Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Informatics

Verifikation von nebenläufigen Programmen in schwachen Speicher-Modellen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Hannes Dallinger
Matrikelnummer 11775789

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Dr. Johann Blieberger
Mitwirkung: Dr. techn. Dipl-Ing. (FH) MSc MBA Patrick Denzler

Wien, 27. Jänner 2025

Hannes Dallinger	Johann Blieberger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

Verification of concurrent programs in weak memory models

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Hannes Dallinger
Registration Number 11775789

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof.Dr. Johann Blieberger
Assistance: Dr. techn. Dipl-Ing. (FH) MSc MBA Patrick Denzler

Vienna, January 27, 2025

Hannes Dallinger   Johann Blieberger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Hannes Dallinger

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.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. Jänner 2025

Hannes Dallinger

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ich möchte meine aufrichtige Dankbarkeit gegenüber meinen Betreuern, Ao.Univ.Prof.Dr. Johann Blieberger und Dr. techn. Dipl.-Ing. (FH) MSc MBA Patrick Denzler, für ihre Unterstützung und ihr Feedback während dieser Thesis zum Ausdruck bringen.

Weiters möchte ich auch meiner Familie für ihre ständige Ermutigung und ihr Verständnis danken. Ihre Unterstützung war von unschätzbarem Wert und hat mir während meines Studiums sehr geholfen.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Ao.Univ.Prof.Dr. Johann Blieberger and Dr. techn. Dipl-Ing. (FH) MSc MBA Patrick Denzler, for their support, guidance, and feedback throughout this thesis.

I also want to thank my family for their constant encouragement and understanding. Their support has been invaluable and helped me during my studies.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Moderne Processoren nutzen schwache Speichermodelle, um die Ausführung von Code zu optimieren. Im Vergleich zu sequentiell konsistenten Speichermodellen erlauben schwache Speichermodelle die Umordnung von Codeanweisungen. Während dies das Verhalten eines Single-Thread-Programmes nicht verändert, kann es in Multicore-Systemen Fehler verursachen indem Ausführungsreihenfolgen zwischen Threads ermöglicht werden, die ansonst nicht möglich wären. Diese Fehler sind für Entwickler oft schwer zu erkennen und zu verstehen.

Diese Thesis schlägt vor, mithilfe von Kronecker Algebra mögliche Ausführungsreihenfolgen von Multithread-Programmen innerhalb schwacher Speichermodelle zu modellieren. Dies kann dann verwendet werden, um Race Conditions im System zu erkennen. Der Fokus liegt dabei speziell auf schwachen Speichermodellen die Release-Acquire Semantiken nutzen.

Im Rahmen der Arbeit wird ein Prototyp entwickelt, der das System zur Erkennung von Race Conditions in Multithread-Programmen einzusetzen, die in LLVM-Bytecode kompiliert wurden, um zu zeigen, dass das System in der Lage ist, in Programmen Probleme zu identifizieren, die unter Verwendung von Release-Acquire Speichermodellen auftreten, obwohl das Programm unter sequentiellen Speichermodellen korrekt war.

Der Prototyp bewies die Korrektheit der sequentiellen Versionen von Petersons und Dekker's Algorithmus und erkannte auch die Fehler, die auftreten, wenn sie nahezu unverändert unter der Verwendung von Release-Acquire-Speichermodellen ausgeführt werden.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Modern Central Processing Units (CPUs) utilize weak memory models to optimize code execution. Compared to sequential consistent memory models, weak memory models allow the reordering of code instructions. While this does not alter the behavior of a single-thread program, it can cause bugs in multicore systems by allowing execution orders between threads that would otherwise not be possible. These errors are often problematic for the developer to detect and understand.

This thesis proposes to use Kronecker Algebra to model possible execution orders of multithreaded programs within weak memory models. The output of such a toolchain can then be used to detect race conditions within the system. It focuses explicitly on weak memory models using release-acquire semantics.

Alongside the thesis, a prototype is developed for the proposed system to detect race conditions in multithreaded programs compiled into LLVM-bytecode. This is done to show that the system can detect problems caused when running a program using release-acquire memory models. However, the program was correct under sequential memory models.

The prototype correctly proved the correctness of the sequential versions of Peterson's and Dekker's Algorithm. Moreover, the prototype detected the errors that arise when Peterson's and Dekker's Algorithms run without modifications using release-acquire memory models.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Aim and Research Questions 3
1.2 Outline 4

2 Background **5**
2.1 Sequential Consistency 5
2.2 Release-Acquire 6
2.3 LLVM 6
2.4 Race Condition 11
2.5 Kronecker Algebra 15

3 Building the Matrix Representation **31**
3.1 Program Skeleton 31
3.2 Setting Dependencies 37

4 Prototype **45**
4.1 Creating Blocks 45
4.2 Building the Skeleton 46
4.3 Dependencies 46
4.4 Optimization 47
4.5 Collecting Further Information 49
4.6 Detecting Dependency Type 49
4.7 Creating the Graph 50
4.8 Detecting Errors 52

5 Examples **55**
5.1 Peterson's Algorithm - Sequential Consistent Version 55
5.2 Peterson's Algorithm - Release-Acquire Version 59

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.3 Peterson's Algorithm - Modified Release-Acquire Version 62
5.4 Dekker's Algorithm - Sequential Consistent Version 64
5.5 Dekker's Algorithm - Release-Acquire Version 68
5.6 Dekker's Algorithm - Modified Release-Acquire Version 71

6 Related Work **75**
6.1 Dynamic Partial Order Reduction 75
6.2 Iris 76
6.3 Context Bound Analysis 77

7 Discussion **79**
7.1 Time 80

8 Conclusion and Future Work **83**

List of Figures **85**

List of Tables **87**

List of Algorithms **87**

Bibliography **89**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

When developing programs, many developers make assumptions about the program and its execution that do not always hold. One of these assumptions is that the compiled machine code is executed with a sequentially consistent memory model.

Sequential consistency is the traditional model for executing code. Executing code sequentially consistently means that the instructions are executed in the order of the program code. While this makes programs very predictable and would make it easy to reason about a program's behavior, this usually does not hold for performance reasons.

Modern Central processing units (CPUs) make use of weak memory models. Weak memory models weaken the constraints of sequential consistency by allowing the compiler and CPU to reorder instructions in ways that do not change the result of those instructions. This allows the CPU to maximize its available resources by, for example, fetching multiple values from memory in parallel or calculating multiple values at once.

Furthermore, it can schedule the execution of slow instructions before faster ones. This has the effect that multiple faster instructions can be done in parallel during the calculation of the slow instruction. Once the slow one is finished, any other instruction that depends on its result can be executed. If, instead, multiple slow instructions were done in parallel, followed by the slow one, all instructions dependent on the slow one now have to wait for them to complete. At the same time, the CPU may run idle since there might be no independent instructions for it to execute in parallel.

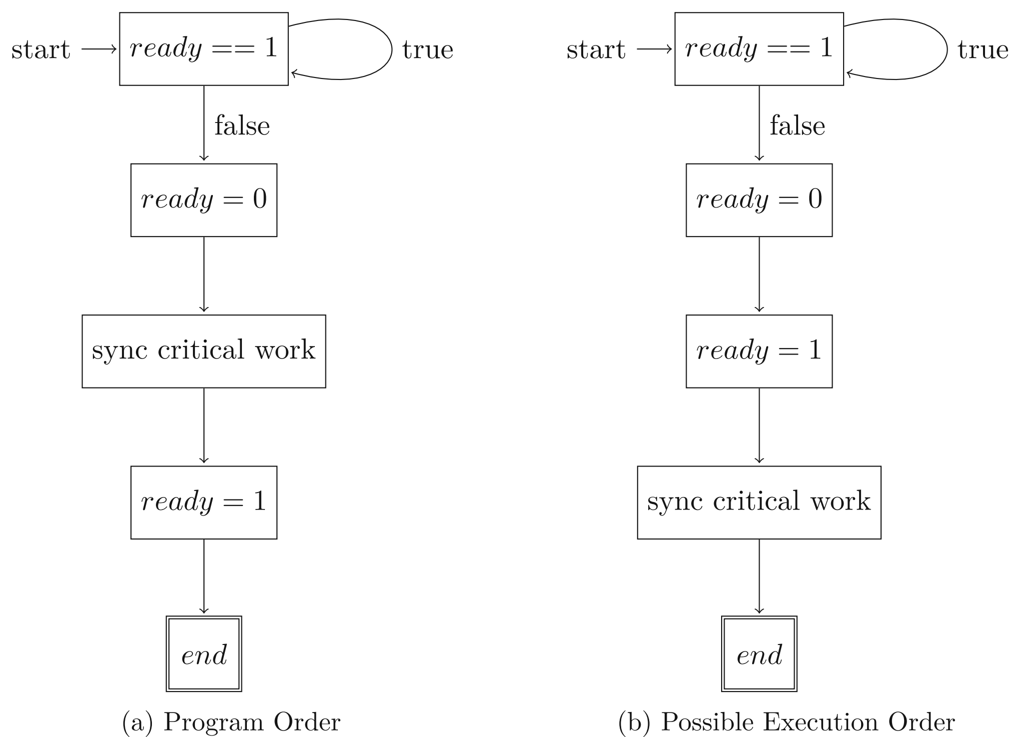
Another case that can cause the execution order to differ from the order written by the programmer is when compile-time optimizations reorder the code. This might be done to better utilize the CPU pipeline, branch prediction, or similar systems used by the CPU to speed up code execution.

