reflect the row and column in the adjacency matrix. If there exists an entry $m_{i,j} = a$ in an adjacency matrix, then a directed edge from node i to node j labeled by a exists in the corresponding directed graph. If $m_{i,j} = 0$, then there is no edge from node i to node j.

Because we usually work with sparse matrices, we suggest adjacency lists as an implementation for the matrices. In adjacency lists zeros are not stored explicitly. Only entries unequal to zero are stored in the lists. Compared to the approach, where all the $o(M)^2$ matrix entries are stored explicitly, this helps to safe memory.

---

[3] A k-by-k matrix is known as square matrix of order k.

To keep things simple, we refer to edges, their labels, the corresponding basic blocks and the corresponding entries of the adjacency matrices synonymously. Analogously, we refer to a node, its node number, and its row and column in the corresponding matrix synonymously.

In the remainder of this dissertation, we often use the node numbers as generated by Kronecker sum, Kronecker product, or, concerning CPGs, we use the node numbers generated by our implementation.

2.8 Limitations

Theoretical results such as [Ram00] state that synchronization-sensitive and context-sensitive analysis is impossible even for the simplest analysis problems. Our system model differs in that it supports subprograms only via inlining and recursions are not allowed.

"Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt."

– LUDWIG WITTGENSTEIN, Austrian-British philosopher, 1889-1951
Proposition 5.6, Tractatus Logico-Philosophicus, 1922

CHAPTER 3

Kronecker Algebra – A Matrix Calculus

> *"I hear and I forget.*
> *I see and I remember.*
> *I do and I understand."*
>
> – CONFUCIUS, Chinese teacher, politician, and philosopher, 551-479 BC

Kronecker product and Kronecker sum form the so-called Kronecker algebra. In this chapter, we define both operations, state properties, and give examples on matrix and graph/automata level. For the Kronecker sum we prove associativity and a new property which we call *Mixed Sum Rule*.

3.1 Kronecker Product / Zehfuss Product

> *"No scientific discovery is named after its original discoverer."*
>
> – Stigler's law of eponymy, 1980

Kronecker product allows to model synchronization [BK02, Pla85]. In this section, we state a definition, give examples, and present properties used in this thesis.

Definition 3 (Kronecker Product) *Given an m-by-n matrix A and a p-by-q matrix B, their* Kronecker product *$A \otimes B$ is an mp-by-nq block matrix defined by*

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}.$$

As stated in [Mil11] the Kronecker product is also being referred to as *Zehfuss product* or *direct product* of matrices. Knuth notes in [Knu11] that Kronecker never published anything about it. Zehfuss was actually the first publishing it in the 19th century [Zeh58].[1] Following Stigler's law of eponymy, Kronecker product is usually not called Zehfuss product.

In terms of formal automata, the Kronecker product calculates the simultaneous executions of the input matrices. Thus, the operation $\otimes$ can be used to synchronize automata. In the following, we give an example.

3.1.1 Examples

Example 1
For this example, we use the following matrices A and B of order 2 and 3, respectively:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \text{ and } B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}.$$

The Kronecker product $A \otimes B$ is a matrix of order 6 given by

$$A \otimes B = \begin{pmatrix} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & a_{1,1}b_{1,3} & a_{1,2}b_{1,1} & a_{1,2}b_{1,2} & a_{1,2}b_{1,3} \\ a_{1,1}b_{2,1} & a_{1,1}b_{2,2} & a_{1,1}b_{2,3} & a_{1,2}b_{2,1} & a_{1,2}b_{2,2} & a_{1,2}b_{2,3} \\ a_{1,1}b_{3,1} & a_{1,1}b_{3,2} & a_{1,1}b_{3,3} & a_{1,2}b_{3,1} & a_{1,2}b_{3,2} & a_{1,2}b_{3,3} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & a_{2,1}b_{1,3} & a_{2,2}b_{1,1} & a_{2,2}b_{1,2} & a_{2,2}b_{1,3} \\ a_{2,1}b_{2,1} & a_{2,1}b_{2,2} & a_{2,1}b_{2,3} & a_{2,2}b_{2,1} & a_{2,2}b_{2,2} & a_{2,2}b_{2,3} \\ a_{2,1}b_{3,1} & a_{2,1}b_{3,2} & a_{2,1}b_{3,3} & a_{2,2}b_{3,1} & a_{2,2}b_{3,2} & a_{2,2}b_{3,3} \end{pmatrix}.$$

Example 2
In this example, we show how we calculate all possible simultaneous executions of the two automata represented by the matrices

$$C = \begin{pmatrix} a & b \\ 0 & 0 \end{pmatrix} \text{ and } D = \begin{pmatrix} 0 & a \\ 0 & b \end{pmatrix}.$$

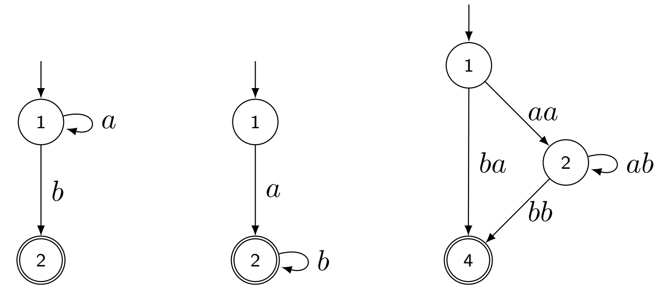
The corresponding automata are depicted in Figure 3.1a and 3.1b, respectively. The Kronecker product $C \otimes D$ is given by

$$C \otimes D = \begin{pmatrix} 0 & aa & 0 & ba \\ 0 & ab & 0 & bb \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

---

[1] Zehfuss proved that $\det(A \otimes B) = \det^n(A)\det^m(B)$, if A and B are matrices of order m and n, respectively, and entries from the domain of real numbers.

In Figure 3.1c the result on automata level is depicted. For each edge in the resulting automaton, actually two edges, i.e. one edge of each of the two input automata, are executed. The first label represents an edge of the first input automata, whereas the second label refers to an edge of the second input automaton. How we determine entry and final nodes is described in Chapter 4. We use this notion here in order to have a complete presentation from an automata point of view.



(a) Graph for C (b) Graph for D (c) Graph for $C \otimes D$

Figure 3.1: Simultaneous Executions via Kronecker Product $C \otimes D$

Example 3
For this example, we use the matrices

$$E = \begin{pmatrix} 0 & a & 0 & d \\ 0 & 0 & b & 0 \\ 0 & 0 & 0 & c \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ and } F = \begin{pmatrix} 0 & e & 0 & 0 \\ 0 & 0 & f & 0 \\ 0 & 0 & 0 & g \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

The corresponding graphs are depicted in Figure 3.2a and 3.2b, respectively. For a concise presentation of the 16-by-16 matrix of the Kronecker product $E \otimes F$ we introduce the helper matrices H_1, H_2, H_3, and H_4 as follows:

$$H_1 = \begin{pmatrix} 0 & a\,e & 0 & 0 \\ 0 & 0 & a\,f & 0 \\ 0 & 0 & 0 & a\,g \\ 0 & 0 & 0 & 0 \end{pmatrix}, \qquad H_2 = \begin{pmatrix} 0 & d\,e & 0 & 0 \\ 0 & 0 & d\,f & 0 \\ 0 & 0 & 0 & d\,g \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

$$H_3 = \begin{pmatrix} 0 & b\,e & 0 & 0 \\ 0 & 0 & b\,f & 0 \\ 0 & 0 & 0 & b\,g \\ 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and } \qquad H_4 = \begin{pmatrix} 0 & c\,e & 0 & 0 \\ 0 & 0 & c\,f & 0 \\ 0 & 0 & 0 & c\,g \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Then we get the resulting matrix

$$E \otimes F = \begin{pmatrix} Z_4 & H_1 & Z_4 & H_2 \\ Z_4 & Z_4 & H_3 & Z_4 \\ Z_4 & Z_4 & Z_4 & H_4 \\ Z_4 & Z_4 & Z_4 & Z_4 \end{pmatrix}.$$

The corresponding graph is depicted in Figure 3.2c. Starting from the entry node 1 *not all nodes are reachable. In the reachable part only the path* $1 \rightarrow 6 \rightarrow 11 \rightarrow 16$ *leads to the final node* 16*. Choosing the edge* $1 \rightarrow 14$ *a further simultaneous execution is impossible.*

3.1.2 Properties

In the following, we list some basic properties of the Kronecker product. Let A, B, C, and D be matrices. Kronecker product is non-commutative because in general

$$A \otimes B \neq B \otimes A.$$

It is permutation equivalent because there exist permutation matrices P and Q such that $A \otimes B = P(B \otimes A)Q$ (cf. [Gra81, Wei62]). If A and B are square matrices, then $A \otimes B$ and $B \otimes A$ are even permutation similar, i.e., $P = Q^T$. It is associative [Gra81, Pla85] as

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C. \tag{3.1}$$

In addition, Kronecker product distributes over $+$ [Gra81], i.e.,

$$A \otimes (B + C) = A \otimes B + A \otimes C, \tag{3.2}$$

$$(A + B) \otimes C = A \otimes C + B \otimes C. \tag{3.3}$$

Hence for example $(A + B) \otimes (C + D) = A \otimes C + B \otimes C + A \otimes D + B \otimes D$.

Properties concerning connectedness of the corresponding undirected and directed graphs can be found in [Wei62] and [McA63, HT66], respectively. A recent publication about connectedness of the resulting graphs is [HIK11]. Additional properties and proofs can be found in [Bel97, Gra81, Dav81, Hur94].

3.2 Kronecker Sum

In this section we define the Kronecker sum of matrices, give examples, and state some important properties. We also relate the operation to Cartesian product graphs.

Definition 4 (Kronecker Sum) *Given a matrix* A *of order* m *and a matrix* B *of order* n*, their* Kronecker sum $A \oplus B$ *is a matrix of order* mn *defined by*

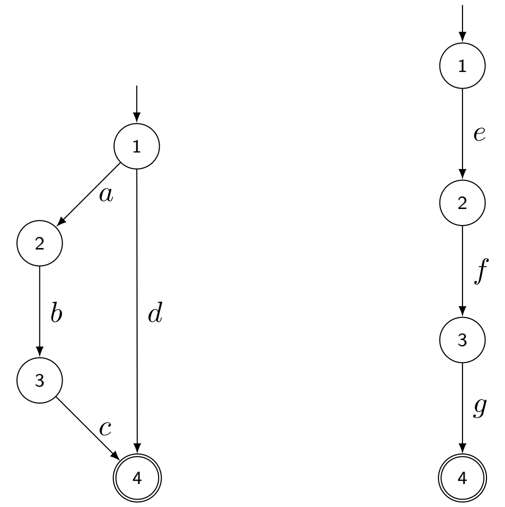
$$A \oplus B = A \otimes I_n + I_m \otimes B,$$

where I_m *and* I_n *denote identity matrices*[2] *of order* m *and* n*, respectively.*

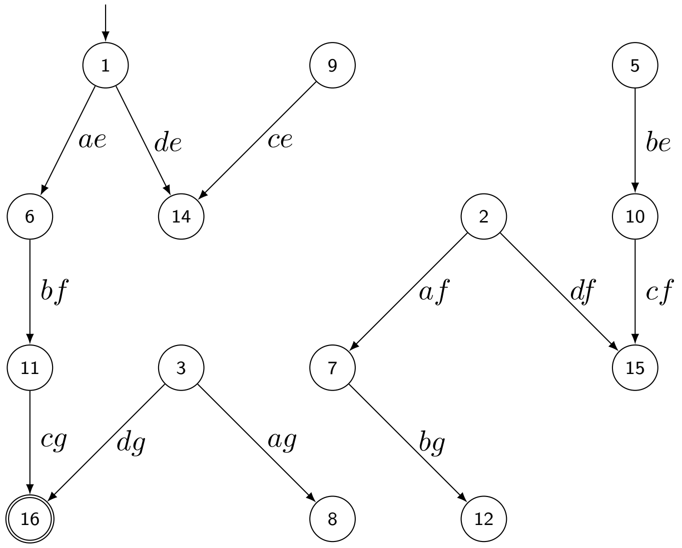
This operation must not be confused with the direct sum of matrices, group direct product or direct product of modules for which the symbol $\oplus$ is used too.

---

[2] Identity matrix I_n is an n-by-n matrix with ones on the main diagonal and zeros elsewhere.



(a) Graph for E (b) Graph for F



(c) Graph represented by $E \otimes F$

Figure 3.2: Simultaneous Executions via Kronecker product $E \otimes F$

3.2.1 Examples

In this section we present two Kronecker sum examples.

Example 4
In this example, we again use the matrices A and B from Example 1. The Kronecker sum $A \oplus B$ is given by

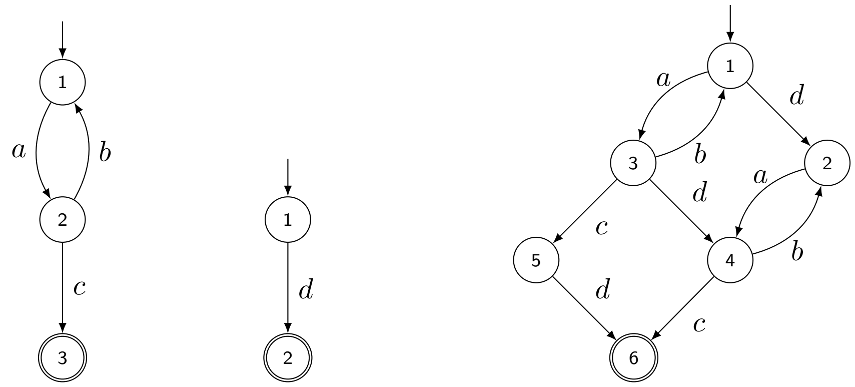
$$
\begin{aligned}
&A \otimes I_3 + I_2 \otimes B = \\
&\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} = \\
&\begin{pmatrix}
a_{1,1} & 0 & 0 & a_{1,2} & 0 & 0 \\
0 & a_{1,1} & 0 & 0 & a_{1,2} & 0 \\
0 & 0 & a_{1,1} & 0 & 0 & a_{1,2} \\
a_{2,1} & 0 & 0 & a_{2,2} & 0 & 0 \\
0 & a_{2,1} & 0 & 0 & a_{2,2} & 0 \\
0 & 0 & a_{2,1} & 0 & 0 & a_{2,2}
\end{pmatrix} + \begin{pmatrix}
b_{1,1} & b_{1,2} & b_{1,3} & 0 & 0 & 0 \\
b_{2,1} & b_{2,2} & b_{2,3} & 0 & 0 & 0 \\
b_{3,1} & b_{3,2} & b_{3,3} & 0 & 0 & 0 \\
0 & 0 & 0 & b_{1,1} & b_{1,2} & b_{1,3} \\
0 & 0 & 0 & b_{2,1} & b_{2,2} & b_{2,3} \\
0 & 0 & 0 & b_{3,1} & b_{3,2} & b_{3,3}
\end{pmatrix} = \\
&\begin{pmatrix}
a_{1,1}+b_{1,1} & b_{1,2} & b_{1,3} & a_{1,2} & 0 & 0 \\
b_{2,1} & a_{1,1}+b_{2,2} & b_{2,3} & 0 & a_{1,2} & 0 \\
b_{3,1} & b_{3,2} & a_{1,1}+b_{3,3} & 0 & 0 & a_{1,2} \\
a_{2,1} & 0 & 0 & a_{2,2}+b_{1,1} & b_{1,2} & b_{1,3} \\
0 & a_{2,1} & 0 & b_{2,1} & a_{2,2}+b_{2,2} & b_{2,3} \\
0 & 0 & a_{2,1} & b_{3,1} & b_{3,2} & a_{2,2}+b_{3,3}
\end{pmatrix}.
\end{aligned}
$$

Example 5
In this example, we use the matrices

$$C = \begin{pmatrix} 0 & a & 0 \\ b & 0 & c \\ 0 & 0 & 0 \end{pmatrix} \text{ and } D = \begin{pmatrix} 0 & d \\ 0 & 0 \end{pmatrix}.$$

The corresponding graphs are depicted in Figure 3.3a and 3.3b. The matrix $C \oplus D$ can be calculated as follows.

$$
\begin{aligned}
&C \oplus D = C \otimes I_2 + I_3 \otimes D = \\
&\begin{pmatrix} 0 & a & 0 \\ b & 0 & c \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & d \\ 0 & 0 \end{pmatrix} = \\
&\begin{pmatrix}
0 & 0 & a & 0 & 0 & 0 \\
0 & 0 & 0 & a & 0 & 0 \\
b & 0 & 0 & 0 & c & 0 \\
0 & b & 0 & 0 & 0 & c \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} + \begin{pmatrix}
0 & d & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & d & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & d \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} = \begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{pmatrix}
0 & d & a & 0 & 0 & 0 \\
0 & 0 & 0 & a & 0 & 0 \\
b & 0 & 0 & d & c & 0 \\
0 & b & 0 & 0 & 0 & c \\
0 & 0 & 0 & 0 & 0 & d \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}.
\end{aligned}
$$



(a) C (b) D (c) $C \oplus D$

Figure 3.3: The Graphs of C, D, and $C \oplus D$

The corresponding graph is depicted in Figure 3.3c.

3.2.2 Properties

In the following, we list and proof some properties of the Kronecker sum of matrices A, B, and C. The Kronecker sum is non-commutative because for element-wise comparison in general $A \oplus B \neq B \oplus A$. However, it essentially commutes because from a graph point of view, the graphs represented by matrices $A \oplus B$ and $B \oplus A$ are structurally isomorphic, i.e., starting from the entry node to the final node the same paths in terms of edge labels are possible. Only the node numbers may be different in $A \oplus B$ compared to $B \oplus A$. Both matrices, namely $A \oplus B$ and $B \oplus A$, consist of the same number of entries unequal to 0 and have the same order.

Now we state a property of the Kronecker sum which we call *Mixed Sum Rule.*

Lemma 1 *Let the matrices A and C have order m and B and D have order n. Then we call*

$$(A \oplus B) + (C \oplus D) = (A + C) \oplus (B + D)$$

the Mixed Sum Rule*.*

Proof 1 *By using Equations* (3.2) *and* (3.3) *and Definition 4 we get*

$$\begin{aligned}(A \oplus B) + (C \oplus D) &= A \otimes I_n + I_m \otimes B + C \otimes I_n + I_m \otimes D \\ &= (A + C) \otimes I_n + I_m \otimes (B + D) \\ &= (A + C) \oplus (B + D).\end{aligned}$$

For example let the matrices A and B be written as $A = \sum_{i \in I} A_i$ and $B = \sum_{j \in J} B_j$, respectively. In addition, let the sets I and J have the same number of elements, i.e., $|I| = |J|$. By using the mixed sum rule we can write $A \oplus B = \sum_{i \in I, j \in J} A_i \oplus B_j$.

We will frequently use the *Mixed Sum Rule* from now on without further notice.

Kronecker sum is also associative, as $(A \oplus B) \oplus C$ and $A \oplus (B \oplus C)$ are equal.

Lemma 2 *Kronecker sum is associative.*

Proof 2 *In the following, we will use $I_m \otimes I_n = I_{m.n}$. Note that Z denotes zero matrices. We have*

$$\begin{aligned}
A \oplus (B \oplus C) &= A \oplus (B \otimes I_{o(C)} + I_{o(B)} \otimes C) \\
\{\textit{adding } Z_{o(A)}\} &= (A + Z_{o(A)}) \oplus (B \otimes I_{o(C)} + I_{o(B)} \otimes C) \\
\{\textit{Lemma 1}\} &= (A \oplus (B \otimes I_{o(C)})) + (Z_{o(A)} \oplus (I_{o(B)} \otimes C)) \\
\{\textit{Equation } (3.1),\ \textit{Definition 4}\} &= (A \oplus (B \otimes I_{o(C)})) + I_{o(A)} \otimes I_{o(B)} \otimes C \\
\{\textit{associativity of } +,\ \textit{Definition 4}\} &= A \otimes I_{o(B).o(C)} + I_{o(A)} \otimes B \otimes I_{o(C)} + \\
&\quad\ I_{o(A).o(B)} \otimes C \\
\{\textit{commutativity of } +\} &= A \otimes I_{o(B)} \otimes I_{o(C)} + I_{o(A).o(B)} \otimes C + \\
&\quad\ I_{o(A)} \otimes B \otimes I_{o(C)} \\
\{\textit{Definition 4}\} &= ((A \otimes I_{o(B)}) \oplus C) + I_{o(A)} \otimes B \otimes I_{o(C)} \\
\{\textit{Definition 4}\} &= ((A \otimes I_{o(B)}) \oplus C) + ((I_{o(A)} \otimes B) \oplus Z_{o(C)}) \\
\{\textit{Lemma 1}\} &= (A \otimes I_{o(B)} + I_{o(A)} \otimes B) \oplus (C + Z_{o(C)}) \\
\{\textit{remove } Z_{o(C)}\} &= (A \otimes I_{o(B)} + I_{o(A)} \otimes B) \oplus C \\
\{\textit{Definition 4}\} &= (A \oplus B) \oplus C.
\end{aligned}$$

The associativity properties of the operations $\otimes$ and $\oplus$ imply that the n-fold operations

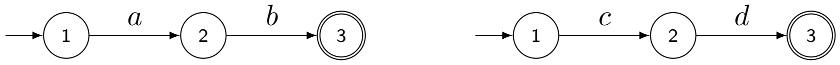
$$\bigotimes_{i=1}^{k} A_i \quad \text{and} \quad \bigoplus_{i=1}^{k} A_i$$

are well defined. Let n_i denote the order of matrix A_i and I_n the identity matrix of order n. Then we can write the n-fold Kronecker sum for k matrices A_i, where $1 \leq i \leq k$ similar to Buchholz and Kemper [BK02] and Ciardo et al. [CM99] as

$$\bigoplus_{i=1}^{k} A_i = \sum_{i=1}^{k} I_{n_1} \otimes \cdots \otimes I_{n_{i-1}} \otimes A_i \otimes I_{n_{i+1}} \otimes \cdots \otimes I_{n_k} = \sum_{i=1}^{k} I_{\prod_{j=1}^{i-1} n_j} \otimes A_i \otimes I_{\prod_{j=i+1}^{k} n_j} .$$

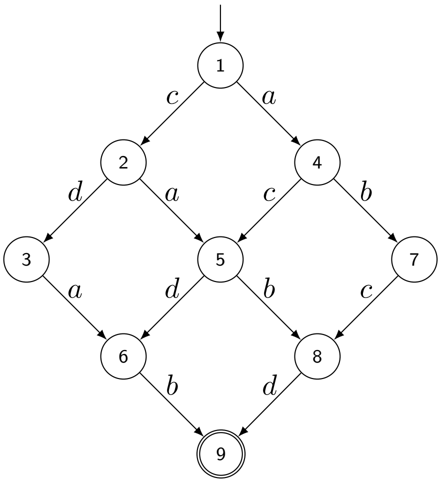
Finally, note that the Kronecker sum of adjacency matrices generates only self-loops out of self-loops in the input matrices. This can easily be seen because on the main diagonal, the Kronecker sum contains only entries of the main diagonal of the two input matrices. A self-loop at node i is defined as an entry $m_{i,i} \neq 0$ in the corresponding adjacency matrix M.

Additional properties of the Kronecker sum can be found in [PA91].



(a) Graph for C (b) Graph for D

Interleavings
$a \cdot b \cdot c \cdot d$
$a \cdot c \cdot b \cdot d$
$a \cdot c \cdot d \cdot b$
$c \cdot a \cdot b \cdot d$
$c \cdot a \cdot d \cdot b$
$c \cdot d \cdot a \cdot b$



(c) Interleavings (d) Graph for $C \oplus D$

Figure 3.4: A Simple Cartesian Product Graph ($C \oplus D$)

3.2.3 Relation of Kronecker Sum to Cartesian Product Graph

By calculating the Kronecker sum of the adjacency matrices of two graphs G_1 and G_2, the adjacency matrix of the Cartesian product graph [IKR08, HIK11] of G_1 and G_2 is computed (cf. [Knu11]).

The Kronecker sum calculates all possible interleavings of two concurrently executing automata (see e.g. [Küs91] for a proof; in that context the Kronecker sum is referred to as Hurwitz product) even for general graphs, e.g., CFGs including conditionals and loops. In Chapter 4 we calculate the maximum number of nodes and edges in such Cartesian graphs, whereas in the Appendix A it is shown how the number of interleavings can be calculated for an arbitrary number of threads and their statements. The following example illustrates interleavings of two threads and how the Kronecker sum handles it.

Example 6 *Let the matrices C and D be defined as follows:*

$$C = \begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix} \qquad D = \begin{pmatrix} 0 & c & 0 \\ 0 & 0 & d \\ 0 & 0 & 0 \end{pmatrix}.$$

The graph corresponding to matrix C is depicted in Figure 3.4a, whereas the graph of matrix D is shown in Figure 3.4b. The regular expressions associated with the CFGs are $a \cdot b$ and $c \cdot d$, respectively. All possible interleavings by executing C and D applying *an interleavings semantics are shown in Figure 3.4c. The adjacency matrix* $C \oplus D$ *is calculated by*

$$
\begin{aligned}
C \oplus D = C \otimes I_3 + I_3 + D = \\
&= \begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & c & 0 \\ 0 & 0 & d \\ 0 & 0 & 0 \end{pmatrix} \\
&= \begin{pmatrix}
0 & 0 & 0 & a & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & a & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & a & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & b & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & b & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & b \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} + \begin{pmatrix}
0 & c & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & d & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & c & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & d & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & c & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & d \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} \\
&= \begin{array}{c|ccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
1 & 0 & c & 0 & a & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & d & 0 & a & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & a & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & c & 0 & b & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & d & 0 & b & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & b \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & d \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} .
\end{aligned}
$$

In Figure 3.4d the graph represented by the adjacency matrix $C \oplus D$ *is depicted. It is easy to see that all possible interleavings are generated.*

As a concluding remark of this chapter, we note that Kronecker algebra is a special case of tensor algebra. The tensor product of matrices is the Kronecker product and the tensor sum of matrices is the Kronecker sum of the matrices (cf. [Pla85]).

CHAPTER 4

Concurrent Program Graphs

> *"A computer program is organized complexity."*
>
> – EDSGER WYBE DIJKSTRA, Dutch computer scientist
> and ACM Turing Award winner 1972, 1930-2002

In this chapter, we introduce *concurrent program graphs* (CPGs) as a data structure modeling concurrent systems. CPGs play a similar role for concurrent systems as control flow graphs (CFGs) do for sequential programs. They describe, e.g., all possible interleavings and incorporate synchronization. In the course of this dissertation, we will see that CPGs can be used as a basis for analyses of issues, e.g., deadlocks, inherent in concurrent programs.

Our system model consists of a finite number of threads and semaphores. Both, threads and semaphores, are represented by *Control Flow Graphs* (CFGs). *Edge splitting* has to be applied to the edges of threads which results in *Refined CFGs* (RCFGs), whereas the CFGs of semaphores are RCFGs without modification. The RCFGs are stored in form of adjacency matrices. The matrices have entries which are referred to as labels $l \in \mathcal{L}$ as defined in Section 2.1. Let $\mathcal{S}$ and $\mathcal{T}$ be the sets of adjacency matrices representing RCFGs of semaphores and threads, respectively. The matrices are manipulated by using conventional Kronecker algebra operations together with some extensions which we define in the course of this chapter. Similar to [BK02] we describe synchronization by Kronecker products and thread interleavings by Kronecker sums.

Formally, the system model consists of the tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where

- $\mathcal{T}$ is the set of RCFG adjacency matrices describing threads,
- $\mathcal{S}$ refers to the set of RCFG adjacency matrices describing semaphores, and

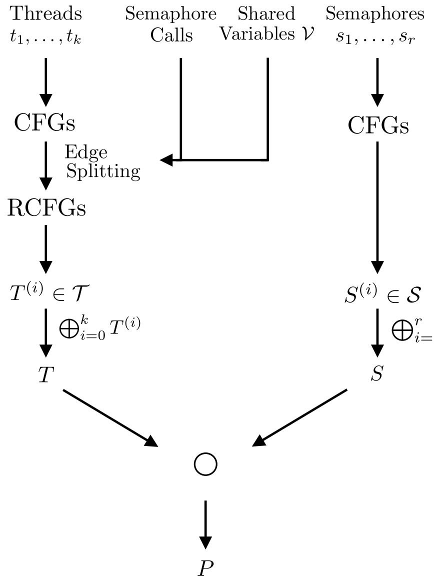

Threads $t_1, \dots, t_k$
Semaphore Calls
Shared Variables $\mathcal{V}$
Semaphores $s_1, \dots, s_r$
CFGs
Edge Splitting
CFGs
RCFGs
$T^{(i)} \in \mathcal{T}$
$\bigoplus_{i=0}^{k} T^{(i)}$
$S^{(i)} \in \mathcal{S}$
$\bigoplus_{i=0}^{r} S^{(i)}$
T
S
$\circ$
P

Figure 4.1: Overview

- $\mathcal{L}$ denotes the set of labels out of the semiring defined in Section 2.1. The labels (or matrix entries) of the ith thread's adjacency matrix $T^{(i)} \in \mathcal{T}$ are elements of $\mathcal{L}$, whereas the labels (or matrix entries) of the jth synchronization primitive's adjacency matrix $S^{(j)} \in \mathcal{S}$ are elements of $\mathcal{L}_S$.

A *concurrent program graph* (CPG) is a graph $C = \langle V, E, n_e, V_f \rangle$ with a set of nodes V, a set of directed edges $E \subseteq V \times V$, a so-called *entry* node $n_e \in V$ and a set of *final nodes* $V_f \subseteq V$. The sets V and E are constructed out of the elements of $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$. Details on how we generate the sets V and E follow in the next sections. Similar to RCFGs, the edges of CPGs are labeled by $l \in \mathcal{L}$.

In Figure 4.1 an overview of how we generate a concurrent program's adjacency matrix is given. As described in Section 2.5, the semaphore calls and the shared variable information may be used as input for *edge splitting*. As an output, we get RCFGs represented by the matrices $T^{(i)}$. The matrices T and S are the interleaved threads and semaphores, respectively. The ring operator $\circ$ is used to generate the matrix P out of T and S. The matrix P statically represents the concurrent programm consisting of k threads and r semaphores. The underlying graph is the CPG. In the following, we define the $\circ$-operation used in Figure 4.1.

We will see that for concurrent programs containing synchronization only a part of the graph represented by the full matrix is reachable. To this part of the CPG we refer to as *reachable CPG* (RCPG). Because our implementation only generates RCPGs, we usually mean the reachable part (RCPG) when referring to a CPG. The full matrix is only interesting when, e.g., arguing about worst-case scenarios regarding the number of nodes.

The remainder of this chapter is organized as follows. In Section 4.1 we show how the matrix of a concurrent program can be generated from a mathematical point of view. How we determine entry and final nodes of CPGs is discussed in 4.2. We prove that CPGs correctly model synchronization in Section 4.3. In Section 4.4 it is shown how unreachable parts arise from synchronization between threads. We prove important properties in Section 4.5. In Section 4.6 we extend the Kronecker algebra with our selective Kronecker product and show how the definitions of Section 4.1 can be simplified. Section 4.7 is devoted to a lazy implementation of our extended Kronecker algebra. Finally, in Section 4.8 it is proven that in general CPGs are irreducible.

4.1 Generating a Concurrent Program's Matrix

Let $T^{(i)} \in \mathcal{T}$ and $S^{(i)} \in \mathcal{S}$ refer to the matrices representing thread i and synchronization primitive i, respectively. Let $M = (m_{i,j}) \in \mathcal{M}$. In addition, we define the matrix M_l with entries equal to l and zeros elsewhere:

$$M_l = (m_{l;i,j}), \text{ where } m_{l;i,j} = \begin{cases} l & \text{if } m_{i,j} = l, \\ 0 & \text{otherwise.} \end{cases}$$

We obtain the matrix T representing k interleaved threads by

$$T = \bigoplus_{i=1}^{k} T^{(i)}, \text{ where } T^{(i)} \in \mathcal{T}.$$

According to Figure 2.3, we have for binary semaphore i and counting semaphore j the following adjacency matrix of order two and three, respectively.

$$S^{(i)} = \begin{pmatrix} 0 & p_i \\ v_i & 0 \end{pmatrix} \text{ and } S^{(j)} = \begin{pmatrix} 0 & p_j & 0 \\ v_j & 0 & p_j \\ 0 & v_j & 0 \end{pmatrix}$$

In a similar fashion we can model counting semaphores of higher order.

The matrix S representing r interleaved synchronization primitives is given by

$$S = \bigoplus_{i=1}^{r} S^{(i)}, \text{ where } S^{(i)} \in \mathcal{S}.$$

The adjacency matrix representing program $\mathcal{P}$ referred to as P is defined as

$$P = T \circ S = \sum_{l \in \mathcal{L}_\mathrm{S}} (T_l \otimes S_l) + \sum_{l \in \mathcal{L}_\mathrm{V}} (T_l \oplus S_l). \tag{4.1}$$

When calculating the left term of Equation 4.1, the Kronecker product exclusively operates on labels $s_x \in \mathcal{L}_\mathrm{S}$, thus on labels referring to synchronization primitive calls. In the resulting matrix, we let the Kronecker product generate only entries consisting of two equal and concatenated labels $s_x \cdot s_x$ and replace it with the single label s_x. This rule means that, e.g., for all v_x and $p_x \in \mathcal{L}_\mathrm{S}$:

- $v_x \cdot v_x$ is replaced by v_x and
- $p_x \cdot p_x$ is replaced by p_x.

Note that, during the evaluation of the Kronecker product, all other combinations, e.g., $v_x \cdot p_x$ and $v_x \cdot v_y$, where $x \neq y$ will not happen by definition because both input matrices are filtered such that both contain only entries of the same label. In Subsection 4.6, we describe how the $\circ$-operation can be implemented efficiently and how we get rid of this rather unaesthetic replacement.

4.2 Determining Entry and Final Nodes of CPGs

Assuming, without loss of generality, that each thread has an entry node with index 1, the entry node of the generated CPG has index 1, too.

When a program uses only binary semaphores as shown in Figure 2.3a, then the program's final node can be calculated as follows. If the r semaphores have an entry and a final node 1 and each thread i of k threads has one final node n_i and n_i is the order of the corresponding adjacency matrix of thread i (thus thread i's final node has the highest row/column number[1]), then the final node of the resulting CPG has final node

$$\prod_{i=1}^{k} n_i \cdot 2^r - 2^r + 1. \tag{4.2}$$

In general, a thread's CPG may have several final nodes and different types of synchronization primitives, e.g., semaphores and/or the barrier synchronization construct introduced below in Chapter 7. For these cases the following formula can be used to determine the final nodes. We refer to a node without outgoing edges as a sink node. Each sink node appears as a zero line in the corresponding adjacency matrix. A CPG's final node may also be a sink node (if the program terminates). However, CPG sink nodes and final nodes can be distinguished as follows. We use a vector determining the final nodes of thread i, namely $F^{(i)}$. In addition, vector $G^{(j)}$ determines the final node of the synchronization primitive j. Both have ones at places q, when node q is a final node, zeros elsewhere. Then the vector

$$\bigotimes_{i=1}^{k} F^{(i)} \otimes \bigotimes_{j=1}^{r} G^{(j)} \tag{4.3}$$

determines the final nodes of the CPG. Again, an one in the resulting vector states that the corresponding node is a final node.

Note if thread i is a daemon[2] thread, we use $F^{(i)} = (1, 1, \ldots, 1)^T$ which informally speaking states that the daemon is always ready to terminate. On the other hand,

---

[1] As stated above, we refer to nodes and their column/row numbers synonymously.

[2] In contrast to non-daemon threads, all daemon threads terminate after the last non-daemon thread terminates.

if thread i does not terminate, then we have $F^{(i)} = (0, 0, \dots, 0)^T$. In such cases the concurrent program's CPG will not have a final node. Thus the concurrent program will not terminate. Only threads within the concurrent program having final nodes may terminate.

Even when a final node in a CPG exists, it may be unreachable due to synchronization. Unreachable parts are discussed below in this chapter (in Section 4.4) and in Chapter 5.

In the remainder of this dissertation, we assume that all threads do have only one single final node. Our results, however, can be generalized easily to an arbitrary number of final nodes.

4.3 ∘-Operation and Synchronization

Theorem 1 *Let $T = \bigoplus_{i=1}^{k} T^{(i)}$ be the matrix representing k interleaved threads and let S be a binary semaphore. Then $T \circ S$ correctly models synchronization of T with semaphore S.*[3]

Proof 3 *First we observe that*

1. *the first term in the definition of Equation* (4.1) *replaces*
   - *each p in matrix T with $\begin{pmatrix} 0 & p \\ 0 & 0 \end{pmatrix}$ and*
   - *each v in matrix T with $\begin{pmatrix} 0 & 0 \\ v & 0 \end{pmatrix}$,*
2. *the second term replaces each $m \in \mathcal{L}_\mathcal{V}$ with $\begin{pmatrix} m & 0 \\ 0 & m \end{pmatrix}$, and*
3. *both terms replace each 0 by $\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$.*

According to these replacements the order of matrix $T \circ S$ has doubled compared to T.

Now, consider the paths in the graph underlying T described by the regular expression

$$\pi = \left(\sum\nolimits_{m \in \mathcal{L}_\mathcal{V}} m\right)^* \left(p \left(\sum\nolimits_{m \in \mathcal{L}_\mathcal{V}} m\right)^* v \left(\sum\nolimits_{m \in \mathcal{L}_\mathcal{V}} m\right)^*\right)^* .$$

By the observations above it is easy to see that paths containing π are present in $T \circ S$. On the other hand, paths not containing π are no more present in $T \circ S$. Thus the semaphore operations always occur in (p, v) pairs in all paths in $T \circ S$. This, however, exactly mirrors the semantics of synchronization via a semaphore.

---

[3] Note that we do not restrict the structure of T.

Generalizing Theorem 1, it is easy to see that the synchronization property is also correctly modeled if we replace the binary semaphore by a counting semaphore of higher order. In addition, the synchronization property is correctly modeled even if more than one semaphore is present on the right-hand side of $T \circ S$.

4.4 Unreachable Parts Caused by Synchronization

In this section, we show that synchronization causes unreachable parts. As an example consider Figure 4.2. The program consists of two threads, namely T_1 and T_2. The RCFGs of the threads are shown in Figure 4.2a and Figure 4.2b. Synchronization is done via a binary semaphore similar to Figure 2.3a. Its operations are referred to as p_1 and v_1. We denote a p and v-call to semaphore x of thread t as $t.p_x$ and $t.v_x$, respectively. T_1 and T_2 access the same shared variable in a and b, respectively. The semaphore is used to ensure that a and b are accessed mutually exclusively. Note that a and b may actually be subgraphs consisting of multiple nodes and edges.

Thus, we get the matrices

$$T_1 = \begin{pmatrix} 0 & p_1 & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & v_1 \\ 0 & 0 & 0 & 0 \end{pmatrix}, T_2 = \begin{pmatrix} 0 & p_1 & 0 & 0 \\ 0 & 0 & b & 0 \\ 0 & 0 & 0 & v_1 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and } S = \begin{pmatrix} 0 & p_1 \\ v_1 & 0 \end{pmatrix}.$$

We obtain the matrix $T = T_1 \oplus T_2$, a matrix of order 16, consisting of the submatrices defined above and zero matrices of order four (instead of Z_4 simply denoted by 0) as follows:

$$T = \begin{pmatrix} T_2 & p_1 \cdot I_4 & 0 & 0 \\ 0 & T_2 & a \cdot I_4 & 0 \\ 0 & 0 & T_2 & v_1 \cdot I_4 \\ 0 & 0 & 0 & T_2 \end{pmatrix}.$$

In order to enable a concise presentation of $T \circ S$ we define the matrices

$$U = \begin{pmatrix} 0 & 0 & 0 & p_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & b & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & b & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, V = \begin{pmatrix} 0 & p_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & p_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & p_1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$W = a \cdot I_8, \text{ and } X = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & v_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & v_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v_1 & 0 \end{pmatrix} \text{ of order 8.}$$

The matrix $T \circ S$, a matrix of order 32, consisting of the submatrices defined above and zero matrices of order eight (instead of Z_8 simply denoted by 0) is given by

$$T \circ S = \begin{pmatrix} U & V & 0 & 0 \\ 0 & U & W & 0 \\ 0 & 0 & U & X \\ 0 & 0 & 0 & U \end{pmatrix}.$$

The reachable part of the CPG is depicted in Figure 4.2c. The resulting adjacency matrix has order 32, whereas the reachable part of the CPG consists only of 12 nodes and 12 edges. Large parts (20 nodes and 20 edges) are unreachable from the entry node. They are depicted in Figure 4.3.

In general, unreachable parts exist if a concurrent program contains synchronization. If a program contains a lot of synchronization, then the reachable parts may be very small. The reasons why parts of the CPG are unreachable can be summarized as follows: Kronecker product limits the number of possible paths such that the p- and v-operations are present in correct p-v-pairs. In contrast $T = \bigoplus_{i=1}^{k} T^{(i)}$ contains all possible paths even those containing semantically wrong uses of the semaphore operations. This observation motivates the lazy implementation described in Section 4.7. In the following, we denote the subgraph of a CPG, whose nodes are reachable from the entry node, *Reachable CPG* (RCPG).

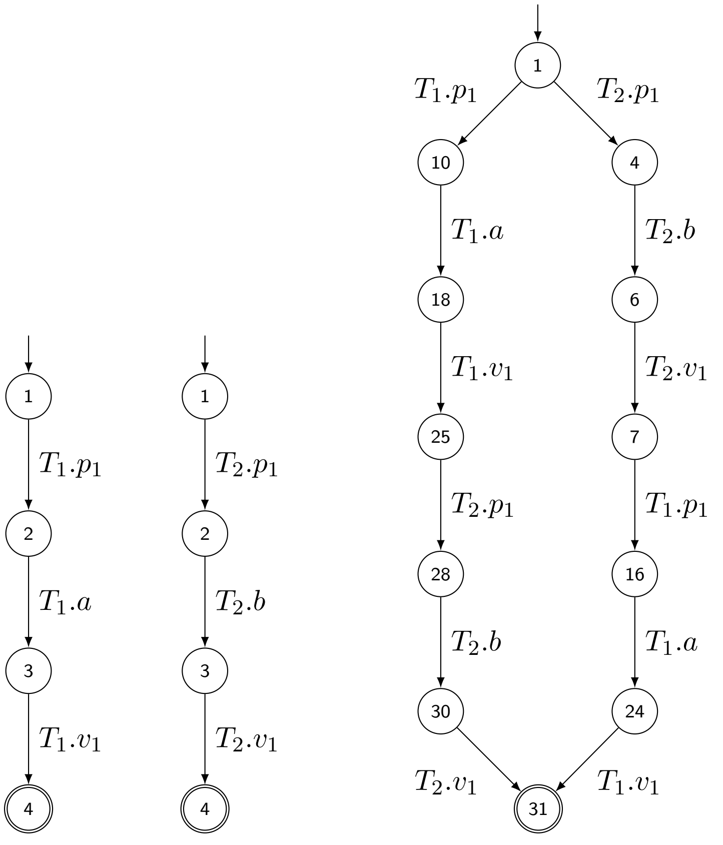
4.5 Properties of CPGs

In this section, we prove interesting properties of the adjacency matrices of CPGs.

A short calculation shows that the Kronecker sum in general generates at most $mn^2 + nm^2 - nm$ entries unequal to zeros. Stated the other way, at least $(mn)^2 - mn^2 - nm^2 + mn$ entries are zero. We will see that for CFGs and RCFGs there are even more zero entries. We will prove that for this case the number of edges is in $O(mn)$. Thus, the number of edges is linear in the order of the resulting adjacency matrix.

Lemma 3 (Maximum Number of Nodes) *Given a program $\mathcal{P}$ consisting of $k > 0$ threads $(t_1, t_2, \ldots, t_k)$, where each t_i has n nodes in its RCFG, the number of nodes in $\mathcal{P}$'s adjacency matrix P is bounded from above by n^k.*

$\square$



(a) T_1 (b) T_2 (c) Reachable Part of the CPG

Figure 4.2: Mutual Exclusion Example

In general, if thread t_i's RCFG has n_i nodes, then the maximum number of nodes in $\mathcal{P}$'s adjacency matrix is

$$\prod_{i=1}^{k} n_i,$$

where k is the number of threads in program $\mathcal{P}$.

As already mentioned in Section 2.3, a CFG node has maximal two outgoing edges. Thus, for RCFGs with n nodes it is easy to see that at most $2n$ edges can be contained. We use this property in the following lemma.

Lemma 4 (Maximum Number of Entries) *Given a program represented by $M_k \in \mathcal{M}$ consisting of $k > 0$ threads represented by the matrices $T^{(i)} \in \mathcal{T}$, where each $T^{(i)}$ has order n, then the number of matrix entries $||M_k||$ is bounded from above by $2k\,n^k$.*

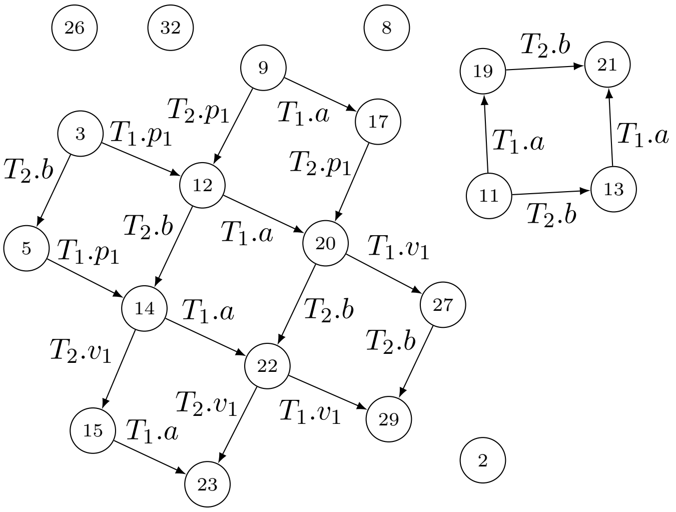


Figure 4.3: Unreachable Parts of the Mutual Exclusion Example

Proof 4 *We prove this lemma by induction on the definition of the Kronecker sum. For* $k = 1$ *the lemma is true. If we assume that for* m *threads* $||M_m|| \leq 2m\, n^m$*, then for* $m + 1$ *threads* $||M_{m+1}|| \leq 2m\, n^m \cdot n + n^m \cdot 2n = 2(m + 1)\, n^{m+1}$*. Thus, we have proved Lemma 4.*

Compared to the full matrix of order n^k with n^{2k} entries the resulting matrix has significantly fewer non-zero entries, namely $2k\, n^k$. Thus a CPG with n^k nodes can have at most $2k\, n^k$ edges. Roughly speaking, this can be explained as follows. At a certain CPG node, each of the k threads can execute one out of its next (at maximum) two basic blocks. From Lemma 3 we know that this is done for n^k nodes.

The observations from above directly lead to a next property, namely the maximum outdegree of CPG nodes. The outdegree of a certain node in a directed graph is the number of outgoing edges. In general, the maximum outdegree of CPG nodes is $2k$.

The following two lemmata, we have originally published in [MB11]. By using Definition 2, we will prove that the matrices of CPGs are sparse.

Lemma 5 (Sparsity of Control Flow Graphs) *CFGs and RCFGs have Sparse Adjacency Matrices.*

Proof 5 *Let* $G = \langle V, E, n_e \rangle$ *be a CFG consisting of* $|V|$ *nodes. From Subsection 2.3, we know that the number of edges* $|E|$ *in* G *is bounded from above by* $2 \cdot |V|$*. By using Definition 2, we get for the maximum number of entries in the adjacency matrix* M*:*

$$||M|| \leq 2 \cdot |V|.$$

Lemma 6 (Sparsity of CPGs) *The Matrix P of a Concurrent Program $\mathcal{P}$ is Sparse.*

Proof 6 *Let $T = \bigoplus_{i=1}^{k} T^{(i)} \in \mathcal{M}$ be an N-by-N adjacency matrix of a program. We require that each of the k threads has order n in its adjacency matrix $T^{(i)}$. From Lemma 4 we know $||T|| \leq 2k\, n^k$. In addition, $N = n^k$ is given by Lemma 3. Hence, for k threads, we get $||T|| \leq 2k\, n^k = 2k\, N$ and by using Definition 2 it is proved that matrix T is sparse because $||T|| \leq 2k\, N$, where $2k$ is constant. A similar result holds for S and $P = T \circ S$.*

Lemma 6 enables the application of memory saving data structures and efficient algorithms. Algorithms may for example work on adjacency lists. Clearly, the space requirements for the adjacency lists are linear in the size of the nodes. In the worst-case, however, the number of CPG nodes increases exponentially in k.

4.6 Efficient Implementation of the ∘-Operation

This section is devoted to an efficient implementation of the ∘-operation. First we define the selective Kronecker product which we denote by $\oslash_L$. This operator synchronizes only labels $l \in L \subseteq \mathcal{L}$ identical in the two input matrices.

Definition 5 (Selective Kronecker Product) *Given an m-by-n matrix A and a p-by-q matrix B, we call $A \oslash_L B$ their selective Kronecker product. For all $l \in L \subseteq \mathcal{L}$ let $A \oslash_L B = (a_{i,j}) \oslash_L (b_{r,s}) = (c_{t,u})$, where*

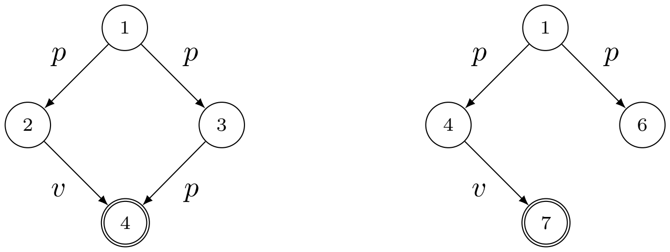
$$c_{(i-1)\cdot p+r,(j-1)\cdot q+s} = \begin{cases} l & \text{if } a_{i,j} = b_{r,s} = l,\ l \in L, \\ 0 & \text{otherwise.} \end{cases}$$

In contrast to the plain Kronecker product, the selective Kronecker product is defined such that

- a label l in the left operand is paired with the same label in the right operand and not with any other label in the right operand and
- for $a_{i,j} = b_{r,s} = l,\ l \in L$ the resulting entry is l and not $l \cdot l$.

Definition 5 is defined for a set of labels L. In practice, we usually constrain $L = \mathcal{L}_{\mathrm{S}}$. Thus, we use this operation only for labels referring to semaphore or other synchronization primitive calls/operations. Used that way, the selective Kronecker product ensures that, e.g., a p-call to semaphore i, i.e., a p_i-call, in the left operand is paired with the corresponding p_i-operation in the right operand and not with any other label, e.g., p_j of a semaphore $j \neq i$, in the right operand. All other entries are 0.

Figure 4.4 shows a small example for the application of the selective Kronecker product $\oslash$. The program in Figure 4.4a has two branches. Both branches are calling operations of an initially unlocked semaphore (cf. Figure 2.4a). The left branch executes the calls of p and v in a correct order. The right branch contains two p-calls. When applying Kronecker



(a) CFG

(b) CPG

$$\begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & p \\ 0 & 0 & 0 & 0 \end{pmatrix} \oslash_{\{p,v\}} \begin{pmatrix} v & p \\ v & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & p & 0 & p & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & p \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(c) Matrices

Figure 4.4: Example Program with Self-Deadlock

algebra (cf. Lemma 7) in Figure 4.4c, we obtain the CPG depicted in Figure 4.4b. Node 6 shows that there is a self-deadlock in the underlying program because it is a sink node but not a final node. The final node can be calculated by applying Formula 4.2 defined in Section 4.2. A short calculation ($4 * 2 - 2 + 1 = 7$) shows that node 7 is the final node. In addition, note that compared to the adjacency matrix of Figure 4.4c some CPG nodes are unreachable from the entry node 1. These unreachable nodes are excluded in the CPG shown in Figure 4.4b.

Remember, we refer to the identity matrix of order n as I_n. In addition, we write $o(M)$ to denote the order of matrix M.

Definition 6 (Filtered Matrix) *We call M_L a* filtered matrix *and define it as a matrix of order $o(M)$ containing entries of $L \subseteq \mathcal{L}$ of $M = (m_{i,j})$ and zeros elsewhere:*

$$M_L = (m_{L;i,j}),\ \textit{where}\ m_{L;i,j} = \begin{cases} m_{i,j} & \textit{if } m_{i,j} \in L, \\ 0 & \textit{otherwise.} \end{cases}$$

Note that

$$\sum_{l \in \mathcal{L}_S} (T_l \otimes S_l) = T \oslash_{\mathcal{L}_S} S. \tag{4.4}$$

Intuitively, Equation 4.4 can be explained as follows. The left side filters the matrices for a certain element of synchronization labels, then builds the Kronecker product of the two results of T_l and S_l, and then sums up the results for all synchronization labels. The right side of the equation does not filter in the first place. It applies the selective Kronecker product for the set of synchronization labels. Note that Equation 4.4 is only correct, when the replacing rules, as done for Equation 4.1, are applied when applying the plain Kronecker product. Thus, for the plain Kronecker product, we replace $v_x \cdot v_x$ by v_x and $p_x \cdot p_x$ by p_x. All other entries of the plain Kronecker product are 0.

In the following, we use $o(S_{\mathcal{L}_{\mathrm{V}}}) = \prod_{i=1}^{r} o(S^{(i)}) = o(S)$. Note that S contains only labels $l \in \mathcal{L}_{\mathrm{S}}$. Hence, when the $\circ$-operator is applied for a label $l \in \mathcal{L}_{\mathrm{V}}$, we get $S_l = Z_{o(S)}$, i.e., a zero matrix of order $o(S)$. Thus we obtain $\sum_{l \in \mathcal{L}_{\mathrm{V}}} (T_l \oplus S_l) = T_{\mathcal{L}_{\mathrm{V}}} \otimes I_{o(S)}$. We will prove this below.

Finally, we can refine Equation (4.1) by stating the following lemma.

Lemma 7 *The $\circ$-operation can be computed efficiently by*

$$P = T \circ S = T \oslash_{\mathcal{L}_S} S + T_{\mathcal{L}_V} \otimes I_{o(S)}.$$

Proof 7 *Using Equation* (4.1) $P = T \circ S$ *is given by* $\sum\limits_{l \in \mathcal{L}_S} (T_l \otimes S_l) + \sum\limits_{l \in \mathcal{L}_V} (T_l \oplus S_l)$. *According to Equation* (4.4) *the first term is equal to* $T \oslash_{\mathcal{L}_S} S$*. By mentioning* $S_l = Z_{o(S)}$ *for* $l \in \mathcal{L}_V$*, Lemma 1, and Definition 4, the second term fulfills*

$$\sum_{l \in \mathcal{L}_V} (T_l \oplus S_l) = \sum_{l \in \mathcal{L}_V} \left(T_l \oplus Z_{o(S)}\right) = T_{\mathcal{L}_V} \oplus Z_{o(S)} = T_{\mathcal{L}_V} \otimes I_{o(S)}.$$

Note that S contains only $l \in \mathcal{L}_S$. It is obvious that the non-zero entries of the first and the second term are $l \in \mathcal{L}_S$ and $l \in \mathcal{L}_V$, respectively. Both terms can be computed by iterating once through the corresponding sparse adjacency matrices, namely T and S.

□

Intuitively, the selective Kronecker product term on the left of Lemma 7 allows for synchronization between the threads represented by T and the synchronization primitives S. Both matrices, namely T and S, are Kronecker sums of the involved threads and synchronization primitives, respectively, in order to represent all possible interleavings of the concurrently executing threads. The right term allows the threads to perform steps that are not involved in synchronization. Summarizing, the threads (represented by matrix T) may perform their steps concurrently, where all interleavings are allowed, except when they call synchronization primitives. In the latter case the synchronization primitives (represented by matrix S) together with Kronecker product ensure that these calls are executed in the order prescribed by the deterministic finite automata (DFA) of the synchronization primitives [BB14].

So, for example, a thread cannot do semaphore calls in the order v followed by p when the semaphore DFA only allows a p-call before a v-call. The CPG of such an erroneous program will contain a node from which the final node of the CPG cannot be reached. This node is the one preceding the v-call. Such nodes can easily be found by traversing CPGs. Thus deadlocks of concurrent systems can be detected with little effort (cf. Chapter 5 and [MB11]).

4.7 Lazy Implementation of Kronecker Algebra

Until now we have primarily focused on a pure mathematical model for shared memory concurrent systems. An alert reader will have noticed that the order of the matrices in our CPG increases exponentially in the number of threads. On the other hand, we have seen that the $\circ$-operation results in parts of the matrix $T \circ S$ that cannot be reached from the entry node of the underlying graph. This comes solely from the fact that synchronization excludes many interleavings.

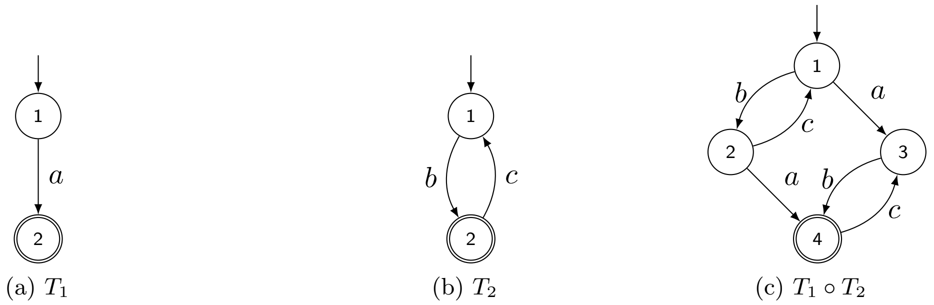
In general, unreachable parts exist if a concurrent program contains synchronization (cf. Section 4.4 and [MB11]). If a program contains a lot of synchronization, the reachable parts may be very small compared to the order of the matrix generated by a naïve approach. This observation motivates the lazy implementation described in this subsection. As introduced in Section 4.4, we denote the subgraph of a CPG whose nodes are reachable from the entry node by *Reachable CPG* (RCPG). An empirical analysis of our approach showed that the runtime complexity of generating an RCPG is linear in the number of RCPG nodes [MB11].

The reasons why parts of the CPG are unreachable can be summarized as follows: The way we adopt the Kronecker product limits the number of possible paths such that, e.g., in case of semaphores, the p- and v-operations are present in correct p-v-pairs in the RCPG. In contrast $T = \bigoplus_{i=1}^{k} T^{(i)}$ contains all possible paths even those containing semantically wrong uses of the semaphore operations.

This contrast can be seen in our example in Figures 8.2 and 8.3. The Kronecker sum (shown in Figure 8.2) of the two threads A and B (as depicted in Figure 8.1) contains five copies of thread A's loop, whereas the RCPG in Figure 8.3 contains this loop only three times. It can be easily seen that the latter reflects the correct use of the semaphore operations p and v.

Choosing a lazy implementation (cf. [HM76]) for the matrix operations ensures that, when extracting the reachable parts of the underlying graph, the overall effort is reduced to exactly these parts. By starting from the RCPG's entry node and calculating all reachable successor nodes, our lazy implementation exactly does this. Thus, for example, if the resulting RCPG's size is linear in terms of the involved threads, only linear effort will be necessary to generate the RCPG. However, if the original problem grows exponentially, then the number of RCPG nodes grows exponentially, too.

Our implementation distinguishes between two types of matrices: Sparse matrices are used for representing threads and semaphores. Lazy matrices are employed for representing all



(a) T_1 (b) T_2 (c) $T_1 \circ T_2$

Figure 4.5: CPGs are Irreducible

the other matrices, i.e., those resulting from the operations of the Kronecker algebra and our ∘-operation. The operations to apply are stored in form of an expression tree. Besides the employed operation, a lazy matrix simply keeps track of its operands. Whenever an entry of a lazy matrix is retrieved, depending on the operation recorded in the lazy matrix, entries of the operands are retrieved and the recorded operation is performed on these entries to calculate the result. In the course of this computation, even the successors of nodes are calculated lazily. Retrieving entries of operands is done recursively if the operands are again lazy matrices, or is done by retrieving the entries from the sparse matrices, where the actual data resides.

In addition, our lazy implementation allows for simple parallelizing. For example, retrieving the entries of left and right operands can be done concurrently. Exploiting this, we achieved further performance improvements for our implementation if run on multi-core architectures.

Even if the RCPG is small (due to synchronization) the RCPG's node numbers may be astronomically big. In general, the biggest node number in the RCPG depends on the order of the CPG's matrix (i.e. $T \circ S$). Even for relatively small multi-threaded programs, the implementation may need more than 64 bit for representing (the positive) matrix row and column indices. Thus we use big integer row and column indices.

When no synchronization primitives, e.g., semaphores or barriers, are present, we expect $P = T$. Because T contains only labels $l \in \mathcal{L}_\mathrm{V}$, we get $P = T_{\mathcal{L}_\mathrm{V}} \otimes I_{o(S)} = T \otimes I_0 = T$. Remember, in Section 2.6 I_0 is defined as (1). This means that the RCPG and the CPG are isomorph when no synchronization is present in the threads.

Note that we often mean RCPGs when we talk about CPGs. Because our implementation never explicitly generates the full matrix of a CPG, we usually get an RCPG. Thus, when we are not explicitly distinguishing between CPGs and RCPGs, we usually mean RCPG.

4.8 CPGs are Irreducible

In general CPGs are irreducible. A graph is called irreducible if it contains loops which can be entered via at least two nodes (cf. [Sre95, SGL98, Mit05]).

As an example consider the two threads T_1 and T_2 depicted in Figure 4.5a and 4.5b, respectively. The system's CPG consisting of these two threads is depicted in Figure 4.5c. Because no semaphores are present in this system, we get the system's matrix $T_1 \circ T_2$ by simply calculate $T_1 \oplus T_2$. The loop $3 \rightarrow 4 \rightarrow 3$ is irreducible because it can be entered via the edges $1 \rightarrow 3$ and $2 \rightarrow 4$.

CHAPTER 5

Deadlocks

> *"Crises and deadlocks when they occur have at least this advantage, that they force us to think."*
>
> – JAWAHARLAL NEHRU, First Indian Prime Minister, 1889-1964

In this chapter, we show how RCPGs, as introduced in Chapter 4, can be used in order to find deadlocks in concurrent programs.

In contrast to livelocks (cf. [BBM07]), where the involved threads cannot make useful progress while still executing (at least some of) their statements, deadlocks force the affected threads to stall.

In this dissertation, we use Stallings' definition of deadlocks [Sta11] and quote it in the following.

Definition 7 (Deadlock [Sta11]) *"An impasse that occurs when multiple processes are waiting for the availability of a resource that will not become available because it is being held by another process that is in a similar wait state."*

In Chapter 9, we also use this definition and relate deadlocks in computer science to deadlocks in railway disposition systems.

In our system model at least two threads and two semaphores are needed for a deadlock to occur. For example consider the threads T_1 and T_2. Thread T_1 starts with calling p of semaphore 1 followed by a call p of semaphore 2. Thread T_2 starts with calling p of semaphore 2 followed by a call p to semaphore 1. If one of the threads can execute its two p calls no deadlocks will occur. But if we come to the point, where each thread has executed its first p-call, then T_1 and T_2 are deadlocked. In this case T_1 holds semaphore 1 and T_2 holds semaphore 2. Each thread now tries to call p of a semaphore which is held by another thread. A similar situation can be seen in Figure 5.1. The two possible paths from node 1 to node 32 in Figure 5.1c reflect the behavior described above. We discuss this example in more detail in Subsection 5.1.

The following theorem shows that deadlocks show up in an RCPG as a pure structural property in the corresponding adjacency matrix. We use Tarjan's algorithm [Tar72] for finding the strongly connected components (SCC) of a graph. If each SCC of a directed graph G is collapsed to a single node, the resulting directed acyclic graph is called the condensation of G. The condensation R_c of RCPG $R = \langle V, E, n_e \rangle$ can be calculated in $O(|V| + |E|)$ time, i.e., in time linear in the size of the RCPG.

As usual, we refer to a node without successor nodes as a sink node. A final node may be a certain sink node in an RCPG or RCFG, where all threads have terminated. If a thread is not supposed to terminate, then its RCFG does not contain a final node.

Theorem 2 *If the program modeled by RCPG R contains a deadlock, then R_c, the condensation of R, contains at least one sink node which is not generated by collapsing final nodes of R. Let ℓ be the node ID of such a sink node. Then R_c's adjacency matrix contains a zero line ℓ, i.e., row ℓ contains only zero entries.*

Proof 8 *Let $D \neq \emptyset$ be the set of nodes in RCPG R from which the final nodes of R cannot be reached. Then R contains at least one deadlock. In this case R_c contains several sinks. Some of these sinks have been generated by collapsing final nodes, some by nodes of set D. The latter are responsible for deadlocks.*

□

False positives may occur. From a static point of view, a deadlock is possible while conditions exclude this case at runtime. Our approach delivers a path to a deadlock in any case. On the other hand, our approach of finding deadlocks is complete. If it states deadlock freedom, then the program under test certainly is deadlock free.

We allow our implementation to stop after the first deadlock is detected, which makes sense since a program containing at least one deadlock is considered to contain an error. The path from the deadlock back to the entry node shows how the deadlock can be reached.

5.1 Deadlock Example

In this subsection, we show how we detect deadlocks. As an example consider Figure 5.1. The program consists of two threads, namely T_1 and T_2. The RCFGs of the threads are shown in Figure 5.1a and Figure 5.1b. Synchronization is done via two binary semaphores similar to Figure 2.3a. Their operations are referred to as p_i and v_i. We denote a p and v-call to semaphore x of thread T_i as $T_i.p_x$ and $T_i.v_x$, respectively.

The matrices for T_1, T_2, S_1, and S_2 are trivially defined as follows:

$$
\begin{aligned}
T_1 &= \begin{pmatrix} 0 & T_1.p_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & T_1.p_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & T_1.a & 0 & 0 \\ 0 & 0 & 0 & 0 & T_1.v_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & T_1.v_1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \\
T_2 &= \begin{pmatrix} 0 & T_2.p_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & T_2.p_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & T_2.b & 0 & 0 \\ 0 & 0 & 0 & 0 & T_2.v_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & T_2.v_2 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \\
S_1 &= \begin{pmatrix} 0 & p_1 \\ v_1 & 0 \end{pmatrix} \text{ and} \\
S_2 &= \begin{pmatrix} 0 & p_2 \\ v_2 & 0 \end{pmatrix}.
\end{aligned}
$$

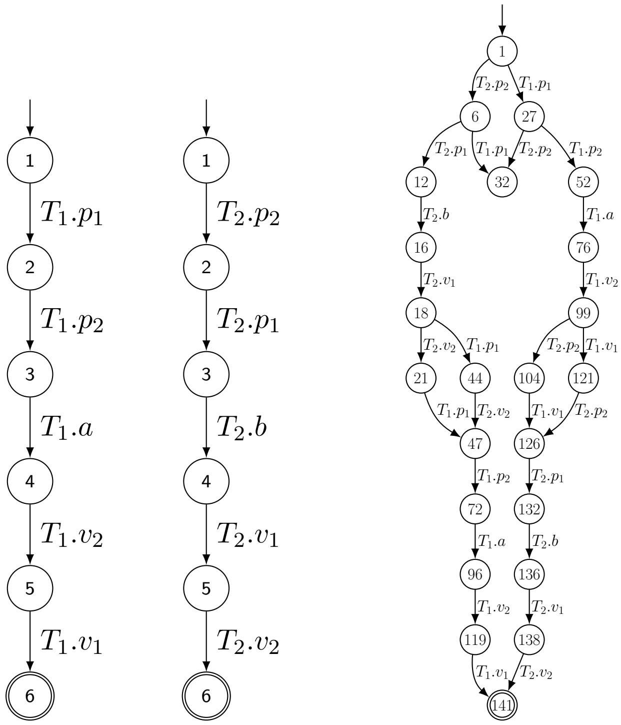
We do not present all intermediate steps. For sake of demonstration, we give the matrix $T \circ S = (T_1 \oplus T_2) \circ (S_1 \oplus S_2)$. Concerning a matrix M, let the tuple (i, j, l) refer to the entry $l \in \mathcal{L}$ at $m_{i,j}$. Similar to adjacency lists, we use a set of such tuples to define a matrix. If an index pair is not mentioned in a set of tuples, then the corresponding entry is assumed to be 0. In order to enable a concise presentation of $T \circ S$ we define the following matrices of order 24:

$$
\begin{aligned}
U = \{ \quad & (1,6,p_2),(3,8,p_2),(5,11,p_1),(6,12,p_1),(9,13,b),(10,14,b), \\
& (11,15,b),(12,16,b),(15,17,v_1),(16,18,v_1),(18,21,v_2),(20,23,v_2)\}, \\
V = \{ \quad & (1,3,p_1),(2,4,p_1),(5,7,p_1),(6,8,p_1),(9,11,p_1),(10,12,p_1), \\
& (13,15,p_1),(14,16,p_1),(17,19,p_1),(18,20,p_1),(21,23,p_1),(22,24,p_1)\}, \\
W = \{ \quad & (1,2,p_2),(3,4,p_2),(5,6,p_2),(7,8,p_2),(9,10,p_2),(11,12,p_2), \\
& (13,14,p_2),(15,16,p_2),(17,18,p_2),(19,20,p_2),(21,22,p_2),(23,24,p_2)\}, \\
X = \{ \quad & (2,1,v_2),(4,3,v_2),(6,5,v_2),(8,7,v_2),(10,9,v_2),(12,11,v_2), \\
& (14,13,v_2),(16,15,v_2),(18,17,v_2),(20,19,v_2),(22,21,v_2),(24,23,v_2)\}, \text{ and} \\
Y = \{ \quad & (3,1,v_1),(4,2,v_1),(7,5,v_1),(8,6,v_1),(11,9,v_1),(12,10,v_1), \\
& (15,13,v_1),(16,14,v_1),(19,17,v_1),(20,18,v_1),(23,21,v_1),(24,22,v_1)\}.
\end{aligned}
$$

Matrix $T \circ S$, a block matrix of order 144, consisting of the submatrices defined above and zero matrices of order 24 (instead of Z_{24} simply denoted by 0) is given by

$$T \circ S = \begin{pmatrix} U & V & 0 & 0 & 0 & 0 \\ 0 & U & W & 0 & 0 & 0 \\ 0 & 0 & U & a \cdot I_{24} & 0 & 0 \\ 0 & 0 & 0 & U & X & 0 \\ 0 & 0 & 0 & 0 & U & Y \\ 0 & 0 & 0 & 0 & 0 & U \end{pmatrix}.$$

The generated RCPG is depicted in Figure 5.1c. The resulting full adjacency matrix has order 144, whereas the resulting RCPG consists only of 23 nodes and 26 edges. Large parts (121 nodes) are unreachable from the entry node. Some of these parts are depicted in Figure 5.2. We already have presented a detailed example with all intermediate matrices and all parts of its unreachable graph in the previous section.



(a) T_1 (b) T_2 (c) Resulting CPG

Figure 5.1: Deadlock Example

In the above example, two semaphores are requested in different orders, thus a deadlock is possible. Node 32 constitutes a zero line because it has no successor. Our approach detects this and colors such nodes red. In the RCPG it is easy to see how the deadlock can be reached. This obviously can happen when T_1 calls p_1 and then T_2 calls p_2 and vice versa. All the other paths through the RCPG are deadlock free.

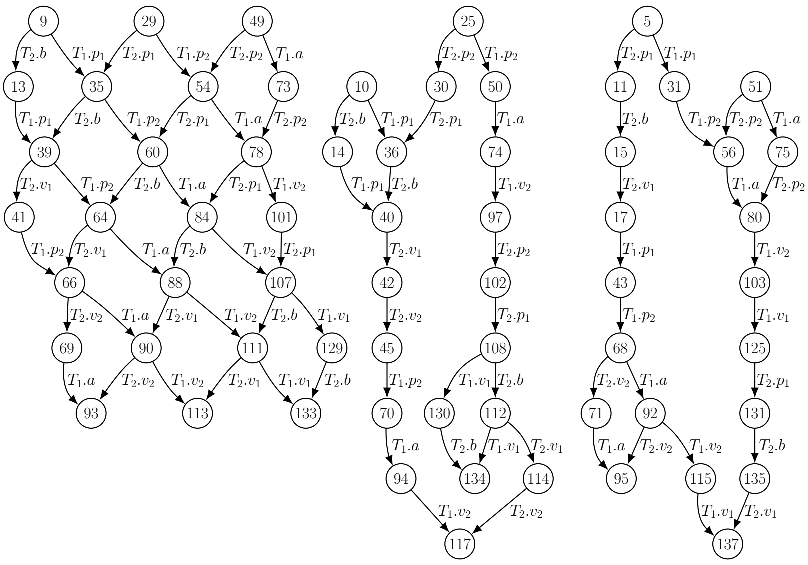


Figure 5.2: Some Unreachable Parts of the Deadlock Example

In Chapter 6, an additional deadlock example is presented.