A second assumption many developers make is to assume that the effect of every instruction is seen immediately by all other threads and in the same order. This is not the case.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  a = 0
2  b = 0
3
4  thread0 {
5      a = 1
6      print(b)
7  }
8
9  thread1 {
10     b = 1
11     print(a)
12 }
```

Figure 1.1: Race Condition Caused by Delayed Synchronization



(a) Program Order

(b) Possible Execution Order

Figure 1.2: Example How Optimization Might Influence Execution Order

Instead, every thread may observe writes to memory in different orders; this allows, for example, both threads in Figure 1.1 to print 0 since both threads might still read an old value even though the other thread has overwritten it.

The previously mentioned changes in execution order are designed not to change the program in ways that impact the result of it. However, this is only true for single-threaded applications since the optimizations are done per thread. Figure 1.2a shows, assuming

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

the check and setting of *ready* is done atomically, a way of how synchronization-critical code can be wrapped to ensure multiple threads cannot enter the critical section at the same time. Without any optimization, this code would work fine even in multithreaded programs. However, during optimization, the instruction reading and modifying the *ready* variable will be considered independent of the synchronization-critical code, which allows the CPU to execute both simultaneously; the resulting possible execution order can be seen in Figure 1.2b. This order makes it possible for the variable *ready* to be reset to 1 before the critical code is finished and, for this reason, allows multiple threads to enter the section simultaneously which can cause a race condition.

A race condition occurs in a system when multiple threads attempt to modify or access shared resources simultaneously. This might cause the program to be unpredictable since its result can depend on the order in which the threads access the shared resources. Race conditions will be further explained in Section 2.4.

Developers can use atomic instructions and memory fences to ensure that neither the compiler nor the CPU reorders synchronization-critical code. Using them allows the programmer to restrict the compiler and CPU's ability to reorder instructions and ensure data synchronization between threads. Thus, they are critical for the correct execution of the program.

However, setting these memory fences is not always easy, and bugs caused by wrong synchronization in weak memory models are not easily detectable by a developer. Even though the problem is undecidable [AAAK19], many verification tools have been developed to help developers detect problems caused by wrong synchronization [LR09, AAC+12, BM08, AAAK19].

1.1 Aim and Research Questions

The aim of this thesis is to develop a new approach to address the previously mentioned issues using Kronecker Algebra. Kronecker Algebra allows the modeling of reordered instruction, the interleaving of concurrent programs [BB14], as well as the detection of race conditions within the program [Bli15].

The following research questions (RQs) were formulated to support the aim and structure of the research process.

- **RQ1:** Can Kronecker Algebra effectively capture the interactions between threads in concurrent programs following the release acquire model?
- **RQ2:** To what extent can Kronecker Algebra assist in detecting and resolving data races in programs using the release acquire model?
- **RQ3:** What are the limitations and challenges of applying Kronecker Algebra to verify the correctness of concurrent programs under the release acquire model?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1.1.1 Delimitations

The system will prioritize using Kronecker Algebra extensively but not exclusively, to ensure that the time required to verify smaller algorithms remains reasonable.

This also means it will not focus on fixing the state explosion caused by the amount of possible interleavings between the threads.

Lastly, the system will not allow for any dynamic creation of threads during execution.

1.2 Outline

The thesis contains two parts. In the first part, a system that uses Kronecker Algebra to detect race conditions within programs using weak memory models will be developed. The second part will develop a prototype that applies the developed system to programs compiled to LLVM-bytecode. Applying the prototype to several examples shows that the system can correctly detect race conditions. Further, related work will be shown in Section 6.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

Background

This chapter will explain the memory models used in the thesis. Further, a short overview of LLVM, race conditions, and Kronecker Algebra will be given.

2.1 Sequential Consistency

Sequential consistency ensures that all threads see every write-to memory in the same order. This makes working with programs using sequential consistency intuitive since every read is guaranteed to read the newest value of the system. Note that this system does not require that every write is immediately synchronized over all threads, only that the write is synchronized to a thread before that thread reads the value.

On its own, this does not prevent race conditions. Take the following program execution as an example:

- Thread A: read value x
- Thread B: read value x
- Thread A: calculate based on x and save it back to x
- Thread B: calculate based on x and save it back to x

In this example, the calculation of thread A is overwritten by thread B with a value that is based on a read that occurred before thread A wrote that value. Algorithms have been developed to prevent this from happening. Section 2.4 will provide further details.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2 Release-Acquire

A concurrent program that is correct under a sequential memory model is not always correct under a weak memory model. For this reason, different types of synchronization techniques exist. One of those techniques is the release-acquire memory model.

In the release-acquire model, memory accesses are categorized into releases and acquires. A release operation typically stores a value in a shared memory location, while an acquire operation is associated with loading a value from a shared memory location. These operations serve as synchronization points between different memory accesses in the program that define which instructions must be performed before and after.

In the case of a thread performing a release operation on a shared memory location, all memory writes performed before the release are guaranteed to be visible to other threads that perform an acquire operation on the same memory afterward. This ensures that ordering constraints are preserved across threads. Furthermore, the acquire-operator guarantees that no memory writes after the instruction is moved to before the acquire-instruction.

Using these two operators, a programmer can ensure the order of execution to make sure no reordering can occur that threatens the correctness of the concurrent system.

2.3 LLVM

Low-Level Virtual Machine [llvb], or LLVM, is a compiler infrastructure designed to generate code as well as to analyze and optimize it. It was initially implemented for C and C++ but has since been used for many other programming languages like Swift, Rust, and Zig, having compilers that utilize LLVM.

At the core of LLVM is an intermediate representation language that serves as a platform-independent abstraction of program code. This intermediate language allows any optimization techniques implemented for LLVM to be used in any programming language that utilizes LLVM for code generation. It also makes it easier to support different architectures since one compiler, compiling the intermediate language to the target architecture, can be reused by different languages.

2.3.1 Code Structure

LLVM programs are divided into one or multiple modules. In the case of compiling C code using LLVM, each module is equivalent to an object file. Each module consists of global variables and functions.

A function is then divided into one or multiple blocks. A block consists of a list of one or multiple instructions containing exactly one terminator instruction at the end. Terminator instructions alter the control flow of the program, like, for example, `RET` or `BR`. Since these terminator instructions are only allowed at the end of the block, and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

all control flow-altering instructions jump to the beginning of a block, this means that assuming sequential consistency, when a block is entered, all instructions within the block are executed one after the other with no control flow allowing any instruction to be skipped or repeated till the execution of the block is completed.

Some important LLVM instructions for this thesis can be seen in Table 2.1.

It is important to mention that `ALLOCA` returns a pointer to the created address and globals pointers to the values. For this reason, to access or change the values, a `STORE` or `LOAD` is needed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Name	Usage	Overview
STORE	STORE <ty> <va>, ptr <po> STORE atomic <ty> <va>, ptr <po> <or>, align <al>	Stores value in <va> to <po> as type <ty> Atomic version that uses the ordering <or> and alignment <al>
LOAD	<re> = LOAD <ty>, ptr <po> <re> = LOAD atomic <ty>, ptr <po> <or>, align <al>	saves the value of <po> of type <ty> into <re> Atomic version that uses ordering <or> and alignment <al>
BR	BR i1 <con>, label <iftrue>, label <iffalse> BR label <dest>	if value in <con> is 0 jump to block <iffalse> else to <iftrue> Unconditional jump to block <dest>
ICMP	<re> = icmp <co> <ty> <op1> <op2> <re> = icmp samesign <co> <ty> <op1> <op2>	Compares the values <op1> and <op2> of type <ty> using the operator in <co> returning 1 to <re> if comparision is true or 0 otherwise, or a list of such values if <op1> and <op2> are lists. Possible values of <co> can be seen in Table 2.2 guarantees that <op1> and <op2> have the same sign
RET	RET <ty> <va> RET void	Returns value <va> of type <ty> from the function Return from a void function
PHI	<re> = PHI <ty> [<val>, <bl>], ...	saves <val> of type <ty> to <re> if last executed block was <bl>
ALLOCA	<re> = ALLOCA <ty>	allocates memory for type <ty> on the stack, returns pointer to data

Table 2.1: List of Important LLVM Instructions [llva], (Optional Values Are Not Included in Usage)

Name	Operation
eq	equal
ne	not equal
ugt	unsigned greater than
uge	unsigned greater than
ult	unsigned less than
ule	unsigned less or equal
sgt	signed greater than
sge	signed greater or equal
slt	signed less than
sle	signed less or equal

Table 2.2: List of Possible `ICMP` Operators

2.3.2 LLVM API

The LLVM API is part of the LLVM project and contains tools to parse and modify the intermediate language of LLVM. This structure makes implementing new tools for analyzing and optimizing code easier and allows them to be used on all languages that compile to LLVM bytecode.

The LLVM API can load a LLVM code file using the `parseIRFile` function and the function returns an LLVM module object that already contains much-needed information. The following will list important types and valuable functions of the LLVM API.

Module

The module class represents an LLVM module and allows access to the filename using `getName`, but most importantly, allows access to an iterator over all functions using `functions` as well as `globals` to access an iterator over global variables.

Function

It represents both function declaration and definition. Both can be differentiated using the `isDeclaration` function. It also contains an iterator to iterate through all blocks of the function.

BasicBlock

Represents a LLVM Block. Most notably, the class contains an iterator that allows iteration through all block instructions in order. Further, the functions `llvm::predecessors` and `llvm::successors`, when called on a block, return all possible predecessor and successor blocks, which allows to follow the control flow through the blocks easily.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Value

`Value` is the base class for many other classes, including `BasicBlock`, `Instruction`, `GlobalValue`, `ConstantInt`, and `ConstantData`.

It is used as the return value of many functions that have to return many different types. The function `llvm::dyn_cast` allows a program to verify the exact type and cast it so that the return value can be handled differently depending on the exact type.

Use

`Use` is another subclass of `Value` and the parent class of everything that can be passed as an argument to an instruction. This includes all the previously mentioned `Value` subclasses: `BasicBlock`, `Instruction`, `GlobalValue`, `ConstantInt`, and `ConstantData`.

Instruction

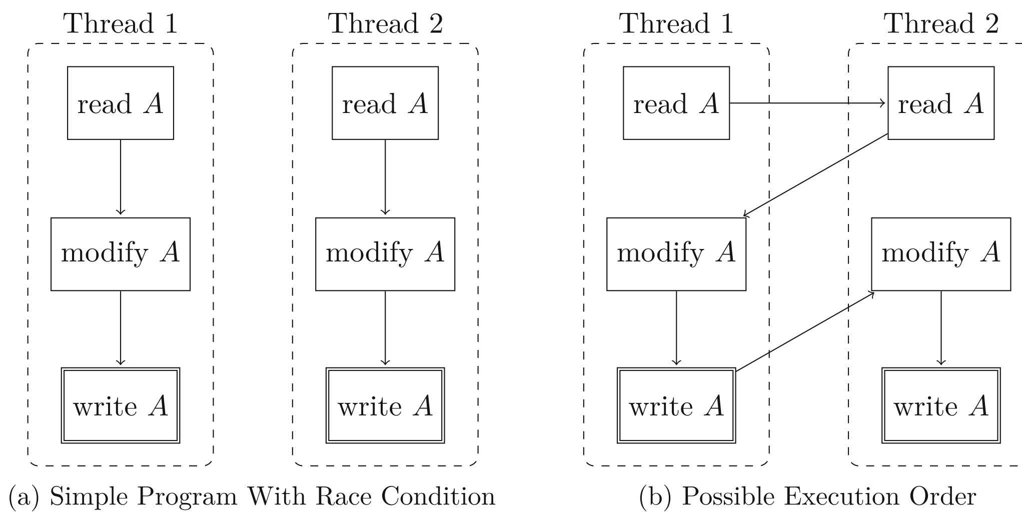
This is the base class for all LLVM Instructions. Some important functions the class provides are:

- `getParent`: returns block instruction is part of
- `operands`: returns iterator over all arguments given to the instruction
- `getOpcode`: allows verification of which type of instruction it is.
- `getNextNode` (`getNextNonDebugInstruction`): returns the instruction that, in code, is written directly after the given one (ignoring debug instructions)
- `getPrevNode` (`getPrevNonDebugInstruction`): returns the instruction that, in code, is written directly above the given one (ignoring debug instructions)

Once cast to the specific subtype representing the exact LLVM instruction, the specific class contains functions to more easily differentiate between the return values of `operands`.

For example, the class `PHINode` contains the functions `getIncomingValue` and `getIncomingBlock` to easily access which value a variable is set to when which `BasicBlock` was the predecessor block, while the class `ICmpInst` contains `getPredicateName` to easily access and differentiate between the different compare operators given in Table 2.2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Simple Program With Race Condition (b) Possible Execution Order

Figure 2.1: Example Read-Modify Race Condition

2.4 Race Condition

Race conditions are a significant problem when dealing with concurrent programs [NM92]. These conditions occur when the outcome of a program depends on the sequence or timing of uncontrollable events. In essence, multiple threads or processes compete for shared resources, and the execution outcome becomes unpredictable due to the timing of their operations.

Usually, the cause of race conditions is the lack of synchronization between concurrent processes that access shared resources like files or variables. When multiple threads or processes attempt to access and modify these resources simultaneously, the final state of the resource becomes dependent on the timing and interleaving of their operations. This leads to inconsistent or erroneous behavior of the program, often resulting in bugs, crashes, or security vulnerabilities.

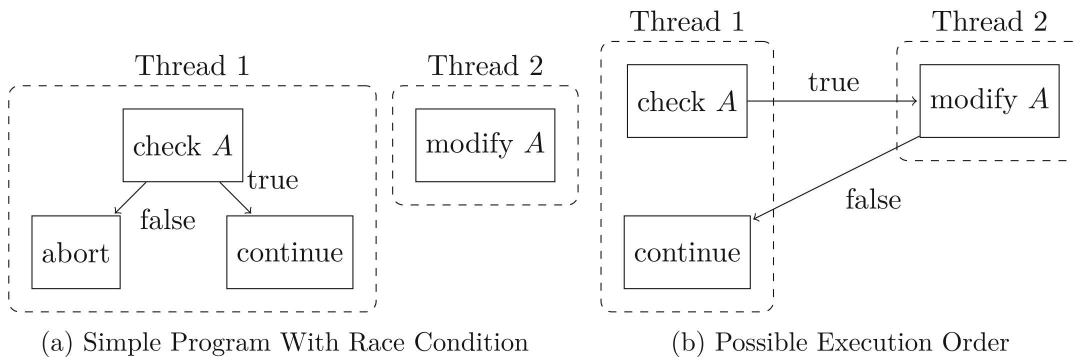
There are different forms of race conditions:

- Read-Modify-Write
- Check-Then-Act

A Read-Modify-Write race occurs when multiple threads access the same resource, modify it, and then save it back. This can result in one of the modifications being lost.

Figure 2.1b shows such an example. In the example, thread 1 and thread 2 read the value of A, followed by thread 1 modifying and saving the values, followed by thread 2 modifying and saving different values.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Simple Program With Race Condition

(b) Possible Execution Order

Figure 2.2: Example Check-Then-Act Race Condition

Since both threads read the original value first, both modify it using that original value. When one thread saves its modified value, the second one completely overwrites it with a value that does not depend on the value written by the previous thread, thus effectively voiding the execution of the first one as if it never occurred.

A Check-Then-Act race happens when a thread checks for a specific condition to hold and then acts depending on the result of the previous check. A second thread may modify resources used in the condition check in between it occurring and the first thread acting on the check, treating the resource as if the condition holds while it no longer does. An example can be seen in Figure 2.2b.

There have been algorithms developed to ensure that such problems cannot arise, and so only one thread has access to a variable from when it reads the value till it saves a calculated value based on it.

Two examples of such algorithms are Dekker's Algorithm [Dij62] and Peterson's Algorithm [Pet81]. Implementations of these algorithms can be seen in Figure 2.3.

2.4.1 Peterson's Algorithm

Peterson's Algorithm [Pet81] works by each thread first setting a variable that signals its intent to enter the critical section, followed by a turn variable that gives priority to another thread. Depending on what `process_1` does during the time `process_0` is setting those variables, one of three scenarios can happen when checking the condition as seen in line 8 in Figure 2.3a.

`process_1` is not trying to enter the critical section, in which case $flag[1]$ is false, and turn equals 1 when `process_0` is checking the while condition. Thus, the condition is false, and `process_0` can enter the critical section.

`process_1` is currently in the critical section. As can be seen in Figure 2.3a, this means that `process_1` has set $flag[1]$ to true. Since `process_0` is setting $turn$ to 1, both parts of the while condition hold, and it awaits `process_1` to exit the critical section and reset $flag[1]$ to false in line 25.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

`process_1` is also trying to enter the critical section. In this case, it depends on which order the *turn* variable is set. If `process_0` sets turn first, then `process_1` will overwrite the value. This causes the while condition of `process_0` to be false and it enters the critical section while `process_1` waits. Otherwise, the roles are reversed, and `process_1` can enter.

2.4.2 Dekker's Algorithm

Dekker's Algorithm [Dij62] uses the same variables, but it differs in when and how they are checked. As can be seen in Figure 2.3b, when it tries to enter the critical section, it only sets its *flag* variable to true in line 6. After this, the same three scenarios can happen.

`process_1` is not trying to enter the critical section, in which case $flag[1]$ is false. Thus, when `process_0` is checking the outer while condition, the condition is false, and the thread skips the loop and enters the critical section.

`process_1` is currently in the critical section. In this case, it depends on the current value of *turn*. If turn is 0, `process_0` is looping in the outer loop till `process_1` exits the critical section. Otherwise, it will enter the if condition and loop within the inner loop till `process_1` exits the section.

`process_1` is also trying to enter the critical section. In this case, both threads will enter the loop, and depending on the current value of *turn*, one of the two threads will enter the if condition, disabling its intent to enter the section to allow the other thread to enter.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```c
bool flag[2] = {false, false};
int turn = 0;

void process_0() {

    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1)
        continue;
    // Begin Critical Section

    // End Critical Section
    flag[0] = false;
}

void process_1() {

    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0)
        continue;
    // Beginning Critical Section

    // End Critical Section
    flag[1] = false;
}
```