CHAPTER 6

Examples and Empirical Data

> *"Wie Sokrates weiß der Stückwerk-Ingenieur, wie wenig er weiß. Er weiß, daß wir nur aus unseren Fehlern lernen können. Deshalb wird er nur Schritt für Schritt vorgehen und die erwarteten Resultate stets sorgfältig mit den tatsächlich erreichten vergleichen, immer auf der Hut vor den bei jeder Reform unweigerlich auftretenden unerwünschten Nebenwirkungen. Er wird sich auch davor hüten, Reformen von solcher Komplexität und Tragweite zu unternehmen, daß es ihm unmöglich wird, Ursachen und Wirkungen zu entwirren und zu wissen, was er eigentlich tut."*
>
> – SIR KARL RAIMUND POPPER, Austrian-British philosopher, 1902-1994
> Das Elend des Historizismus, 1965

Now we pause for a moment and have a look to examples and empirical data. In this chapter, we give an example for deadlock analysis in Section 6.1. Then we show how we are able to model a static form of fork-join parallelism in Section 6.2. Section 6.3, a client-server example is presented. In Section 6.4, a data race example is given. It indicates that, based on CPGs, dataflow analysis can be performed. Finally, we present empirical data in Section 6.5 and show that a speedup can be achieved by a parallel version of our lazy CPG generating algorithm.

6.1 Deadlock Example

In this example, we use the RCFGs of the threads $T1$ and $T2$ which are depicted in Figure 6.1a and Figure 6.1b, respectively. We assume that the four semaphores are from the type depicted in Figure 2.3a.

The graph of $T = T1 \oplus T2$ is diamond shaped similar to the graph in Figure 3.4d but it contains $13^2 = 169$ nodes. The CPG represented by the matrix $(T1 \oplus T2) \circ (S1 \oplus S2 \oplus S3 \oplus S4)$ consists of $13^2 \cdot 2^4 = 2704$ nodes. In contrast, the RCPG contains only 82 nodes being reachable from the entry node of the CPG. Thus only 3% of the potential graph nodes are reachable. Our lazy implementation generates exactly the 82 reachable nodes in 0.02 s.[1] The RCPG is shown in Figure 6.1c.

In order to distinguish deadlock from final nodes we calculate the final node of the example first. The final node of the graph can be calculated by Formula 4.2 introduced in Section 4.2. Because $13^2\,2^4 - 2^4 + 1 = 2689$, node 2689 is the final node of R.

In this example, semaphores are requested in different orders, thus deadlock is possible. The nodes 316, 912, 984, 1272, and 1948 constitute zero lines because they have no successors. Hence five different deadlocks can occur. Our approach detects them and highlights the corresponding nodes. In the RCPG it is easy to see how the deadlock can be reached. Four deadlocks are reachable via exactly one path, whereas the other one (node 912) can be reached via multiple paths. In case of the deadlock caused by zero line 316, the tasks $T1$ and $T2$ acquire semaphore 1 and 3 in different orders. The zero line 912 is similarly caused by different orders in acquiring semaphores 3 and 4.

Note that the RCPG R in Figure 6.1c is a directed acyclic graph. Thus it is isomorphic to its condensation. Hence we do not generate R_c, the condensation of R.

By redirecting the edges $12 \to 13$ to $12 \to 1$ and removing the nodes 13 in the RCFGs in Figure 6.1a and Figure 6.1b, the tasks would execute their bodies in endless loops. The modified program does not terminate and thus no final node is present. In this case each zero line constitutes a deadlock. The resulting RCPG R' contains five deadlocks again. A naïve implementation of Kronecker algebra would generate a matrix of order $12^2 \cdot 2^4 = 2304$. Our implementation calculates the 57 reachable nodes of R' in 0.01 s. Although R' contains fewer nodes than R, it is more complex. This is caused by the loops in the tasks. Thus for sake of simplicity, we have chosen a "terminating" program to be presented here.

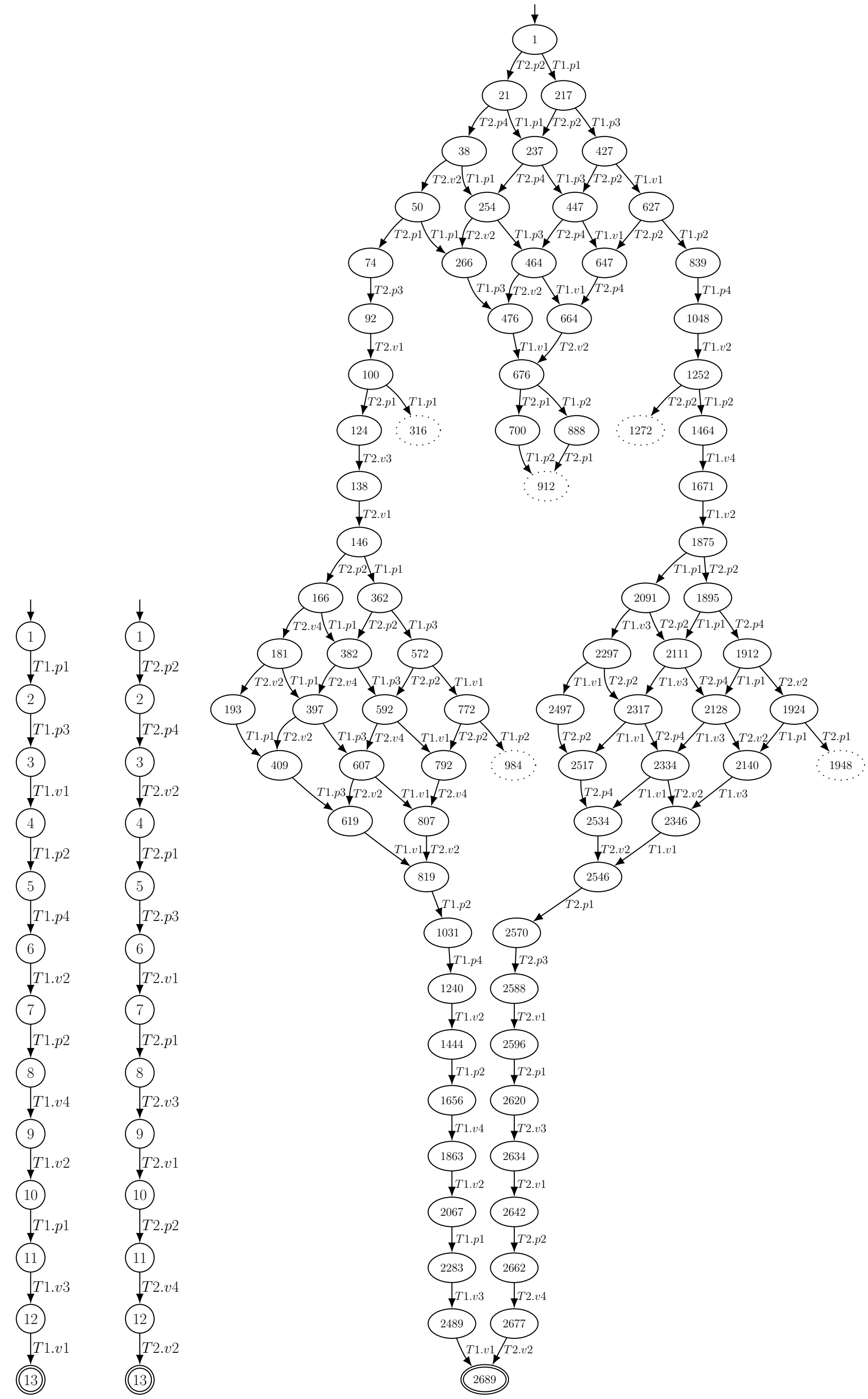
6.2 Modeling Static Fork Join

In this section, we show how we can model a static fork-join mechanism. The forked thread still has to be known statically. However, the proposed approach helps to keep the number of generated interleavings small and exact in the sense that basic blocks before fork and after join are not interleaved with the forked/joined thread.

One semaphore is added for each forked thread. If the thread is joined too, then an additional semaphore is introduced. Each of the added semaphores is an initially locked semaphore (cf. Figure 2.4b). In the following, we assume that thread T_1 forks and joins thread T_2. In order to model the fork-join mechanism, we introduce the initially locked semaphores s_1 and s_2. We replace both, namely fork and join, statements of T_1 by calls to semaphores as follows. Thread T_2 gets additional statements at the beginning and at the end. In the RCFG of thread T_1, we replace the fork statement by a semaphore call $s_1.v$. Thread T_2 gets the statement $s_1.p$ at the beginning immediately before its original

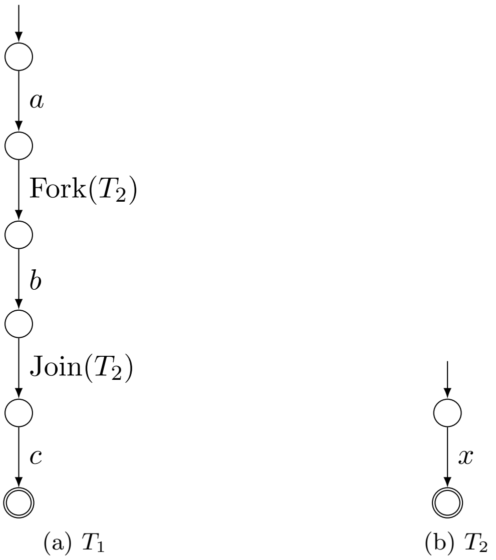
---

[1] The analysis for this example was done on an Intel Core i7-870 (3 GHz, 4 GB RAM) running CentOS 6.0.



(a) $T1$ (b) $T2$ (c) RCPG R

Figure 6.1: A Second Deadlock Example



(a) T_1 (b) T_2

Figure 6.2: RCFGs for Example Consisting of T_1 and T_2

first basic block. So far, this construct allows T_2 to execute its statements only after T_1 has executed its statement $s_1.v$. In order to model the join functionality, T_2 gets the additional statement $s_2.v$ immediately after its original final node. The join statement in T_1 is replaced by the semaphore call $s_2.p$. In a similar fashion as T_1 enables the execution of T_2 by calling $s_1.v$, T_2 allows T_1 to go on with its execution by calling $s_2.v$.

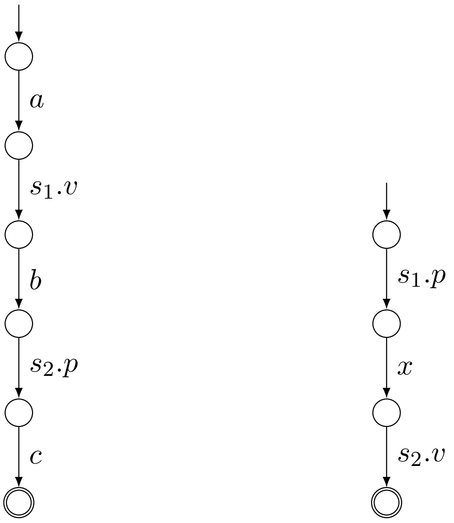
6.2.1 Example

In this example, we use the two threads T_1 and T_2 as depicted in Figure 6.2. The threads call the semaphore operations exactly as described above. In addition, the thread T_1 (Figure 6.2a) executes a before forking thread T_2, c after joining T_2, and b between the fork and join statements. The thread T_2 (Figure 6.2b) initially executes the edge x. The two RCFGs are adapted as described above. The resulting RCFGs are depicted in Figure 6.3. The expectation is that the parts of T_1 before the fork statement and the basic blocks below the join statement are not interleaved with the only edge of T_2, namely x. It can easily be seen in the resulting RCPG, which is depicted in Figure 6.4, that this expectation is met.

6.3 Client-Server Example

We have done analysis on client-server scenarios using our lazy implementation. For the example presented here we have used multiple clients and a semaphore of the form shown in Figure 6.5a and 6.5b, respectively.

In Table 6.5c statistics for 1, 2, 4, 8, 16, and 32 clients are given. Figure 6.5d shows the resulting graph for 8 clients. The few nodes in the resulting matrix and the node IDs indicate that most nodes in the resulting matrix are superfluous. The case of 32



(a) Adapted T_1 (b) Adapted T_2

Figure 6.3: Adapted RCFGs for Example Consisting of T_1 and T_2

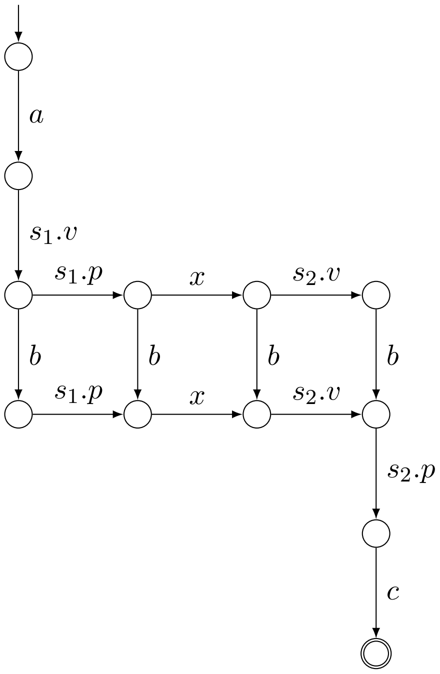
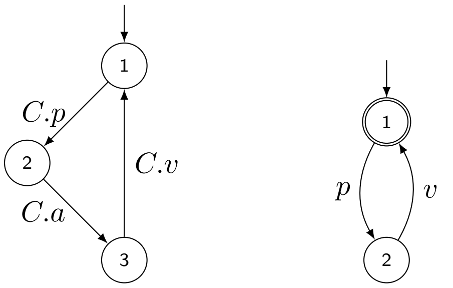


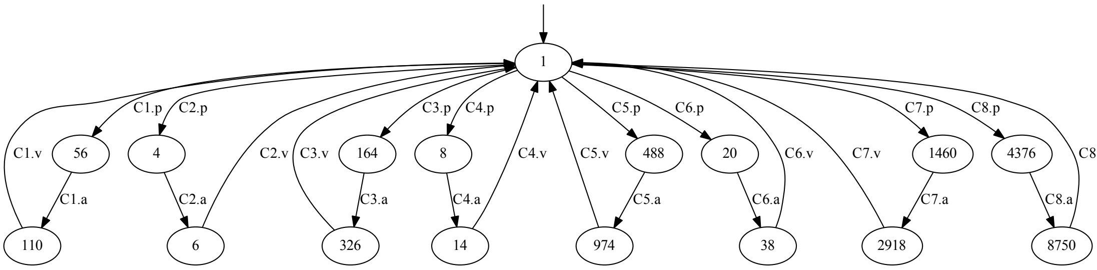
Figure 6.4: RCPG of the T_1-T_2-System



(a) Client (b) Semaphore

Clients	Nodes	Edges	Exec. Time	Potential Nodes
1	3	3	0.0013s	6
2	5	6	0.0013s	18
4	9	12	0.0045s	162
8	17	24	0.012s	13,122
16	33	48	0.068s	86,093,422
32	65	96	0.43s	3.706×10^{15}

(c) Statistics



(d) Result for 8 Clients

Figure 6.5: Client-Server Example

clients and one semaphore forms a matrix with an order of approx. 3.706×10^{15}. Our implementation generated only 65 nodes in 0.43s. In fact we observed a linear growth in the number of clients for the number of nodes and edges and for the execution time. We did our analysis on an Intel Xeon 2.8 GHz with 8GB DDR2 RAM. Note that an implementation of the matrix calculus for shared memory concurrent systems has to provide node IDs of a sufficient size. The order of $T \circ S$ can be quite big, although the resulting RCPG is small.

```
$T_1$ ()
1  $s.p$            {edge T1.p}
2  $r \leftarrow sv$       {edge a}
3  $r \leftarrow r+1$      {edge b}
4  $sv \leftarrow r$       {edge b}
5  $s.v$            {edge T1.v}
$T_2$ ()
1  $t \leftarrow sv$       {edge c}
2  $s.p$            {edge T2.p}
3  $t \leftarrow t+1$      {edge d}
4  $sv \leftarrow t$       {edge d}
5  $s.v$            {edge T2.v}
```

Figure 6.6: Data Race Example

6.4 A Data Race Example

With this example, we indicate that data flow analysis (cf. [RP86, RP88, SGL98, KU76]) can be done on CPGs.

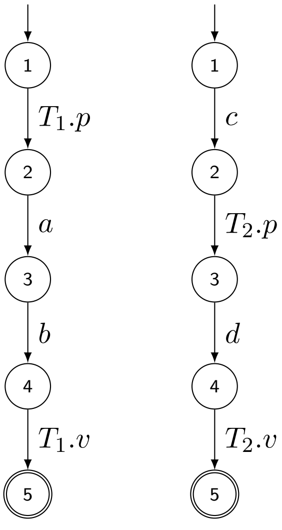
We give an example, where a programmer is supposed to have used synchronization primitives in a wrong way. The program consisting of two threads, namely T_1 and T_2, and a semaphore s are given in Figure 6.6. Comments in the pseudocode denote each code line's RCFG edge (thus after edge splitting). Before edge splitting each thread consists of exactly one basic block. We assume that $sv = 0$ at program start. It is supposed that the program delivers $sv = 2$ when it terminates. Both threads in the program access the shared variable sv. The variables r and t are local to the corresponding threads. The programmer inadvertently has placed line 1 in front of line 2 in T_2.

After edge splitting, we get the RCFGs depicted in Figure 6.7. We assume a semaphore as shown in Figure 2.3a.

The adjacency matrices of the threads T_1 and T_2 are

$$T_1 = \begin{pmatrix} 0 & T_1.p & 0 & 0 & 0 \\ 0 & 0 & a & 0 & 0 \\ 0 & 0 & 0 & b & 0 \\ 0 & 0 & 0 & 0 & T_1.v \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \text{ and } T_2 = \begin{pmatrix} 0 & c & 0 & 0 & 0 \\ 0 & 0 & T_2.p & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & T_2.v \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Although the following matrices are not computed by our lazy implementation, we give them here to allow the reader to see a complete example. To enable a concise presentation, we define the following submatrices of order five:



(a) T_1 (b) T_2

Figure 6.7: RCFGs for Data Race Example

$$H = \begin{pmatrix} 0 & c & 0 & 0 & 0 \\ 0 & 0 & T_2.p & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & T_2.v \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, I = \begin{pmatrix} T_1.p & 0 & 0 & 0 & 0 \\ 0 & T_1.p & 0 & 0 & 0 \\ 0 & 0 & T_1.p & 0 & 0 \\ 0 & 0 & 0 & T_1.p & 0 \\ 0 & 0 & 0 & 0 & T_1.p \end{pmatrix},$$

$$J = a \cdot I_5, K = b \cdot I_5, \text{ and } L = \begin{pmatrix} T_1.v & 0 & 0 & 0 & 0 \\ 0 & T_1.v & 0 & 0 & 0 \\ 0 & 0 & T_1.v & 0 & 0 \\ 0 & 0 & 0 & T_1.v & 0 \\ 0 & 0 & 0 & 0 & T_1.v \end{pmatrix}.$$

Now, we get $T = T_1 \oplus T_2$, a matrix of order 25, consisting of the submatrices defined above and zero matrices of order five (instead of Z_5 simply denoted by 0).

$$T = \begin{pmatrix} H & I & 0 & 0 & 0 \\ 0 & H & J & 0 & 0 \\ 0 & 0 & H & K & 0 \\ 0 & 0 & 0 & H & L \\ 0 & 0 & 0 & 0 & H \end{pmatrix}.$$

To shorten the presentation of $P = T \circ S$ we define the following submatrices of order ten: $W = a \cdot I_{10},\ X = b \cdot I_{10}$,

$$U = \begin{pmatrix} 0 & 0 & c & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & T_2.p & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & T_2.v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$V = \begin{pmatrix} 0 & T_1.p & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & T_1.p & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & T_1.p & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & T_1.p & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & T_1.p \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and}$$

$$Y = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ T_1.v & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & T_1.v & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & T_1.v & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & T_1.v & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & T_1.v & 0 \end{pmatrix}.$$

With the help of zero matrices of order ten, we can state the program's matrix

$$P = T \circ S = T \circ \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix} = \begin{pmatrix} U & V & 0 & 0 & 0 \\ 0 & U & W & 0 & 0 \\ 0 & 0 & U & X & 0 \\ 0 & 0 & 0 & U & Y \\ 0 & 0 & 0 & 0 & U \end{pmatrix}.$$

The final node can be calculated by using Formula 4.2 from Section 4.2. Thus the final node is $5^2 * 2 - 2 + 1 = 49$.

Matrix P has order 50. The corresponding RCPG is shown in Figure 6.8. Our lazy implementation computes only the 19 reachable nodes. Due to synchronization the other

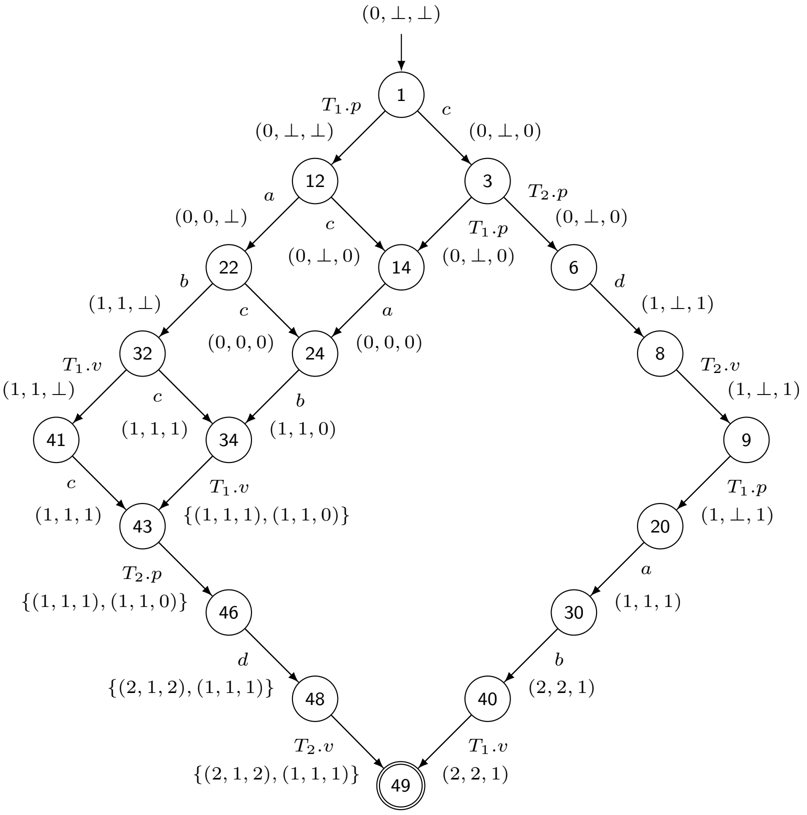


Figure 6.8: CPG of Data Race Example

parts are not reachable. In addition to the usual labels we have add a set of tuples to each edge in the CPG of Figure 6.8. Tuple (x, y, z) denotes values of variables, such that $sv = x$, $r = y$ and $t = z$. We use $\perp$ to refer to an undefined value. A tuple shows the values after the basic block on the corresponding edge has been evaluated. The entry node of the CPG is Node 1. At program start we have the variable assignment $(0, \perp, \perp)$. At Node 49 we result in the set of tuples $\{(1, 1, 1), (2, 1, 2), (2, 2, 1)\}$. Due to the interleavings different tuples may occur at join nodes. This is reflected by a set of tuples. As stated above the program is supposed to deliver $sv = 2$. Thus the tuple $(1, 1, 1)$ shows that the program is erroneous. The error is caused by a data race between the edges c of thread T_2 and the edges a and b of thread T_1.

6.5 Empirical Data

In Section 6.3, we already gave some empirical data concerning client-server examples. In this section, we give empirical data for ten additional examples.

Let $o(P)$ and $o(R)$ refer to the order of the adjacency matrix P (which is not computed by our lazy implementation) and the order of the adjacency matrix R of the RCPG, respectively. In addition k and r refer to the number of threads and the number of semaphores, respectively.

k	r	$o(P)$	$\sqrt{(o(P))}$	$o(R)$	Runtime [s]
2	4	256	16.00	12	0.03
3	5	4800	69.28	30	0.097542
4	6	124416	352.73	98	0.48655
3	6	75264	274.34	221	1.057529
4	7	614400	783.84	338	2.537082
4	8	1536000	1239.35	277	2.566587
4	8	737280	858.65	380	3.724364
4	13	298721280	17283.56	2583	96.024073
4	11	55050240	7419.58	3908	146.81
5	6	14929920	3863.93	7666	309.371395

Table 6.1: Empirical Data

In the following, we use the data depicted in Table 6.1.[2] The numbers in the fourth column are rounded to two decimal places. As a first observation we note that except for one example all values of $o(R)$ are smaller than the corresponding values of $\sqrt{(o(P))}$. In addition, the runtime of our implementation shows a strong correlation to the order $o(R)$ of the adjacency matrix C of the generated CPG with a Pearson product-moment correlation coefficient of 0.9990277130. In contrast the values of the theoretical order $o(P)$ of the resulting adjacency matrix P correlate to the runtime only with a correlation coefficient of 0.2370050995.[3]

These observations show that the runtime complexity does not depend on the order $o(P)$ which grows exponentially in the number of threads. We conclude this chapter by stating that the collected data give strong indication that the runtime complexity of our approach is linear in the number of nodes present in the RCPG, i.e., the size of the solution.

6.5.1 Improvements via a Parallel Version

By choosing a worklist algorithm our lazy implementation allows for simple parallelizing. For example, retrieving the entries of left and right operands can be done concurrently. Exploiting this, we achieved further performance improvements for our implementation if run on multi-core architectures. We present some results for an example modeling five dining philosophers in such a way that it does not deadlock. Thus the full reachable state space has to be generated. On a machine able simultaneously executing 16 threads we achieved the reductions depicted in Figure 6.9. The number of threads generating the RCPG is situated on the x-axis. Note that the diagram is actually defined for natural numbers only. The achieved reduction in time can be seen on the y-axis. For example using 16 threads it took 1/14.6 of the time compared to using a single thread.

---

[2] We did our analysis on an Intel Pentium D 3.0 GHz machine with 1GB DDR RAM running CentOS 5.6.

[3] Both correlation coefficients are rounded to ten decimal places.

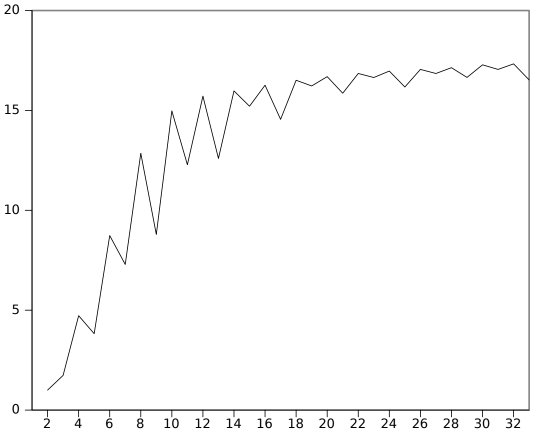


Figure 6.9: Improvements via a Parallel Version

Another deadlock free program consisting of five threads calling 15 binary semaphores. For this example, each thread's RCFG contains 16 nodes. The theoretical adjacency matrix representing the program has an order of $16^5 \cdot 2^{15} = 8,153,726,976$. Our implementation generates the reachable part (RCPG) containing 65,944 nodes in 2.49 seconds.[4] Note that only 0.00081% of the potential graph are reachable.

---

[4] The analysis was done on an Intel Core i7-6700k, 4.0 GHz, 16 GB RAM, Ubuntu 14.04.2, 64 bit

CHAPTER 7

Static Analysis of Barriers

"Auch aus Steinen,
die einem in den Weg gelegt werden,
kann man Schönes bauen."

– JOHANN WOLFGANG VON GOETHE, German writer and statesman, 1749-1832

In this chapter, we present a new synchronization primitive modeling barriers. This barrier synchronization construct was developed during the work for this doctoral thesis. It appeared for the first time in [MB16b]. By adopting this synchronization primitive, we are able to statically analyze Ada multi-tasking programs that employ barriers for synchronization issues. It turns out that we can use our lazy Kronecker algebra implementation out-of-the-box (as presented in Section 4.7) for concurrent program graphs (CPGs) using such barrier synchronization primitives. In addition, we show how our Kronecker algebra-based approach can be used to prove both, namely semaphore-based barrier implementations and their usage scenarios, correct.

Kronecker algebra is an useful vehicle to model multi-threaded shared memory programs and stochastic automata [BK02, MB11, MB12b, Pla85]. Kronecker sum, Kronecker product, and a slightly adapted version of Kronecker product are applied to the adjacency matrices of the underlying concurrent programs' control flow graphs (CFGs) and the synchronization primitives' graph representations. Until now we have applied Kronecker algebra analysis to multi-threaded concurrent programs being synchronized via *semaphores*. In [BB14] also a Kronecker algebra-based approach is applied to higher level synchronization primitives like Ada's *protected objects* (POs). Applying Kronecker algebra to Ada's *barriers* is novel in this area.

The contributions of this chapter are as follows.

1. We show how to model Ada's barriers such that Kronecker algebra can be employed

for static analysis. This is done by introducing a novel synchronization primitive modeling the semantics of barriers.

2. We compare our barrier synchronization primitive with a barrier implementation based on semaphores. As a byproduct, we show how our CPG-based approach can be used as a basis for proving such implementations correct. It turns out that our barrier construct is better suited for program analysis because it fully can be analyzed by static analysis[1], while the implementations using semaphores, in order to omit dead paths, require advanced techniques (e.g. symbolic analysis [BSB12]).

The outline of this chapter is as follows: In Section 7.1, we introduce novel synchronization primitives for modeling the semantics of barriers. We also compare our barrier construct with barriers modeled by semaphores and show how semaphore-based barrier implementations can be analyzed. Related work is discussed in Section 10.3. Finally, we conclude this chapter in Section 11.1.

7.1 Barriers

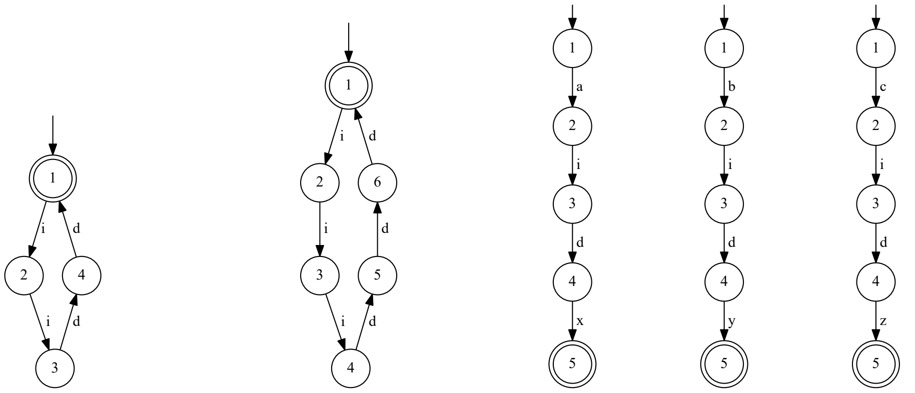
Kronecker algebra until now has only been applied to concurrent programs that use semaphores and Ada's protected objects (POs) for synchronization. In this section, we propose a new synchronization primitive modeling the behavior of barriers. We will give examples and we will compare our solution with the barrier implementation found in [Dow05]. Several other implementations of barriers can be found in [HS98].

A barrier is a synchronization construct available in most modern programming languages, e.g., Ada and Java. It is used when threads have to wait for each other. Thus a barrier is used to synchronize a set of n threads. The first thread(s) reaching the barrier will be blocked. When the nth thread reaches the barrier all the threads are released and continue their work. A barrier is called reusable, when it can be re-used after the threads are released. In general, dynamic and static barriers are distinguished. Static barriers have a statically fixed number of participating threads, while the number of threads can vary at runtime for dynamic barriers.

In Ada, barriers are available in form of synchronous barriers [Bru16, D.10.1] available in the package `Ada.Synchronous_Barriers`. Tasks calling its procedure `Wait_For_Release` will be blocked until the `Release_Threshold` is reached. Java supports barriers in form of the class `CyclicBarrier` [Gon12]. The method `await` is called when the barrier is reached. Also a dynamic barrier, namely `Phaser` [Gon12], is supported in Java. Both, Ada's synchronous barrier and Java's `CyclicBarrier` are reusable. With the class `CountDownLatch`, Java has also some sort of a non-reusable barrier, where one or more threads can wait until a set of operations being performed in other threads completes. The approach presented in this doctoral thesis works for both, Ada's and Java's static barriers. However, in the remainder of this chapter, we elaborate on Ada's barrier in more detail. Hence, the term task will be used instead of thread.

---

[1] Programs using our barrier synchronization primitive from within loops or conditional statements will still require advanced techniques.



(a) Barrier (n=2) (b) Barrier (n=3) (c) T1 (d) T2 (e) T3

Figure 7.1: RCFGs of Tasks T1, T2, and T3 Using a Barrier

We model a call of the barrier operation `Wait_For_Release` with label i. This indicates that the counter within the barrier implementation is incremented by one during such a call. In order to set the current counter to zero all the tasks call d (decrements counter). Both labels i and $d \in \mathcal{L}_{\mathrm{S}}$. We require that the barrier labels have to be unique, i.e., the labels of two different barriers have to be different (the same applies for all synchronization primitives, e.g., semaphores) thus the jth barrier uses the labels i_j and d_j. Because the examples in this chapter use at most one barrier, we do not have to pay much attention to this fact.

Similar to the semaphore synchronization primitives, we now model our barrier synchronization primitive. A barrier construct includes n subsequent edges labeled by i followed by n subsequent edges labeled by d. Examples for such barrier constructs are depicted in Figure 7.1a and Figure 7.1b. These barriers synchronize two and three threads, respectively. In a similar way it is possible to model barriers synchronizing any number of threads. After releasing the tasks, each of them, before allowed to continue, calls the barrier's d-operation. This sets the counter back to zero and ensures that the barrier is reusable.

During CFG construction each call of the procedure `Wait_For_Release` is being replaced such that i and d of the corresponding barrier synchronization primitive are called. This replacement implies that the actually used barrier implementation has to be provably correct. Otherwise, a proof could state correctness while abstracting away from a faulty implementation. This proof can be done independently from proving a barrier usage scenario correct. From a certain point of view, our barrier construct is based on the semantics of barriers. A different approach is to use any implementation of a barrier based on semaphores to verify a barrier usage scenario together with the

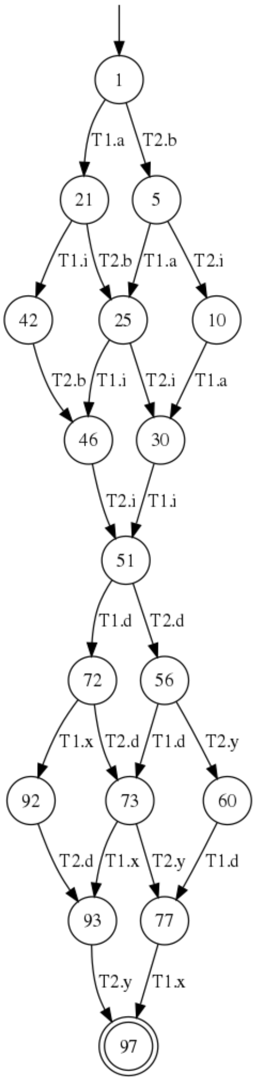


Figure 7.2: CPG for Program Consisting of T1 and T2

barrier's implementation. As we will see in the following examples, the verification of programs using our barrier synchronization primitive will be easier compared to barriers implemented using semaphores.

7.1.1 Examples

As an example consider the CFGs of the tasks T1, T2, and T3 shown in Figure 7.1. The CPG of a program consisting of the two tasks T1 and T2 is depicted in Figure 7.2, whereas the CPG of the program consisting of the three tasks T1, T2, and T3 is shown in Figure 7.3. The first program uses the barrier construct presented in Figure 7.1a, whereas the second program uses the barrier depicted in Figure 7.1b. Note that these graphs are generated with our standard CPG generating software. The input programs just use our new barrier synchronization primitive. It can easily be seen that the barrier synchronizes the tasks correctly.