(a) Peterson's Algorithm

```c
bool flag[2] = {false, false};
int turn = 0;

void process_0() {

    flag[0] = true;
    while (flag[1]) {
        if (turn != 0) {
            flag[0] = false;
            while (turn != 0)
                continue;
            flag[0] = true;
        }
    }
    // Beginning Critical Section

    // End Critical Section
    turn = 1;
    flag[0] = false;
}

void process_1() {

    flag[1] = true;
    while (flag[0]) {
        if (turn != 1) {
            flag[1] = false;
            while (turn != 1)
                continue;
            flag[1] = true;
        }
    }
    // Beginning Critical Section

    // End Critical Section
    turn = 0;
    flag[1] = false;
}
```

(b) Dekker's Algorithm

Figure 2.3: C Implementation of Peterson's and Dekker's Algorithm (*turn* and *flag* Must Be Sequentially Consistent Atomics)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

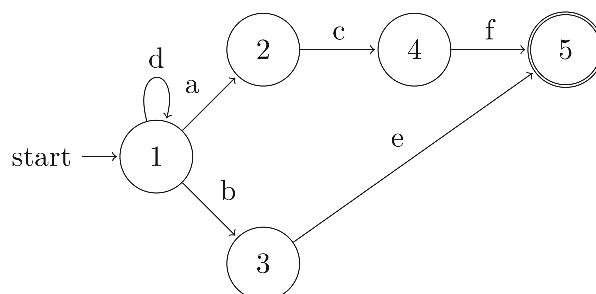


Figure 2.4: FSM F_0

2.5 Kronecker Algebra

Kronecker Algebra expands the standard matrix operations (e.g., matrix addition, matrix multiplication). At its core is the Kronecker Product.

The following is an overview of the necessary parts of Kronecker Algebra as described in the Kronecker Booklet [Bli15].

2.5.1 Finite State Machine (FSM) as Matrix

Every FSM can be converted into an n by n matrix, with n being the number of states in the FSM. This is done by labelling every node with a unique number from 1 to n and then creating an n by n matrix where every transition within the FSM from a starting node with the ID s to an end node with the ID e is added to the matrix by setting the value at row s and column e to the label of the transition and all elements not set in this way being set to 0.

Take the FSM F_0 in Figure 2.4 as an example. This finite state machine can be converted into a 5 by 5 matrix by setting the following positions.

- $(1, 1)$ to value d
- $(1, 2)$ to value a
- $(1, 3)$ to value b
- $(2, 4)$ to value c
- $(3, 5)$ to value e
- $(4, 5)$ to value f

This results in the following matrix:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$\begin{pmatrix} d & a & b & 0 & 0 \\ 0 & 0 & 0 & c & 0 \\ 0 & 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 & f \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

It is important to know that this loses the information about which nodes are considered end and starting nodes. To not lose this information we can create an initial node vector and an end node vector.

The initial node vector of an m by m matrix is a vector of size m in which all elements are set to 0 except for elements with the ID of starting nodes. In the example above this results in the vector $S_{F_0} = (1, 0, 0, 0, 0)$.

Equally, we can create the end node vector by only marking elements that share the ID with end nodes. This results in the end node vector

$$E_{F_0} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

for the example above.

From now on, all 0 values of matrices will be represented with a dot for readability. Furthermore, it is assumed that all used FSMs only have one starting node, which will always be given the id 1. If this is applied to the matrix above, the result is the following:

$$\begin{pmatrix} d & a & b & . & . \\ . & . & . & c & . \\ . & . & . & . & e \\ . & . & . & . & f \\ . & . & . & . & . \end{pmatrix}$$

2.5.2 Kronecker Product

The Kronecker Product is denoted by $\otimes$ and defined as:

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

For a matrix A of dimensions m by n.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

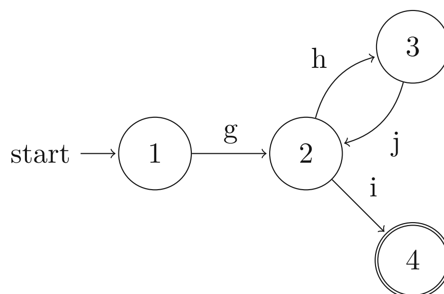


Figure 2.5: FSM F_1

As an example, the Kronecker Product of the FSM in Figures 2.4 and 2.5 is:

$$\begin{pmatrix} d & a & b & . & . \\ . & . & . & c & . \\ . & . & . & . & e \\ . & . & . & . & f \\ . & . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & g & . & . \\ . & . & h & i \\ . & j & . & . \\ . & . & . & . \end{pmatrix} =$$

$$\begin{pmatrix}
. & dg & . & . & . & ag & . & . & . & bg & . & . & . & . & . & . & . & . & . & . \\
. & . & dh & di & . & . & ah & ai & . & . & bh & bi & . & . & . & . & . & . & . & . \\
. & dj & . & . & . & aj & . & . & . & bj & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & cg & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & ch & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & ci & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & cj & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & eg & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & eh & ei \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & ej & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & fg & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & fh & fi \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & fj & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{pmatrix}$$

Its graph representation is given in Figure 2.6 and the resulting FSM with all not reachable nodes removed in Figure 2.7.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

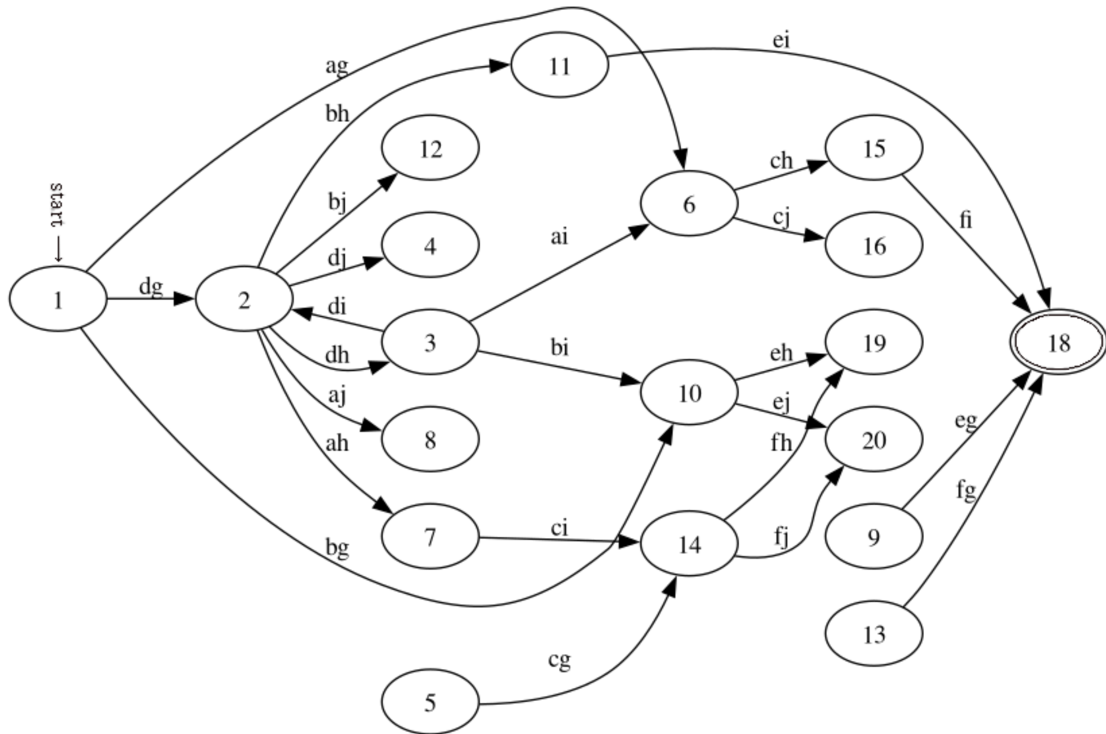


Figure 2.6: Result of the Kronecker Product of the Automata Shown in 2.4 and 2.5

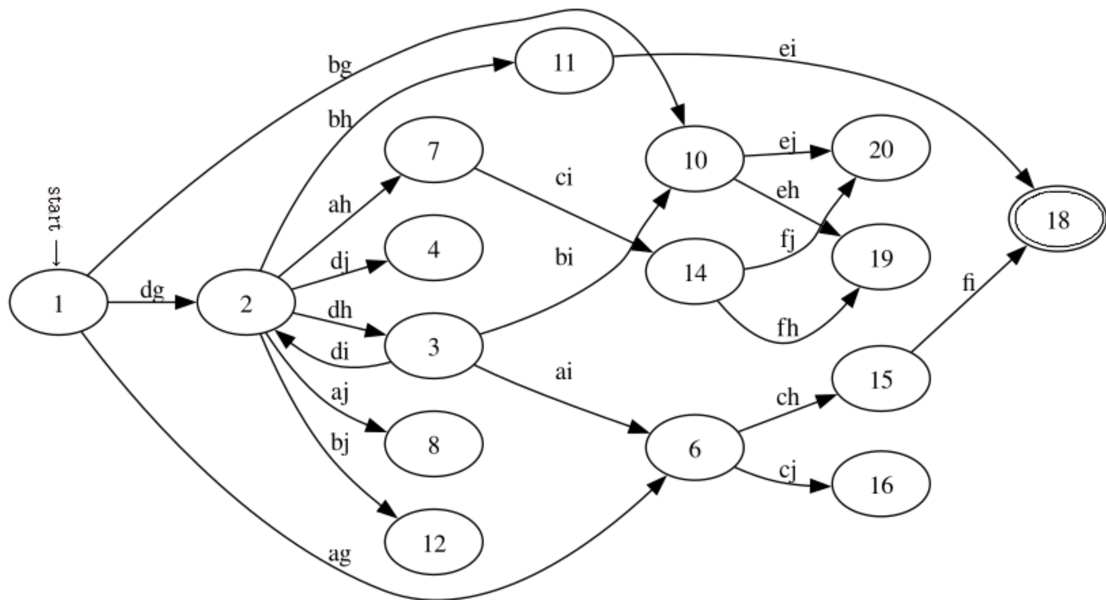


Figure 2.7: Result of the Kronecker Product of the Automata Shown in 2.4 and 2.5 Without Unreachable Nodes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As seen in the figures, this results in a new FSM equivalent to the two automata running in lockstep, which ends as soon as one of them runs into a dead end. But note that not all nodes that have no outgoing edges are valid end nodes of our new graph.

To calculate all valid start and end nodes of a graph $A \otimes B$ we first have to create the initial state vectors S_A and S_B as well as the final state vectors E_A and E_B. Then the initial state vector of $A \otimes B$ can be calculated using $S_A \otimes S_B$ with the final state vector being calculated as $E_A \otimes E_B$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.5.3 Synchronized Kronecker Product

The Synchronized Kronecker Product, denoted for clarity as $\dot{\otimes}$, is equivalent to the standard Kronecker Product with one difference. Every matrix element resulting from the Kronecker Product is set to 0 if its value is not equal to xx, with x being some label name or x otherwise.