Figure 7.5 presents an example program consisting of two tasks TL1 (Figure 7.4a) and TL2 (Figure 7.4b). Each task contains a loop and a `Wait_For_Release` inside the loop. If the number of loop iterations is the same in both tasks, the final node 61 is reached; otherwise, the program stalls at nodes 30 or 54. The number of loop iterations cannot be calculated by the Kronecker approach. For this purpose some sort of symbolic analysis is needed. In the simplest case, only lower and upper bounds of for-loops have to be compared.

7.1.2 Comparison to a Barrier Implementation using Semaphores

In the following, we compare our barrier construct with the barrier implementation on page 41 in [Dow05]. The following pseudocode is out of [Dow05].

Listing 7.1: Reusable Barrier Solution using Semaphores

```
# rendezvous

mutex.wait()                # ps
    count += 1              # i
    if count == n:
        turnstile2.wait()   # pb2, lock the second
        turnstile.signal()  # vb1, unlock the first
    else # empty            # T1.a; T2.e
mutex.signal()              # vs

turnstile.wait()            # pb1, first turnstile
turnstile.signal()          # vb1

# critical point            # T1.b; T2.f

mutex.wait()                # ps
    count -= 1              # d
    if count == 0:
        turnstile.wait()    # pb1, lock the first
        turnstile2.signal() # vb2, unlock the second
    else # empty            # T1.c; T2.g
mutex.signal()              # vs

turnstile2.wait()           # pb2, second turnstile
turnstile2.signal()         # vb2
```

Three semaphores, namely `mutex`, `turnstile`, and `turnstile2`, are used in order to implement the barrier functionality. Two of them, namely `mutex` and `turnstile2`, are initially unlocked semaphores as shown in Figure 2.4a. The semaphore `turnstile` is initially locked as depicted in Figure 2.4b. We assume that the two threads T1 and T2 execute the code. Some lines are modeled by the same labels for both threads (e.g. both threads use `ps` in order to get access to the variable `count`). Other lines are modeled by different labels (e.g. T1 and T2 execute `b` and `f`, respectively, as their critical point).

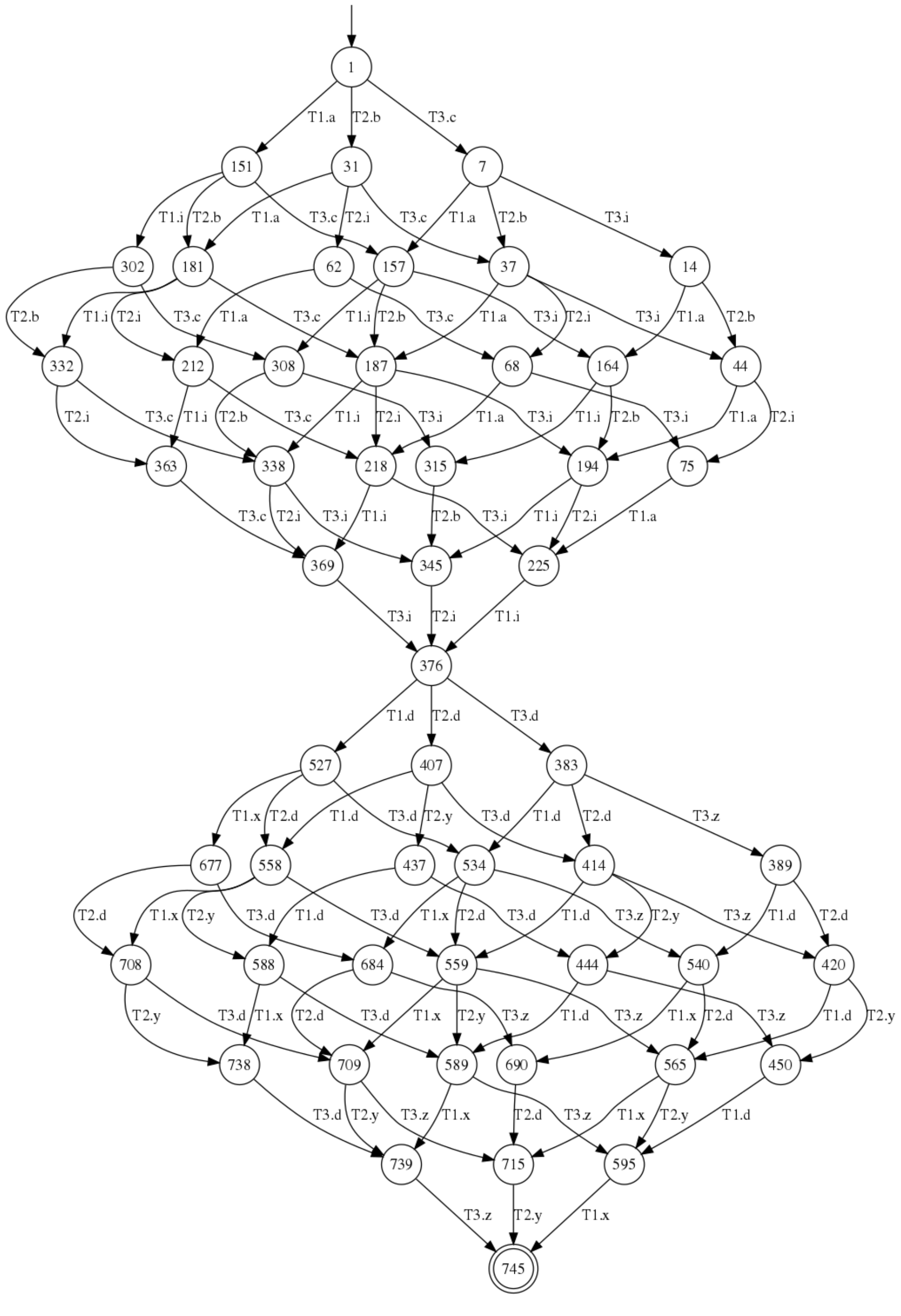
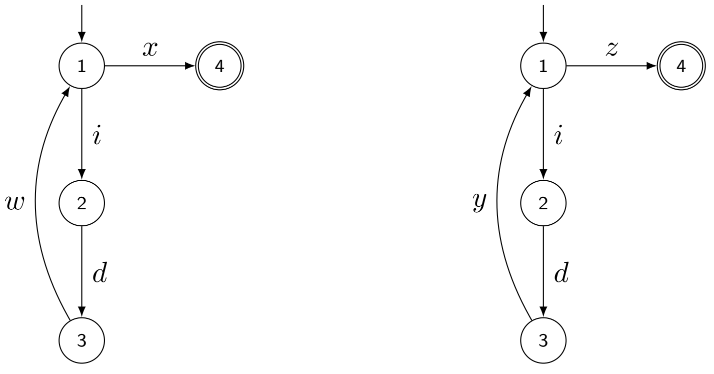


Figure 7.3: CPG for Program Consisting of T1, T2, and T3



(a) CFG for TL1 (b) CFG for TL2

Figure 7.4: CFGs for Example Consisting of TL1 and TL2

The CPG of the reusable barrier solution is depicted in Figure 7.6. The graph contains the potential deadlock nodes 681, 761, 1774, 1790, 1961 and 2030. The red dotted edges are dead paths which can be ruled out by a value-sensitive (e.g. symbolic) analysis (cf. [BSB12]). Due to these red edges some nodes are unreachable which are indicated in red, too. As an effect of this, it is easy to see, that all the potential deadlock nodes are unreachable. We can conclude that the implementation using three semaphores is correct but it is obviously more complex as our solution (cf. Figure 7.2) and thus it is harder to prove its correctness. In addition, to exclude the dead paths in Figure 7.6, advanced approaches like symbolic analysis are needed.

Similar to the reusable barrier solution above, we discuss a non-reusable barrier solution. The following pseudocode can be found on page 29 in [Dow05].

Listing 7.2: Non-reusable Barrier Solution using Semaphores

```
# rendezvous

mutex.wait()                    # ps
    count = count + 1           # c
mutex.signal()                  # vs

if count == n: barrier.signal() # T1.v, T1.a; T2.v, T2.x
else # empty                    # T1.b; T2.y

barrier.wait()                  # p
barrier.signal()                # v

# critical point
```

Two semaphores, namely `mutex` and `barrier`, are used in order to implement the barrier's functionality. The first, namely `mutex`, is an initially unlocked semaphore as shown in Figure 2.4a. The second semaphore, namely `barrier`, is an initially locked semaphore as depicted in Figure 2.4b. The CPG of the non-reusable barrier solutionis depicted in Figure 7.7. Again there are dead paths (the corresponding edges are presented dotted

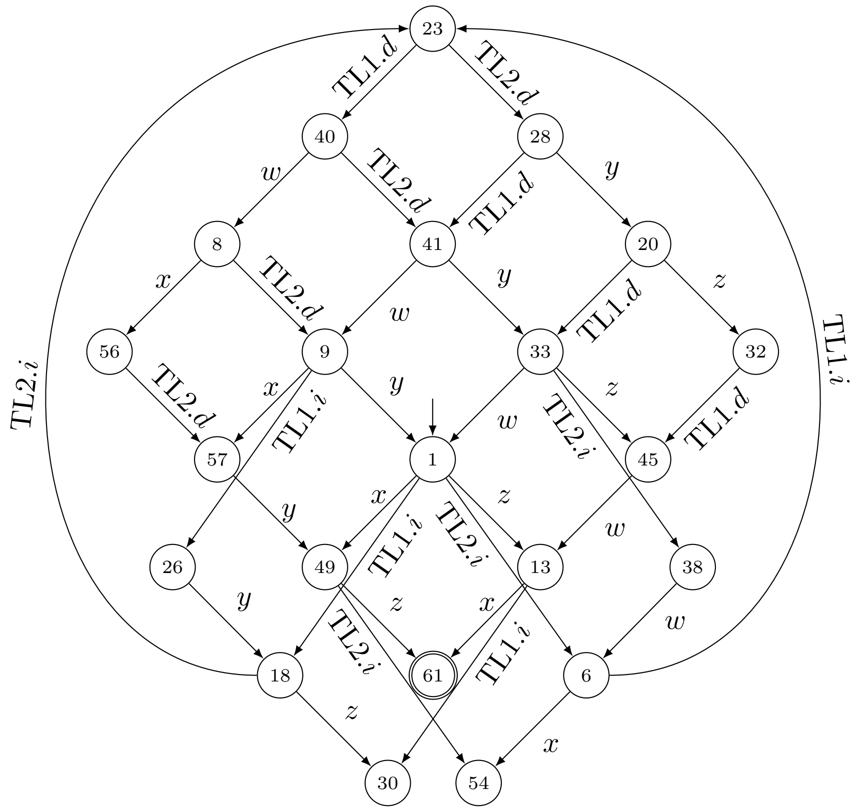


Figure 7.5: CPG for Program Consisting of TL1 and TL2

in red) in the resulting CPG. This graph includes also a deadlock node (i.e. node 181). Again, the paths to this node can be revealed as dead paths, e.g., by symbolic analysis. Thus the node 181 is shown in red which states that this node is unreachable (as several other nodes). In contrast to that, our approach does not generate any deadlock nodes nor dead paths.

In [Dow05] it is stated that analyzing barrier implementations can be quite tricky. We can conclude this section by stating that we introduced a technique for formally and automatically analyze barrier implementations and programs using barriers. Thus, the informal proof of barriers, which is mentioned in [Dow05], can be replaced by an automatic and a more reliable one.

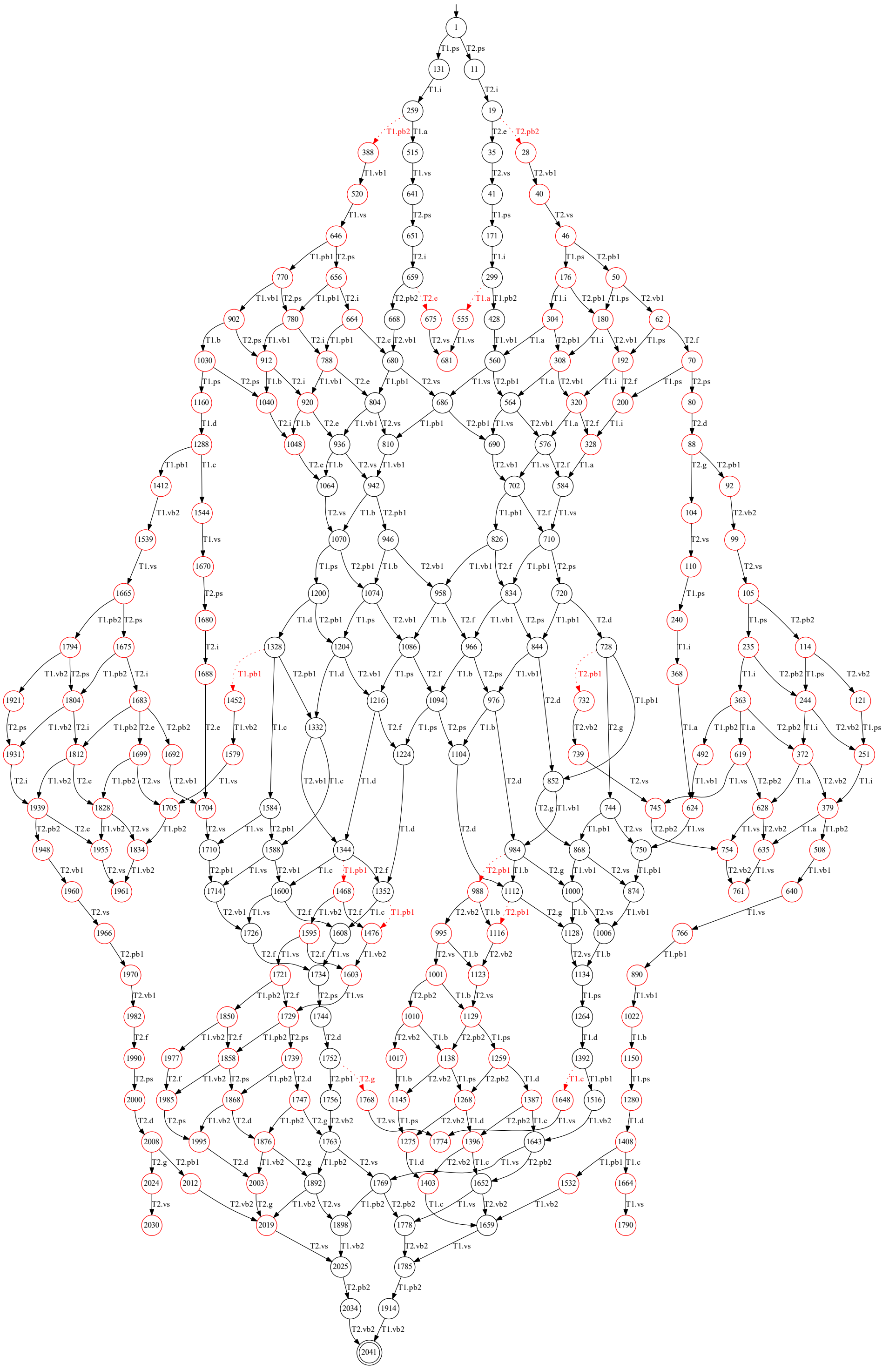


Figure 7.6: CPG of Reusable Barrier Solution using Semaphores

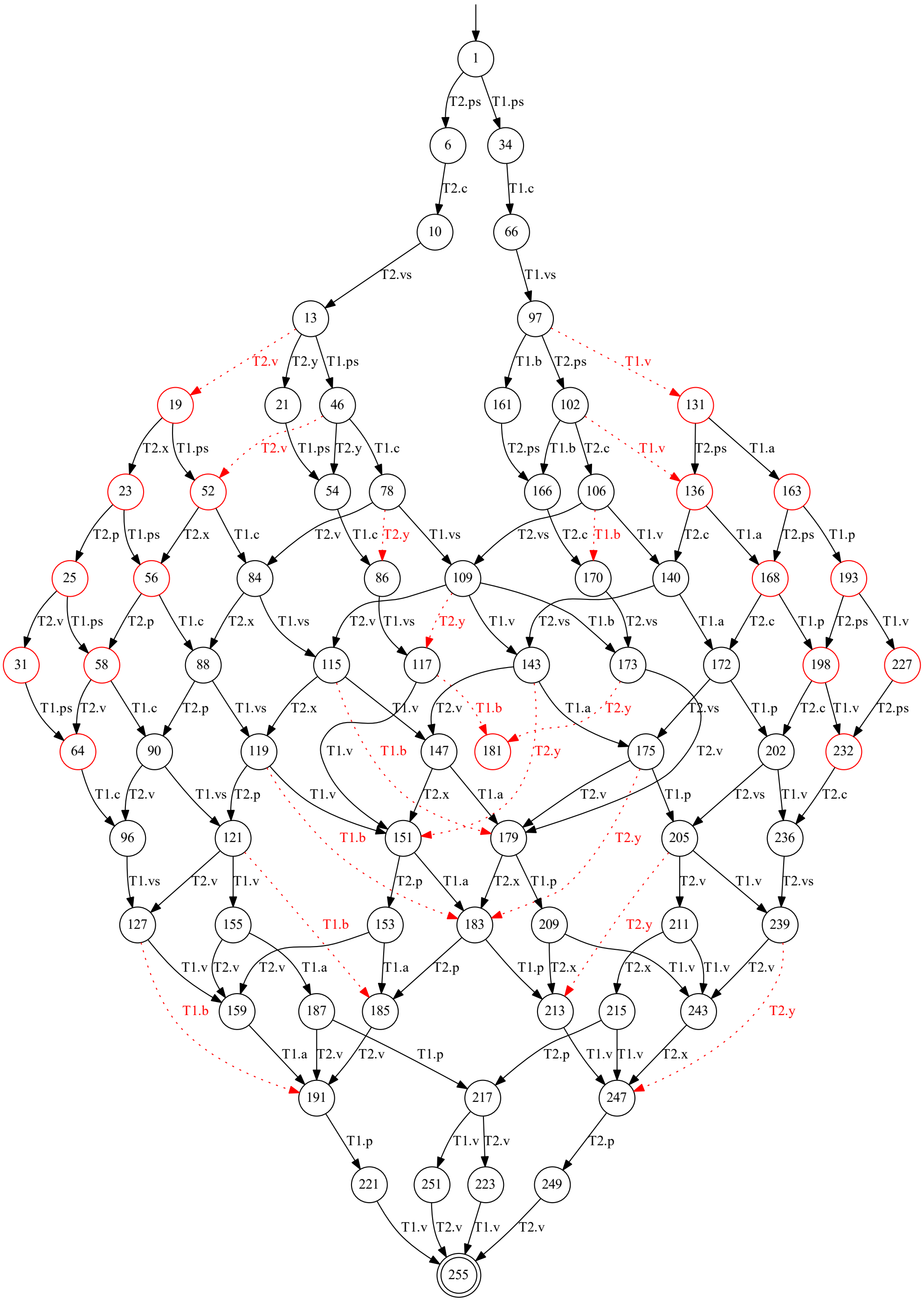


Figure 7.7: CPG of Non-Reusable Barrier Solution using Semaphores

CHAPTER 8

Worst-Case Execution Time Analysis

"The only reason for time is so that everything doesn't happen at once."

– ALBERT EINSTEIN, German-born theoretical physicist,
Nobel Prize in Physics 1921, 1879-1955

The *worst-case execution time* (WCET) of a program is the maximum time it can take to execute that program. It is somehow the opposite of the best-case execution time (BCET). The average case execution time is in between of these two values. For reliable and/or safety relevant real-time systems, WCET is very important property in order to understand the timing behaviour of programs. In the ideal case, WCET analysis determines the exact WCET. WCET analysis at least has to estimate safe upper bounds, i.e., the resulting value does not underestimate the real WCET.

It is widely agreed that the problem of determining upper bounds on worst-case execution times for sequential programs has been more or less solved [WEE$^+$08]. With the advent of multi-core processors scientific and industrial interest focuses on analysis and verification of multi-threaded applications. WCET analysis of multi-threaded software is still a challenge. Beside other reasons, this comes from the fact that synchronization has to be taken into account.

In this chapter, which mainly summarizes the contributions of [MB16a, MB12b], we focus on this issue and on automatically incorporating stalling times (e.g. caused by lock contention) in a WCET analysis of shared memory concurrent programs running on a multi-core architecture. The idea that thread interleavings of concurrent programs can be studied with a matrix calculus (as established in the Chapters 3 and 4) is novel in this research area. Our sparse matrix representations of the program are manipulated using a lazy implementation of our *extended Kronecker algebra*. We describe synchronization by *extended Kronecker product* and thread interleavings by Kronecker sums.

In [NYP15a] a good overview of state-of-the-art WCET techniques with their advantages and disadvantages is given. Static, measurement-based, hybrid, and probabilistic WCET analysis techniques are distinguished. The approach presented in this chapter can be categorized as a static WCET analysis technique and together with measurements can be extended to a hybrid technique. We focus on a formal definition and description of the data flow equations for timing analysis. The low-level analysis mentioned in [NYP15a] is omitted.

Previous work done in the field of timing analysis for multi-core (e.g. [ORS13]) assumes that the threads are more or less executed in parallel and the threads do not heavily synchronize with each other, except when forking and joining. Our approach supports critical sections and the corresponding stalling times (e.g. caused by lock contention) in the heart of its matrix operations. Forking and joining of threads can also easily be modeled. Thus, our model is suitable for systems using both, namely concurrent and parallel (e.g. fork and join) execution models. However, the focus in this chapter is on a concurrent execution model.

We allow communication between threads in multiple ways, e.g., via shared memory accesses protected by critical sections. However, we use a rather abstract view on synchronization primitives. Modeling thread interactions on the hardware-level is out of the scope of this dissertation. A lot of research projects have been launched to make time predictable multi-core hardware architectures available. Our approach may benefit from this research.

Reachable Concurrent Program Graphs (RCPGs) as introduced in Chapter 4 represent concurrent and parallel programs similar as control flow graphs (CFGs) do for sequential programs. In this chapter, we use RCPGs as a basis for the calculation of the WCET of the underlying concurrent system. With this graph model, we are able to calculate the WCET of multi-threaded programs including stalling times which are due to synchronization. In contrast to [MB12b], we (1) adopt the generating functions-based approach presented in [Bli02, Section 4] for timing analysis and (2) are able to handle loops. For timing analysis, we set up a data flow equation for each RCPG node. It turns out that at certain synchronizing nodes, stalling times (e.g. caused by lock contention) can be formulated within data flow equations as simple maximum operations. Choosing this approach, the calculated WCET includes stalling time. This is in contrast to most of the work done in this field (e.g. [ORS13]), which usually adopts a partial approach, where stalling times are calculated in a second step. However, the data flow equations are solved by well-known elimination based data flow methods or an off-the-shelf equation solver. The WCET of multi-threaded programs can finally be calculated with a non-linear function solver.

In this chapter, we show how to calculate the WCET of an underlying concurrent system. Since multi-threaded programs may contain blocking because of synchronization between threads, the terms execution time and WCET do not apply directly to concurrent systems.

In this doctoral thesis, we stick to the term WCET for concurrent systems, too. The reader, however, has to be aware of the fact that, in general, the WCET includes stalling time.

We successively apply the following steps:

1. Generate CFGs out of binary or source code (cf. Subsection 2.3).
2. Generate RCPG out of the CFGs (cf. Chapter 4).
3. Apply hardware-level analysis based on the RCPG. This is *Phase 2* mentioned in [NYP15a]. Such an analysis may take into account, e.g., shared resources like memory, CPU caches, and buses, and other hardware components like instruction caches and pipelining. Annotate this information at the corresponding RCPG edges. As mentioned above this step is out of scope of this dissertation. However, in order to get tight bounds this step is necessary (cf. [PBP13]). Some of these analyses (e.g. cache analysis) may be performed together with the next step.
4. Establish and solve data flow equations based on the RCPG (cf. Section 8.1). Stalling times are incorporated via these equations and not in a measured manner. The latter would be likely to underestimate stalling times [NYP15a].

Similar to [Bli02] and [PS97], which provide exact WCET for sequential programs, our approach calculates an exact WCET for concurrent programs running on a multi-core CPU (not only an upper bound) provided that the number of how often each loop is executed, the execution frequencies and execution times of the basic blocks (also of the semaphore operations p and v)[1] on RCFG level are known, and hardware impact is given. We assume timing predictability on the hardware level, e.g., as discussed in [GKUR12].

In the remainder of this chapter, we refer to both, a processor and a core, as a processor. Our *computational model* can be described as follows. We model concurrent programs by threads which use semaphores for synchronization. We assume that on each processor exactly one thread is running and each thread immediately executes its next statement if the thread is not stalled.

Stalling may occur only in succession of synchronization primitive calls. Because the data flow equations defined in this chapter do not expect v-edges blocking threads, we restrict how semaphores can be used to the following two variants:

1. One possibility is to exclusively allow semaphores with non-blocking v-calls (as depicted in Figure 2.4).
2. When also semaphores with potentially blocking v-calls (as shown in Figure 2.3) shall be used, then we require a well structured program such that no thread is

---

[1] These execution times do not include stalling time which we calculate automatically.

> allowed to call the v-operation of such a semaphore, when it is unlocked. This can e.g. be ensured by a similar concept as an owner is for mutexes, where only the owner of a mutex is allowed to unlock it.

The outline of this chapter is organized as follows. Section 8.1 is devoted to WCET analysis of multi-threaded programs. We introduce so-called execution frequencies and describe how we establish data flow equations and constraints for solving the equations. An example is given in Section 8.2. Related work is discussed in Section 10.4. Finally, the draw our conclusion in Section 11.2.1.

8.1 Worst-Case Execution Time Analysis on RCPGs

In order to calculate the WCET of a concurrent program, we adopt the generating functions-based approach introduced in [Bli02, Section 4]. We generalize this approach such that we are able to analyze multi-threaded programs. Each node of the RCPG is assigned a data flow variable and a data flow equation is set up based on the predecessors of the RCPG node. A data flow variable is represented by a vector. Each component of the vector reflects a processor and is used to calculate the WCET of the corresponding thread. Recall that only one single thread is allocated to a processor. Even though RCPGs support multiple concurrent threads on one CPU, we restrict the WCET analysis to one thread per processor. This assumption eases the definition of the data flow equations and it is not an inherent restriction of CPGs.

8.1.1 Execution Frequencies

In the remaining part of this chapter, we will use execution frequencies [Bli02, BW09, Ram96]. The execution frequency $e(k \to n)$ is a measure of how often the edge $k \to n$ is taken compared to the other outgoing edges of node k.[2] Thus each execution frequency is a rational number. Its values range from 0 (which models a dead path) to 1, i.e., $0 \leq e(k \to n) \leq 1$.

For each node k, it is required that the execution frequencies of all outgoing edges sum[3] to 1, i.e.,

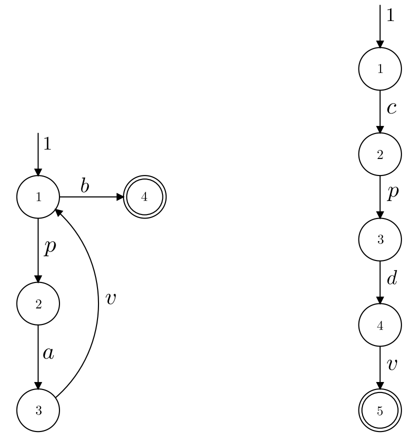
$$\sum_{n \in \mathrm{Succs}(k)} e(k \to n) = 1.$$

If node k has at least two outgoing edges, then we have a so-called node constraint $\sum_{n \in \mathrm{Succs}(k)} e(k \to n) = 1$. We assign a variable to each $e(k \to n)$. A concrete value is assigned to each of these variables during the maximization process which is described below in Section 8.1.6. If a node m has only one outgoing edge to node n, then the execution frequency $e(m \to n) = 1$ is statically known and neither a node constraint nor an additional variable for the execution frequency is needed.

---

[2] The execution frequency $e(m \to n)$ is similar to ORACLE(P, P') in [Bli02].

[3] Similar to Kirchhoff's point rule for electrical circuits [Kir45], but we consider only outgoing edges.



(a) RCFG of thread A (b) RCFG of thread B

Figure 8.1: RCFGs of Threads A and B

8.1.2 Loops

Let $\mathbb{N}_0$ refer to the set of natural numbers including zero, i.e., $\mathbb{N}_0 = \{0, 1, 2, \dots\}$. From now on, we use the variable $\ell_i \in \mathbb{N}_0$ to refer to the number of loop iterations of loop i at CFG level. For each loop, we require that this number is statically known.

In this chapter, we use a running example consisting of the two threads A and B. The corresponding CFGs are depicted in the Figures 8.1a and 8.1b.

The Kronecker sum of the two threads is depicted in Figure 8.2. It can be seen that the Kronecker sum calculates all possible interleavings of the two threads. In addition, we observe that the Kronecker sum generates several copies of basic blocks (in our case edges) and loops in different places. In particular, note that thread A's loop is copied five times (B's number of nodes).

Since RCPGs model all interleavings of the involved threads, a certain execution of the underlying concurrent program (a certain path in the RCPG) may divide the code of a loop in the CFG among all its copies in the RCPG. In particular, we do not know a priori how a loop will be split among its copies in the RCPG for the path producing the WCET. For this reason, we assign variables (with unknown values) to the number of loop iterations of the loop copies in the RCPG. Later on (during the maximization process), concrete values for this loop iteration variables are chosen such that the execution time is maximized. Note that assigning variables to loop iteration numbers implies that some execution frequencies have also to be considered variable. These execution frequencies also get concrete values during the maximization process.

We refer to the number of loop iterations of the jth copy of loop i as $\ell_i^j \in \mathbb{N}_0$. This variable denotes the number of how often the loop entry edge of the jth copy of loop i is executed. The other edges belonging to the same loop copy may be executed differently often. The loop entry of the jth copy (out of n) of loop i gets assigned the execution

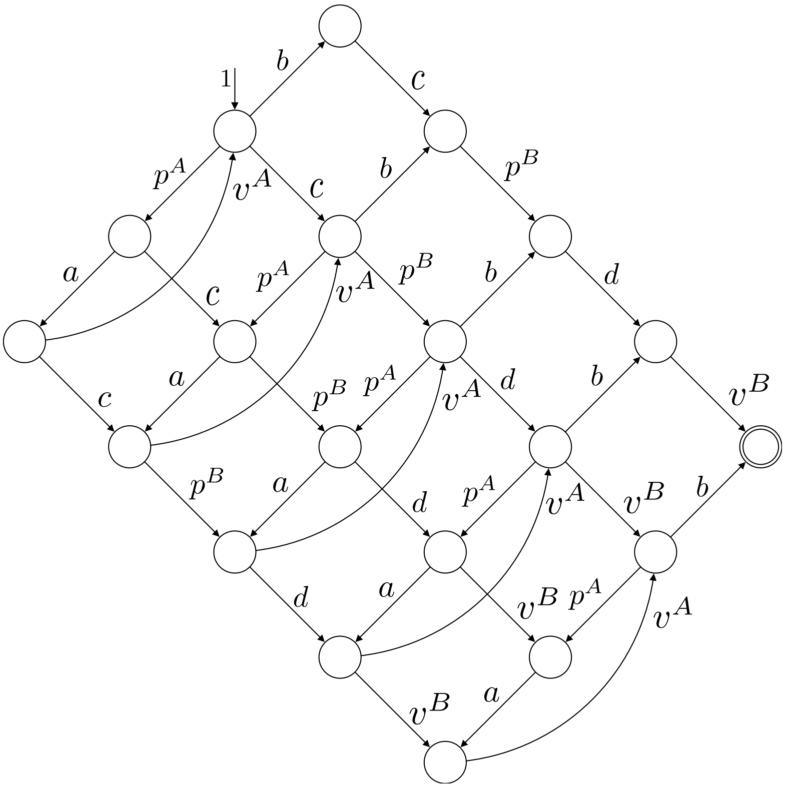


Figure 8.2: Kronecker Sum $A \oplus B$ of Threads A and B

frequency variable

$$e_i^j = \frac{\ell_i^j}{\ell_i^j + 1}, \text{ where } \sum\nolimits_{j=1}^{n} \ell_i^j = \ell_i.$$

Note that the variables ℓ_i^j get numerical values during the maximization process. Thus, the execution frequency of each loop entry edge is calculated automatically.

If node m has multiple outgoing loop entry edges for the loops $1, 2 \ldots n$ and there exists exactly one outgoing non loop entry edge, then the execution frequency for the loop entry edge of loop i is $\frac{\ell_i}{(\sum_{j=1}^{n} \ell_j)+1}$.

Loop Iteration Constraints. Assume CFG loop i is executed ℓ_i times and n copies (as mentioned above due to the Kronecker sum) of that loop are in the RCPG, then we have the constraint $\sum_{j=1}^{n} \ell_i^j = \ell_i$. We assume that the value of variable ℓ_i, i.e., the number of loop iterations on thread (CFG) level is known a priori. The variables ℓ_i^j are used as variables during the maximization process.

In general, RCPGs contain irreducible loops. Ramalingam writes in [Ram99] “A vertex can be an entry vertex of at most one irreducible loop.”. This property does not hold for RCPGs. Remember that RCPGs have the basic blocks on the edges.

A similar property, however, holds for each copy of a loop in RCPGs: An edge can be an entry edge of at most one irreducible loop.

During the generation of the RCPG it is possible to remember each copy of a CFG loop entry edge. In order to establish the loop iteration constraints, we go through this information directly after generating the RCPG.

Loop Exit Constraints. For loop i's jth copy, we have ℓ_i^j iterations. Then we have the loop exit constraint

$$x_i^j = \frac{1}{\ell_i^j + 1},$$

where x_i^j is the sum of execution frequencies of all loop exiting edges of the jth copy of loop i. In general, such loop exiting edges do also include edges from other threads which do not execute any part of loop i. Note that we can calculate the loop exit constraints automatically. Our approach does support nested loops [KKP$^+$11] which result in non-linear constraints. This is one reason which prohibits applying an ILP-based approach e.g. [PS97] for solving the multi-core WCET problem.

8.1.3 Synchronizing Nodes

A thread calling a semaphore's p-operation potentially blocks [Sta11]. On the other hand, a thread calling a semaphore's v-operation may unblock a waiting thread [Sta11]. In RCPGs, blocking occurs at what we call *synchronizing nodes.* We distinguish between two types of synchronizing nodes, namely *vp*- and *pp*-synchronizing nodes.

Each *vp-synchronizing node* has an incoming edge labeled by a semaphore v-operation, an outgoing edge labeled by a p-operation of the same semaphore, and these two edges are part of different threads. In this case, the thread calling the p-operation (potentially) has to wait until the other thread's v-operation is finished.

Definition 8 *A vp*-synchronizing node *is an RCPG node s such that*

- *there exists an edge $e_{in} = (i, s)$ with label v_k and*
- *there exists an edge $e_{out} = (s, j)$ with label p_k,*

where k denotes the same semaphore and the edges e_{in} and e_{out} are mapped to different processors, i.e., $\mathcal{P}(e_{in}) \neq \mathcal{P}(e_{out})$.

For *vp*-synchronizing nodes, we establish specific data flow equations as described in the following subsection.

Definition 9 *A pp*-synchronizing node *is an RCPG node s such that*

- *there exists an edge* $e_{out1} = (s, i)$ *with label* p_k *and*
- *there exists an edge* $e_{out2} = (s, j)$ *with label* p_k*,*

where k *denotes the same semaphore and the edges* e_{out1} *and* e_{out2} *are mapped to different processors, i.e.,* $\mathcal{P}(e_{out1}) \neq \mathcal{P}(e_{out2})$*.*

For *pp*-synchronizing nodes, we establish *fairness constraints* ensuring a deterministic choice when e.g. the time of both involved CPUs at node s is exactly the same.

8.1.4 Setting Up And Solving Data Flow Equations

In this section, we extend the generating function [GKP94] based approach of Section 4 of [Bli02] such that we are able to calculate the WCET of concurrent programs modeled by RCPGs. Each RCPG node's data flow equation is set up according to its predecessors and the incoming edges (including execution frequencies, execution time and, in case of *vp*-synchronizing nodes, stalling time). These data flow equations can e.g. be solved according to [Sre95, SGL98].

Let the vector $\mathbf{P}(z) = (P_1(z), \ldots, P_i(z), \ldots, P_n(z))^{\intercal}$. We write $\mathbf{P}^i(z)$ to denote the ith component $P_i(z)$ of vector $\mathbf{P}(z)$. In addition, $\mathbf{P_x}(z)$ refers to the vector of node x.

Let a be a scalar and let $\mathbf{P}(z)$ and $\mathbf{Q}(z)$ be two n-dimensional vectors. The addition and multiplication of vectors and the multiplication of a scalar with a vector are defined as follows:

$$\begin{aligned}\mathbf{P}(z) + \mathbf{Q}(z) =& (P_1(z) + Q_1(z), P_2(z) + Q_2(z), \ldots, P_n(z) + Q_n(z))^{\intercal},\\ \mathbf{P}(z)\,\mathbf{Q}(z) =& (P_1(z)\,Q_1(z), P_2(z)\,Q_2(z), \ldots, P_n(z)\,Q_n(z))^{\intercal},\\ a\,\mathbf{P}(z) =& (a\,P_1(z), a\,P_2(z), \ldots, a\,P_n(z))^{\intercal}.\end{aligned}$$

Definition 10 (Setting up data flow equations) *Let* $\mathbf{time}(m \to n)$ *refer to the time assigned to edge* $m \to n$*. In addition, the set of predecessor nodes of node* n *is referred to as* $Preds(n)$*.*

If n *is a non-vp-synchronizing node and edge* $m \to n$ *is mapped to processor* k*, then*

$$\mathbf{P_n}(z) = \sum_{m \in Preds(n)} e(m \to n)\ \mathfrak{t}(m \to n)\ \mathbf{P_m}(z),$$

where the k*th component of vector* $\mathfrak{t}(m \to n)$ *is* $z^{\mathbf{time}(m \to n)}$ *and the other components of that vector are equal to* 1*.*

Let s *be a vp-synchronizing node. In addition, let* π_v *and* π_p *be the processors which the edges* $i \to s$ *and* $s \to j$ *are mapped to, i.e,* $\pi_v = \mathcal{P}(i \to s)$ *and* $\pi_p = \mathcal{P}(s \to j)$*.*[4] *Then for* $k \neq \pi_p$

$$\mathbf{P_j}^k(z) = e(s \to j)\ \mathfrak{t}(s \to j)^k\ \mathbf{P_s}^k(z)$$

---

[4] According to the definition of *vp*-synchronizing nodes $\pi_v \neq \pi_p$.

and

$$\mathbf{P_j}^{\pi_p}(z) = e(s \to j)\ \mathfrak{t}(s \to j)^{\pi_p}\ \max\Big(\mathbf{P_s}^{\pi_v}(z), \mathbf{P_s}^{\pi_p}(z)\Big) + \sum_{m \neq s, m \in Preds(j)} e(m \to j)\ \mathfrak{t}(m \to j)^{\pi_p}\ \mathbf{P_m}^{\pi_p}(z)$$

where the first term considers the incoming p-edge and the second term takes into account all other incoming edges of the potentially blocking thread running on processor π_p.

The max-operator in Definition 10 is not an ordinary maximum operation for numbers. During the maximization process, we actually do the whole calculation twice. One time, we replace $\max(\mathbf{P_s}^{\pi_v}(z), \mathbf{P_s}^{\pi_p}(z))$ by $\mathbf{P_s}^{\pi_v}(z)$ and then, we do this calculation using the second solution $\mathbf{P_s}^{\pi_p}(z)$. In the end, the solution with the highest WCET value will be taken.

The entry node's equation $\mathbf{P_{entry}}(z)$ follows the rules above and, in addition, for n threads adds an n-dimensional vector $(1, 1, 1, \dots, 1)^T$.

The data flow equations can intuitively be explained as follows. We cumulate the execution times in an interleavings semantics fashion. One can think of taking one edge after the other. Nevertheless, edges may be executed in parallel and the execution and stalling times are added to the corresponding vector components. In the overall process, we get the WCET of the concurrent program.

The system of data flow equations can be solved efficiently by applying an algorithm presented in [SGL98]. It relies on two operations: inserting one equation into another and solving recursions by so-called loop breaking. The order of these operations is completely determined by the DJ graph introduced in [SGL98]. As a result, we get an explicit formula for the final node.

In order to double-check that we calculate a correct solution, we used Mathematica© to solve the node equations, too. Both of the two approaches for solving the node equations calculate the same and correct results.

8.1.5 Partial Loop Unrolling

For *vp*-synchronizing nodes having at least one outgoing loop entry edge[5], we have to partly unroll the corresponding loop such that one iteration is statically present in the RCPG's equations. Partial loop unrolling ensures that synchronization is modeled correctly. Only the unrolled part contains a synchronizing node. Some execution frequencies and equations have to be added or adapted. Edges have to be added to ensure that the original and the unrolled loop behave semantically equivalent. For example, if the original loop was able to iterate $n \geq 0$ times, then the new construct must also allow

---

[5] Note that we can detect these nodes when generating the RCPG.

the same number of iterations. In order to define some execution frequencies correctly, we are using the Kronecker delta function. We do such partial loop unrolling for our example in Subsection 8.2.2. In our example, we e.g. have to add edge $b_5^{'}$ to allow a zero number of iterations (compare Figure 8.3 and Figure 8.4). Note that partial loop unrolling can be fully automated.

8.1.6 Maximization Process

In order to determine the WCET, we have to differentiate the solution for the final node n_f with respect to z and after that set $z = 1$.

Let function^k refer to the function representing the solution of the kth component of the final node n_f. According to well-known facts of generating functions [GKP94, Bli02] it is defined as

$$\text{function}^k = \frac{\mathrm{d}}{\mathrm{d}z}\,\mathbf{P_f}^k(z)\,\Bigg|_{z=1}.$$

In order to calculate the loop iteration count for all loop copies and to calculate the undefined execution frequencies within the given constraints, we maximize this function. This goes beyond the approaches given in [MB12b, Bli02]. During this maximization step, for which we used `NMaximize` of Mathematica©, e.g. all ℓ_i^j are treated as variables. For each of these variables, Mathematica finds values within the given constraints. Thus Mathematica assigns valid values for all ℓ_i^j and all the unknown execution frequencies. Of course, instead of Mathematica, any non-linear function solver capable of handling constraints can be used. The WCET of the kth CPU core is given by

$$\text{WCET}^k = \text{Maximize}(\text{function}^k, \{\mathit{constraints}\}).$$

In the following, the variable configuration found during this maximization is used. The WCET of a concurrent program consisting of n threads is defined as

$$\max\Big(\text{WCET}^1, \ldots, \text{WCET}^n\Big),$$

where max is the ordinary maximum operator for numbers.

If the RCPG contains s *vp*-synchronizing nodes, then the maximization process has to be done 2^s times. One time for each possible value of $\max(\ldots,\ldots)$ originating from the *vp*-synchronizing nodes. At last the largest value of those 2^s results represents the WCET of the concurrent program. Hence, the computational complexity may increase exponentially in s. However, s is usually small. For n threads and r semaphores, the number of *vp*-synchronizing nodes in the CPG is bounded above by $\sum_{j=1}^{r}\sum_{i=1}^{n} v_j^i \sum_{k=0,k\neq i}^{n} p_j^k$, where v_j^i is the number of v-operations of semaphore j in thread i and p_j^k is the number of p-operations of semaphore j in thread k. Depending on how the semaphores are used not all *vp*-synchronizing nodes may be part of the RCPG. In addition, information may be available which allows to conclude that even some of the present cases cannot result in the final WCET value. Then, these cases need not be considered in the maximization process. In [VBS12] an example with CPG matrix size of 298721280 has been analyzed within 400 ms. It contained 13 semaphores and only 15 synchronization nodes. Even though the graphs for travel time analysis do not contain loops, the number of synchronizing nodes is comparable.

8.2 Example

In this section, we go on with our running example. It consists of two threads, namely A and B, sharing one single semaphore with operations p and v. Thus, the example includes mutual exclusion via a semaphore. In addition, the example includes one single loop in thread A. The CFGs of the two threads A and B are depicted in Figures 8.1a and 8.1b, respectively. Each edge is labeled by a basic block.

Together with a RCFG of a binary semaphore, we calculate the adjacency matrix P of the corresponding RCPG in the following steps: The interleaved threads are given by $T = A \oplus B$. Because we have only one semaphore, the interleaved semaphores are trivially defined as

$$S = \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix}.$$

The program's matrix P is given by

$$P = T \oslash_{\mathcal{L}_\mathrm{S}} S + T_{\mathcal{L}_\mathrm{V}} \otimes I_2,$$

where $\mathcal{L}_\mathrm{S} = \{p, v\}$ and $\mathcal{L}_\mathrm{V} = \{a, b, c, d\}$. The RCPG of the A-B-system is depicted in Figure 8.3. The edges of the RCPG are labeled by their execution frequencies on RCPG level. However, we indicate for each execution frequency l_x that it is the execution frequency for the xth copy of basic block l.[6] Due to synchronization through the used semaphore, two out of five loop copies (compared to the Kronecker sum in Figure 8.2) are not reachable in the RCPG. In order to establish each node's data flow equation, we (compared to Figure 8.2) introduce node numbers. Note that these numbers do not refer to the corresponding matrix indices.

We assume that both threads access shared variables in the basic blocks a and d. Thus the basic blocks a and d are only allowed to be executed in a mutually exclusive fashion. This is ensured by using a semaphore. The basic blocks a and d are protected by p-calls. After the corresponding thread finishes its execution of a or d, the semaphore is released by a v-call. We assume that all the other basic blocks do not access shared variables. Note that the threads are mapped to distinct processors and that these mappings are immutable.

Each variable x in this example (except ℓ, ℓ^1, ℓ^2 and ℓ^3) is a rational number such that $0 \leq x \leq 1$. We assume that thread A's loop is executed ℓ times and the three copies of

---

[6] A hardware analysis may detect that the copies of a basic block have different execution times due to e.g. shared data caches or instruction pipelining.

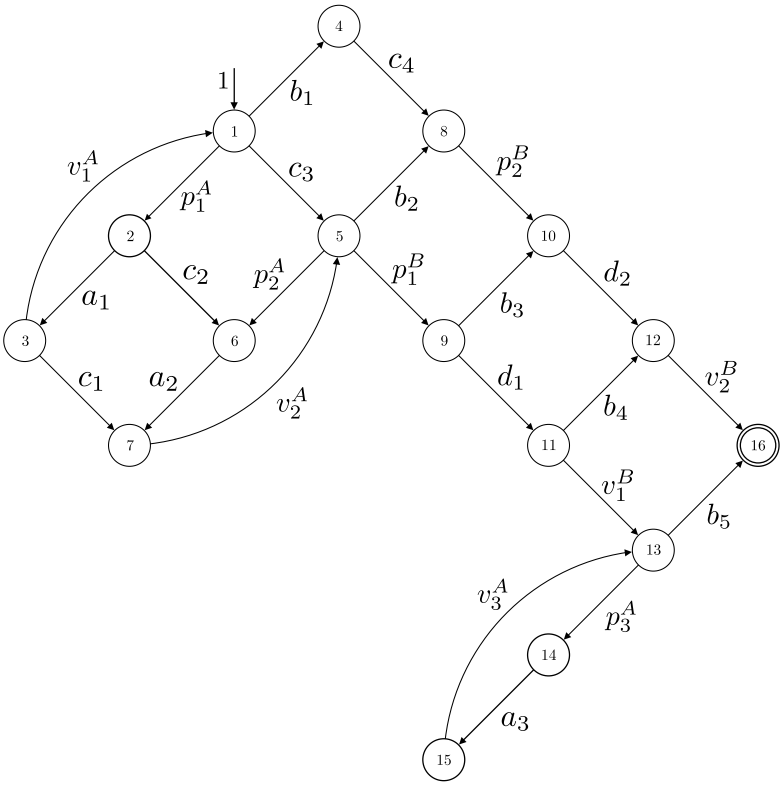


Figure 8.3: RCPG

the loop are executed ℓ^1, ℓ^2 and ℓ^3 times, respectively. Thus, we have the loop iteration constraint $\ell^1 + \ell^2 + \ell^3 = \ell$, where $\ell^i \in \mathbb{N}_0$.

For this example, we set up the data flow equations such that thread A's and B's equations are defined in the vector's first and second, respectively, vector component.

8.2.1 Equations Not Affected By Partial Loop Unrolling

Following the rules of Section 8.1, we obtain the following equations.

$$\mathbf{P_1}(z) = v_1^A \begin{pmatrix} z^{\tau_v} \\ 1 \end{pmatrix} \mathbf{P_3}(z) + \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\mathbf{P_2}(z) = p_1^A \begin{pmatrix} z^{\tau_p} \\ 1 \end{pmatrix} \mathbf{P_1}(z)$$

$$\mathbf{P_3}(z) = a_1 \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P_2}(z)$$

$$\mathbf{P_4}(z) = b_1 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_1}(z)$$

$$
\begin{aligned}
\mathbf{P_5}(z) &= v_2^A \begin{pmatrix} z^{\tau_v} \\ 1 \end{pmatrix} \mathbf{P_7}(z) + c_3 \begin{pmatrix} 1 \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z) \\
\mathbf{P_6}(z) &= p_2^A \begin{pmatrix} z^{\tau_p} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + c_2 \begin{pmatrix} 1 \\ z^{\tau_c} \end{pmatrix} \mathbf{P_2}(z) \\
\mathbf{P_7}(z) &= a_2 \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P_6}(z) + c_1 \begin{pmatrix} 1 \\ z^{\tau_c} \end{pmatrix} \mathbf{P_3}(z) \\
\mathbf{P_8}(z) &= b_2 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + c_4 \begin{pmatrix} 1 \\ z^{\tau_c} \end{pmatrix} \mathbf{P_4}(z) \\
\mathbf{P_9}(z) &= p_1^B \begin{pmatrix} \mathbf{P_5}^1(z) \\ z^{\tau_p} \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big) \end{pmatrix} \\
\mathbf{P_{10}}(z) &= b_3 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_9}(z) + p_2^B \begin{pmatrix} 1 \\ z^{\tau_p} \end{pmatrix} \mathbf{P_8}(z) \\
\mathbf{P_{11}}(z) &= d_1 \begin{pmatrix} 1 \\ z^{\tau_d} \end{pmatrix} \mathbf{P_9}(z) \\
\mathbf{P_{12}}(z) &= b_4 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_{11}}(z) + d_2 \begin{pmatrix} 1 \\ z^{\tau_d} \end{pmatrix} \mathbf{P_{10}}(z) \\
\mathbf{P_{15}}(z) &= a_3 \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P_{14}}(z)
\end{aligned}
$$

Note again that the max-operators in $\mathbf{P_9}(z)$ and $\mathbf{P'_{14}}(z)$ (the latter is stated in Subsection 8.2.2) originate because the original nodes 9 and 14, respectively, are *vp*-synchronizing nodes and that max is not the ordinary maximum operation for numbers. During the maximization process, for each max-operator, we do the whole calculation twice, once for each potential solution.

8.2.2 Partial Loop Unrolling

Node 13 is a *vp*-synchronizing node and edge 13 → 14 constitutes a loop entry edge. Hence, we have to apply partial loop unrolling. We get the following additional data flow equations.

$$
\begin{aligned}
\mathbf{P'_{13}}(z) &= v_1^B \begin{pmatrix} 1 \\ z^{\tau_v} \end{pmatrix} \mathbf{P_{11}}(z) \\
\mathbf{P'_{14}}(z) &= p_3^{A'} \begin{pmatrix} z^{\tau_p} \max\Big(\mathbf{P'_{13}}^1(z), \mathbf{P'_{13}}^2(z)\Big) \\ \mathbf{P'_{13}}^2(z) \end{pmatrix}
\end{aligned}
$$

$$\begin{aligned}
\mathbf{P'_{15}}(z) &= \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P'_{14}}(z) \\
\mathbf{P_{13}}(z) &= v_3^A \begin{pmatrix} z^{\tau_v} \\ 1 \end{pmatrix} \mathbf{P_{15}}(z) + \begin{pmatrix} z^{\tau_v} \\ 1 \end{pmatrix} \mathbf{P'_{15}}(z) \\
\mathbf{P_{14}}(z) &= p_3^A \begin{pmatrix} z^{\tau_p} \\ 1 \end{pmatrix} \mathbf{P_{13}}(z) \\
\mathbf{P_{16}}(z) &= b_5 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_{13}}(z) + v_2^B \begin{pmatrix} 1 \\ z^{\tau_v} \end{pmatrix} \mathbf{P_{12}}(z) + b_5^{'} \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P'_{13}}(z)
\end{aligned}$$

The changes in the equations can be interpreted on RCPG-level as depicted in Figure 8.4 (compare to Figure 8.3). For edges whose execution frequency is 1, we write $1(a)$ in order to state that the edge refers to the basic block a. For these edges, the execution time would otherwise be unclear. In the following, we use the Kronecker delta function. Kronecker delta $\delta_{i,j}$ is defined as

$$\delta_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

By partially unrolling the loop, we get the execution frequencies:

$$b_5^{'} = \delta_{\ell^3,0}, \qquad p_3^{A'} = 1 - b_5^{'}, \qquad p_3^A = \begin{cases} \frac{\ell^3-1}{\ell^3} & \text{if } \ell^3 > 1, \\ 0 & \text{otherwise.} \end{cases}$$

Note that the non-linear function solver employed for the maximization process must be able to handle $\delta_{i,j}$ and case functions (like that used in the right hand side of p_3^A) correctly.

8.2.3 Execution Frequencies and Constraints

The following execution frequencies and constraints are extracted out of the RCPG. The execution frequencies of the loop entry edges p_1^A and p_2^A are established as follows:

$$p_1^A = \tfrac{\ell^1}{\ell^1+1}, \qquad p_2^A = \tfrac{\ell^2}{\ell^2+1}.$$

From the node constraints, we get a_2, a_3, c_4, d_2, v_2^A, v_3^A, p_2^B and v_2^B statically set to 1. The remaining node constraints contribute execution frequency variables and the corresponding constraints for the final maximization process.

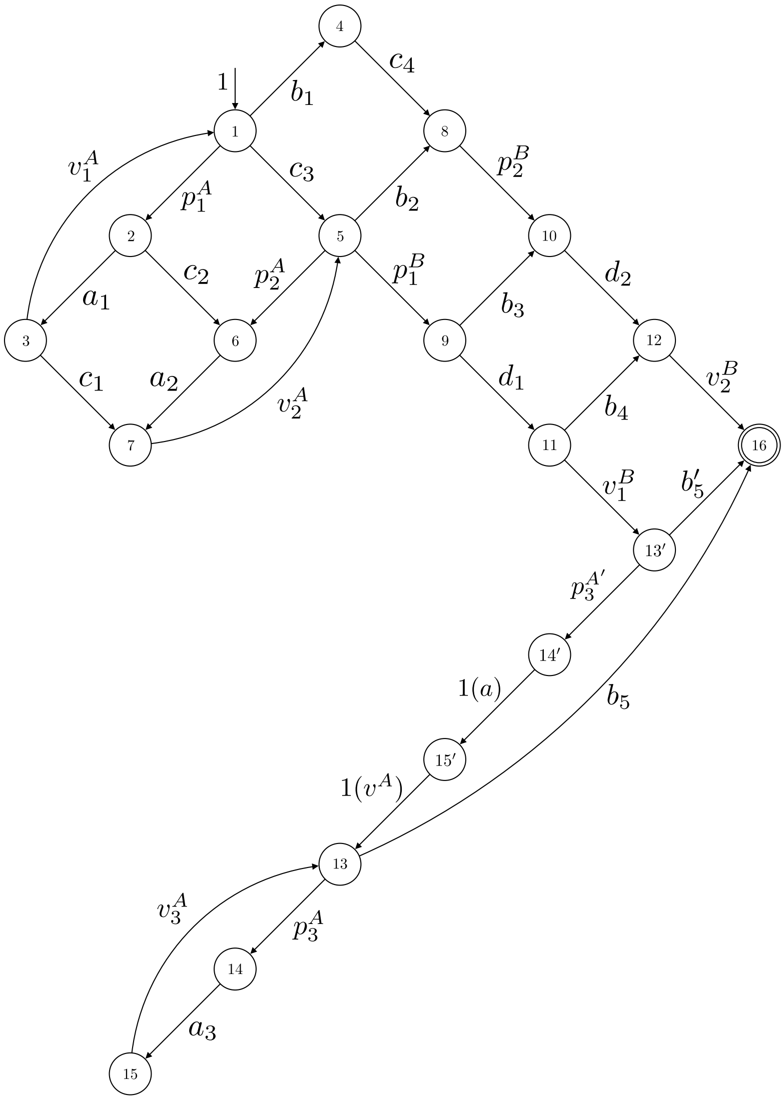


Figure 8.4: Adapted RCPG

$$a_1 = 1 - c_2, \qquad b_4 = 1 - v_1^B,$$

$$b_1 + c_3 + p_1^A = 1,$$

$$b_2 + p_1^B + p_2^A = 1, \qquad b_5 = 1 - p_3^A,$$

$$b_3 = 1 - d_1, \qquad c_1 = 1 - v_1^A.$$

The loop exit constraints are as follows:

$$b_2 + p_1^B = \tfrac{1}{\ell^2+1},$$

$$b_1 + c_1 + c_2 + c_3 = \tfrac{1}{\ell^1+1}.$$

The time needed for executing basic block b is referred to as τ_b. We assume that all copies of a certain basic block lead to the same execution time. Thus, e.g., each one out of b_1, b_2 and b_3 has an execution time of τ_b. Finally, for node 5, which is a *pp*-synchronizing node, we have the following constraints. These conditions follow from our computational model described in the introduction of this chapter and the fairness constraints from Subsection 8.1.3:

$$(\ell^1 + \ell^2 - 1)\,(\tau_p + \tau_a + \tau_v) < \tau_c,$$

$$(\ell^1 + \ell^2)\,(\tau_p + \tau_a + \tau_v) \geq \tau_c \vee \ell\,(\tau_p + \tau_a + \tau_v) < \tau_c,$$

$$\ell\,(\tau_p + \tau_a + \tau_v) < c \Rightarrow \ell^3 \equiv 0.$$

8.2.4 Solving the Equations

We used two approaches to solve the equations. At first, we applied the eager elimination method [Sre95, SGL98]. To double-check the solution, we used an off-the-shelf equation solver, namely Mathematica©, too. Both approaches calculate the same and correct result. In the following, we solve the equations by applying the eager elimination method. For a concise presentation, we use the notation $\tau_a + \tau_b = \tau_{a,b}$. From now on, we assume that $\max(\mathbf{P'_{13}}^1(z), \mathbf{P'_{13}}^2(z)) = \mathbf{P'_{13}}^2(z)$. This obviously leads to the WCET because thread A has to wait for thread B at synchronizing node $13'$. A calculation for the other case reveals that, it does not find the WCET.

Note that the elements of our data flow framework are vectors of rational functions, i.e., one polynomial divided by another polynomial. Hence insertion and loop breaking are straight-forward.

We start by substituting equation 13' into 14':

$$\mathbf{P'_{13}}^2(z) = v_1^B\, z^{\tau_v}\, \mathbf{P_{11}}^2(z)$$

$$\mathbf{P}'_{\mathbf{14}}(z) = p_3^{A'} \begin{pmatrix} z^{\tau_p}\, {\mathbf{P}'_{\mathbf{13}}}^{2}(z) \\ {\mathbf{P}'_{\mathbf{13}}}^{2}(z) \end{pmatrix}$$

$$\mathbf{P}'_{\mathbf{14}}(z) = p_3^{A'} \begin{pmatrix} v_1^B\, z^{\tau_{p,v}}\, {\mathbf{P_{11}}}^{2}(z) \\ v_1^B\, z^{\tau_v}\, {\mathbf{P_{11}}}^{2}(z) \end{pmatrix}$$

Substitute 14’ into 15’

$$\mathbf{P}'_{\mathbf{15}}(z) = p_3^{A'} \begin{pmatrix} v_1^B\, z^{\tau_{a,p,v}}\, {\mathbf{P_{11}}}^{2}(z) \\ v_1^B\, z^{\tau_v}\, {\mathbf{P_{11}}}^{2}(z) \end{pmatrix}$$

Substitute 15’ into 13

$$\mathbf{P_{13}}(z) = v_3^A \begin{pmatrix} z^{\tau_v} \\ 1 \end{pmatrix} \mathbf{P_{15}}(z) + p_3^{A'} \begin{pmatrix} v_1^B\, z^{\tau_{a,p,2v}}\, {\mathbf{P_{11}}}^{2}(z) \\ v_1^B\, z^{\tau_v}\, {\mathbf{P_{11}}}^{2}(z) \end{pmatrix}$$

Substitute 15 into 13

$$\mathbf{P_{13}}(z) = a_3\, v_3^A \begin{pmatrix} z^{\tau_{a,v}} \\ 1 \end{pmatrix} \mathbf{P_{14}}(z) + p_3^{A'} \begin{pmatrix} v_1^B\, z^{\tau_{a,p,2v}}\, {\mathbf{P_{11}}}^{2}(z) \\ v_1^B\, z^{\tau_v}\, {\mathbf{P_{11}}}^{2}(z) \end{pmatrix}$$

Substitute 14 into 13

$$\mathbf{P_{13}}(z) = a_3\, p_3^A\, v_3^A \begin{pmatrix} z^{\tau_{a,p,v}} \\ 1 \end{pmatrix} \mathbf{P_{13}}(z) + p_3^{A'} \begin{pmatrix} v_1^B\, z^{\tau_{a,p,2v}}\, {\mathbf{P_{11}}}^{2}(z) \\ v_1^B\, z^{\tau_v}\, {\mathbf{P_{11}}}^{2}(z) \end{pmatrix}$$

Now, we loop break the equation of node 13:

$${\mathbf{P_{13}}}^{1}(z) = \frac{p_3^{A'}\, v_1^B\, z^{\tau_{a,p,2v}}\, {\mathbf{P_{11}}}^{2}(z)}{1 - a_3\, p_3^A\, v_3^A\, z^{\tau_{a,p,v}}}$$

$${\mathbf{P_{13}}}^{2}(z) = \frac{p_3^{A'}\, v_1^B\, z^{\tau_v}\, {\mathbf{P_{11}}}^{2}(z)}{1 - a_3\, p_3^A\, v_3^A}$$

Substitute 13 into 16

$$\mathbf{P_{16}}(z) = b_5 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_{13}}(z) + v_2^B \begin{pmatrix} 1 \\ z^{\tau_v} \end{pmatrix} \mathbf{P_{12}}(z) + b_5^{'} \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P}'_{\mathbf{13}}(z)$$

$$\mathbf{P_{16}}(z) = \begin{pmatrix} \frac{b_5\, p_3^{A'}\, v_1^B\, z^{\tau_{a,b,p,2v}}\, {\mathbf{P_{11}}}^{2}(z)}{1-a_3\, p_3^A\, v_3^A\, z^{\tau_{a,p,v}}} \\ \frac{b_5\, p_3^{A'}\, v_1^B\, z^{\tau_v}\, {\mathbf{P_{11}}}^{2}(z)}{1-a_3\, p_3^A\, v_3^A} \end{pmatrix} + v_2^B \begin{pmatrix} 1 \\ z^{\tau_v} \end{pmatrix} \mathbf{P_{12}}(z) + b_5^{'} \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P}'_{\mathbf{13}}(z)$$

Substitute 13’ into 16

$$\mathbf{P}'_{\mathbf{13}}(z) = v_1^B \begin{pmatrix} 1 \\ z^{\tau_v} \end{pmatrix} \mathbf{P_{11}}(z)$$

$$\mathbf{P_{16}}(z) = \begin{pmatrix} \frac{b_5\, p_3^{A'}\, v_1^B\, z^{\tau_{a,b,p,2v}}\, {\mathbf{P_{11}}}^{2}(z)}{1-a_3\, p_3^A\, v_3^A\, z^{\tau_{a,p,v}}} \\ \frac{b_5\, p_3^{A'}\, v_1^B\, z^{\tau_v}\, {\mathbf{P_{11}}}^{2}(z)}{1-a_3\, p_3^A\, v_3^A} \end{pmatrix} + v_2^B \begin{pmatrix} 1 \\ z^{\tau_v} \end{pmatrix} \mathbf{P_{12}}(z) + b_5^{'}\, v_1^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_v} \end{pmatrix} \mathbf{P_{11}}(z)$$

Substitute 11 into 12

$$\mathbf{P_{11}}(z) = d_1 \begin{pmatrix} 1 \\ z^{\tau_d} \end{pmatrix} \mathbf{P_9}(z)$$

$$\mathbf{P_{12}}(z) = b_4 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_{11}}(z) + d_2 \begin{pmatrix} 1 \\ z^{\tau_d} \end{pmatrix} \mathbf{P_{10}}(z)$$

$$\mathbf{P_{12}}(z) = b_4\, d_1 \begin{pmatrix} z^{\tau_b} \\ z^{\tau_d} \end{pmatrix} \mathbf{P_9}(z) + d_2 \begin{pmatrix} 1 \\ z^{\tau_d} \end{pmatrix} \mathbf{P_{10}}(z)$$

Substitute 11 into 16

$$\mathbf{P_{11}}(z) = d_1 \begin{pmatrix} 1 \\ z^{\tau_d} \end{pmatrix} \mathbf{P_9}(z)$$

$$\mathbf{P_{16}}(z) = \begin{pmatrix} \frac{b_5\, d_1\, p_3^{A'}\, v_1^B\, z^{\tau_{a,b,d,p,2v}}\, \mathbf{P_9}^2(z)}{1 - a_3\, p_3^A\, v_3^A\, z^{\tau_{a,p,v}}} \\ \frac{b_5\, d_1\, p_3^{A'}\, v_1^B\, z^{\tau_{d,v}}\, \mathbf{P_9}^2(z)}{1 - a_3\, p_3^A\, v_3^A} \end{pmatrix} + v_2^B \begin{pmatrix} 1 \\ z^{\tau_v} \end{pmatrix} \mathbf{P_{12}}(z) + b_5^{'}\, d_1\, v_1^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{d,v}} \end{pmatrix} \mathbf{P_9}(z)$$

Substitute 9 into 10

$$\mathbf{P_9}(z) = p_1^B \begin{pmatrix} \mathbf{P_5}^1(z) \\ z^{\tau_p} \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big) \end{pmatrix}$$

$$\mathbf{P_{10}}(z) = b_3 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_9}(z) + p_2^B \begin{pmatrix} 1 \\ z^{\tau_p} \end{pmatrix} \mathbf{P_8}(z)$$

$$\mathbf{P_{10}}(z) = b_3\, p_1^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_p} \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big) \end{pmatrix} + p_2^B \begin{pmatrix} 1 \\ z^{\tau_p} \end{pmatrix} \mathbf{P_8}(z)$$

Substitute 9 into 12

$$\mathbf{P_{12}}(z) = b_4\, d_1 \begin{pmatrix} z^{\tau_b} \\ z^{\tau_d} \end{pmatrix} \mathbf{P_9}(z) + d_2 \begin{pmatrix} 1 \\ z^{\tau_d} \end{pmatrix} \mathbf{P_{10}}(z)$$

$$\mathbf{P_{12}}(z) = b_4\, d_1\; p_1^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p}} \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big) \end{pmatrix} + d_2 \begin{pmatrix} 1 \\ z^{\tau_d} \end{pmatrix} \mathbf{P_{10}}(z)$$

Substitute 9 into 16

$$\mathbf{P_9}(z) = p_1^B \begin{pmatrix} \mathbf{P_5}^1(z) \\ z^{\tau_p}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix}$$

$$\mathbf{P_{16}}(z) = \begin{pmatrix} \frac{b_5\, d_1\, p_1^B\, p_3^{A'}\, v_1^B\, z^{\tau_{a,b,d,2p,2v}}\, \mathrm{mP}_{\mathbf{5}}(z)}{1 - a_3\, p_3^A\, v_3^A\, z^{\tau_{a,p,v}}} \\ \frac{b_5\, d_1\, p_1^B\, p_3^{A'}\, v_1^B\, z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z)}{1 - a_3\, p_3^A\, v_3^A} \end{pmatrix} + v_2^B \begin{pmatrix} 1 \\ z^{\tau_v} \end{pmatrix} \mathbf{P_{12}}(z)$$

$$+ \, b_5^{'} \, d_1 \, p_1^B \, v_1^B \begin{pmatrix} z^{\tau_b} \, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p,v}} \, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix}$$

Substitute 3 into 1

$$\mathbf{P_3}(z) = a_1 \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P_2}(z)$$

$$\mathbf{P_1}(z) = v_1^A \begin{pmatrix} z^{\tau_v} \\ 1 \end{pmatrix} \mathbf{P_3}(z) + \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\mathbf{P_1}(z) = a_1 \, v_1^A \begin{pmatrix} z^{\tau_{a,v}} \\ 1 \end{pmatrix} \mathbf{P_2}(z) + \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Substitute 3 into 7

$$\mathbf{P_7}(z) = a_2 \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P_6}(z) + c_1 \begin{pmatrix} 1 \\ z^{\tau_c} \end{pmatrix} \mathbf{P_3}(z)$$

$$\mathbf{P_7}(z) = a_2 \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P_6}(z) + a_1 \, c_1 \begin{pmatrix} z^{\tau_a} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_2}(z)$$

Substitute 2 into 6

$$\mathbf{P_2}(z) = p_1^A \begin{pmatrix} z^{\tau_p} \\ 1 \end{pmatrix} \mathbf{P_1}(z)$$

$$\mathbf{P_6}(z) = p_2^A \begin{pmatrix} z^{\tau_p} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + c_2 \begin{pmatrix} 1 \\ z^{\tau_c} \end{pmatrix} \mathbf{P_2}(z)$$

$$\mathbf{P_6}(z) = p_2^A \begin{pmatrix} z^{\tau_p} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + c_2 \, p_1^A \begin{pmatrix} z^{\tau_p} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z)$$

Substitute 2 into 7

$$\mathbf{P_7}(z) = a_2 \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P_6}(z) + a_1 \, c_1 \begin{pmatrix} z^{\tau_a} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_2}(z)$$

$$\mathbf{P_7}(z) = a_2 \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P_6}(z) + a_1 \, c_1 \, p_1^A \begin{pmatrix} z^{\tau_{a,p}} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z)$$

Substitute 4 into 8

$$\mathbf{P_4}(z) = b_1 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_1}(z)$$

$$\mathbf{P_8}(z) = b_2 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + c_4 \begin{pmatrix} 1 \\ z^{\tau_c} \end{pmatrix} \mathbf{P_4}(z)$$

$$\mathbf{P_8}(z) = b_2 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + b_1 \, c_4 \begin{pmatrix} z^{\tau_b} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z)$$

Now, we have a strongly connected component (SCC) [SGL98, Tar72] containing the nodes 5, 6, and 7. We have to collapse the SCC now. We substitute 6 into 7:

$$
\begin{aligned}
\mathbf{P_6}(z) &= p_2^A \begin{pmatrix} z^{\tau_p} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + c_2\, p_1^A \begin{pmatrix} z^{\tau_p} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z), \\
\mathbf{P_7}(z) &= a_2 \begin{pmatrix} z^{\tau_a} \\ 1 \end{pmatrix} \mathbf{P_6}(z) + a_1\, c_1\, p_1^A \begin{pmatrix} z^{\tau_{a,p}} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z), \\
\mathbf{P_7}(z) &= a_2\, p_2^A \begin{pmatrix} z^{\tau_{a,p}} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + a_2\, c_2\, p_1^A \begin{pmatrix} z^{\tau_{a,p}} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z) \\
&\quad + a_1\, c_1\, p_1^A \begin{pmatrix} z^{\tau_{a,p}} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z).
\end{aligned}
$$

Next, we substitute 7 into 5:

$$
\begin{aligned}
\mathbf{P_5}(z) &= v_2^A \begin{pmatrix} z^{\tau_v} \\ 1 \end{pmatrix} \mathbf{P_7}(z) + c_3 \begin{pmatrix} 1 \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z), \\
\mathbf{P_5}(z) &= a_2\, p_2^A\, v_2^A \begin{pmatrix} z^{\tau_{a,p,v}} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + (a_2\, c_2 + a_1\, c_1)\; p_1^A\, v_2^A \begin{pmatrix} z^{\tau_{a,p,v}} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z) \\
&\quad + c_3 \begin{pmatrix} 1 \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z).
\end{aligned}
$$