As an example, the Kronecker Product of the matrix representations of the automata shown in Figure 2.8 and Figure 2.9 is:

$$\begin{pmatrix} . & a & . & . & . \\ . & . & b & . & . \\ . & . & . & c & . \\ . & . & . & . & c \\ . & . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & a & . \\ . & . & b \\ . & . & c \end{pmatrix} =$$

$$\begin{pmatrix} . & . & . & . & aa & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & ab & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & ac & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & ba & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & bb & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & bc & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & ca & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & cb & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & cc & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & ca & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & cb \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & cc \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \end{pmatrix}$$

While its Synchronized Kronecker Product is:

$$\begin{pmatrix} . & a & . & . & . \\ . & . & b & . & . \\ . & . & . & c & . \\ . & . & . & . & c \\ . & . & . & . & . \end{pmatrix} \dot{\otimes} \begin{pmatrix} . & a & . \\ . & . & b \\ . & . & c \end{pmatrix} =$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

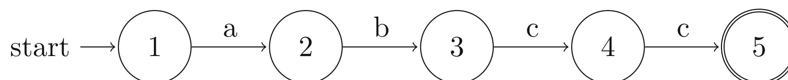


Figure 2.8: FSM F_2

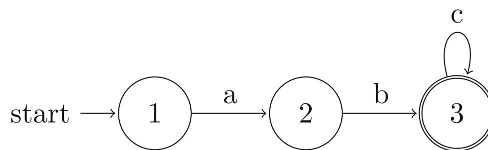


Figure 2.9: FSM F_3

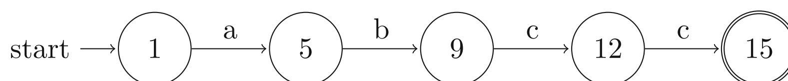
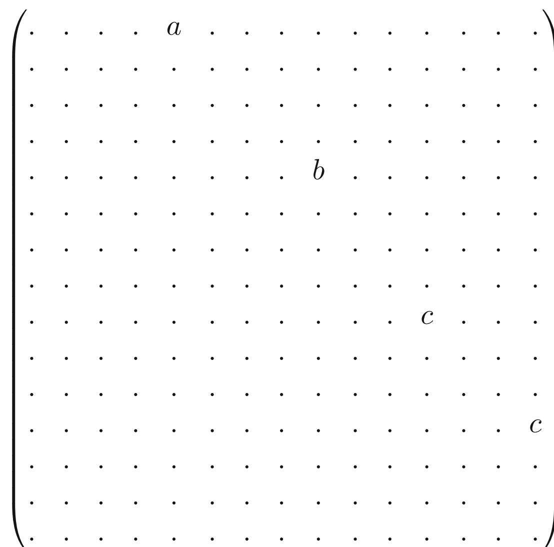


Figure 2.10: Result of the Synchronized Kronecker Product of the Automata F_2 and F_3



The graphical representation can be seen in Figure 2.10. As can be seen, the graphs in Figures 2.10 and 2.8 are equivalent except for the node IDs and thus isomorphic.

However, there are matrices where this is not the case. Take the matrix representations of the Figures 2.11 and 2.9 as an example:

$$\begin{pmatrix} \cdot & a & \cdot & \cdot \\ \cdot & \cdot & c & \cdot \\ \cdot & \cdot & \cdot & c \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \dot{\otimes} \begin{pmatrix} \cdot & a & \cdot \\ \cdot & \cdot & b \\ \cdot & \cdot & c \end{pmatrix} =$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

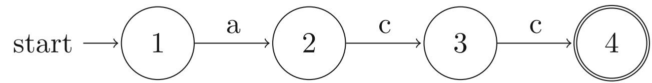


Figure 2.11: FSM F_4

$$\begin{pmatrix}
\cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}$$

Note that the label a is in the 5th column of the result matrix, but the two c labels are in rows 6 and 9, which means they are unreachable. In this example, the resulting graph only contains one edge with the label a, meaning it is not isomorphic to the left operator of the Synchronized Kronecker Product.

When analyzing the two examples, someone might realize that in the first example, the FSM in Figure 2.8 can be simulated by the FSM in Figure 2.9, while in the second 2.9 cannot simulate 2.11 and make the conclusion that the operation results in an isomorphic graph if the right side of the operator simulates the left side and a non-isomorphic graph if not.

This is not necessarily the case. As an example, take the Synchronized Kronecker Product between Figure 2.12 and Figure 2.9:

$$\begin{pmatrix}
\cdot & a & \cdot \\
\cdot & \cdot & b \\
\cdot & \cdot & \cdot
\end{pmatrix} \dot{\otimes} \begin{pmatrix}
\cdot & a & \cdot \\
\cdot & \cdot & b \\
\cdot & \cdot & c
\end{pmatrix} =$$

$$\begin{pmatrix}
\cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

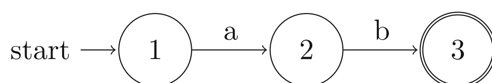


Figure 2.12: FSM F_5

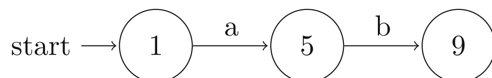


Figure 2.13: Result of Synchronized Kronecker Product of the Automata F_5 and F_3

While the resulting graph in Figure 2.13 is isomorphic to the one in Figure 2.12, the FSM in 2.9 does not halt in the end node when reaching the end node of 2.12.

The final nodes of a graph calculated using the (Synchronized) Kronecker Product can be calculated by building the Kronecker Product of the vectors containing the final nodes of the underlying graphs.

If both conditions are satisfied, graph A is isomorphic to graph $A \dot{\otimes} B$, and both have equivalent final nodes. All execution orders of graph B can also be found in graph A with the difference that graph A might encounter labels not present in graph A.

2.5.4 Kronecker Sum

Kronecker Sum is defined for matrix A and matrix B as:

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

With A being an m by m, B an n by n matrix, and I_k the k by k identity matrix.

As an example, the Kronecker Sum between FSMs shown in Figures 2.14a and 2.14b is:

$$\begin{pmatrix} \cdot & a & b & \cdot \\ \cdot & \cdot & \cdot & c \\ \cdot & \cdot & \cdot & d \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \oplus \begin{pmatrix} \cdot & e & \cdot \\ \cdot & \cdot & f \\ \cdot & \cdot & \cdot \end{pmatrix} =$$

$$\begin{pmatrix} \cdot & a & b & \cdot \\ \cdot & \cdot & \cdot & c \\ \cdot & \cdot & \cdot & d \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} 1 & \cdot & \cdot \\ \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \end{pmatrix} + \begin{pmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \end{pmatrix} \otimes \begin{pmatrix} \cdot & e & \cdot \\ \cdot & \cdot & f \\ \cdot & \cdot & \cdot \end{pmatrix} =$$

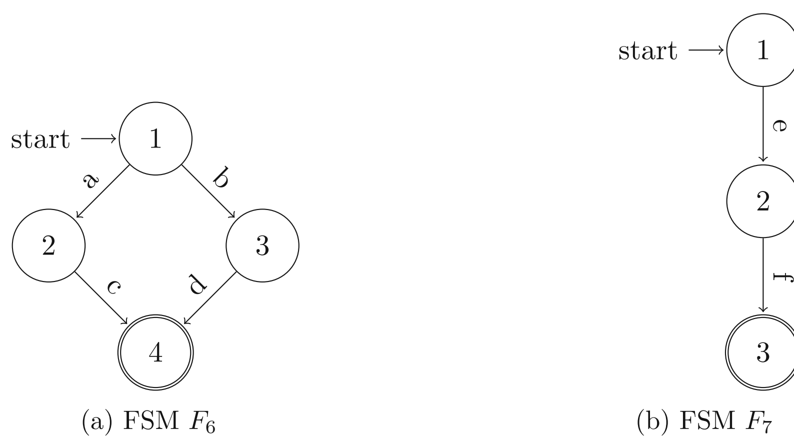
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$
\begin{pmatrix}
\cdot & \cdot & \cdot & a & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & b & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & d & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & d & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & d \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}
+
\begin{pmatrix}
\cdot & e & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & f & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & e & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & f & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & e & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & f & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & e & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & f \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}
=
$$

$$
\begin{pmatrix}
\cdot & e & \cdot & a & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & f & \cdot & a & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & b & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & e & \cdot & \cdot & \cdot & \cdot & c & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & f & \cdot & \cdot & \cdot & \cdot & c & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & e & \cdot & d & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & f & \cdot & d & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & d \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & e & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & f \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}
$$

Similar to the Kronecker Product, this produces a new FSM, a combination of the two input FSMs. However, unlike the Kronecker Product, where the result is that the two input FSMs are executed in lockstep, the Kronecker Sum results in a Finite State Machine equivalent to executing all interleavings of the input FSMs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) FSM F_6

(b) FSM F_7

Figure 2.14: Examples FSM for Kronecker Sum

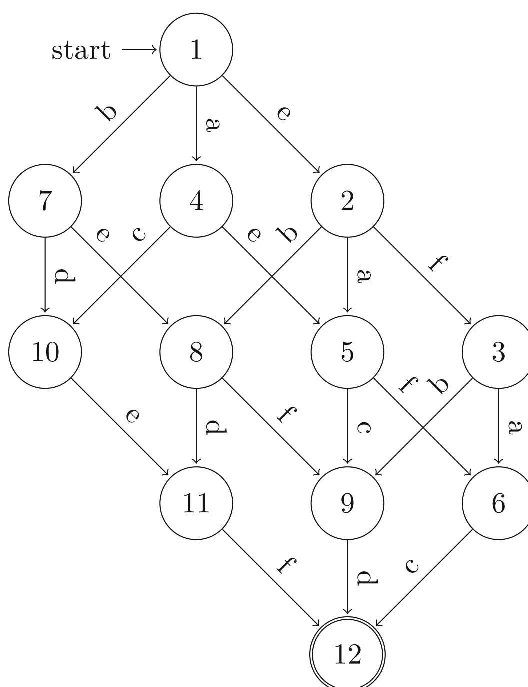
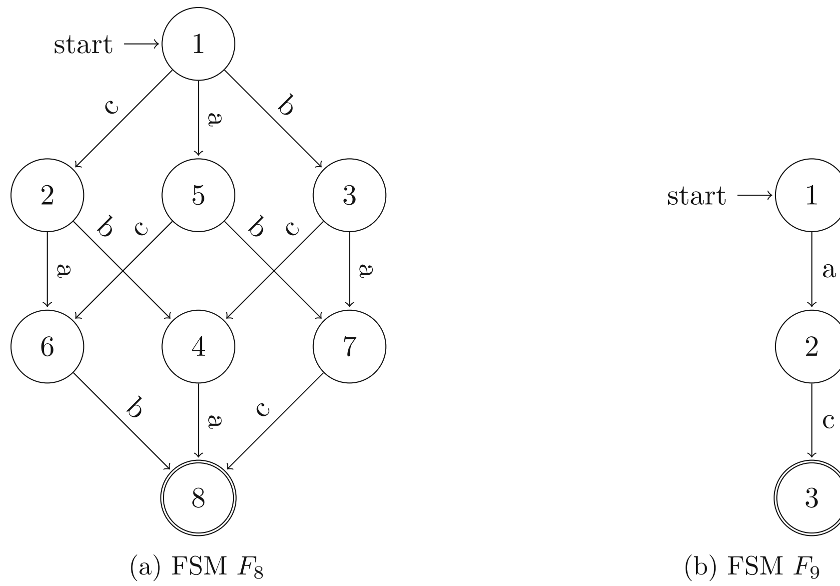


Figure 2.15: Result of Kronecker Sum Between F_6 and F_7

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) FSM F_8

(b) FSM F_9

Figure 2.16: Examples FSM for Kronecker Skip

2.5.5 Kronecker Skip

Kronecker Skip is defined as:

$$A \odot U = A_V \otimes I_m + A_S \dot{\otimes} U$$

With A_V being the matrix A with all elements that exist in U set to 0, A_S being A with all elements that do not exist in U set to 0, and I_m being an identity matrix of size m with m being the size of matrix U.

As an example, take the Kronecker Skip of the graphs in Figures 2.16a and 2.16b:

$$\begin{pmatrix} . & c & b & . & a & . & . & . \\ . & . & . & b & . & a & . & . \\ . & . & . & c & . & . & a & . \\ . & . & . & . & . & . & . & a \\ . & . & . & . & c & b & . & . \\ . & . & . & . & . & . & . & b \\ . & . & . & . & . & . & . & c \\ . & . & . & . & . & . & . & . \end{pmatrix} \odot \begin{pmatrix} . & a & . \\ . & . & c \\ . & . & . \end{pmatrix} =$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

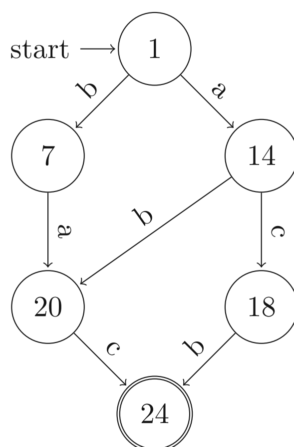


Figure 2.17: Result of Kronecker Skip Between F_8 and F_9

$$\begin{pmatrix} \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} 1 & \cdot & \cdot \\ \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \end{pmatrix} + \begin{pmatrix} \cdot & c & \cdot & \cdot & a & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot \\ \cdot & \cdot & \cdot & c & \cdot & \cdot & a & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a \\ \cdot & \cdot & \cdot & \cdot & c & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \dot{\otimes} \begin{pmatrix} \cdot & a & \cdot \\ \cdot & \cdot & c \\ \cdot & \cdot & \cdot \end{pmatrix} =$$