Finally, we loop break the equation of node 5. For the first component, we obtain:

$$
\begin{aligned}
\mathbf{P_5}^1(z) &= a_2\, p_2^A\, v_2^A\, z^{\tau_{a,p,v}}\, \mathbf{P_5}^1(z) + (a_2\, c_2 + a_1\, c_1\,)\; p_1^A\, v_2^A\, z^{\tau_{a,p,v}}\, \mathbf{P_1}^1(z) \\
&\quad + c_3\, \mathbf{P_1}^1(z), \\
\mathbf{P_5}^1(z) &= \frac{(a_1\, c_1 + a_2\, c_2)\; p_1^A\, v_2^A\, z^{\tau_{a,p,v}}\, \mathbf{P_1}^1(z) + c_3\, \mathbf{P_1}^1(z)}{1 - a_2\, p_2^A\, v_2^A\, z^{\tau_{a,p,v}}},
\end{aligned}
$$

and for the second component, we get

$$
\begin{aligned}
\mathbf{P_5}^2(z) &= a_2\, p_2^A\, v_2^A\, \mathbf{P_5}^2(z) + (a_1\, c_1 + a_2\, c_2)\; p_1^A\, v_2^A\, z^{\tau_c}\, \mathbf{P_1}^2(z) + c_3\, z^{\tau_c}\, \mathbf{P_1}^2(z), \\
\mathbf{P_5}^2(z) &= \frac{(a_1\, c_1 + a_2\, c_2)\; p_1^A\, v_2^A\, z^{\tau_c}\, \mathbf{P_1}^2(z) + c_3\, z^{\tau_c}\, \mathbf{P_1}^2(z)}{1 - a_2\, p_2^A\, v_2^A}.
\end{aligned}
$$

Hence, we have

$$
\mathbf{P_5}(z) = \begin{pmatrix} \frac{(a_1\, c_1 + a_2\, c_2)\, p_1^A\, v_2^A\, z^{\tau_{a,p,v}}\, \mathbf{P_1}^1(z) + c_3\, \mathbf{P_1}^1(z)}{1 - a_2\, p_2^A\, v_2^A\, z^{\tau_{a,p,v}}} \\ \frac{(a_1\, c_1 + a_2\, c_2)\, p_1^A\, v_2^A\, z^{\tau_c}\, \mathbf{P_1}^2(z) + c_3\, z^{\tau_c}\, \mathbf{P_1}^2(z)}{1 - a_2\, p_2^A\, v_2^A} \end{pmatrix}.
$$

Substitute 5 into 8

$$
\mathbf{P_8}(z) = b_2 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \mathbf{P_5}(z) + b_1\, c_4 \begin{pmatrix} z^{\tau_b} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z)
$$

$$\mathbf{P_8}(z) = b_2 \begin{pmatrix} z^{\tau_b} \\ 1 \end{pmatrix} \begin{pmatrix} \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_{a,p,v}}\,\mathbf{P_1}^1(z)+c_3\,\mathbf{P_1}^1(z)}{1-a_2\,p_2^A\,v_2^A\,z^{\tau_{a,p,v}}} \\ \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_c}\,\mathbf{P_1}^2(z)+c_3\,z^{\tau_c}\,\mathbf{P_1}^2(z)}{1-a_2\,p_2^A\,v_2^A} \end{pmatrix} + b_1\,c_4 \begin{pmatrix} z^{\tau_b} \\ z^{\tau_c} \end{pmatrix} \mathbf{P_1}(z)$$

Substitute 8 into 10

$$\mathbf{P_{10}}(z) = b_3\,p_1^B \begin{pmatrix} z^{\tau_b}\,\mathbf{P_5}^1(z) \\ z^{\tau_p}\ \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big) \end{pmatrix} + p_2^B \begin{pmatrix} 1 \\ z^{\tau_p} \end{pmatrix} \mathbf{P_8}(z)$$

$$\begin{aligned} \mathbf{P_{10}}(z) = {} & b_3\,p_1^B \begin{pmatrix} z^{\tau_b}\,\mathbf{P_5}^1(z) \\ z^{\tau_p}\ \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big) \end{pmatrix} \\ & + b_2\,p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_p} \end{pmatrix} \begin{pmatrix} \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_{a,p,v}}\,\mathbf{P_1}^1(z)+c_3\,\mathbf{P_1}^1(z)}{1-a_2\,p_2^A\,v_2^A\,z^{\tau_{a,p,v}}} \\ \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_c}\,\mathbf{P_1}^2(z)+c_3\,z^{\tau_c}\,\mathbf{P_1}^2(z)}{1-a_2\,p_2^A\,v_2^A} \end{pmatrix} \\ & + b_1\,c_4\,p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{c,p}} \end{pmatrix} \mathbf{P_1}(z) \end{aligned}$$

Substitute 5 into 10

We postpone this insertion. We instead introduce variable $\mathrm{mP}_{\mathbf{5}}(z)$ in order to get a concise presentation.

$$\mathbf{P_5}(z) = \begin{pmatrix} \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_{a,p,v}}\,\mathbf{P_1}^1(z)+c_3\,\mathbf{P_1}^1(z)}{1-a_2\,p_2^A\,v_2^A\,z^{\tau_{a,p,v}}} \\ \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_c}\,\mathbf{P_1}^2(z)+c_3\,z^{\tau_c}\,\mathbf{P_1}^2(z)}{1-a_2\,p_2^A\,v_2^A} \end{pmatrix}$$

$$\begin{aligned} \mathbf{P_{10}}(z) = {} & b_3\,p_1^B \begin{pmatrix} z^{\tau_b}\,\mathbf{P_5}^1(z) \\ z^{\tau_p}\,\mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix} + b_2\,p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_p} \end{pmatrix} \begin{pmatrix} \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_{a,p,v}}\,\mathbf{P_1}^1(z)+c_3\,\mathbf{P_1}^1(z)}{1-a_2\,p_2^A\,v_2^A\,z^{\tau_{a,p,v}}} \\ \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_c}\,\mathbf{P_1}^2(z)+c_3\,z^{\tau_c}\,\mathbf{P_1}^2(z)}{1-a_2\,p_2^A\,v_2^A} \end{pmatrix} \\ & + b_1\,c_4\,p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{c,p}} \end{pmatrix} \mathbf{P_1}(z) \end{aligned}$$

where

$$\mathrm{mP}_{\mathbf{5}}(z) = \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big)$$

Substitute 10 into 12

$$\mathbf{P_{12}}(z) = b_4\,d_1\;p_1^B \begin{pmatrix} z^{\tau_b}\,\mathbf{P_5}^1(z) \\ z^{\tau_{d,p}}\ \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big) \end{pmatrix} + d_2 \begin{pmatrix} 1 \\ z^{\tau_d} \end{pmatrix} \mathbf{P_{10}}(z)$$

$$\mathbf{P_{12}}(z) = b_4\,d_1\;p_1^B \begin{pmatrix} z^{\tau_b}\,\mathbf{P_5}^1(z) \\ z^{\tau_{d,p}}\ \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big) \end{pmatrix} + b_3\,d_2\,p_1^B \begin{pmatrix} z^{\tau_b}\,\mathbf{P_5}^1(z) \\ z^{\tau_{d,p}}\,\mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix}$$

$$
\begin{aligned}
&+ b_2\, d_2\, p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{d,p}} \end{pmatrix} \begin{pmatrix} \frac{(a_1\, c_1 + a_2\, c_2)\, p_1^A\, v_2^A\, z^{\tau_{a,p,v}}\, \mathbf{P_1}^1(z) + c_3\, \mathbf{P_1}^1(z)}{1 - a_2\, p_2^A\, v_2^A\, z^{\tau_{a,p,v}}} \\ \frac{(a_1\, c_1 + a_2\, c_2)\, p_1^A\, v_2^A\, z^{\tau_c}\, \mathbf{P_1}^2(z) + c_3\, z^{\tau_c}\, \mathbf{P_1}^2(z)}{1 - a_2\, p_2^A\, v_2^A} \end{pmatrix} \\
&+ b_1\, c_4\, d_2\, p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{c,d,p}} \end{pmatrix} \mathbf{P_1}(z)
\end{aligned}
$$

Substitute 5 into 12

Again, we are postponing this step. We use the variable $\mathrm{mP}_{\mathbf{5}}(z)$ in order to shorten the presentation of the equation.

$$
\begin{aligned}
\mathbf{P_{12}}(z) = {} & b_4\, d_1\; p_1^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix} + b_3\, d_2\, p_1^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix} \\
&+ b_2\, d_2\, p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{d,p}} \end{pmatrix} \begin{pmatrix} \frac{(a_1\, c_1 + a_2\, c_2)\, p_1^A\, v_2^A\, z^{\tau_{a,p,v}}\, \mathbf{P_1}^1(z) + c_3\, \mathbf{P_1}^1(z)}{1 - a_2\, p_2^A\, v_2^A\, z^{\tau_{a,p,v}}} \\ \frac{(a_1\, c_1 + a_2\, c_2)\, p_1^A\, v_2^A\, z^{\tau_c}\, \mathbf{P_1}^2(z) + c_3\, z^{\tau_c}\, \mathbf{P_1}^2(z)}{1 - a_2\, p_2^A\, v_2^A} \end{pmatrix} \\
&+ b_1\, c_4\, d_2\, p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{c,d,p}} \end{pmatrix} \mathbf{P_1}(z)
\end{aligned}
$$

Substitute 12 into 16

$$
\begin{aligned}
\mathbf{P_{12}}(z) = {} & b_4\, d_1\; p_1^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix} + b_3\, d_2\, p_1^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix} \\
&+ b_2\, d_2\, p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{d,p}} \end{pmatrix} \mathbf{P_5}(z) + b_1\, c_4\, d_2\, p_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{c,d,p}} \end{pmatrix} \mathbf{P_1}(z) \\
\mathbf{P_{16}}(z) = {} & \begin{pmatrix} \frac{b_5\, d_1\, p_1^B\, p_3^{A'}\, v_1^B\, z^{\tau_{a,b,d,2p,2v}}\, \mathrm{mP}_{\mathbf{5}}(z)}{1 - a_3\, p_3^A\, v_3^A\, z^{\tau_{a,p,v}}} \\ \frac{b_5\, d_1\, p_1^B\, p_3^{A'}\, v_1^B\, z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z)}{1 - a_3\, p_3^A\, v_3^A} \end{pmatrix} \\
&+ b_4\, d_1\; p_1^B\, v_2^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix} + b_3\, d_2\, p_1^B\, v_2^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix} \\
&+ b_2\, d_2\, p_2^B\, v_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{d,p,v}} \end{pmatrix} \mathbf{P_5}(z) + b_1\, c_4\, d_2\, p_2^B\, v_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{c,d,p,v}} \end{pmatrix} \mathbf{P_1}(z) \\
&+ b_5'\, d_1\, p_1^B\, v_1^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix}
\end{aligned}
$$

Substitute 5 into 16 (postponed)

Substitute 2 into 1

$$
\mathbf{P_2}(z) = p_1^A \begin{pmatrix} z^{\tau_p} \\ 1 \end{pmatrix} \mathbf{P_1}(z)
$$

$$\mathbf{P_1}(z) = a_1\, v_1^A \begin{pmatrix} z^{\tau_{a,v}} \\ 1 \end{pmatrix} \mathbf{P_2}(z) + \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\mathbf{P_1}(z) = a_1\, p_1^A\, v_1^A \begin{pmatrix} z^{\tau_{a,p,v}} \\ 1 \end{pmatrix} \mathbf{P_1}(z) + \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Now, we have to loop break the equation of node 1, i.e., $\not\circlearrowright 1$:

$$\mathbf{P_1}(z) = \begin{pmatrix} \frac{1}{1-a_1\, p_1^A\, v_1^A\, z^{\tau_{a,p,v}}} \\ \frac{1}{1-a_1\, p_1^A\, v_1^A} \end{pmatrix}$$

Finally, we substitute 1 into 16 and get the resulting equations for the final node 16:

$$\begin{aligned}
\mathbf{P_{16}}(z) = {} & \begin{pmatrix} \frac{b_5\, d_1\, p_1^B\, p_3^{A'}\, v_1^B\, z^{\tau_{a,b,d,2p,2v}}\, \mathrm{mP}_{\mathbf{5}}(z)}{1-a_3\, p_3^A\, v_3^A\, z^{\tau_{a,p,v}}} \\ \frac{b_5\, d_1\, p_1^B\, p_3^{A'}\, v_1^B\, z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z)}{1-a_3\, p_3^A\, v_3^A} \end{pmatrix} \\
& + b_4\, d_1\; p_1^B\, v_2^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix} + b_3\, d_2\, p_1^B\, v_2^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix} \\
& + b_2\, d_2\, p_2^B\, v_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{d,p,v}} \end{pmatrix} \mathbf{P_5}(z) \\
& + b_1\, c_4\, d_2\, p_2^B\, v_2^B \begin{pmatrix} z^{\tau_b} \\ z^{\tau_{c,d,p,v}} \end{pmatrix} \begin{pmatrix} \frac{1}{1-a_1\, p_1^A\, v_1^A\, z^{\tau_{a,p,v}}} \\ \frac{1}{1-a_1\, p_1^A\, v_1^A} \end{pmatrix} \\
& + b_5'\, d_1\, p_1^B\, v_1^B \begin{pmatrix} z^{\tau_b}\, \mathbf{P_5}^1(z) \\ z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z) \end{pmatrix}.
\end{aligned}$$

The equations per component are

$$\begin{aligned}
\mathbf{P_{16}}^1(z) = {} & \frac{b_5\, d_1\, p_1^B\, p_3^{A'}\, v_1^B\, z^{\tau_{a,b,d,2p,2v}}\, \mathrm{mP}_{\mathbf{5}}(z)}{1 - a_3\, p_3^A\, v_3^A\, z^{\tau_{a,p,v}}} \\
& + b_4\, d_1\; p_1^B\, v_2^B\, z^{\tau_b}\, \mathbf{P_5}^1(z) + b_3\, d_2\, p_1^B\, v_2^B\, z^{\tau_b}\, \mathbf{P_5}^1(z) \\
& + b_2\, d_2\, p_2^B\, v_2^B\, z^{\tau_b}\, \mathbf{P_5}^1(z) + b_1\, c_4\, d_2\, p_2^B\, v_2^B\, z^{\tau_b}\, \frac{1}{1 - a_1\, p_1^A\, v_1^A\, z^{\tau_{a,p,v}}} \\
& + b_5'\, d_1\, p_1^B\, v_1^B\, z^{\tau_b}\, \mathbf{P_5}^1(z)
\end{aligned}$$

and

$$\begin{aligned}
\mathbf{P_{16}}^2(z) = {} & \frac{b_5\, d_1\, p_1^B\, p_3^{A'}\, v_1^B\, z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z)}{1 - a_3\, p_3^A\, v_3^A} \\
& + b_4\, d_1\; p_1^B\, v_2^B\, z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z) + b_3\, d_2\, p_1^B\, v_2^B\, z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z) \\
& + b_2\, d_2\, p_2^B\, v_2^B\, z^{\tau_{d,p,v}}\, \mathbf{P_5}^2(z) + b_1\, c_4\, d_2\, p_2^B\, v_2^B\, z^{\tau_{c,d,p,v}}\, \frac{1}{1 - a_1\, p_1^A\, v_1^A} \\
& + b_5'\, d_1\, p_1^B\, v_1^B\, z^{\tau_{d,p,v}}\, \mathrm{mP}_{\mathbf{5}}(z),
\end{aligned}$$

Thread	1	2	3	4	5	6	7	8	9	10	11	12	13	Parameter τ_c	WCET PROG. ℓ^x	Time [s]
A	p	a	a	v				p	a	a	v	b		$1, 2, 3$	12	48,45
B	c	·	c		p	d	v								$\ell^1 = \ell^3 = 1, \ell^2 = 0$	
A	p	a	a	v				p	a	a	v	b		4	12	53,21
B	c	c	c	c	p	d	v								$\ell^1 = \ell^3 = 1, \ell^2 = 0$	
A	p	a	a	v	p	a	a	v	b					$5, 6, 7, 8$	11	9,11
B	c	c	c	c	c	·	·	c	p	d	v				$\ell^1 = \ell^2 = 1, \ell^3 = 0$	
A	p	a	a	v	p	a	a	v	b					9	12	8,59
B	c	c	c	c	c	c	c	c	c	p	d	v			$\ell^1 = \ell^2 = 1, \ell^3 = 0$	
A	p	a	a	v	p	a	a	v	b					10	13	8,77
B	c	c	c	c	c	c	c	c	c	c	p	d	v		$\ell^1 = \ell^2 = 1, \ell^3 = 0$	

Table 8.1: WCET for $\ell = 2$ and Multiple Values of τ_c

where

$$\mathbf{P_5}(z) = \begin{pmatrix} \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_{a,p,v}}+c_3}{(1-a_1\,p_1^A\,v_1^A\,z^{\tau_{a,p,v}})(1-a_2\,p_2^A\,v_2^A\,z^{\tau_{a,p,v}})} \\ \frac{(a_1\,c_1+a_2\,c_2)\,p_1^A\,v_2^A\,z^{\tau_c}+c_3\,z^{\tau_c}}{(1-a_1\,p_1^A\,v_1^A)(1-a_2\,p_2^A\,v_2^A)} \end{pmatrix}$$

and

$$\mathrm{mP}_{\mathbf{5}}(z) = \max\Big(\mathbf{P_5}^1(z), \mathbf{P_5}^2(z)\Big).$$

8.2.5 Maximization Process

Finally, we have to differentiate $\mathbf{P_{16}}^n(z)$ with respect to z and then set $z = 1$.

$$\text{function}^k = \frac{\mathrm{d}}{\mathrm{d}z}\,\mathbf{P_{16}}^k(z)\,\Big|_{z=1}$$

$$\text{WCET}^k = \text{Maximize}(\text{function}^k, \{\textit{constraints}\}),$$

where the set *constraints* consists of the constraints set up in Section 8.2.3. The WCET of the concurrent program consisting of two threads is defined by

$$\max\Big(\text{WCET}^1, \text{WCET}^2\Big).$$

In Table 8.1 some WCET values of the program and its components, namely the threads A and B, are depicted. The time needed for executing basic block b is referred to as τ_b. We assume that all copies of a certain basic block lead to the same execution time. Further we set $\tau_a = \tau_b = \tau_d = \tau_p = \tau_v = 1$, $\ell = 2$, and let τ_c range from 1 to 10. As described above, during the maximization process, we let Mathematica© choose the values of the variables ℓ^1, ℓ^2, ℓ^3 and all the unknown execution frequencies. We used the execution time τ_c as an input parameter to see how it affects the WCET of the program. Note that the calculated values are exact WCET values and that they do not grow linear with τ_c. In the rightmost column of Table 8.1, we present the time needed by Mathematica to calculate the time of the component leading to the WCET. Note that the maximization dominates the overall CPU time. Generating the RCPG and solving the data flow equations takes only a few milli seconds. Using a specialized non-linear solver would probably lead to better maximization times. We consider finding the best non-linear function solver as future work. Mathematica 10 was executed on a CentOS 6.0, Intel Core i7 870 CPU, 2.96GHz, 8MB cache and 4GB RAM.

CHAPTER 9

Deadlock Avoidance for Railway Systems

> *"A man who has never gone to school may steal from a freight car;*
> *but if he has a university education, he may steal the whole railroad."*
>
> – THEODORE ROOSEVELT, 26th U.S. American president, 1858-1919

In this chapter, we mainly summarize the contributions of [MBS12]. At first, we relate deadlocks in computer science to deadlocks in railway disposition systems. The major part of this chapter is devoted to how our Kronecker algebra approach can be used in order to avoid deadlocks in railway systems. The adaptations required for railway systems were done during the work for this dissertation in [MBS12]. Our approach was already extended in multiple publications [VBS12, VBS13, BSV14, SBS15, SBS16], e.g., to save energy by minimizing stop and go of trains.

Deadlock analysis for railway systems differs in several aspects from deadlock analysis in computer science. While the problem of deadlock analysis for standard computer systems is well-understood, multi-threaded embedded computer systems pose new challenges. A novel approach in this area can easily be applied to deadlock analysis in the domain of railway systems. The approach is based on Kronecker algebra. A lazy implementation of the matrix operations even allows analyzing exponentially sized systems in a very efficient manner. The running time of the algorithm does not depend on the problem size but on the size of the solution. While other approaches suffer from the fact that additional constraints make the problem and its solution harder, our approach delivers its results faster if constraints are added. In addition, our approach is complete and sound for railway systems, i.e., it generates neither false positives nor false negatives.

The *deadlock* problem and its solutions were studied in the earliest days of computer science. Although computer science borrowed several concepts and terminology from the railway domain such as *semaphores* or *tokens*, this was not the case for the deadlock problem. Deadlocks must have been impending in railway systems from the very beginning and railwaymen certainly were aware of them. In contrast to these facts, no solutions were commonly known in the midst of the 20th century. So computer scientists were the first to study the problem intensively and to deliver adequate solutions.

In this chapter, we again use Stallings' [Sta11] definition of deadlocks quoted in Chapter 5 as Definition 7.

For a deadlock to occur, four necessary conditions were found [JES71]:

1. Mutual exclusion: a resource that cannot be used by more than one process at a time.
2. Hold and Wait: processes already holding resources may request new resources held by other processes.
3. No preemption: No resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process.
4. Circular wait: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds.

If one of these conditions cannot be met, a deadlock cannot occur. Clearly all four conditions apply to railway systems.

To handle deadlock problems one distinguishes between *deadlock prevention*, *deadlock avoidance*, and *deadlock detection*.

- *Deadlock prevention* tries to remove one of the four conditions above in order to make it impossible for a deadlock to occur.
- *Deadlock avoidance* utilizes certain information about processes known in advance to decide whether or not to allocate resources to processes. Examples for deadlock avoidance algorithms include the Banker's algorithm [Dijnda], Wait/Die and Wound/Wait algorithms to name a few.
- When a deadlock is detected at runtime, a process is terminated and restarted later on.

Deadlocks in railway systems attracted attention when railway simulation systems became available. Since simulations were supposed to run without human interaction, deadlocks became an obstacle. Obviously autonomous dispatching systems also suffer from deadlock problems if no adequate solutions can be found.

The outline of this chapter is as follows. In Section 9.1, we contrast deadlocks in computer systems with deadlocks in railway systems. Section 9.2 gives our standard railway system

Computer systems	Railway systems
In standard computer applications all resources are held until the process terminates (not true for embedded systems where processes may not terminate at all).	A train seizes a track section shortly before it enters it, and holds it until it leaves it (which may be long before it reaches its destination).
Program code consists of straight line code, if, and loop statements.	Routes compare to straight line code; there may be alternative routes which compare to if statements; loops may be useful for model railroad applications.
If a multi-threaded computer program contains a deadlock, the program is erroneous.	If in a railway system it is possible to bypass a deadlock situation, then the system is correct. Deadlocks are always possible, but can usually be bypassed.
A computer program is “correct”, if we can prove it deadlock free.	A set of trains is schedulable if all possible deadlocks can be bypassed.
If a deadlock is detected, it is viable to terminate a program and restart it later on (not true for safety related embedded systems).	Trains cannot simply be taken out of the overall system.

Table 9.1: Differences between the Deadlock Problem in Computer and Railway Systems

model. In Section 9.3, we present a simple example. Section 9.4 shows how our resulting graphs can be employed for finding optimal solutions. In Section 9.5, we extend our standard model in order to solve several important practical problems. We relate our approach to existing work in Section 10.5.

9.1 Deadlocks in Computers vs. Deadlocks in Railway Systems

In railway systems trains, routes, and track sections correspond to processes, program code, and shared resources in computer systems, respectively.

Table 9.1 shows major differences between the deadlock problem in computer and railway systems.

From the table above it becomes obvious that deadlock analysis methods and algorithms well-suited for standard computer systems cannot simply be transferred to railway systems. However, safety related embedded systems show more resemblance to railway system. This is the reason why we try to apply the approach, we have developed for embedded systems [MB11] to railway systems. We will show in this chapter that this approach is in fact well-suited for railway systems. In the following, we will argue that only deadlock avoidance makes sense in the domain of railway systems; deadlock prevention and deadlock detection cannot be applied in this domain. *Deadlock prevention* is done by ensuring that one of the four necessary conditions stated above is not fulfilled. It is easy to see that this cannot be done for railway systems:

1. Mutual exclusion: Track sections have to be seized by trains exclusively.
2. Hold and Wait: It is typical that a train occupies two or more track sections at the same time.
3. No preemption: A track section cannot be removed from a train "holding" it.
4. Circular wait: It is impossible to impose a strict order on the track sections without posing the problem that some trains would seize all track sections of their route from start to destination before they start running.

Deadlock detection is of no interest for railway systems because "terminating a train" is no option.

The only remaining option is *deadlock avoidance* which will be discussed in the rest of this paper.

Before we give an introduction to our railway system model, we state a few additional differences between computer and railway systems:

- Computer systems usually consist of a number of processes (threads) and a number of shared resources. In most cases the number of resources is much smaller than the number of processes. In railway systems the number of track sections usually is much greater than the number of trains.
- Besides the track sections being shared resources in railway systems there are additional constraints that have to be met. For example train connections and overtaking have to be taken care of, tardy trains may cause penalty, ...

While previous approaches perform bad if additional constraints are added to the problem domain, our approach delivers its results faster if constraints are added. This makes it a very promising approach. Details will be given in the following sections.

9.2 Railway System Model

For an introductory example, we come back to the Example 6 of Chapter 3. We use the graph corresponding to matrix C which is depicted in Figure 3.4a, whereas the graph of matrix D is shown in Figure 3.4b. Now interpret C and D as being trains and a, b, c, and d as being actions of the trains with the following meaning: a denotes train C enters track section T_a, b means train C has left track section T_a, c denotes train D enters track section T_b, and d means train D has left track section T_b. All possible temporal interleavings of these actions are shown in Table 3.4c. In Figure 3.4d the graph represented by the adjacency matrix $C \oplus D$ is depicted. It is easy to see that all possible temporal interleavings are generated correctly.

Now assume that T_a and T_b denote the same track section. It is clear that in this case the temporal interleavings of Table 3.4c are no more valid. The trains have to *synchronize* in

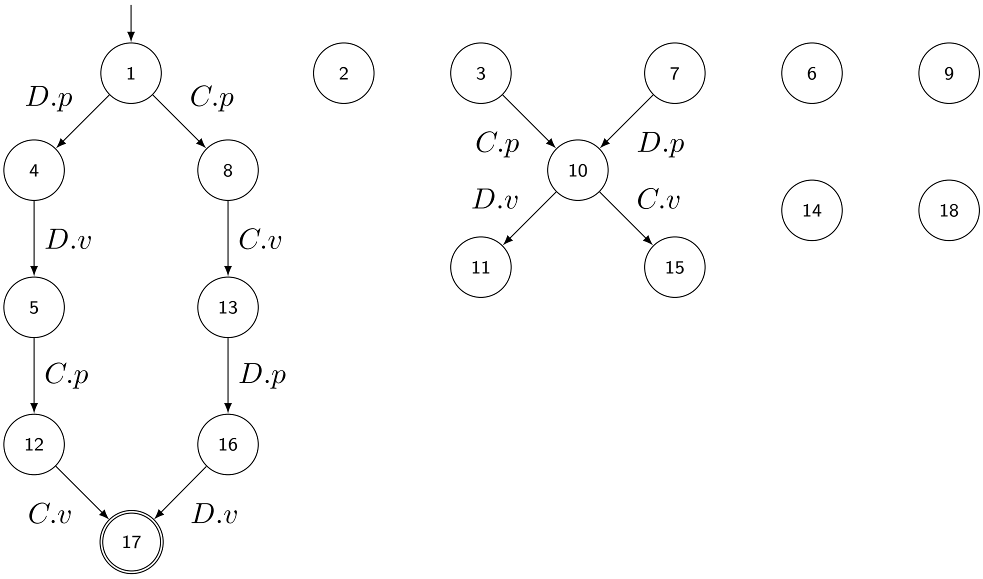


Figure 9.1: Graph $(C \oplus D) \otimes S$

order to perform their actions correctly. This can be modeled by Kronecker product and an additional matrix of the form

$$S = \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix},$$

where p denotes the action "Enter the track section" and v means "Train has left the track section". The correct system behavior can be described by the matrix $R = (C \oplus D) \otimes S$. The corresponding graph is shown in Figure 9.1. It is interesting that the graph became decomposed into seven parts (sub-graphs). Clearly only the part reachable from the entry node is responsible for the system behavior, the other six parts can safely be ignored. Thus, we concentrate on the sub-graph with entry node 1. Studying this sub-graph, we see that now the trains enter the track section one after the other. Note that the two paths in the subgraph correctly mirror the two cases where C enters the track section before D and vice versa. A proof that Kronecker product models synchronization correctly can be found in [MB11]. It is also worth noting, that parts unreachable from the entry node always occur if synchronization via Kronecker product takes place. This observation is the major reason for the lazy implementation described in Section 4.7.

We model a general railway system S by a set of track sections $T = T_i | 1 \leq i \leq r$. Each track section T_i is modeled by matrix

$$T_i = \begin{pmatrix} 0 & p_i \\ v_i & 0 \end{pmatrix}.$$

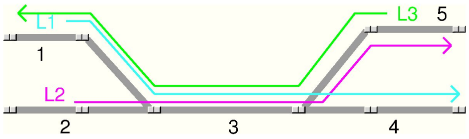


Figure 9.2: A Simple Example

In addition, a railway system consists of a set of trains $L = L_j | 1 \leq j \leq t$. The route R_j of train L_j is a sequence of track sections $T_{l_1}, \dots, T_{l_s}$ for $1 \leq l_n \leq r$ and $1 \leq n \leq s$. Each route is modeled by a $2s \times 2s$-matrix. In particular, $R_j = (r_{i,k})$. The set of routes is denoted by $R = R_j | 1 \leq j \leq t$.

The behavior of railway system $S\langle T, L, R\rangle$ is modeled by

$$S = \left(\bigoplus_{j=1}^{t} R_j\right) \bigotimes \left(\bigoplus_{i=1}^{r} T_i\right),$$

where during the evaluation of the Kronecker product, we replace each $p_i \cdot p_i$ by p_i and each $v_i \cdot v_i$ by v_i. This is similar to the replacements done in Section 4.1. In addition to that, we replace all other combinations, e.g., $p_i \cdot v_i$ and $p_i \cdot p_j$, where $i \neq j$ by 0.

Similar to Section 4.6, we get rid of these rather unaesthetic replacements by using the selective Kronecker product. By using Definition 5 we write

$$S = \left(\bigoplus_{j=1}^{t} R_j\right) \oslash_{\mathcal{L}_\mathrm{S}} \left(\bigoplus_{i=1}^{r} T_i\right).$$

The different paths in the graph corresponding to matrix S mirror *all possible behaviors* of the railway system in terms of temporal interleavings of the actions of trains, namely entering and leaving track sections. From the discussion above and from Chapter 5 it is clear that deadlocks appear as purely structural properties of the underlying graph. Deadlocks manifest themselves as non-final nodes[1] with no successors.

9.3 A Simple Example

In this section, we give a small example on how deadlocks can be avoided by our Kronecker algebra based approach. In Figure 9.2 a typical scenario is shown that may lead to a deadlock. The routes of the three involved trains are given in Table 9.2. If train L_3 enters track section T_3 before the other trains move, a deadlock is unavoidable.

The matrices for the three routes are setup as follows:

---

[1] A final node corresponds to the destination of a route. A final node of a railway system corresponds to the state, where all trains have reached their destinations.

Trains	Routes
L_1	p_3, v_1, p_4, v_3, v_4
L_2	p_3, v_2, p_5, v_3, v_5
L_3	p_3, v_5, p_1, v_3, v_1

Table 9.2: Routes of Trains L_1, L_2, and L_3

$$R_1 = \begin{pmatrix} 0 & p_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & v_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & v_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & v_4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \ R_2 = \begin{pmatrix} 0 & p_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & v_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_5 & 0 & 0 \\ 0 & 0 & 0 & 0 & v_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & v_5 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and}$$

$$R_3 = \begin{pmatrix} 0 & p_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & v_5 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & v_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & v_1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The matrices for the five track sections have the form

$$T_i = \begin{pmatrix} 0 & p_i \\ v_i & 0 \end{pmatrix} \text{ for } 1 \leq i \leq 5,$$

thus are semaphores like the one depicted in Figure 2.3a with the difference that for $i \in \{1, 2, 5\}$ the semaphore's node 2 is its entry node. This reflects the fact that in the beginning of this example the corresponding tracks are occupied. The order of the resulting matrix can be computed by multiplying the order of the involved matrices, i.e., the order of the matrices R_j and the order of the matrices T_i. In our example the resulting matrix will have order $6912 = 6 \cdot 6 \cdot 6 \cdot 2^5$. Due to synchronization only a small part of the resulting graph is reachable from the entry node. Only the reachable part – consisting of 42 nodes – is depicted in Figure 9.3a. To increase readability our implementation distinguishes between the following node types:

- Diamond nodes denote deadlocks or nodes from which only deadlocks can be reached.
- Solid nodes denote *safe states*. A state is *safe* if all trains can perform their actions without having to take into account the moves of the other trains in the system, provided that the track section which they are to enter is not occupied by another train.[2]

---

[2] If a track section is occupied by another train, the movement of the train wanting to enter may be delayed, but no deadlock can occur.

	Algorithm says system deadlocks	Algorithm says system does not deadlock
System deadlocks	okay	False negative
System does not deadlock	False positive	okay

Table 9.3: Four Possible Outcomes of an Analyzing Algorithm

- From dotted nodes both, diamond and solid nodes, can be reached.

If only deadlock avoidance is of interest, graphs like that in Figure 9.3a can be scaled down. In fact, diamond nodes are of no interest because we want to avoid deadlocks. From the set of solid nodes only those are of interest, which have dotted predecessors only. In addition, we do not have to show the node labels, i.e., the matrix line numbers. If we eliminate the non-interesting nodes from Figure 9.3a, we obtain the graph in Figure 9.3b. We call such graphs NDLS-graphs because they contain **n**o **d**eadlock nodes and only a **l**imited number of **s**afe nodes. From the two solid states the trains can proceed with their movements in any order and no deadlock will happen. From the remaining possibilities one path can be freely chosen. The trains will reach the final node in each of the possible cases. An implementation may choose an energy saving and/or travelling time saving path [VBS12].

In order to avoid deadlocks the minimal number of actions can be obtained by selecting a path from the entry node to a solid node which is situated nearest to the entry node. This, however, may not be the best alternative. In the next section, we elaborate on the problem of finding the “cheapest” path.

In order to evaluate the quality of algorithms or tools it is convenient to introduce *false positives* and *false negatives*. Assume we have an algorithm at hand that examines a system to find out whether the system deadlocks or not. Then the four possible outcomes depicted in Figure 9.3 exist.

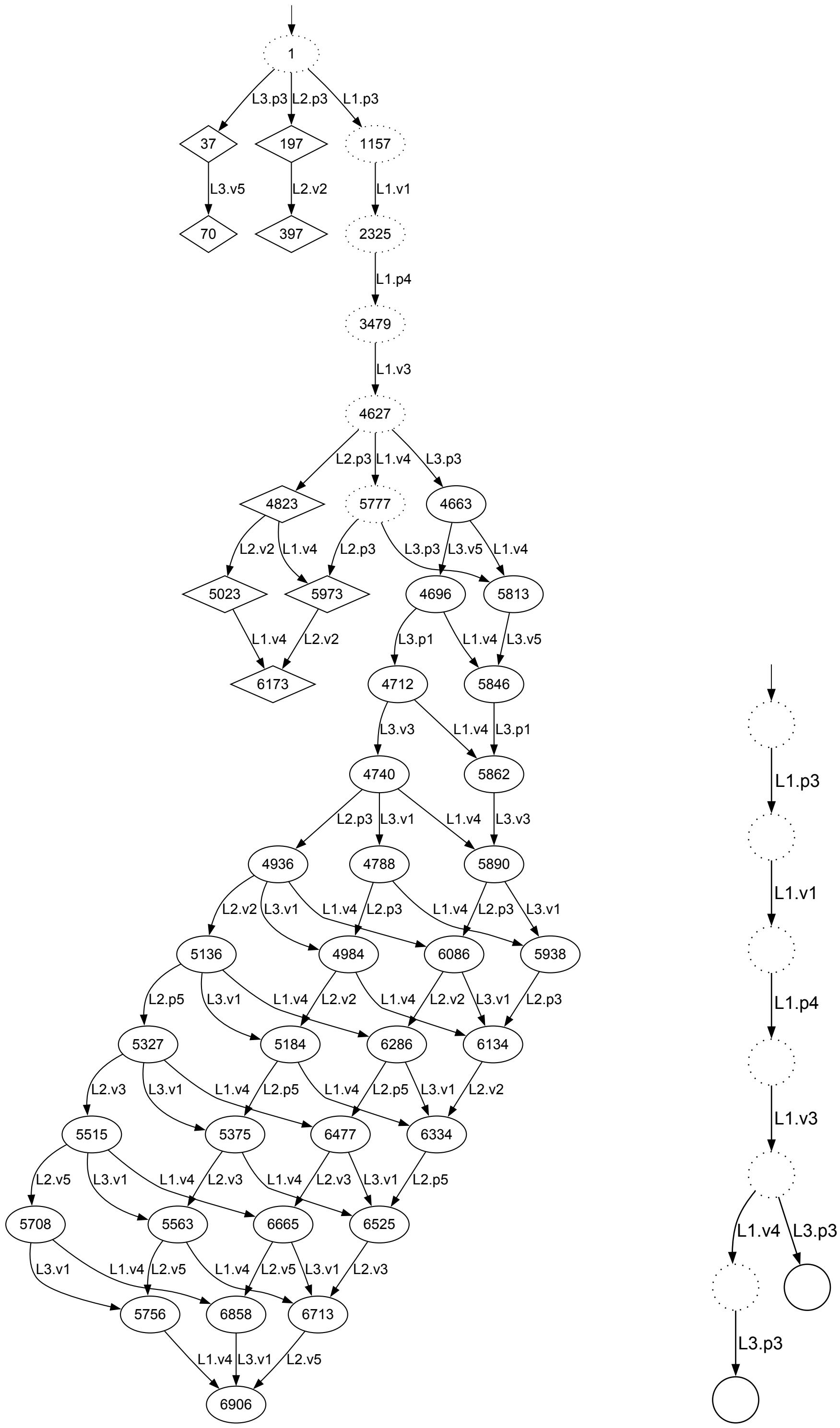
It is worth noting that our approach does neither deliver false positives nor false negatives[3]. Thus it is *complete* and *sound*. There is only one exception: In case of alternative routes (cf. Section 9.5) deadlocks in all possible paths will be reported although only one path can be chosen by the train. This situation may somehow be classified as reporting false positives.

For computer systems our approach can be modified such that it stops as soon as the first deadlock is found[4]. For railway systems the algorithm may be modified to stop as soon as a safe state is found.

---

[3] This is true for railway systems. It is not true for computer systems because computer programs may contain dead code, i.e., code that is never executed. Our approach will report a deadlock even if it is contained in dead code. Thus it will deliver false positives for computer systems but no false negatives.

[4] This makes sense because a computer system is considered erroneous if at least one deadlock is present.



(a) Graph Containing Deadlock Nodes (b) NDLS graph

Figure 9.3: Possible Train Movements

9.4 Avoiding Deadlocks under Additional Constraints

Our approach produces a graph which gives all temporal interleavings of train moves. A short reflection shows that not all of these interleavings may result in an "optimal" behavior. However, a more sophisticated analysis may choose to assign weights to the edges of the graph. These weights can be chosen such that the term "optimal" from above makes sense. Weights may include temporal constraints originating from physical properties of the trains such as accelerating and braking characteristics depending, e.g., on the weight of the train, or penalties due to tardiness.

In any case, well-known algorithms such as those of [Dij59] or [FT84] can be applied to graphs with nonnegative weights to find the optimal (shortest) path. Dijkstra's algorithm performs in time quadratic in the number of nodes of the graph. The Fredman and Tarjan version uses time proportional to the sum of the number of edges and the number of nodes times the logarithm of the number of nodes.

If only deadlock avoidance is of concern, the optimizations given in the previous section apply.

9.5 Extensions of the Standard Railway Model

In this section, we discuss how our standard model defined in Section 4 can be extended in order to solve certain important problems.

The first problem we consider occurs if the length of a train is greater than the length of a track section. Assume a route of a train containing three consecutive track sections t_1, t_2, and t_3, where t_2 is shorter than the length of the train. In addition assume that the train occupies track section t_1. If t_2 would be long enough, the train's moves would be $p_2, v_1, p_3, v_2, \ldots$ In case of the short track section the train's moves are $p_2, p_3, v_1, v_2, \ldots$

It is clear that this approach can easily be extended if more than three track sections are used by a train simultaneously.

Furthermore note that double slip switches can be seen as two normal switches with a zero length track section in between. Thus routes containing double slip switches represent typical examples of the method above.

Sometimes it is useful to model a railway system from a more abstract view. For example one abstracts away from the track details of a station and instead would like to use only the fact that the station has a capacity of holding c trains at the same time, where each train is able to enter and leave the station independently from the other trains in the station. Such a station can be modeled by a $(c+1) \times (c+1)$ matrix of the following form

$$\begin{pmatrix} 0 & p & 0 & \cdots & 0 \\ v & 0 & p & \cdots & 0 \\ 0 & v & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & v & 0 & p \\ 0 & 0 & \cdots & v & 0 \end{pmatrix}.$$

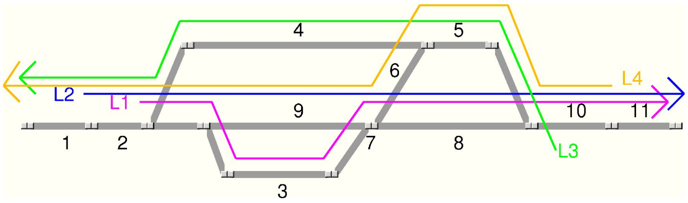


Figure 9.4: A More Elaborate Example

Thus our approach is able to handle abstract views as well. One can envisage even more abstract levels such as abstracting whole railway lines to their capacity. This allows to model nets of railway lines.

Sometimes it is useful when a train is allowed to take alternative routes. This gives more freedom in bypassing a potential deadlock situation. Since our approach was developed for handling embedded computer systems, it is able to handle alternatives which are present as if-statements in computer programs. Generating matrices that model alternative routes is very easy. A line in the matrix representing a fork state of a route has two successors, i.e., two non-zero entries. Later on the different alternatives can be joined and a single (straightforward) route continues. Clearly this works even for more than two alternative routes.

As already noted our approach was developed for embedded computer systems. Thus it also supports loop statements. Matrices modeling loops can be employed to model trains running in loops. This might be of interest for model railroaders.

By adding further matrices, we can model synchronization of trains which can be used to ensure connections and overtaking of trains. For example assume that we want to ensure that train A shall not leave track section t_A before train B leaves track section t_B. We introduce two "artificial track sections" d and e for synchronization with the usual operations p_d, v_d, p_e, and v_e. In addition, we insert immediately in front of the moves v_A and v_B the actions v_d, p_e, and p_d, v_e, respectively. Thus the two trains are synchronized and we are able to model connections correctly. Instead of using additional semaphores for synchronizing multiple trains, we may use our barrier synchronization primitive introduced in Chapter 7.

We conclude this section with a more elaborate example containing some of the extensions described above. In particular it contains a double slip switch (track section 7) and it shows how overtaking can be modeled.

The system is depicted in Figure 9.4. The routes are defined as follows:

$$\begin{aligned}
R_1 &= p_3, v_2, p_7, p_8, v_3, v_7, p_{10}, v_8, p_{11}, v_{10}, v_{11},\\
R_2 &= p_2, v_1, p_9, v_2, p_7, p_8, v_9, v_7, p_{10}, v_8, p_{11}, v_{10}, v_{11},\\
R_3 &= p_5, v_{10}, p_4, v_5, p_2, v_4, p_1, v_2, v_1, \text{ and}\\
R_4 &= p_{10}, v_{11}, p_5, v_{10}, p_6, v_5, p_7, p_9, v_6, v_7, p_2, v_9, p_1, v_2, v_1.
\end{aligned}$$

Note that track section 7 has zero length; it originates from the double-slip switch located between track sections 3, 9, 6, and 8. Kronecker algebra calculations produce a graph with 55,050,240 potential nodes. After the reductions presented in Section 5 only 992 nodes are left. Now we introduce additional constraints. We assume that train L_2 overtakes L_1 and L_4 overtakes L_3 within the station. We need one additional "artificial track section" for each train pair. The corresponding operations are denoted by p_{12}, v_{12}, p_{13}, and v_{13}. Now the routes read as follows:

$$\begin{aligned}
R_1 &= p_3, v_2, v_{12}, p_7, p_8, v_3, v_7, p_{10}, v_8, p_{11}, v_{10}, v_{11},\\
R_2 &= p_2, v_1, p_9, v_2, p_7, p_8, v_9, v_7, p_{10}, v_8, p_{12}, p_{11}, v_{10}, v_{11},\\
R_3 &= p_5, v_{10}, p_4, v_5, v_{13}, p_2, v_4, p_1, v_2, v_1, \text{ and}\\
R_4 &= p_{10}, v_{11}, p_5, v_{10}, p_6, v_5, p_7, p_9, v_6, v_7, p_2, v_9, p_{13}, p_1, v_2, v_1.
\end{aligned}$$

Kronecker algebra produces 298,721,280 potential nodes. After the usual reductions, however, only 144 nodes are left (cf. Figure 9.5). This means that the number of potential nodes has grown by a factor of five, but the size of the solution has dropped by factor of approximately 6.8. The resulting NDLS-graph is depicted in Figure 9.5. This example shows that if constraints are added to a railway system, our approach delivers its solution faster. This is rather atypical for recent approaches.

9.6 Concluding Remarks

In this chapter, we have presented a Kronecker algebra based approach for deadlock analysis in railway systems. It can solve a lot of important practical problems concerning train scheduling in railway disposition systems. By employing a lazy implementation for the Kronecker matrix operations the solution is computed very efficiently.

By assigning weights to the edges of the resulting graph various optimizations can be performed. This includes optimizations due to physical properties of the trains such as accelerating and braking characteristics depending, e.g., on the weight of the train, or penalties due to tardiness.

The running time of the algorithm does not depend on the problem size but on the size of the solution. While other approaches suffer from the fact that additional constraints make the problem and its solution harder, our approach delivers its results faster if constraints are added.

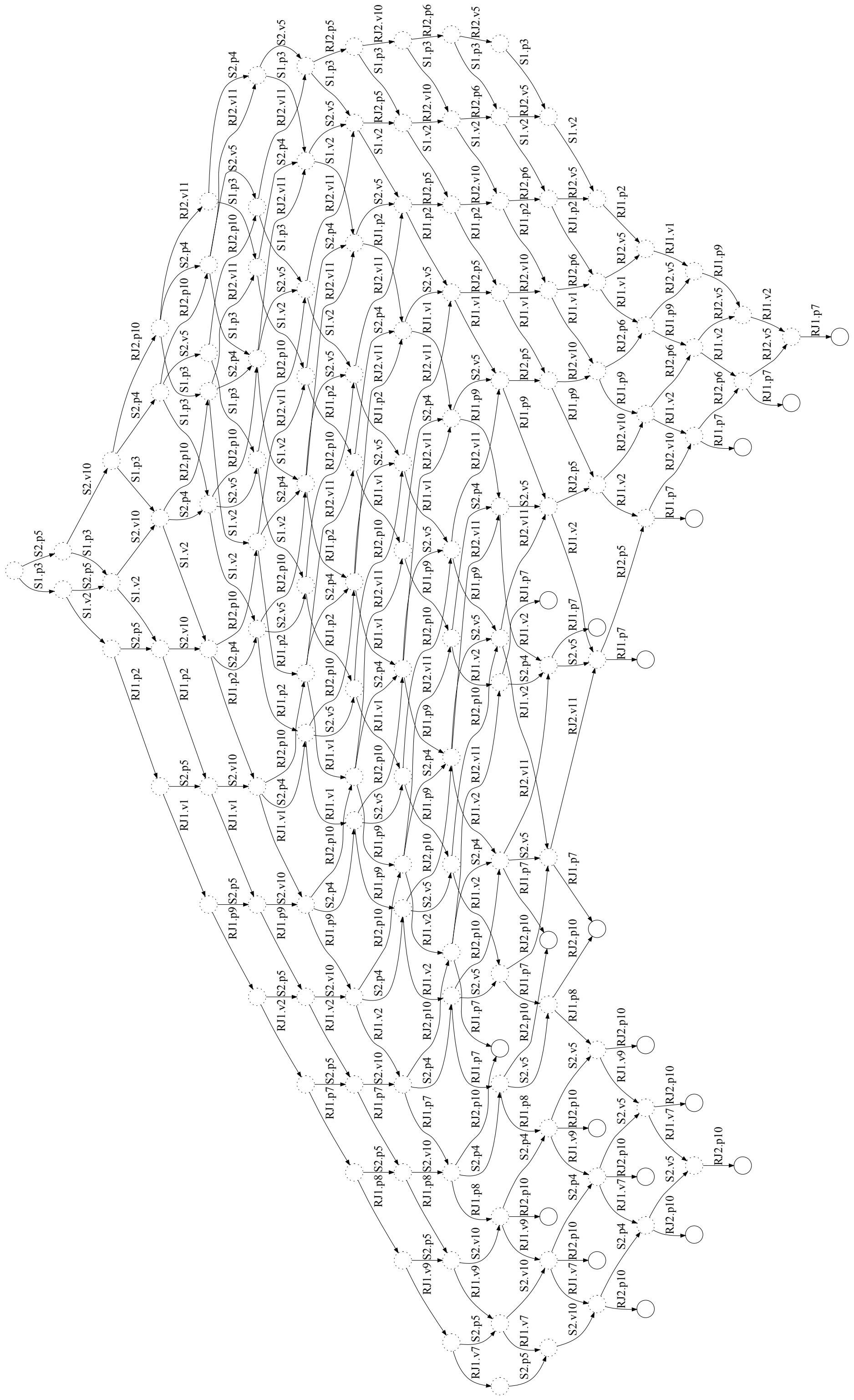


Figure 9.5: Resulting NDLS Graph

CHAPTER 10

Related Work

> *"It might be well for all of us to remember that, while differing widely in the various little bits we know, in our infinite ignorance we are all equal."*
>
> – SIR KARL RAIMUND POPPER, Austrian-British philosopher, 1902-1994
> Conjectures and Refutations - The Growth of Scientific Knowledge, 1963

In this chapter, we relate the work done for this dissertation to existing approaches.

10.1 Kronecker Algebra

Although not studying similar problems, we consider the following work in terms of how we generate the graph model for concurrent programs as related work.

In this area, the closest work to ours was probably done by Buchholz and Kemper [BK02]. Buchholz and Kemper worked on generating reachability sets in composed automata. The plain Kronecker product is used in order to describe networks of synchronized automata. It differs from our work as follows. Boolean matrices (one matrix for each label) are used, whereas the entries of our matrices are labels from a semiring. We use similar definitions of Kronecker algebra and extend it such that we are able to use semaphores and barriers for synchronization. Our approach can be used e.g. for analyzing the timing behavior of concurrent programs, detecting deadlocks in multi-threaded software, and avoiding deadlocks in railway systems. In addition, we propose lazy calculation of matrix entries to optimize running time. Different techniques, including data flow analysis (e.g. [RP86, RP88, SGL98, KU76]) and symbolic analysis (e.g. [BSB12]), can be applied to the generated CPGs.

Although not closely related, we recognize the work done in the field of *stochastic automata networks* (SAN) which is based on the work of Plateau [Pla85] and in the field of *generalized stochastic petri nets* (GSPN) (e.g. [CM99]) as related work. Compared to ours these fields are completely different. Nevertheless, basic operators are shared and some properties influenced this doctoral thesis.

10.2 Data Structures Modeling Concurrent Systems

Petri nets [Pet62, Rei85] are widely used to model concurrent systems. They model a rather monolithic global system view. The focus is on the whole system instead of on the individual threads. Petri nets have to be constructed from scratch and cannot be generated from source code efficiently. They use markings to indicate a certain state. The set of all markings defines all possible states. In contrast our Kronecker algebra approach enables us to model the more or less independent components, namely threads and synchronization primitives, of a concurrent system in form of CFGs. CFGs are frequently used data structures anyways and Kronecker algebra (which purely relies on simple algebraic definitions) is used to construct a global system view in a fully automated way. The resulting concurrent program graphs (CPGs) represents the corresponding multi-threaded program such that all interleavings and synchronization are taken into account.

10.3 Static Analysis of Barriers

Barriers can be employed in various parallel programming models, such as single program multiple data (SPMD, e.g., CUDA [NVI16], OpenCL [Ope16], and OpenMP [Ope13]), fork/join, and shared memory interleaving semantics based models. In the following, we compare some of the work done in these areas to our work.

In [AG98] the concept of structural correctness is defined to ensure that all threads execute the same number of barriers. Static analysis is used to determine if or not a program is structural correct. Combining this approach with ours, a large number of programs can be automatically verified.

In [KY06, ZDG08] the focus is on determining which portions of the program may execute in parallel. However, such analyses do not verify the correctness of barrier synchronization. Our approach delivers whether or not a statement is concurrently executed via the structure of the CPG.

Paper [ZD07] generalizes [AG98] by introducing barrier matching which allows to prove a larger set of barrier scenarios correct. Our approach combined with symbolic analysis [BSB12], however, is capable to verify an even larger set of such scenarios.

In [LCT13] a bounded permission system and a concurrent separation logic are presented for verifying fork/join programs with static and dynamic barriers. Since Ada supports static barriers only, our approach can only be compared to that part of [LCT13]. We are sure, that our approach via symbolic analysis can verify the same set of scenarios.

In [MB12a] several barrier scenarios are verified. It will be interesting future work to see whether these proofs have counterparts in our graph-based model.

10.4 Worst-Case Execution Time Analysis

A good overview for WCET techniques for parallel applications can be found in [NYP15a]. It presents the different approaches for determining estimations and upper bounds for WCET with all their advantages and disadvantages. Because we proposed a static technique, our approach has the advantages and disadvantages compared to measurement based, hybrid, and probabilistic approaches as mentioned in that paper.

Our approach is the first one capable of handling parallel and concurrent software. There exist several approaches for parallel systems which we will discuss in the following.

The P-SOCRATES project [PQB^{+}14, FNNP16] establishes a response time analysis technique for sporadic directed acyclic graph (DAG) tasks under partitioned scheduling. It aims to include interference analysis in a holistic and integrated perspective, i.e., allowing multiple sources of interference (e.g cache, memory bus and network) in an analysis and study how these analyses work together. The used hardware and software stack are described in [NYP^{+}15b]. Each task is implemented using OpenMP [Ope13] which uses compiler directives in order to define task regions. These tasks are mapped to clusters and then to OS threads by the OpenMP runtime environment. The OS threads are scheduled dynamically. The adopted WCET methodology is a purely measurement-based approach and thus has the drawback that for measurements hardware needs to be available and it might underestimate the WCET [NYP15a]. On the other hand it has, compared to our static approach, the advantage of higher portability and, until now, we do not take scheduling into account. In contrast, we have parallelism and concurrency in our graph model. Our approach can be used for a purely static WCET technique (where the hardware needs to be modeled) and for a hybrid WCET method, where the CPG represents the control flow and a measurement-based analysis delivers the execution times.

In [PBP13] an IPET based approach is presented. Communication between code regions in form of message passing is detected via source code annotations specifying the recipient and the latency of the communication. For each communication between code regions, the corresponding CFGs are connected via an additional edge. Hence, the data structure are CFGs connected via communication edges. This is not enough for programs containing recurring communication between threads. In contrast to that, our approach generates a new data structure (RCPG) out of the input CFGs in a fully automated way. The RCPG incorporates thread synchronization of the multi-threaded program and thus contains only the reachable interleavings. Our approach is not limited to one single synchronization mechanism, it can be used to model e.g. semaphores or locks. In addition, RCPGs play a similar role for multi-threaded programs as CFGs do for sequential programs and can be used for further analysis purposes. The hardware analysis on basic block level of [PBP13] can be applied to our approach too.

As our approach for loops, the work presented in [ORS13] also relies on annotations. The worst case stalling time is estimated for each synchronization operation. This time is added to the time of the corresponding basic block. Our approach exactly detects the points, where stalling will occur, i.e., at the *vp*-synchronizing nodes, and establishes dataflow equations to handle that problem in an explicit and natural way. It calculates the stalling times which need not be given by the user. At these points (e.g. critical section protected via a semaphore), we can also incorporate hardware penalties for all kinds of external communication and optimizations for e.g. shared data caches. Our approach allows synchronization within loops in a concurrent program, whereas [ORS13] does not support that. This is the main reason why [ORS13] can use an ILP approach.

Similar to [ORS13], we use a rather abstract view of synchronization primitives and assume timing predictability on the hardware level as discussed e.g. in [GKUR12].

Current steps towards multi-core analysis including hardware modelling try to restrict interleavings and use a rigorous bus protocol (e.g. TDMA) that increases the predictability [NPB$^+$14]. A worst-case resource usage bound to compute the WCET overlap is used. Hence, it finds a WCET upper bound only, while our approach determines the exact WCET that includes stalling times.

In [GELP10], a method based on model checking of multi-core applications modeled as timed automata is investigated. The tool box UPPAAL is used and synchronization is modeled by using spinlock-like primitives. Since the model-checking attempt in [GELP10] has scalability problems the authors switched to the abstract execution approach of [GGL14]. It allows to calculate safe approximations of the WCET of programs using threads, shared memory and locks. Locks are modeled in a spinlock-like fashion. The problem of nontermination is inherent in abstract execution. Thus, it is not guaranteed in [GGL14] that the algorithm will terminate. This issue is partly solved by setting timeouts.

10.5 Railway Systems

In [Cui10] Banker's algorithm [Dijnda] is modified such that it can be employed for deadlock analysis in railway systems. Since Banker's algorithm has been designed for standard computer systems it is not well-suited for railway systems. For example it may prohibit allocating a resource (track section) although a potential deadlock can be bypassed. In contrast to our approach, both, namely track sections and switches, have to be modeled as resources in [Cui10].

An operations research approach is proposed in [Mar95] to do deadlock analysis in railroad systems.

In [Pac93] Movement Consequence Analysis (MCA) and Dynamic Route Reservation (DRR) are introduced for deadlock analysis. Both are rule-based methods for which correctness cannot be proved. It delivers false positives.

An algorithm which can handle only simple railway networks is given in [PT83]. In addition it produces false positives. The same restrictions apply to the approach presented in [MPWH03].

Deadlock free algorithms are presented in [LDL04]. All of them may generate false positives.

A colored Petri net model is used in [FGS03] to describe a railway network system and to derive the traffic controller. Safeness and deadlock freedom are guaranteed. A deadlock prevention strategy is defined and expressed by a set of linear inequality constraints. It is shown how collision and deadlock prevention constraints can be expressed as colored generalized mutual exclusion constraints and that the controller can be implemented by a set of monitor places. It is however not clear how this approach can be extended to find optimal solutions such as our approach suggests (cf. Section 9.4).

In [Ž08] a colored Petri net model of a simple railway station operation is constructed and Banker's algorithm is employed for deadlock avoidance. It turned out that Banker's algorithm is not able to handle a large number of processes (trains) and resources (track sections).

Our Kronecker algebra model for the railway domain was already extended. The adaptations required for railway systems were establish during the work for this doctoral thesis. This approach was published at first in [MBS12]. It was extended by several publications (e.g. [VBS12, VBS13, VBS14, SVB14, BSV14, Vol14, SBS15, SBS16]). These publications are working on travel time analysis and global optimization concerning energy saving by reducing stop and go of multiple trains.

CHAPTER 11

Conclusion

"Nach manchen mißglückten Versuchen, meine Ergebnisse zu einem solchen Ganzen zusammenzuschweißen, sah ich ein, daß mir dies nie gelingen würde. Daß das Beste, was ich schreiben konnte, immer nur philosophische Bemerkungen bleiben würden; daß meine Gedanken bald erlahmten, wenn ich versuchte, sie, gegen ihre natürliche Richtung, in einer Richtung weiterzuzwingen."

– LUDWIG WITTGENSTEIN, Austrian-British philosopher, 1889-1951
Philosophische Untersuchungen, 1953 (published posthumously)

In this chapter, we draw our conclusion for this dissertation. In Section 11.1 we summarize important aspects of this doctoral thesis. Finally, in Section 11.2 an outlook suggests directions for future work.

11.1 Summary

We established a framework for analysis of shared memory concurrent programs. In its heart, we introduced a generic graph model representing multi-threaded programs. Thread synchronization is modeled by synchronization primitives like semaphores and barriers. Threads and synchronization primitives are modeled as matrices. The matrices are manipulated by Kronecker algebra, which we extended in the course of this dissertation. By using this matrix calculus, we are immediately able to support conditionals, loops, and synchronization between threads. The resulting matrix represents the analyzed multi-threaded program. The underlying graph – which we call *Concurrent Program Graph* (CPG) – plays a similar role for concurrent programs as control flow graphs do for sequential programs. Thus a suitable graph model for analysis of multi-threaded software has been set up which forms a basis for studying various properties (e.g. deadlock and execution time) of programs.

In this dissertation, we extended the Kronecker algebra and proved some properties of this matrix calculus. We used CPGs for detecting deadlocks, analysing programs using barriers, and execution time analysis of concurrent programs. In addition, we adopted our

Kronecker algebra based approach in the railway domain. In the following, we summarize these contributions.

Kronecker Algebra. For the Kronecker sum, we introduced and proved a new rule which we call *Mixed Sum Rule*. Also [BK02] implicitly rely on this rule. We proved the associativity of the Kronecker sum of matrices, too. In addition, we introduced the *selective Kronecker product*. The usefulness of our approach has been proved by a lazy implementation of our extended Kronecker algebra. The implementation is very memory efficient and has been parallelized to exploit modern many-core hardware architectures. Because it focuses on the matrix operations, our implementation can be used for program analysis of concurrent systems and in the railway domain.

Properties of CPGs. In the following, we list some properties of CPGs discussed in this doctoral thesis.

- We proved that, in general, CPGs can be represented by sparse matrices. Hence the number of entries in the matrices is linear in their number of lines. Thus memory-conserving data structures such as adjacency lists are a natural fit.
- Synchronization leads to unreachable parts.
- By choosing a lazy implementation, we are able to reduce, compared to the full CPG, the computation time effort to the reachable CPG (RCPG).

Deadlocks. We proposed an approach for statically detecting deadlocks. The method is complete because when a deadlock exists, then it will find it. On the other hand, because it's a static method, it is not sound, i.e., it may find false positives. This drawback can be alleviated by applying symbolic evaluation [BSB12].

Barriers. We have shown how Kronecker algebra can be employed for static analysis of concurrent programs (e.g. written in Ada or Java) that use reusable static barriers for synchronization. The implementation of our novel barrier synchronization primitive has to be provably correct. Otherwise, a proof could state correctness while abstracting away from a faulty implementation. This proof can be done independently from proving a barrier usage scenario correct.

In addition, we have compared our novel barrier synchronization primitive with a barrier implementation based on semaphores. As a byproduct we have shown how our CPG-based approach can be used as a basis for proving such implementations correct. In fact it is possible to use any implementation of a barrier based on semaphores to verify a barrier usage scenario together with the barrier's implementation.

Our barrier construct is better suited for program analysis because it fully can be analyzed by static analysis, while the implementations using semaphores, in order to omit dead paths, require advanced techniques (e.g. symbolic analysis). However, programs using our barrier synchronization primitive from within loops or conditional statements will still require advanced techniques.

Since Kronecker algebra is based on the theory of finite automata, dynamically allocated tasks and dynamically allocated protected objects cannot be modeled by our approach. As our analysis targets safety related systems, we do not consider this a severe limitation.

Worst-Case Execution Time Analysis. In the chapter on WCET analysis, we focused on incorporating stalling times automatically in a WCET analysis of shared memory concurrent programs running on a multi-core architecture. The emphasis was on a formal definition of the dataflow equations for timing analysis. Our approach is suited for parallel and concurrent systems. CPGs, as introduced in this dissertation, serve as a basis for WCET analysis of multi-threaded concurrent programs. We applied a generating functions approach. Dataflow equations are set up. The WCET is calculated by a non-linear function solver. Non-linearity is inherent to the multi-threaded WCET problem. The reasons are that (1) several copies of loops show up in the RCPG and (2) partial loop unrolling has to be done in certain cases. (1) implies that loop iteration numbers for loop copies have to be considered variable until the maximization process takes place. Thus, nested loops cause non-linear constraints to be handed to the function solver. (2) generates additional non-linear constraints.

Deadlock Avoidance in Railways Systems. Finally, we proposed a *deadlock avoidance* approach for railway systems. For multiple trains and a given track topology, we are able to calculate all possible train movements. Some lead and some do not lead to deadlocks. From some points on no deadlocks are reachable and the trains can proceed with their movements in any order. From certain other points a deadlock is inevitable. Our approach, which may take into account additional constraints, such as weight of the train or penalties due to tardiness, can be used to avoid such situations and to suggest deadlock-free train movements only.

11.2 Outlook

In this section, we suggest directions for future work.

11.2.1 WCET Analysis

In terms of WCET analysis a lot of work remains to be done. One future work may be modelling low-level hardware features in order to form a fully static technique. In general, without taking into account instruction pipelining, shared caches, arbitration of shared memory bus, branch prediction, prefetching, etc., we might overestimate the WCET. Our approach could benefit from e.g. [CCR$^+$12, LDM$^+$12, YZ08] which support shared L2 instruction caches. With the advent of these performance-enhancing hardware features, the WCET estimation became [AEF$^+$14] more difficult. Thus, a second direction for future work could be to combine our CPGs and flow analysis with measurements in order to form a hybrid WCET technique as described in [NYP15a].

Finding the best non-linear function solver is ongoing research. Mathematica© was just the first attempt. This will probably lead to better maximization times. A direction of future work is to generalize for multiple threads running on one CPU core. What we currently are working on is to extend our WCET analysis approach such that, beside semaphore based barriers, it can also handle programs adopting barrier synchronization constructs as introduced in Chapter 7.

We believe that WCET analysis and schedulability analysis should work closely together. In multi-core systems both influence each other. In [ADI+15] "cross-core inference on shared hardware resources" is mentioned as a main reason for that. From our point of view, it is even worth to investigate how both, namely WCET analysis and schedulability analysis, can be done in one step. For example, the number of paths, which have to be taken into account in a WCET analysis, can be reduced, when dispatching points are statically known.

Further, it would be interesting how an implicit path enumeration technique (IPET) approach [PS97] together with non-linear solvers can produce similar results to our approach. Another direction for future work is to generalize the monitors of [BB14] in order to enable WCET analysis. Finally, a possible direction for future work could be a WCET analysis of barrier implementations as presented in this dissertation and semaphore-based implementations.

11.2.2 Value-Sensitive Analysis

Value-sensitive analysis such as symbolic analysis (e.g. [BSB08]) can also be performed on CPGs. First steps were done in [BB14]. By using a value-sensitive analysis on CPGs we could gain precision in order to exclude paths unreachable during runtime which we observed e.g. in Chapter 7. A value-sensitive analysis may also help excluding false positives in static deadlock detection.

11.2.3 Data Flow Analysis

Performing data flow analysis (e.g. [RP86, RP88, SGL98, KU76]) on CPGs is also a direction for future work. There exists already ample work in the field of concurrent software (e.g. [CVJL08, FM07, DCCN04, NAC99]). Because CPGs for concurrent programs serve as a similar data structure as CFGs do for sequential programs, we believe that data flow analysis techniques can be applied on CPGs.

11.2.4 Graph Reductions

Thread synchronization usually helps to reduce the system size. Currently there is also work in progress for a GPGPU implementation generating RCPGs. The first results are very promising. However, in order to enable analysis of industrial-sized systems, the number of nodes has to be reduced further.

For certain analyses, the size of RCPGs may be further reduced. The Kronecker sum generates all interleavings (in our case) of the basic blocks. One path out of the generated interleavings may be enough to represent all the interleavings even for a value-sensitive analysis. On the other hand, at certain points in the code, information may be available, where the dispatcher is not allowed to do context switches. As mentioned above, this is an area, where the applied scheduling method may influence an analysis.

Future work could include collapsing of the input RCFGs. This coarsens the granularity of the RCFGs. Hence, the resulting RCPGs will be smaller.

We consider optimizations similar to partial order reduction (cf. [CGP99]) as future work. Partial-Order Reduction (POR) is a way to reduce the number of interleavings. It is vital that this reduction does not lead to results different from those one would obtain by taking into account all possible interleavings. Early reduction algorithms were developed by Overman [Ove81]. Beside others, the relatively new papers [DHRR04, KSG09] are trying to reduce the amount of interleavings in shared-memory programs. A vast amount of work has been published building on the papers mentioned next, even on combining several approaches to achieve better results. In [Val96, Chapters 6 and 7] Valmari gives a good survey of basic approaches. In the following, we present some fundamental approaches.

Virtual Coarsening. The idea is that in a concurrent program only the ordering of actions visible to other threads is important. This reduction can be made without loss of information [AM71, Pnu86].

Sequentialization. In [GM11] Garg and Madhusudan present sequentialization of concurrent programs. It is shown that if for a concurrent program a compositional proof exists, then it can be translated to a sequential program. Nevertheless, a drawback of the approach is that it generates recursive programs, even when the concurrent program contains no recursion.

Partial Order Reduction. In [Val96] Valmari proposes the theory of stubborn sets, which is based on commutativity. This method tries to “save effort by postponing the investigation of structural transitions to future states. . . ” [Val96]. Two versions, weak and strong stubborn sets, are distinguished. The weak theory is more complicated and more difficult to implement, but it leads to better reduction results.

Godefroid presents in [God96] sleeping sets and persistent sets. Sleeping sets capture information of the past of the search. Persistent sets can be seen as an enhancement of stubborn sets. The semantic model was inspired by Mazurkiewicz’s traces [Maz95].

Peled [Pel94] uses ample sets which are persistent sets satisfying additional conditions sufficient for LTL model checking. Minea [Min99] uses also ample sets, but with a less restrictive independence relation. A description how ample sets are calculated can be found in [CGMP99]. A proof for the correctness of the reduction and an algorithm calculating ample sets using heuristics is given in [CGP99].

Kahlon et al. propose in [KSG09] a framework for static analysis of concurrent programs. Partial order reduction and synchronization constraints are used to reduce thread interleavings. In order to gain further reductions abstract interpretation is applied.

Symmetric Reduction. A system may contain several identical components that are coupled to each other. Symmetric reduction tries to find such symmetries. Its complexity is proved to be the same as that of the graph isomorphism problem [Jun03, p. 22].

(Symbolic) Model Checking. An important approach in the field of model checking is the work of McMillan [McM93] which is based on *Ordered Binary Decision Diagrams* (OBDD). The performance of this approach is highly dependent on the variable order. Finding an optimal order is known to be NP-complete [Bry92]. Thus BDD representations do not improve worst-case complexity. In addition, problems exist which have exponential size OBDDs for any variable order. Also notable is the work of Clarke et al. [CGJ$^+$01] with their counter-example guided abstraction refinement (CEGAR) approach. In [RGG$^+$95] a model checking tool is presented that builds up a system gradually, at each stage compressing the subsystems to find an equivalent CSP process with many less states. With this approach systems of exponential size ($\geq 10^{20}$) can be model checked successfully. In [GG08] Ganai and Gupta studied modeling concurrent systems for bounded model checking (BMC). Like all BMC approaches it has the drawback that it can only show correctness within a bounded number of k steps.

There exists also a lot of work combining the approaches listed above. In [ABH$^+$97] symbolic model checking is combined with partial order reduction. Partial order reduction and symmetry reduction are combined in [EJP97].

It would be interesting to see whether or not it is possible to incorporate something similar to our lazy RCPG generation algorithm.

11.2.5 Automata Based Model Checking

Linear temporal logic (LTL) was at first used by Amir Pnueli for the verification of computer programs in [Pnu77]. For each CPG, it is possible to generate an equivalent Büchi automaton [Büc62]. A property of interest, which is stated as a LTL specification, can be formulated. LTL specifications can be translated into Büchi automata, too [WVS83, VW94]. Thus a second Büchi automaton can be generated out of the negated property.

By using both automata it is possible to apply LTL model checking [GG08, Var07, Pel00, CGP99, VW86]. When the intersection of both automata is empty, then the (non negated) property is satisfied. There were also attempts to do a computational tree logic (CTL) model checking based on Kronecker algebra [KL98].