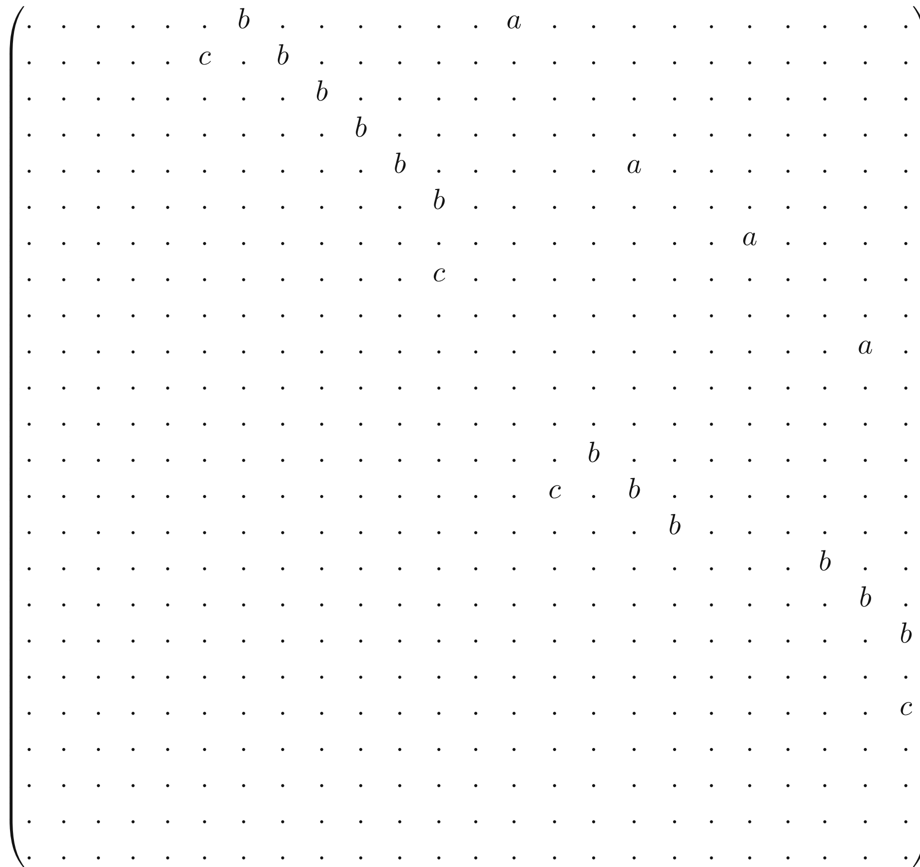
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$
\begin{pmatrix}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix} +
$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$
\begin{pmatrix}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & c & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix} =
$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub
TU WIEN



The graph representation can be seen in Figure 2.17. The graph shows that the resulting FSM is equivalent to the graph left of the operator but with all paths removed in which an edge with the label a does not precede an edge labeled c.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 3

Building the Matrix Representation

The following sections explain how a list of instructions and LLVM blocks can be converted into a matrix representation.

3.1 Program Skeleton

The first step is to generate a matrix that represents the program's basic structure. This means the matrix should contain all instructions and should already respect control flow dependencies. This means the matrix should model the program as if all its instructions are entirely independent.

3.1.1 Generating Blocks

A list of matrices can be generated in the first step, with each matrix representing all possible execution orders within one LLVM block. The simplest version to do this is to calculate the Kronecker Sum of a list of matrices of the form

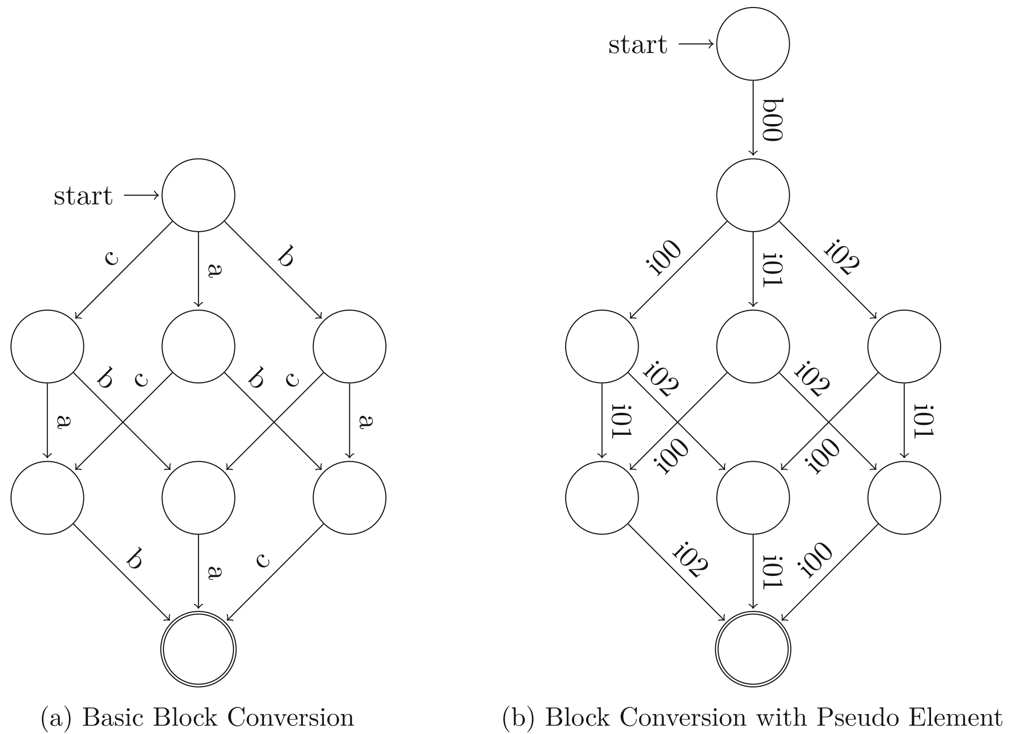
$$\begin{pmatrix} . & x \\ . & . \end{pmatrix}$$

for every instruction x contained in the block. Figure 3.1a shows the result of a Kronecker Sum operation between three matrices with the instructions a, b, and c.

While this way of modeling a block works for many blocks, it has two flaws:

Firstly, modeling many dependencies between instructions in two different blocks is challenging. For this reason, a pseudo instruction will be added that will be labeled using

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Basic Block Conversion (b) Block Conversion with Pseudo Element

Figure 3.1: Example Creation of Blocks

bxx (xx being a unique numeric ID for each block). Further, all regular instructions will, from now on, be labeled using ixx. The purpose of these pseudo instructions is to model dependencies that ensure that all instructions within a block must be executed after a specific instruction. For us to be able to use the label for this purpose, we have to ensure that our pseudo instruction is guaranteed to be executed first within the block. This can be done by taking the Kronecker Sum of all instructions, including the pseudo-instruction, and applying the Kronecker Skip with matrices of the form:

$$\begin{pmatrix} . & bxx & . \\ . & . & iyy \\ . & . & . \end{pmatrix}$$

For each instruction ID, y within the block, and xx is the ID of that block's pseudo-instruction. So, for a block with the ID 00 and the instructions $i00$, $i01$, and $i02$, the block conversion would be done using the following formula:

$$\begin{pmatrix} . & b00 \\ . & . \end{pmatrix} \oplus \begin{pmatrix} . & i00 \\ . & . \end{pmatrix} \oplus \begin{pmatrix} . & i01 \\ . & . \end{pmatrix} \oplus \begin{pmatrix} . & i02 \\ . & . \end{pmatrix} \odot$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$\begin{pmatrix} . & b00 & . \\ . & . & i00 \\ . & . & . \end{pmatrix} \odot \begin{pmatrix} . & b00 & . \\ . & . & i01 \\ . & . & . \end{pmatrix} \odot \begin{pmatrix} . & b00 & . \\ . & . & i02 \\ . & . & . \end{pmatrix}$$

The resulting FSM can be seen in Figure 3.1b.

The second problem is that if we continue with these blocks, we will run into the problem of them not being able to model loops. For now, the goal of the blocks is that a Kronecker Sum of all blocks in a program results in a matrix that models all possible executions if all dependencies are ignored. However, since the Kronecker Sum only creates possible interleavings between the FSM, the result cannot contain a loop if none of the input matrices contain one.

For this reason, if a block is part of a loop, which means there is some path through the program that starts and ends at that block, the possibility of every instruction within the block being succeeded by itself at some point must be modeled. Therefore, the matrices to model the existence of an instruction with the ID xx are modified to the following:

$$\begin{pmatrix} . & ixx \\ ixx & . \end{pmatrix}$$

and

$$\begin{pmatrix} . & byy \\ ixx & . \end{pmatrix}$$

It is used to model the dependency that the instruction ixx must be preceded by the block-specific pseudo instruction byy. A resulting example graph can be seen in Figure 3.2. Note that this introduces the limitation that an instruction has to be executed before the block is executed a second time. This is deliberate and will be explained in Section 3.1.2.

Also note that the block in Figure 3.2 can be simplified by creating a graph that merges both nodes that have outgoing $b00$ edges, meaning all incoming edges of the node that is marked green instead point to the starting node marked in blue. This halves the size of the graph representing that block and makes all graphs using that block smaller. From now on, the “loop-block” will be simplified.

3.1.2 Limitations of Loops

Loops cause a unique challenge when modeling them using matrix representation. When modeling the function in Figure 3.3a, the matrix representation should show that there is the possibility that the calculation within the loop is executed multiple times. To model this, the last edge of the part of the graph that models the inside of the loop can target the node before the check within the while loop is done, as seen in Figure 3.3b.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

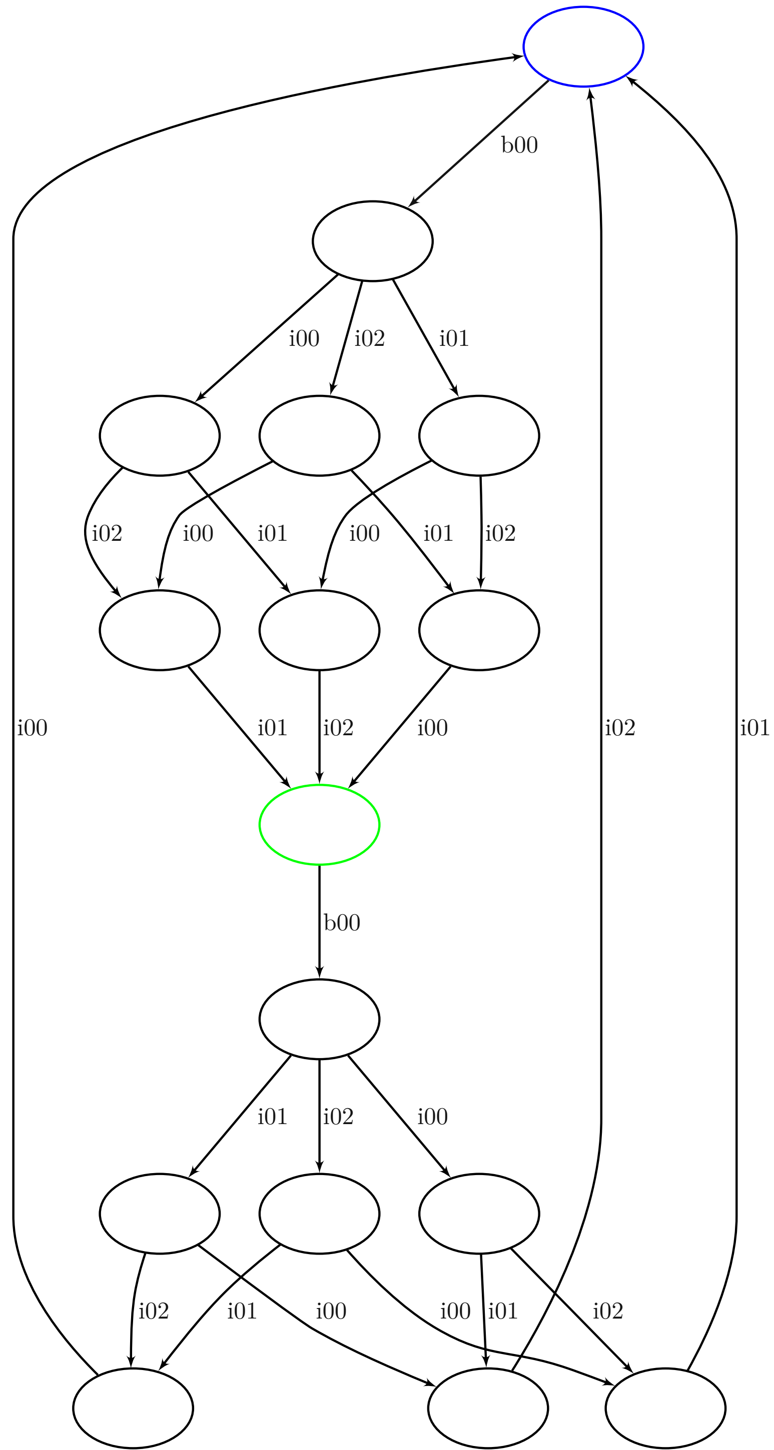
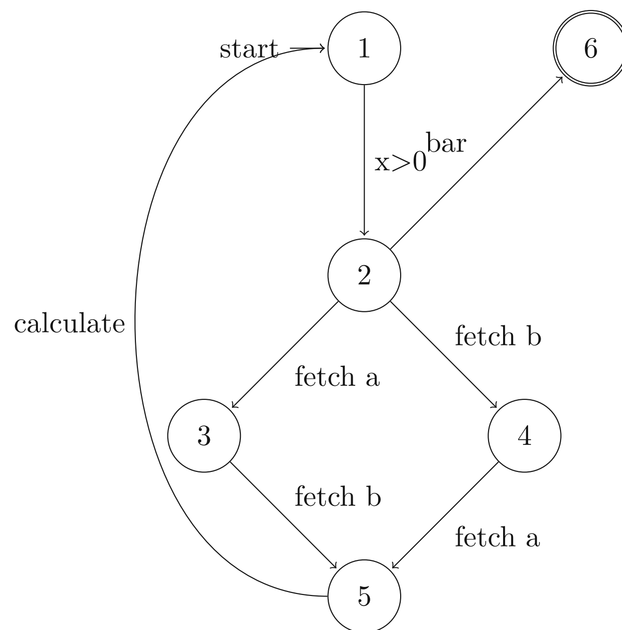