> *"One never notices what has been done,*
> *one can only see what remains to be done."*
>
> – MARIE CURIE, Polish and naturalized-French physicist and chemist,
> Nobel Prize in Physics 1903, Nobel Prize in Chemistry 1911, 1867-1934

APPENDIX A

The State Explosion Problem

"Controlling complexity is the essence of computer programming."

– BRIAN WILSON KERNIGAN, Canadian computer scientist, 1942-

In this appendix, we summarize the main results of [MB08].

In general, for analysis of multi-threaded software it is important to analyze all possible execution sequences. This ensures that each possible state is reached. We will start with an example which shows the problem in practice. The simple example will be followed by a theoretical analysis of the state explosion problem.

As a motivating example consider program $\mathcal{P}$ executing the threads C and D in an interleaving semantics. Please note that we define all statements to be atomic and (in this appendix), immediately after execution of a statement, a context switch to the other thread may happen. Let thread C consist of $a \cdot b$, whereas thread D contains the statements $c \cdot d$ as follows:

$$\mathcal{P}\colon (\underbrace{x{:=}4;}_{a} \underbrace{x{:=}x+3}_{b}) \parallel (\underbrace{x{:=}2;}_{c} \underbrace{x{:=}(x*x)+1}_{d}) \ .$$

We can reuse the figures of Example 6 in Subsection 3.2.3. The graphs depicted in Figures 3.4a and 3.4b can be interpreted as the CFGs for the threads C and D, respectively. Their Kronecker sum is shown in Figure 3.4d. By using Lemma 3 we calculate the number of nodes in the corresponding CPG which obviously is 9. But what about the number of interleavings?

Program $\mathcal{P}$ may result in one out of six states caused by the corresponding six interleavings. All possible final states of program $\mathcal{P}$ are depicted in Table A.1. From now on we focus on how the number of interleavings of an arbitrary number of threads and the corresponding statements or basic blocks can be calculated.

Order	x
a b c d	5
a c b d	26
a c d b	8
c a b d	50
c a d b	20
c d a b	7

Table A.1: Computation Results of Program $\mathcal{P}$

For enumerating the number of interleavings for n threads $(t_1, t_2, \ldots, t_n)$, where each t_i has k_i statements (and $1 \leq i \leq n$), the multinomial theorem can be applied. This results in

$$\binom{k_1 + k_2 + k_3 + \cdots + k_n}{k_n} \cdots \binom{k_1 + k_2 + k_3}{k_3} \cdot \binom{k_1 + k_2}{k_2} = \frac{(k_1 + k_2 + k_3 + \cdots + k_n)!}{k_1! k_2! k_3! \ldots k_n!}$$

interleavings (sometimes also referred to as orderings).

Lemma 8 (Number of Interleavings) *Given n threads $(t_1, t_2, \ldots, t_n)$, where each t_i has k_i statements, the number of interleavings is given by*

$$\frac{\left(\sum\limits_{i=1}^{n} k_i\right)!}{\prod\limits_{i=1}^{n} k_i!}. \tag{A.1}$$

□

Going back with our example from above, we can calculate the number of interleavings of program $\mathcal{P}$ by applying Lemma 8 and get the expected result $\frac{(2+2)!}{2!*2!} = \frac{4*3*2}{2*2} = 6$.

Lemma 8 shows how simple it is to get astronomically high numbers of interleavings. If we have 20 statements in each of three threads, then we get $\frac{(60!}{(20!)^3} \approx 5 \times 10^{26}$ interleavings. Please note that the formulæ in this chapter apply to both, statements or blocks, depending on the used granularity.

In order to find the maximum of Equation (A.1), we have to maximize

$$\frac{k!}{\prod\limits_{i=1}^{n} k_i!} \tag{A.2}$$

where $k = \sum_{i=1}^{n} k_i$. In the following, we use the gamma function $\Gamma(x)$ (cf. [AS64]) to replace the integer factorial by a real-valued function. Note that $\Gamma(m + 1) = m!$.

In order to find the extreme value of Equation (A.2), we employ the logarithmic derivative of Equation (A.2) which simplifies the calculations significantly.

The derivative of

$$log(k!) - \sum_{i=1}^{n} log(\Gamma(k_i + 1)) + \lambda(\sum_{i=1}^{n} k_i - k)$$

with respect to k_i is

$$-\frac{\Gamma'(k_i + 1)}{\Gamma(k_i + 1)} + \lambda = 0,$$

where $1 \leq i \leq n$. This is valid for all k_i, in particular for $i = s$ and $i = t$, i.e.,

$$\frac{\Gamma'(k_s + 1)}{\Gamma(k_s + 1)} = \frac{\Gamma'(k_t + 1)}{\Gamma(k_t + 1)}.$$

Using the digamma function $\psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$ (cf. [AS64]) we can write

$$\psi(k_s + 1) = \psi(k_t + 1).$$

Because $\psi(x)$ is monotonically increasing for $x \geq 0$ (cf. e.g. [AS64]) we get

$$k_s = k_t, \text{ for all } 1 \leq s, t \leq n$$

which implies $k_i = \frac{k}{n}$, provided that n divides k, i.e., $n \,|\, k$.

Thus, we have proved the following lemma.

Lemma 9 *For a given number of statements k, the worst-case number of interleavings appears if all n threads have the same number of statements. In this case the number of interleavings is given by*

$$\frac{k!}{\left(\left(\frac{k}{n}\right)!\right)^n} \tag{A.3}$$

where n is a divisor of k, i.e., $n \,|\, k$.

□

In the following, we write $k = \beta \cdot n$. If the number of statements per thread $\beta \geq 1$ is fixed, Formula (A.3) can be estimated by Stirling's approximation $m! = (\frac{m}{e})^m \sqrt{2\pi m}(1 + O(\frac{1}{m}))$ as $m \to \infty$ (cf. e.g. [AS64]) giving

$$\frac{(\beta\, n)!}{(\beta!)^n} \sim n^{\beta\, n + \frac{1}{2}} \, (2\pi\beta)^{-\frac{n}{2} + \frac{1}{2}}, \; (n \to \infty). \tag{A.4}$$

For $\beta \in \{5, 10, 15, 20, 25, 30\}$ the characteristics of this formula are depicted in Figure A.1. This case describes the practical case when there is e.g. an Ada task type defining tasks with β statements.

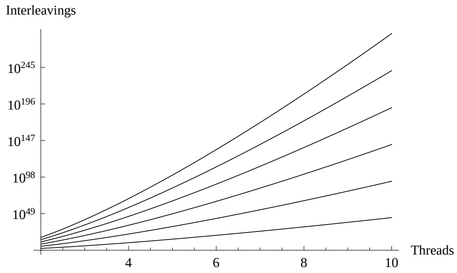


Figure A.1: Interleavings for Fixed Number of Statements per Thread

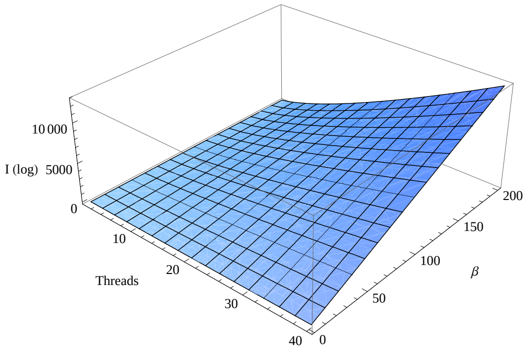


Figure A.2: Interleavings for a Variable Number of Threads and Statements per Thread

In Figure A.2 the behavior of a variable number of statements per thread $2 \leq \beta \leq 200$ and a variable number of threads $1 \leq n \leq 40$ is depicted in logarithmic scale. Please note that the functions depicted in Figure A.1 and A.2 are actually defined for natural numbers only.

The theoretical analysis in this chapter showed an exponential growth of interleavings in terms of the number of statements. This number of interleavings is generated with the Kronecker sum of the adjacency matrices of the CFGs of the threads present of the multi-threaded program. In this dissertation, we work on the granularity of basic blocks. Thus, we have an exponential growth in terms of the number of basic blocks. Thread synchronization usually helps to reduce this number.

Bibliography

[ABH$^+$97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-Order Reduction in Symbolic State Space Exploration. In *Proceedings of the 9th International Conference on Computer Aided Verification – CAV 1997, Haifa, Israel, June 22-25, 1997*, pages 340–351, 1997.

[ADI$^+$15] Sebastian Altmeyer, Robert I. Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. A Generic and Compositional Framework for Multicore Response Time Analysis. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems – RTNS 2015, Lille, France, Nov. 4-6, 2015*, pages 129–138, New York, NY, USA, 2015. ACM.

[AEF$^+$14] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building Timing Predictable Embedded Systems. *ACM Transactions on Embedded Computer Systems*, 13(4):82:1–82:37, March 2014.

[AG98] Alexander Aiken and David Gay. Barrier Inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages – POPL 1998, San Diego, California, USA, January 19 - 21*, pages 342–354, New York, NY, USA, 1998. ACM.

[AM71] Edward A. Ashcroft and Zohar Manna. Formalization of Properties of Parallel Programs. In Bernard Meltzer and Donald Michie, editors, *Proceedings of the 6th Annual Machine Intelligence Workshop, Edinburgh, UK, July 1970*, pages 17–41. Edinburgh University Press, 1971.

[AP02] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java, 2nd edition.* Cambridge University Press, 2002.

[AS64] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover, New York, 1964.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachusetts, 1986.

[BB98] Johann Blieberger and Bernd Burgstaller. Symbolic Reaching Definitions Analysis of Ada Programs. In *Proceedings of the 3rd International Conference on Reliable Software Technologies – Ada-Europe 1998, Uppsala, Sweden*, volume 1411 of *Lecture Notes in Computer Science (LNCS)*, pages 238–250. Springer Press, June 1998.

[BB03] Johann Blieberger and Bernd Burgstaller. Eliminating Redundant Range Checks in GNAT Using Symbolic Evaluation. In *Proceedings of the 8th International Conference on Reliable Software Technologies – Ada-Europe 2003, Toulouse, France*, volume 2655 of *Lecture Notes in Computer Science (LNCS)*, pages 153–167. Springer Press, June 2003.

[BB14] Bernd Burgstaller and Johann Blieberger. Kronecker Algebra for Static Analysis of Ada Programs with Protected Objects. In *Proceedings of the 19th International Conference on Reliable Software Technologies – Ada-Europe 2014*, volume 8454 of *Lecture Notes in Computer Science (LNCS)*, pages 27–42, Paris, France, June 2014. Springer Press.

[BBM06] Bernd Burgstaller, Johann Blieberger, and Robert Mittermayr. Static Detection of Access Anomalies in Ada95. In *Proceedings of the 11th International Conference on Reliable Software Technologies – Ada-Europe 2006, Porto, Portugal*, volume 4006 of *Lecture Notes in Computer Science (LNCS)*, pages 40–55. Springer Press, June 2006.

[BBM07] Johann Blieberger, Bernd Burgstaller, and Robert Mittermayr. Static Detection of Livelocks in Ada Multitasking Programs. In *Proceedings of the 12th International Conference on Reliable Software Technologies – Ada-Europe 2007, Geneve, Switzerland*, volume 4498 of *Lecture Notes in Computer Science (LNCS)*, pages 69–83. Springer Press, June 2007.

[BBS99] Johann Blieberger, Bernd Burgstaller, and Bernhard Scholz. Interprocedural Symbolic Evaluation of Ada Programs with Aliases. In *Proceedings of the 4th International Conference on Reliable Software Technologies – Ada-Europe 1999, Santander, Spain*, volume 1622 of *Lecture Notes in Computer Science (LNCS)*, pages 136–145. Springer Press, June 1999.

[Bel97] Richard Bellman. *Introduction to Matrix Analysis.* Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2nd edition, 1997.

[BK02] Peter Buchholz and Peter Kemper. Efficient Computation and Representation of Large Reachability Sets for Composed Automata. *Discrete Event Dynamic Systems*, 12(3):265–286, 2002.

[Bli02] Johann Blieberger. Data-Flow Frameworks for Worst-Case Execution Time Analysis. *Real-Time Systems*, 22(3):183–227, 2002.

[Bru16] Randall L. Brukardt. Annotated Ada Reference Manual, ISO/IEC 8652:2012/Cor 1:2016. `http://www.ada-auth.org/standards/aarm12_w_tc1/AA-Final.pdf`, 2016. [Online; accessed 2016-08-17].

[Bry92] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BSB08] Bernd Burgstaller, Bernhard Scholz, and Johann Blieberger. *Symbolic Analysis: An Algebra-Based Approach.* VDM Verlag, Saarbrücken, 2008.

[BSB12] Bernd Burgstaller, Bernhard Scholz, and Johann Blieberger. A symbolic analysis framework for static analysis of imperative programming languages. *Journal of Systems and Software*, 85(6):1418–1439, 2012.

[BSV14] Johann Blieberger, Andreas Schöbel, and Mark Volcic. Kronecker-Algebra und ihre breit gefächerten Anwendungen im Eisenbahnbereich. *Signal + Draht*, 7/8:15–18, 2014.

[Büc62] Julius Richard Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Proceeding of the 1st International Congress on Logic, Methodology, and Philosophy of Science – CLMPS 1960, Stanford, California, USA, 1960*, pages 1–11. Stanford University Press, 1962.

[BW09] Raymond P. L. Buse and Westley R. Weimer. The Road Not Taken: Estimating Path Execution Frequency Statically. In *Proceedings of the 31st International Conference on Software Engineering – ICSE 2009, Vancouver, Canada, May 16-24, 2009*, pages 144–154, 2009.

[CCR$^+$12] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A Unified WCET Analysis Framework for Multi-core Platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 99–108, 2012.

[CGJ$^+$01] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the State Explosion Problem in Model Checking. In Reinhard Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science (LNCS)*, pages 176–194. Springer Press, 2001.

[CGMP99] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State Space Reduction using Partial Order Techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(3):279–287, 1999.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[CM99] Gianfranco Ciardo and Andrew S. Miner. A Data Structure for the Efficient Kronecker Solution of GSPNs. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models – PNPM 1999, Zaragoza, Spain, 8-10 Sept. 1999*, pages 22–31. IEEE Computer Society Press, 1999.

[Cui10] Yong Cui. *Simulation-Based Hybrid Model for a Partially-Automatic Dispatching of Railway Operation.* PhD thesis, Universität Stuttgart, 2010.

[CVJL08] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. Dataflow Analysis for Concurrent Programs Using Datarace Detection. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation – PLDI 2008, Tucson, AZ, USA, June 07 - 13, 2008*, pages 316–326, New York, NY, USA, 2008. ACM.

[Dav81] Marc Davio. Kronecker Products and Shuffle Algebra. *IEEE Transactions on Computers*, 30(2):116–125, 1981.

[DCCN04] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow Analysis for Verifying Properties of Concurrent Software Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(4):359–430, Oct. 2004.

[DHRR04] Matthew B. Dwyer, John Hatcliff, Robby, and Venkatesh Prasad Ranganath. Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004.

[Dij59] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Dij65] Edsger Wybe Dijkstra. Cooperating sequential processes. University of Texas at Austin, EWD-123, 1965.

[Dijnda] Edsger Wybe Dijkstra. Een algorithme ter voorkoming van de dodelijke omarming. circulated privately, n.d.

[Dijndb] Edsger Wybe Dijkstra. Over de sequentialiteit van procesbeschrijvingen. circulated privately, n.d.

[Dow05] Allen B. Downey. *The Little Book of Semaphores.* Green Tea Press, 2005.

[EJP97] Ernest Allen Emerson, Somesh Jha, and Doron Peled. Combining Partial Order and Symmetry Reductions. In *Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems – TACAS 1997, Enschede, The Netherlands, April 2–4, 1997*, volume 1217 of *Lecture Notes in Computer Science (LNCS)*, pages 19–34. Springer Press, 1997.

[FGS03] Maria Pia Fanti, Alessandro Giua, and Carla Seatzu. A deadlock prevention method for railway networks using monitors for colored Petri nets. In *Proceedings of 2003 IEEE International Conference on Systems, Man and Cybernetics*, volume 2 of *SMC '03*, pages 1866–1873, October 2003.

[FM07] Azadeh Farzan and P. Madhusudan. Causal Dataflow Analysis for Concurrent Programs. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007*, pages 102–116, 2007.

[FNNP16] José Carlos Fonseca, Geoffrey Nelissen, Vincent Nélis, and Luís Miguel Pinho. Response Time Analysis of Sporadic DAG Tasks under Partitioned Scheduling. In *Proceedings of the 11th IEEE International Symposium on Industrial Embedded Systems – SIES 2016, Krakow, Poland, May 23-25, 2016*, pages 1–10, 2016.

[FT84] Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science – FOCS 1984, Singer Island, Florida, USA, 24-26 Oct., 1984*, pages 338–346, 1984.

[GELP10] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In Björn Lisper, editor, *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis – WCET 2010, Brussels, Belgium, July 6, 2010*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 101–112, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[GG08] Malay K. Ganai and Aarti Gupta. Efficient Modeling of Concurrent Systems in BMC. In *Proceedings of the 15th International Workshop on Model Checking Software*, SPIN '08, pages 114–133, Berlin, Heidelberg, 2008. Springer Press.

[GGL14] Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Timing Analysis of Parallel Software Using Abstract Execution. In Kenneth McMillan and Xavier Rival, editors, *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation – VMCAI 2014, San Diego, USA, January 19-21, 2014*, volume 8318 of *Theoretical Computer Science and General Issues*, pages 59–77, 2014.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[GKUR12] Mike Gerdes, Florian Kluge, Theo Ungerer, and Christine Rochange. The Split-Phase Synchronisation Technique: Reducing the Pessimism in the WCET Analysis of Parallelised Hard Real-Time Programs. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications – RTCSA 2012, Seoul, Korea, August 19-22, 2012*, pages 88–97. IEEE Computer Society, 2012.

[GM11] Pranav Garg and P. Madhusudan. Compositionality Entails Sequentializability. In *Proceedings of the Seventeenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2011, Saarbruecken, Germany, March 26 - April 3, 2011*, volume 6605 of *Lecture Notes in Computer Science (LNCS)*, pages 26–40. Springer Press, 2011.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science (LNCS).* Springer Press, 1996.

[Gon12] Javier Fernández González. *Java 7 Concurrency Cookbook.* Packt Publishing Ltd., 2012.

[Gra81] Alexander Graham. *Kronecker Products and Matrix Calculus with Applications.* Ellis Horwood Ltd., New York, 1981.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[HIK11] Richard Hammack, Wilfried Imrich, and Sandi Klavžar. *Handbook of Product Graphs.* Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL, second edition, 2011. With a foreword by Peter Winkler.

[HM76] Peter Henderson and James H. Morris, Jr. A Lazy Evaluator. In *Proceedings of the 3rd ACM Symposium on Principles of Programming Languages – POPL 1976, Atlanta, Georgia, USA*, pages 95–103. ACM, January 1976.

[HS98] Jonathan M. D. Hill and David B. Skillicorn. Practical Barrier Synchronisation. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing – PDP 1998, Madrid, Spain, 21-23 Jan, 1998*, pages 438–444, 1998.

[HT66] Frank Harary and Charles A. Trauth, Jr. Connectedness of Products of Two Directed Graphs. *SIAM Journal on Applied Mathematics*, 14(2):250–254, 1966.

[Hur94] Adolf Hurwitz. Zur Invariantentheorie. *Mathemathische Annalen*, 45:381–404, 1894.

[IKR08] Wilfried Imrich, Sandi Klavžar, and Douglas F. Rall. *Topics in Graph Theory: Graphs and Their Cartesian Product.* A K Peters Ltd, 2008.

[JES71] Edward G. Coffman Jr., M. J. Elphick, and Arie Shoshani. System Deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[Jun03] Tommi Junttila. *On The Symmetry Reduction Method For Petri Nets and Similar Formalisms.* PhD thesis, Helsinki University of Technology, 2003.

[Kir45] Gustav Robert Kirchhoff. Ueber den Durchgang eines elektrischen Stromes durch eine Ebene, insbesondere durch eine kreisförmige. *Annalen der Physik und Chemie*, LXIV(4):497–514, 1845.

[KKP$^+$11] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software and Systems Modeling*, 2011. (online edition).

[KL98] Peter Kemper and Ralf Lübeck. Model Checking Based on Kronecker Algebra. Technical report, Universität Dortmund, Fachbereich Informatik, Forschungsbericht Nr. 669, 1998.

[Knu11] Donald E. Knuth. *Combinatorial Algorithms*, volume 4A of *The Art of Computer Programming.* Addison-Wesley, 2011.

[KS86] Werner Kuich and Arto Salomaa. *Semirings, Automata, Languages.* Springer Press, 1986.

[KSG09] Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta. Semantic Reduction of Thread Interleavings in Concurrent Programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2009*, volume 5505, pages 124–138, Berlin, Heidelberg, 2009. Springer Press.

[KU76] John B. Kam and Jeffrey D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23:158–171, January 1976.

[Küs91] Gerhard Küster. On the Hurwitz Product of Formal Power Series and Automata. *Theoretical Computer Science*, 83(2):261–273, 1991.

[KY06] Amir Kamil and Katherine Yelick. Concurrency Analysis for Parallel Programs with Textually Aligned Barriers. In E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing – LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005*, volume 4339 of *Theoretical Computer Science and General Issues*, pages 185–199, 2006.

[LCT13] Duy-Khanh Le, Wei-Ngan Chin, and Yong-Meng Teo. Verification of Static and Dynamic Barrier Synchronization Using Bounded Permissions. In Lindsay Groves and Jing Sun, editors, *Formal Methods and Software Engineering*

- *Proceedings of the 15th International Conference on Formal Engineering Methods – ICFEM 2013, Queenstown, New Zealand, October 29 to November 1, 2013*, volume 8144 of *Lecture Notes in Computer Science (LNCS)*, pages 231–248, 2013.

[LDL04] Quan Lu, Maged Dessouky, and Robert C. Leachman. Modeling Train Movements through Complex Rail Networks. *ACM Transactions on Modeling Computer Simulation*, 14(1):48–75, Jan. 2004.

[LDM$^+$12] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Syst.*, 48(6):638–680, Nov. 2012.

[Mar95] Ullrich Martin. *Verfahren zur Bewertung von Zug- und Rangierfahrten bei der Disposition.* PhD thesis, TU Braunschweig, Institut für Eisenbahnwesen und Verkehrssicherung, 1995.

[Maz95] Antoni Mazurkiewicz. Introduction to Trace Theory. In Volker Diekert and Grzegorz Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific Publishing Co., Inc., 1995.

[MB08] Robert Mittermayr and Johann Blieberger. Static Partial-Order Reduction of Concurrent Systems in Polynomial Time. In *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation – ISoLA'08, Chalkidiki, Greece*, volume 17 of *Communications in Computer and Information Science (CCIS)*, pages 619–633. Springer Press, 2008.

[MB11] Robert Mittermayr and Johann Blieberger. Shared Memory Concurrent System Verification using Kronecker Algebra. Technical Report 183/1-155, Automation Systems Group, TU Vienna, `http://arxiv.org/abs/1109.5522`, Sept. 2011. [Online; accessed 2016-08-17].

[MB12a] Alexander Malkis and Aindya Banerjee. Verification of Software Barriers. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPoPP 2012, New Orleans, Louisiana, USA, February 25-29, 2012*, pages 313–314, New York, NY, USA, 2012. ACM.

[MB12b] Robert Mittermayr and Johann Blieberger. Timing Analysis of Concurrent Programs. In Tullio Vardanega, editor, *Proceedings of the 12th International Workshop on Worst-Case Execution Time Analysis – WCET 2012, Pisa, Italy, July 10, 2012*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 59–68, Dagstuhl, Germany, 2012. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[MB16a] Robert Mittermayr and Johann Blieberger. A Generic Graph Model for WCET Analysis of Multi-Core Concurrent Applications. *Journal of Software Engineering and Applications (JSEA), Special Issue On Parallel & Concurrent Computing*, 9(5):182–198, May 2016.

[MB16b] Robert Mittermayr and Johann Blieberger. Kronecker Algebra for Static Analysis of Barriers in Ada. In M. Bertogna, L. M. Pinho, and E. Quinones, editors, *Proceeding of the 21th International Conference on Reliable Software Technologies – Ada-Europe 2016, Pisa, Italy, June 13-17, 2016*, volume 9695 of *Lecture Notes in Computer Science (LNCS)*, pages 145–159. Springer Press, June 2016.

[MBS12] Robert Mittermayr, Johann Blieberger, and Andreas Schöbel. Kronecker Algebra based Deadlock Analysis for Railway Systems. *Scientific Journal on Traffic and Transportation Research (PROMET)*, pages 359–369, 2012.

[McA63] M. H. McAndrew. On the Product of Directed Graphs. *Proceedings of the American Mathematical Society*, 14(4):600–606, 1963.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[Mil11] Jeff Miller. Earliest Known Uses of Some of the Words of Mathematics. `http://jeff560.tripod.com/k.html`, Revision August 1st, 2011. [Online; accessed 2016-08-17].

[Min99] Marius Minea. *Partial Order Reduction for Verification of Timed Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1999. Chair-Edmund M. Clarke.

[Mit05] Robert Mittermayr. Statische Analyse von Multi-Threading Java-Programmen. Master's thesis, Vienna University of Technology, 2005.

[MPWH03] Graham Mills, Peter J. Pudney, Kevin White, and John Hewitt. The Effects of Deadlock Avoidance on Rail Network Capacity and Performance. In *Proceedings of the 2003 Mathematics-in-Industry Study Group*, 2003.

[NAC99] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data Flow Analysis for Checking Properties of Concurrent Java Programs. In *Proceedings of the 21st International Conference on Software Engineering – ICSE 1999, Los Angeles, CA, USA, May 16 - 22, 1999*, pages 399–410, 1999.

[NPB$^+$14] Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems – ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 109–118, Los Alamitos, CA, USA, 2014. IEEE Computer Society.

[NVI16] NVIDIA Corporation. CUDA Parallel Computing Platform. `http://www.nvidia.com/CUDA`, 2016. [Online; accessed 2016-08-17].

[NYP15a] Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. Methodologies for the WCET Analysis of Parallel Applications on Many-Core Architectures. In *Proceedings of the 18th Euromicro Conference on Digital System Design – DSD 2015, Funchal, Madeira, Portugal, Aug. 26-28, 2015*, pages 748–755, Aug. 2015.

[NYP$^+$15b] Vincent Nélis, Patrick Meumeu Yomsi, Luís Miguel Pinho, Eduardo Quiñones, Marko Bertogna, Andrea Marongiu, Paolo Gai, and Claudio Scordino. A system model and stack for the parallelization of time-critical applications on many-core architectures. Technical Report CISTER-TR-141206, CISTER Research Center, Polytechnic Institute of Porto, Nov. 2015.

[Ope13] OpenMP Architecture Review Board. OpenMP Application Program Interface – Version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013. [Online; accessed 2016-08-17].

[Ope16] OpenCL. The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl`, 2016. [Online; accessed 2016-08-17].

[ORS13] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET Analysis of Real-Time Parallel Applications. In Claire Maiza, editor, *Proceedings of the 13th International Workshop on Worst-Case Execution Time Analysis – WCET 2013, Paris, France, Jul 9, 2013*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 11–20, Dagstuhl, Germany, 2013. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Ove81] William T. Overman. *Verification of Concurrent Systems: Function and Timing.* PhD thesis, University of California, Los Angeles, 1981.

[PA91] Brigitte Plateau and Karim Atif. Stochastic Automata Network For Modeling Parallel Systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

[Pac93] Jörn Pachl. *Steuerlogik für Zuglenkanlagen zum Einsatz unter stochastischen Betriebsbedingungen.* PhD thesis, TU Braunschweig, Institut für Eisenbahnwesen und Verkehrssicherung, Heft 49, 1993.

[PBP13] Dumitru Potop-Butucaru and Isabelle Puaut. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In Claire Maiza, editor, *Proceedings of the 13th International Workshop on Worst-Case Execution Time Analysis – WCET 2013, Paris, France, Jul 9, 2013*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 21–31, Dagstuhl, Germany, 2013. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Pel94] Doron Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In David L. Dill, editor, *Proceedings of the 6th International Conference on Computer Aided Verification – CAV 1994, Stanford, California, USA, June 21-23, 1994*, volume 818 of *Lecture Notes in Computer Science (LNCS)*, pages 377–390, London, UK, 1994. Springer Press.

[Pel00] Doron Peled. Model Checking Using Automata Theory. In M. Kemal Inan and Robert P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series*, pages 55–79. Springer Press, 2000.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[Pla85] Brigitte Plateau. On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms. *ACM SIGMETRICS Performance Evaluation Review*, 13:147–154, aug 1985.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science – FOCS 1977, Providence, Rhode Island, USA, Oct. 31 - Nov. 2, 1977*, pages 46–57, Oct. 1977.

[Pnu86] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science (LNCS)*, pages 510–584, New York, NY, USA, 1986. Springer Press.

[PQB$^+$14] Luís Miguel Pinho, Eduardo Quiñones, Marko Bertogna, Andrea Marongiu, Jorge Pereira Carlos, Claudio Scordino, and Michele Ramponi. P-SOCRATES: a Parallel Software Framework for Time-Critical Many-Core Systems. In *17th Euromicro Conference on Digital System Design – DSD 2014, Verona, Italy, August 27-29, 2014*, pages 214–221, 2014.

[PS97] Peter Puschner and Anton Schedl. Computing Maximum Task Execution Times - A Graph-Based Approach. *Journal of Real-Time Systems*, 13:67–91, 1997.

[PT83] E. R. Petersen and Allison James Taylor. Line Block Prevention in Rail Line Dispatch and Simulation Models. *Information Systems and Operations Research (INFOR journal)*, 21(1):46–51, 1983.

[Ram96] Ganesan Ramalingam. Data Flow Frequency Analysis. In Charles N. Fischer, editor, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation – PLDI 1996, Philadelphia, Pennsylvania, USA, May 21-24, 1996*, pages 267–277. ACM, 1996.

[Ram99] Ganesan Ramalingam. Identifying Loops in Almost Linear Time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):175–188, March 1999.

[Ram00] Ganesan Ramalingam. Context-Sensitive Synchronization-Sensitive Analysis Is Undecidable. *ACM Transactions Programming Language Systems*, 22(2):416–430, 2000.

[Rei85] Wolfgang Reisig. *Petri Nets – An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer Press, 1985.

[RGG$^+$95] Andrew William Roscoe, Paul H. B. Gardiner, Michael H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems – TACAS 1995*, pages 133–152, London, UK, 1995. Springer Press.

[RP86] Barbara Gershon Ryder and Marvin C. Paull. Elimination Algorithms for Data Flow Analysis. *ACM Computing Surveys (CSUR)*, 18(3):277–316, sep 1986.

[RP88] Barbara Gershon Ryder and Marvin C. Paull. Incremental Data-Flow Analysis Algorithms. *ACM Transactions on Programming Language and Systems (TOPLAS)*, 10(1):1–50, jan 1988.

[SBF00] Bernhard Scholz, Johann Blieberger, and Thomas Fahringer. Symbolic Pointer Analysis for Detecting Memory Leaks. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation – PEPM 2000, Boston, Massachusetts, USA, January 22 - 23, 2000*, volume 1845, pages 104–113. ACM, 2000.

[SBS15] Mark Stefan, Johann Blieberger, and Andreas Schöbel. Kronecker Algebra zur Optimierung des Eisenbahnbetriebes. *Eisenbahntechnische Rundschau (ETR)*, 9:78–84, 2015.

[SBS16] Mark Stefan, Johann Blieberger, and Andreas Schöbel. Application of Kronecker Algebra in Railway Operation. *Tehnički vjesnik – Technical Gazette (TV-TG)*, 2016.

[SGL98] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. A New Framework for Elimination-Based Data Flow Analysis Using DJ Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(2):388–435, 1998.

[Sre95] Vugranam C. Sreedhar. *Efficient program analysis using DJ graphs*. PhD thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, 1995.

[Sta11] William Stallings. *Operating Systems – Internals and Design Principles.* Prentice Hall, 7th edition, 2011.

[SVB14] Andreas Schöbel, Mark Volcic, and Johann Blieberger. Analysis and optimisation of railway systems. In *EURO-ŽEL 2014*, Žilina, Slovak Republic, May 2014.

[Tar72] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing (SICOMP)*, 1(2):146–160, 1972.

[Tar81] Robert Endre Tarjan. A Unified Approach to Path Problems. *Journal of the ACM (JACM)*, 28(3):577–593, 1981.

[Val96] Antti Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science (LNCS)*, pages 429–528. Springer Press, September 1996.

[Var07] Moshe Y. Vardi. Automata-Theoretic Model Checking Revisited. In Byron Cook and Andreas Podelski, editors, *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation – VMCAI 2007, Nice, France, January 14-16, 2007*, pages 137–150. Springer Press, 2007.

[VBS12] Mark Volcic, Johann Blieberger, and Andreas Schöbel. Kronecker Algebra based Travel Time Analysis for Railway Systems. In *FORMS/FORMAT 2012 – 9th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 273–281, Braunschweig, Germany, December 2012.

[VBS13] Mark Volcic, Johann Blieberger, and Andreas Schöbel. Kronecker algebra based modelling of railway operation. In *Proceeding of the 3rd International Conference on Models and Technologies for Intelligent Transport Systems – MT-ITS 2013, Dresden, Germany, December, 2013*, pages 345–356, 2013.

[VBS14] Mark Volcic, Johann Blieberger, and Andreas Schöbel. Optimisation of railway operation by application of kronecker algebra. In *Proceeding of the 3rd International Conference on Road and Rail Infrastructure – CETRA 2014, Split, Croatia, April 28-30, 2014*, pages 37–42, 2014.

[Vol14] Mark Volcic. *Energy-efficient Optimization of Railway Operation – An Algorithm Based on Kronecker Algebra.* PhD thesis, TU Vienna, Treitlstr. 1-3, 1040 Vienna, Dec. 2014.

[VW86] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science – LICS 1986, Cambridge, MA, USA, June 16-18, 1986*, pages 332–344. IEEE Computer Society, Jun. 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning About Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.

[Ž08] M. Žarnay. Solving deadlock states in model of railway station operation using coloured Petri nets. In *Proceedings of Symposium FORMS/FORMAT - Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 205–213, October 2008.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, May 2008.

[Wei62] Paul M. Weichsel. The Kronecker Product of Graphs. *Proceedings of the American Mathematical Society*, 13(1):47–52, 1962.

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning About Infinite Computation Paths. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science – SFCS 1983, Tucson, Nov. 7-9, 1983*, pages 185–194, Washington, DC, USA, 1983. IEEE Computer Society.

[YZ08] Jun Yan and Wei Zhang. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008.

[ZD07] Yuan Zhang and Evelyn Duesterwald. Barriers Matching for Programs with Textually Unaligned Barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPoPP 2007, San Jose, California, USA*, pages 194–204. ACM, 2007.

[ZDG08] Yuan Zhang, Evelyn Duesterwald, and Guang R. Gao. Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. *Languages and Compilers for Parallel Computing: 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007, Revised Selected Papers*, pages 95–109, 2008.

[Zeh58] Johann Georg Zehfuss. Ueber eine gewisse Determinante. *Zeitschrift für Mathematik und Physik*, 3:298–301, 1858.

Curriculum Vitæ
Robert Mittermayr

Contact Information

Institute of Computer-Aided Automation
Vienna University of Technology
Treitlstr. 1-3, 1040 Vienna, Austria
`www.auto.tuwien.ac.at/~robert`
`robert@auto.tuwien.ac.at`

Education

10/2006–08/2016	As a sideline Doctoral Dissertation "*Kronecker Algebra Based Analysis of Shared Memory Concurrent Systems*" in Computer Science, Supervisor: Prof. J. Blieberger.
06/2005	**Dipl.-Ing.**[1] in Computer Science at Vienna University of Technology, Austria. Thesis: "*Statische Analyse von Multi-Threading Java-Programmen*", Supervisor: Prof. J. Blieberger.
01/2004–05/2004	Exchange Student (Erasmus/Sokrates) at Helsinki University of Technology, Finland.
10/1998–05/2005	Studies in Computer Science at Vienna University of Technology, Austria.

Professional Activities

since 08/2014	Software Engineer in the field of railway interlocking systems
10/2006–03/2014	Software Engineer in the field of professional audio and video broadcasting
05/2005–09/2006	Software Engineer in the field of railway interlocking systems

Miscellaneous

12/1997–07/1998	Military Service

---

[1] Five year undergraduate degree directly to master's level (**M.Sc.** equivalent).