Figure 3.2: Block Conversion with Loop

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1 void foo(){
2     while(x > 0){
3         x = x - a - b;
4     }
5     bar();
6 }
```

(a) Simple loop program



(b) Tree Representation

Figure 3.3: Example of a Loop That Can Be Modelled in Matrix Form

However, not every loop can be modeled so easily. When looking at an example where an instruction is not producing any output used by any other instruction within the loop or its conditions, problems in modeling them occur. Following are two groups of loops where this is the case:

1. Produced value is only used after the loop terminates (3.4a).
2. Produced value is only used by other threads (3.4b).

In the example in Figure 3.3b all instructions within the loop must be finished before a new iteration can start. This means the instructions can only be delayed till the beginning of the next loop iteration. In the Figures 3.4a and 3.4b this is not the case, instead the instructions calculating and modifying y can be delayed till the following iteration of the loop or even till after termination of the loop. This has the effect that not only do we have to model the situation in which our first iteration of the loop executes this instruction, but we also have to model the situation in which this iteration delays it. This has the effect that the model has to show that the second iteration of the loop can either execute the instruction not at all or up to twice, depending on if it delays the execution, executes the instruction of the previous loop, or executes the instruction of this and the previous loop.

Every following iteration suggests that the next iteration might execute the instruction again. Since the modeled graph does not model the exact amount of iterations, and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1 void foo(){
2     while(x > 0){
3         y += 1;
4         x -= b;
5     }
6     print(y);
7 }
```

(a) Loop Instruction is Independent of Other Instruction Within the Loop

```
1 void foo(){
2     while(x > 0){
3         x -= b;
4         y = foobar();
5     }
6     bar();
7 }
```

(b) Produced Value is Independent of Everything That is Executed Within the Thread

Figure 3.4: Examples of Loops Not Easily Modeled

detecting the maximum amount of iterations in many cases is impossible, an infinite amount of iterations would need to be modeled to model all possible execution orders. This would result in an infinite graph and, therefore, an infinite matrix, which is not usable for any further analysis.

The way of generating the blocks is described in the previous sections, results in a graph in which those situations are not possible. Instead, the resulting graph models a program in which all loop instructions must be executed before the next iteration starts. This, of course, might cause false negatives in such cases.

An argument might be made that only modeling a small number of iterations of every loop might be enough to detect the majority of errors. This would allow for unrolling the loops and would reduce the size required to model such problems to a finite space, but this will not be explored any further in this thesis.

3.1.3 Generating the Skeleton

The matrix resulting from the Kronecker Sum of the matrices from the previous steps does not yet respect the program's control flow. Currently, it allows all blocks and their containing instructions to be executed in an arbitrary order. The following describes how a matrix can be created that, when applied using Kronecker Skip, results in a new matrix that respects control flow dependencies.

We start by adding an edge from node 1 to 2 for the first block and labeling the edge with the name of the pseudo instruction of that block. After that, we repeat the following.

If the block has a single successor, add an edge labeled with the successor block name to either a new node, in case the successor block has not been reached before, or a node that already has a predecessor edge with that label.

If the block has multiple successors, add a new edge labeled with the name of the instruction that decides which successor block is chosen to a new node. Then, add an

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

edge for each successor block, labeled with that successor's name, to an existing node with a predecessor edge with that label or a new node if such a node does not exist.

Adding the jump instructions ensures that all instructions of the successor blocks are executed after the condition that decides which successor is chosen. This enforces dependencies caused by control flow.

This skeleton-matrix could be replaced with several matrices of similar style as the one explained in Section 3.2 between the LLVM BR and similar end-instructions and its successor blocks pseudo-instructions. However, merging those dependencies into one skeleton matrix makes setting dependencies easier. Furthermore, as will be seen in Section 4.4, this is an optimization implemented in the prototype that decreases the calculation time by reducing the number of Kronecker operations and allows the removal of multiple BR instructions from the graph.

3.2 Setting Dependencies

This section focuses on how the matrices are constructed to model an instruction $i00$ depending on an instruction $i01$. For the simplest form, this was already shown in Section 3.1.1 where the following matrix

$$\begin{pmatrix} . & b00 & . \\ . & . & i00 \\ . & . & . \end{pmatrix}$$

was used for non-loops and the matrix

$$\begin{pmatrix} . & b00 \\ i00 & . \end{pmatrix}$$

for loops to ensure $b00$ always precedes $i00$.

This was enough to create the blocks since, by the nature of LLVM blocks, all instructions within the block are executed whenever the block is entered without any conditional executions, which would result in some instructions only being executed sometimes or multiple times.

3.2.1 Dependency on Previous Iteration of a Loop

Even if both instructions are within the same LLVM block, the previously shown two "simple" rules are insufficient. Take the code in Figure 3.5a as an example; the print statement and the assignment to a will be combined into the same LLVM block. Since the print statement is outputting the value of the variable a, starting from the second iteration of the loop, it depends on the assignment of the previous iteration.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  a = 0;
2
3  foo{
4      while X:
5          print(a)
6          a = a + 1
7  }
```

(a) Example in Which Instruction Depends on the Previous Iteration of Loop

```
1  a = 0;
2
3  foo{
4      if X:
5          a = 1
6      print(a)
7  }
```

(b) Example in Which Instruction Depends on Instruction Inside Condition

```
1  a = 0;
2
3  foo{
4      a = 1
5      while X:
6          print(a)
7  }
```

(c) Example in Which Instruction in Loop Depends on Instruction Outside It

Figure 3.5: Different Dependency Examples

If we label the print statement $i00$ and the assignment $i01$, then the rule:

$$\begin{pmatrix} . & i01 \\ i00 & . \end{pmatrix}$$

would ensure that the print cannot be moved before the assignment. More precisely, it would ensure that, when executing the code, any $i00$ is always preceded by an $i01$ without another $i00$ being in between them.

But our execution order is $i00, i01, i00, i01, i00, i01, \ldots$, with the first $i00$ not being preceded by an $i01$. Therefore, the matrix would not cause the intended result.

In this specific case, the fix is to ignore that dependency altogether. As explained before, the graph is modeled so that all instructions must be executed before the block can be executed again. This means the construction of the graph already ensures that $i00$ and $i01$ of the first iterations are both executed before the block is reentered and, therefore, that the assignment $i01$ of the first iteration is executed before the print $i00$ of the second iteration.

Once we look outside a single block, we see many more cases in which the above rules are insufficient. Two further examples can be seen in Figures 3.5b and 3.5c.

3.2.2 Dependency With Condition

The basics of Figure 3.5b can be expressed with the block-matrices:

$$\begin{pmatrix} . & b00 & . \\ . & . & i00 \\ . & . & . \end{pmatrix} \begin{pmatrix} . & b01 & . \\ . & . & i01 \\ . & . & . \end{pmatrix} \begin{pmatrix} . & b02 & . \\ . & . & i02 \\ . & . & . \end{pmatrix}$$

combined with the skeleton

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$\begin{pmatrix} . & b00 & . & . & . \\ . & . & i00 & . & . \\ . & . & . & b01 & b02 \\ . & . & . & . & b02 \\ . & . & . & . & . \end{pmatrix}$$

with $i00$ being the condition, $i01$ being the assignment, $i02$ the print statement, and the bxx being block instructions. Applying the skip operator with the basic rule:

$$\begin{pmatrix} . & i01 & . \\ . & . & i02 \\ . & . & . \end{pmatrix}$$

results in the matrix:

$$\begin{pmatrix} . & b00 & . & . & . & . & . & . & . \\ . & . & i00 & . & . & . & . & . & . \\ . & . & . & b02 & b01 & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & b02 & i01 & . & . \\ . & . & . & . & . & . & . & i01 & . \\ . & . & . & . & . & . & . & b02 & . \\ . & . & . & . & . & . & . & . & i02 \\ . & . & . & . & . & . & . & . & . \end{pmatrix}$$

A graphical representation with and without the applied rule can be seen in Figures 3.6a and 3.6b.

In the context of FSM as matrices, note that this is an equivalent but simplified version of the matrix where unnecessary rows and columns were removed and the rest reordered since otherwise, the matrix would be significantly harder to read and not fit into the document.

The path $i00$, $i02$ in the resulting matrix is not correctly modeled. While in row 3 the execution splits between the blocks $b02$ and $b01$, the first one leads to row 4, which does not contain further instructions. This means this path terminates before $i02$ is ever executed. This instruction is missing because the rule says that $i02$ has to be preceded by an $i01$, which, in this case, it would not. Therefore, the calculation removes the instruction.

Thus, the rule has to model that $i02$ can be executed without $i01$ being executed before, but only if $i01$ is bypassed; otherwise, $i01$ has to be contained in the path before $i02$. The corrected matrix is:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

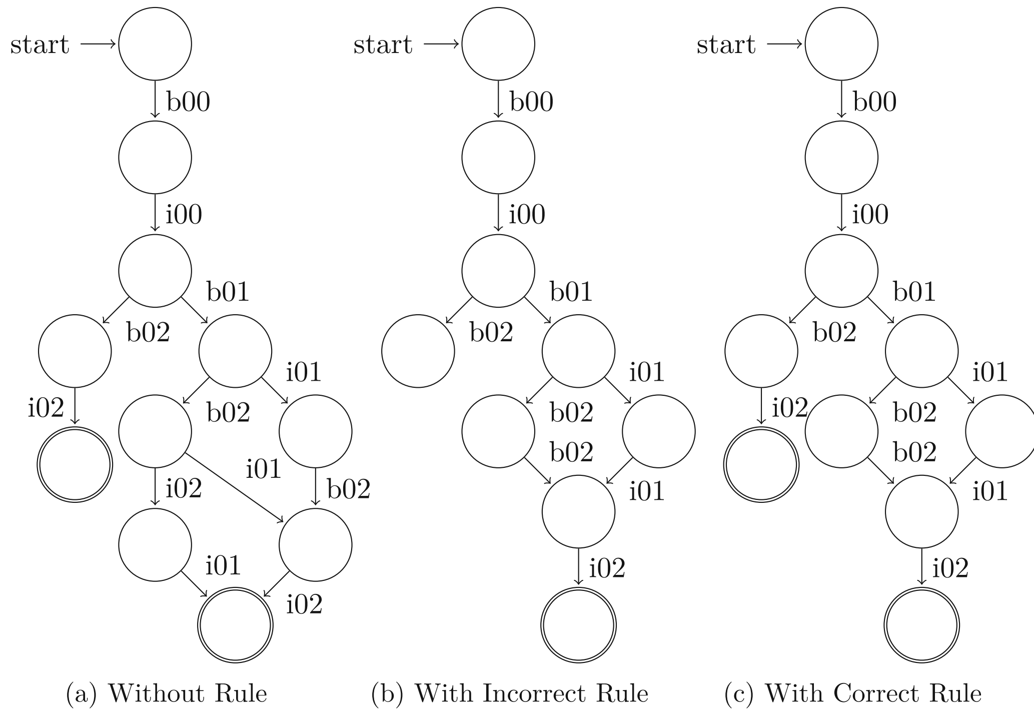


Figure 3.6: Example of Applied Dependencies in Program With Rule

$$\begin{pmatrix} . & b01 & . & i02 \\ . & . & i01 & . \\ . & . & . & i02 \\ . & . & . & . \end{pmatrix}$$

which, when applied with the skip operator, results in the following simplified matrix.

$$\begin{pmatrix} . & b00 & . & . & . & . & . & . & . \\ . & . & i00 & . & . & . & . & . & . \\ . & . & . & b02 & b01 & . & . & . & . \\ . & . & . & . & . & . & . & . & i02 \\ . & . & . & . & . & b02 & i01 & . & . \\ . & . & . & . & . & . & . & i01 & . \\ . & . & . & . & . & . & . & b02 & . \\ . & . & . & . & . & . & . & . & i02 \\ . & . & . & . & . & . & . & . & . \end{pmatrix}$$

3.2.3 Dependency With Only One Part in Loop

The example in Figure 3.5c can be modeled with the matrices:

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$\begin{pmatrix} . & b00 & . \\ . & . & i00 \\ . & . & . \end{pmatrix} \begin{pmatrix} . & b01 \\ i01 & . \end{pmatrix} \begin{pmatrix} . & b02 \\ i02 & . \end{pmatrix}$$

combined with the skeleton

$$\begin{pmatrix} . & b00 & . & . \\ . & . & b01 & . \\ . & . & . & i01 \\ . & b02 & . & . \end{pmatrix}$$

with $i00$ being the assignment, $i01$ being the condition and $i02$ being the print statement. Applying any of the simple rules results in the following simplified matrix:

$$\begin{pmatrix}
. & b00 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & b01 & i00 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & i01 & i00 & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & b01 & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & b02 & i00 & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & i01 & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & i00 & b01 & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & b02 & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & i02 & b01 & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & i00 & i01 & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & b01 & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & i02 & i01 & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & i00 & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & i01 & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & i02 & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & b02 & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & b01 & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & i01 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{pmatrix}$$

Their graph representations are given in Figure 3.7 to make them easier to read. The graph shows that both rules only model the program if it is guaranteed to be terminated after precisely one iteration.

This is because both simple rules demand that an $i00$ precedes each $i02$. Since $i00$ is only executed once, this limits $i02$ to be executed only once. The correct rule has to model that after $i00$, $i02$ can be executed infinitely often or never. For this reason, the rule has to be:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

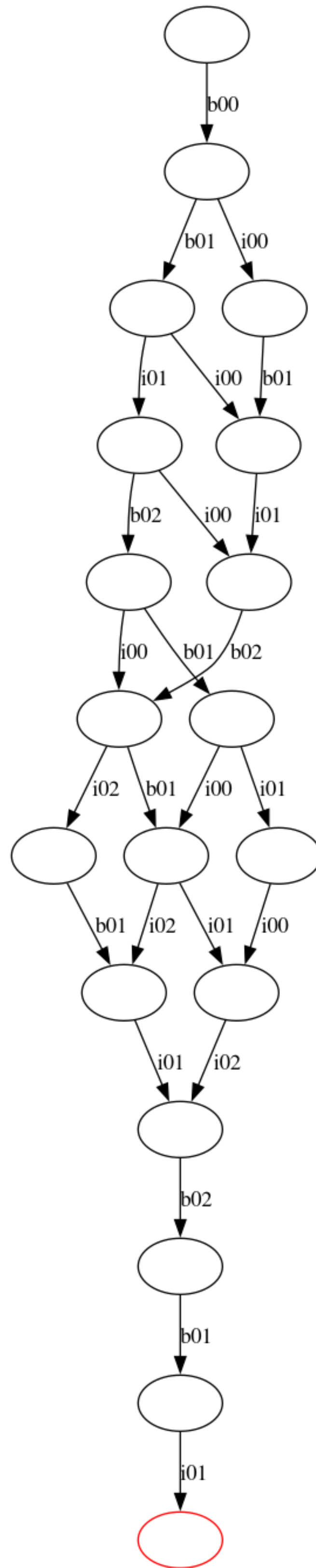


Figure 3.7: Visualization of Figure 3.5c With Applied Simple Rules

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$\begin{pmatrix} . & i00 \\ . & i02 \end{pmatrix}$$

which, when applying it using Kronecker Skip on the original matrix, results in:

$$\begin{pmatrix}
. & b00 & . & . & . & . & . & . & . & . & . & . & . \\
. & . & b01 & i00 & . & . & . & . & . & . & . & . & . \\
. & . & . & . & i00 & . & i01 & . & . & . & . & . & . \\
. & . & . & . & b01 & . & . & . & . & . & . & . & . \\
. & . & . & . & . & i01 & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & b02 & . & . & . & . \\
. & . & . & . & . & i00 & . & b02 & . & . & . & . & . \\
. & . & . & . & . & . & . & . & i00 & . & b01 & . & . \\
. & . & . & i02 & . & . & . & . & . & b01 & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & i01 \\
. & . & . & . & . & . & . & . & . & i00 & . & i01 & . \\
. & . & . & . & . & . & . & . & . & . & . & . & i00 \\
. & . & . & . & . & i02 & . & . & . & . & . & . & .
\end{pmatrix}$$

The graphical representation can be seen in Figure 3.6c

3.2.4 More Complex Rules

From the previous sections, we can derive a set of rules:

- Rule if both instructions are part of the same loop
  If both instructions are part of the same loop then what would be the end node of the rule, meaning the node that has no outgoing edges, is instead replaced with the ID 0 which is the ID of the starting node.
- Rule for conditionals
  If an instruction might be bypassed during execution, meaning one might be executed without the other, then apply the rule as seen in 3.2.2.
- Rule if only one instruction is part of a loop and the second is not
  In that case, apply the rule as seen in 3.2.3.
- Rule if the instruction is dependent on the previous iteration of a loop
  Ignore the dependency.

By combining these rules, more complex rules can be derived.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Prototype

The prototype can be divided into three components. A C++ component that interacts with the LLVM API to extract all required information from the LLVM bytecode. A framework that is written in Ada to handle the Kronecker operation, for the operations, it uses a modified version of the lazy implementation explained the Kronecker Booklet [Bli15]. An analyzer that takes the resulting graph and checks it for possible race conditions.

4.1 Creating Blocks

The first step is to build the blocks needed to apply the system that was explained in the previous section to LLVM. After loading the byte file using `parseIRFile`, the API allows iterating over all functions and their blocks within the module and further over each instruction of each block.

Assigning every block and instruction a unique ID gives the prototype all the information needed to build the blocks. Nevertheless, at this stage, the prototype already filters out instructions that will not be needed later. Since the prototype builds one matrix per function and considers that matrix one thread, it is not important what the function returns, making `RET` instructions uninteresting. Thus, they are removed. `ALLOCA` instructions are also removed. They only create new space on the stack. Since we are only interested in when and which instructions interact with this piece of memory, not when it is created, they can be safely removed. In the case of `BR` instructions, not all are removed. First, conditional `BR` instructions are used since they are used in constructing the skeleton. Second, if a block is part of a loop and would otherwise not contain instructions, the `BR` instruction is kept. Otherwise, this would cause errors for some optimizations that will be explained in Section 4.4.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4.2 Building the Skeleton

The skeleton is created as described in Section 3.1.3. To get the successor blocks of an LLVM block, the function `llvm::successor` is used and, in cases with more than one successor block, the instruction that decides which successor will be executed can be accessed using `BasicBlock::getTerminator`.

4.3 Dependencies

In the next step, the prototype has to find all instruction dependencies. Since the skeleton already handles control flow dependencies, there are three remaining types of dependencies:

- Operator Dependencies
- Memory Dependencies
- Atomic Dependencies

4.3.1 Operator Dependencies

An operator dependency is a dependency between two instructions where one instruction directly uses the output of a second instruction. The `llvm::Instruction` type contains the `operands` function to access all arguments of an instruction. This function returns an iterator over objects of type `llvm::Use`. This object is of subtype `llvm::Instruction` if the argument is calculated directly (without being saved into any local or global variables in between) by a preceding instruction. Which means there is a dependency between those two instructions.

4.3.2 Memory Dependencies

Memory dependencies can be found similarly. Suppose an object returned by `operands` is an `ALLOCA` instruction. In that case, it usually loads or stores to the memory address reserved by the `ALLOCA` instruction. Similarly, if it is of type `llvm::GlobalVariable`, it reads or writes to a global variable. Since the value returned by `ALLOCA` as well as global variables are pointers to their values, it requires `LOAD` and `STORE` instructions to access and modify. Therefore, to track memory dependencies, it is enough to track which values those two operations modify or read.

To access which variable is written or read, the instruction can be cast to either an `llvm::LoadInst` or `llvm::StoreInst`, which both contain a `getPointerOperand` function returning which variable is modified. Note that the prototype assumes no pointer arithmetic occurs and that the memory addresses are not passed to the function using arguments or other means.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To set write dependencies, it iterates through all instructions in written order, and for all `STORE` that write to a global variable or to a memory created using `ALLOCA`, it iterates forward and backward till all paths either have no successor or it reaches a `STORE` writing to the same part of memory. If during this, a `LOAD` to the same memory is encountered, a dependency is set, but it continues to search for a `STORE`. This is because while any modification to execution order should not change what value is read by a `LOAD` since the load does not change the value, multiple reads to the same value can be reordered. Only its order relative to `STORE` operations to the same value has to be fixed.

4.3.3 Atomic Dependencies

The last type of dependencies is the restrictions caused by release-acquire or sequentially consistent atomic instructions.

The `getOrdering` function can detect if a load or store instruction is atomic. The values it can return that are implemented in the prototype are:

- `NonAtomic`
- `Acquire`
- `Release`
- `AcquireRelease`
- `SequentiallyConsistent`

Once an instruction is detected to be sequentially consistent, every preceding and following instruction can be marked as depending on the atomic instruction; this can be repeated till another sequentially consistent instruction is reached. The same is true for acquire and release operations, only that since they only limit the reordering into one direction, for release, only dependencies to preceding instructions have to be set, in case of release only succeeding instructions.

4.4 Optimization

While the above-described way of creating a graph representing all possible execution orders works, it results in a needlessly big matrix, which in the prototype results in very long calculation times, even with minimal programs.

As seen by the definition of the Kronecker Product in Section 2.5.2, the Kronecker Product between an $m \times m$ and an $n \times n$ matrix results in a new matrix of size $m \cdot n \times m \cdot n$. This is also true for the Kronecker Sum and Kronecker Skip because of the use of the Kronecker Product in their definitions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Since all our dependencies are modeled by a matrix of at least size 2×2, every dependency at least doubles the size of the resulting matrix. For this reason, reducing the number of Kronecker operations will significantly reduce the time needed for calculation.

Some ways of reducing dependencies have already been mentioned in the sections explaining the detection of dependencies.

One way of ultimately reducing the problem of slow calculations would be to circumvent Kronecker Algebra and build the matrix concerning all dependencies in code, which would remove the slowdown caused by big matrix calculations. While this would work, this thesis's point is to explore its building using Kronecker Algebra. Therefore, only a limited number of optimizations are used. The following describes some optimizations that are used in the prototype.

Another way of reducing the amount of dependencies was already described in Section 3.1. In many cases, it merges a significant amount of dependencies needed to ensure the correct control flow between blocks into a single matrix. Similarly, we can reduce the amount of dependencies by building the diamond-shaped block matrix in code. This also allows it to model a block part of a loop without the inefficiency explained in 3.1.1 and shown in Figure 3.2.

A second optimization in the prototype is to respect trivial dependencies when generating blocks. A dependency between two instructions is considered trivial if both are within the same block. So when generating the block that contains the instructions $i00$ and $i01$, and there exists a dependency that enforces $i00$ has to execute before $i01$, then instead of generating all possible paths the way it was described before, it only generates the paths in which $i00$ is reached before $i01$.

This not only reduces the size of the block matrix by reducing the number of nodes required to model it but also reduces the number of skip operators since the dependency does not have to be calculated later on, reducing the matrix size even further.

The last form of optimization that is included in the prototype makes use of the pseudo instructions as well as the way the skeleton is built. The acquire memory fence ensures no instruction after it is moved in front of the fence. This means there must be a dependency from every dependency after the fence to the fence itself.

This causes many dependencies that can be simplified by creating a dependency from the succeeding block to the fence. By construction, all block instructions depend on the block's pseudo instruction and all succeeding blocks due to the dependencies from the control flow. This means that all dependencies from an acquire memory fence to an instruction outside the block of the memory fence can be simplified by dependencies between the direct succeeding blocks to the memory fence. The same can be done for sequential memory fences.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.5 Collecting Further Information

Extracting the dependencies from the program is not enough to perform the complete analysis. The phase that will be explained in Chapter 4.8, information about which of the IDs represents which instructions, what arguments it, as well as what arguments it is called with, and whether it is a global variable, local variable, or constant value is needed.

How to access the needed information has already been mentioned. One important argument that was not mentioned is constant values. In LLVM, those are represented using the class `llvm::ConstantData`. Therefore, the `llvm::dyn_cast` function can be used to check if an argument of type `Use` is constant data. The prototype currently only checks for integer values. Thus, a cast to `llvm::ConstantInt` is performed instead, which allows it to extract the number value of the argument using `getSExtValue`.

4.6 Detecting Dependency Type

As explained in Section 3.2, there is not one matrix that works for all dependencies that have to be set. For this reason, the program needs to collect some data to check which rules from Section 3.2 it has to apply.

These checks can be done by simply analyzing if the (both direct and indirect) successors and predecessors satisfy certain conditions.

- Dependencies part of the same loop: Start at the block that contains one of the instructions and follow its successors, if a path can be found that ends at the same block and contains the block of the second instruction the two instructions are part of the same loop.

- Dependency part of independent loop: Do the same as above, but this time, check if a path can be found that ends at the same block but does not contain the block of the second instruction.

- Dependency can be bypassed: Check if, starting from the instruction, there is a path that terminates by reaching an end block but does not contain the instructions that depend on it.

- Instructions are not part of the same block: Trivially check if the parent blocks of the instructions are the same.

The rules in Section 3.2 can be used to build a matrix fitting each dependency; using these conditions,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.7 Creating the Graph

As mentioned, the prototype uses an Ada library using the lazy implementation as explained in the Kronecker Booklet [Bli15]. The lazy implementation intends to reduce the amount of unnecessary calculated values. While this cuts down on calculation speed, it still causes the Kronecker calculation to be a significant bottleneck.

Table 4.1 shows the time required to calculate:

$$\begin{pmatrix} . & a & . \\ . & . & b \\ . & . & . \end{pmatrix} \odot \begin{pmatrix} . & a & . \\ . & . & b \\ . & . & . \end{pmatrix} \odot \begin{pmatrix} . & a & . \\ . & . & b \\ . & . & . \end{pmatrix} \odot \ldots$$

The table shows exponential time scales with the number of skip operators. Section 5 shows that the amount of dependencies, even with optimizations, is already greater than the maximum 15 shown in the table. This means that, even with better hardware, this way of calculating the control flow graph will hit a limit at small examples.

The problem is that the lazy algorithm used by the library is not very good at handling situations that make the internal tree used in the calculation very tall instead of wide, since then the number of values that have to be calculated repeatedly outweighs the number of calculations saved by not calculating values that are not required.

While the Kronecker Sum is associative and can be reordered to form a wider tree, this is not true for Kronecker Skip.

The prototype uses the library, but after each calculation step, it is forced to resolve all rows of the matrix that are reachable in the context of considering it an FSM to optimize the calculation. Using those values creates a new matrix with all unnecessary, unreachable rows and columns removed. Any following operation is done with this new matrix.

This still has the effect that many (but not all) unnecessary values are never calculated while also not recalculating values repeatedly. Table 4.2 shows that this system can handle significantly more skip operations than before, but as can be seen in Tables 4.3 and 4.4, the inefficiency when calculating Kronecker Sums remains.

The calculation performed for the last two mentioned tables is:

$$\begin{pmatrix} . & a \\ . & . \end{pmatrix} \oplus \begin{pmatrix} . & a \\ . & . \end{pmatrix} \oplus \begin{pmatrix} . & a \\ . & . \end{pmatrix} \oplus \ldots$$

While the lazy algorithm is better equipped at handling the Kronecker Sums when the internal tree created is very flat, meaning instead of $A \oplus B \oplus C \oplus D$ calculate $(A \oplus B) \oplus (C \oplus D)$, this is still significantly slower than the above-described method. This will be shown in Section 7.1.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
time	0.018	0.018	0.019	0.019	0.020	0.019	0.027	0.038	0.056	0.108	0.207	0.416	0.840	1.873	4.190

Table 4.1: Average Execution Time in Seconds of Library for n Kronecker Skips

n	10	20	30	40	50	60	70	80	90	100
time	0.181	0.314	0.449	0.581	0.716	0.853	0.982	1.120	1.256	1.400

Table 4.2: Average Execution Time in Seconds of Custom Calculation for n Kronecker Skips

n	1	2	3	4	5	6	7	8	9	10	11	12
time	0.018	0.018	0.019	0.027	0.046	0.097	0.229	0.550	1.377	3.367	8.267	20.200

Table 4.3: Average Execution Time in Seconds of Library for n Kronecker Sums

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
time	0.024	0.041	0.056	0.068	0.084	0.109	0.137	0.192	0.301	0.526	1.030	2.067	4.370	9.387	21.183

Table 4.4: Average Execution Time in Seconds of Custom Calculation for n Kronecker Sums

```
1 thread0 {
2     a = 1;
3 }
4
5 thread1 {
6     while a != 1:
7         continue
8     foo()
9 }
```

Figure 4.1: Example Where Kronecker Sum of Threads is Not Enough

The previous sections provide all the information to the prototype to create all matrices needed; the prototype now calculates the Kronecker Sum of all block matrices within one function, followed by the Kronecker Skip of all found dependencies to create a graph representation of a single thread using the way of calculating described. Another Kronecker Sum with a second thread gives a new graph that shows all possible orders in which the two functions can be executed if they are entirely independent. However, if the two threads are independent of each other, the problem of detecting bugs caused by weak memory models is trivial since they are caused when multiple threads interact.

The graph is then passed to the analyzation unit described in the next section for less trivial examples.

4.8 Detecting Errors

For example, thread 0 sets global variable a to value 1 when the threads interact. At the same time, thread 1 waits till variable a takes on the value 1, as seen in Figure 4.1, the Kronecker Sum of their matrices will allow for the path, thread 1 loads value for a, thread 1 compares the loaded value with 1. Thread 1 executes foo, even though that path should be impossible, assuming there is no third thread writing to a.

An interpreter is executed on the graph from the starting node to respect interactions like this. If there is more than one successor node, then a copy of the interpreter, including its current state, is run on each of the successor states.

The state of the interpreter contains the following:

- the ID of the node in which it was reached
- a list of variable-value pairs saving the current variable assignments
- the last two blocks that were reached for each thread
- the next block that has to be executed for each thread

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The interpreter deals with many trivial cases like add, greater than, equal, and similar operations by simply calculating the result based on the current state and modifying the current state accordingly. The following will explain how the interpreter handles more interesting operations.

4.8.1 Conditional Jumps

LLVM has multiple types of instructions that alter the control flow. Since the BR instruction is the only required for the examples in Section 5, it will be explained here. Note that other control flow-altering instructions can be handled similarly.

The BR instruction comes in two forms: unconditional and conditional. The unconditional BR instruction can be ignored since the skeleton used in constructing the graph already guarantees the following next pseudo-instruction following an unconditional BR instruction, which must be the pseudo-instruction representing the block it would jump to.

This is different in the case of the conditional BR instruction. Since the construction of the graph does not take into account what values the variables that the jump instruction depends on have, there will be a path where the jump instruction is followed by the, in this situation, false block as well as a path where the correct block follows it.

For this reason, when reaching a BR instruction, the interpreter saves which block should be entered next. If the interpreter then reaches a pseudo-instruction that does not represent the block that should follow the jump instruction, that specific interpreter terminates, and its result is ignored.

Note that due to the construction of the skeleton containing the conditional BR instruction, with dependencies to all successor blocks, as well as the construction of the blocks guaranteeing that the BR instruction of a block always depends on the pseudo instruction of that block, a situation in which two BR instructions follow each other without a pseudo instruction in between is not possible. For this reason, only saving one successor block per thread is enough.

4.8.2 `PHI` Instructions

The LLVM `PHI` instruction returns a specific value depending on which block the second last block entered within the thread of the `PHI` instruction. For this reason, the prototype always has to save the last two pseudo instructions that got reached in each thread. When a `PHI` instruction is executed, the value can be set depending on the saved last block.

Due to the skeleton used in the construction of the graph, which guarantees the control flow between blocks is equivalent between the pseudo instruction in the graph and the blocks in LLVM byte code, this models `PHI` instructions equivalent to how they are handled in code.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4.8.3 Preventing Infinite Loops

Like every interpreter running on Turing-complete languages, there is a chance it runs into an infinite loop. If we take the example given in Figure 4.1, the resulting graph from this program will contain the cycle that indefinitely checks for a not to have the value 1 without the value of a ever being set.

The prototype saves a list of all states it has reached to counteract this situation. It terminates if any interpreter reaches a state that has been seen before, independently of whether it was reached by the interpreter itself or any copy.

For the examples in Section 5, this would be enough. All examples given have a finite amount of variables that take on only a finite amount of values, with the number of nodes in the graph and the number of blocks being finite in all programs. This means the amount of states is also finite. However, this might not necessarily be the case (for example, any program that contains an (infinite) loop that increases the value of a variable).

A simple counter can be added that counts the number of steps an interpreter has taken to counteract this, with the interpreter terminating when it has reached a maximum explored depth.

4.8.4 Detecting Race Conditions

In the examples given in Section 5, the critical section is represented by a single store operation to a non-atomic global variable, which LLVM compiles to a single `STORE` operation each. Suppose a path is found where those two instructions are executed back to back, meaning without any other instruction between them. In that case, there is no possibility that an instruction enforces which of the two comes first. Note that the two instructions are from different threads and, for this reason, do not have direct data dependencies between them that would have been set using Kronecker Algebra, meaning their execution order is not restricted relative to each other. This further means that if the critical section would be made out of more than one instruction (as an example, a typical three-instruction load value, modify value, and save value could be such a critical section), those instructions could be reordered with the instructions of the critical section of the second thread, resulting in a race condition. If a situation like this is found by the interpreter, the path taken so far is displayed and the interpreter stopped.

The prototype is not designed to detect which instructions are critical; therefore, it takes two instructions representing the critical sections as input. Each copy of the interpreter saves the path it has so far taken; if it reaches a point in which both the last reached instruction and the current one are equal to the given critical sections, then it has found a possible execution order in which the program runs into a race condition and outputs the taken path.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Examples

In the following section, the results of the prototype when applied to some examples will be presented.

5.1 Peterson's Algorithm - Sequential Consistent Version

It is expected that no problems will be found in the sequentially consistent version of Peterson's Algorithm since it is known to be correct. The implementation's C and LLVM codes can be seen in Figures 5.1 and 5.2. Since the prototype is not implemented to handle arrays, the array holding the intent to enter is split into two separate variables.

The first stage is to build the graph for all possible execution orders. The following list lists all dependencies the prototype detects, excluding those it optimizes away. Note instruction ix is the x'th instruction of the function as found in Figure 5.2 with numbering starting at 0 and with the second thread continuing to number after the last instruction of the first thread. When applying this system to the code in Figure 5.2, the critical instructions get the labels $i12$ and $i27$.

Dependencies in the first thread caused by sequential instructions are:

i00 – i01	i01 – b01	i03 – i04	i03 – i05	i03 – b02	i03 – b03	i01 – i03
i11 – i03	i06 – i07	i06 – b03	i05 – i06	i04 – i06	i03 – i06	i12 – i13
i09 – i13	i10 – i13					

Dependencies caused by dependency chains are:

i03 – i04	i04 – i05	i06 – i07	i07 – i09	i09 – i10

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
atomic_int flag_0 = 0;
atomic_int flag_1 = 0;
atomic_int turn = 0;
int critical;

void process_0() {

    flag_0 = 1;
    turn = 1;
    while (flag_1 && turn == 1)
        continue;

    critical = 17;

    flag_0 = 1;
}
```

Figure 5.1: One Thread of Peterson's Algorithm in C

Dependencies caused by reading memory another instruction writes to, as well as dependencies caused by writing to memory other reads from:

i01 – i06 | i00 – i13

Dependencies for the second thread look the same in all examples.

A graphical representation of the resulting matrix can be seen in Figure 5.3. Since the code of the second thread is almost identical, the graph looks identical to this graph. As to be expected, when the interpreter is run on the Kronecker Sum of this graph, it cannot find any errors, meaning it does not find a valid path in which the two instructions that represent our critical section, $i12$, and $i27$, can be executed back to back.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
@flag_0 = dso_local global i32 0, align 4
@flag_1 = dso_local global i32 0, align 4
@turn = dso_local global i32 0, align 4
@critical = dso_local global i32 0, align 4

define dso_local void @thread1() #0 {
  store atomic i32 1, ptr @flag_0 seq_cst, align 4
  store atomic i32 1, ptr @turn seq_cst, align 4
  br label %1
1:
  %2 = load atomic i32, ptr @flag_1 seq_cst, align 4
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %4, label %7
4:
  %5 = load atomic i32, ptr @turn seq_cst, align 4
  %6 = icmp eq i32 %5, 1
  br label %7
7:
  %8 = phi i1 [ false, %1 ], [ %6, %4 ]
  br i1 %8, label %9, label %10
9:
  br label %1, !llvm.loop !6
10:
  store i32 19, ptr @critical, align 4
  store atomic i32 0, ptr @flag_0 seq_cst, align 4
  ret void
}
```

Figure 5.2: LLVM Version of Peterson's Algorithm

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

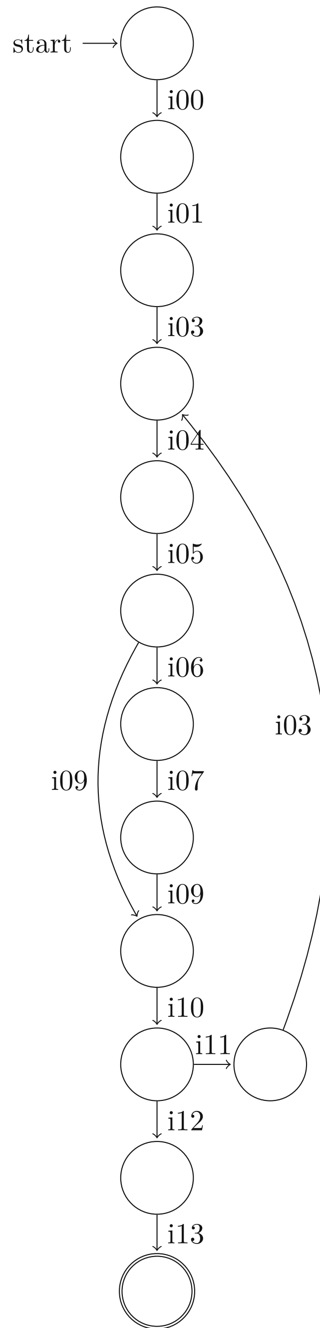


Figure 5.3: Single Thread Graphical Representation of Peterson's Algorithm With Pseudo Instructions Removed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

5.2 Peterson's Algorithm - Release-Acquire Version

The second example will be made with a modified version of Peterson's Algorithm that uses Release-Acquire semantics instead. The only changes that have been made are to change all atomic `STORE` instructions to use release memory ordering and for atomic `LOAD` instructions to use acquire memory ordering. The modified LLVM code can be found in Figure 5.4.

The dependencies found that are caused by release-acquire instructions are the following:

i00 – i01	i03 – i04	i03 – i05	i03 – b02	i03 – b03	i06 – i07	i06 – b03
i12 – i13	i10 – i13	i09 – i13				

Dependencies caused by dependency chains and dependencies caused by memory reads/writes stay the same.

As can be seen, release acquire already has significantly fewer dependencies than the sequential version. The resulting graph can be seen in Figure 5.5. Immediately, this graph is significantly less restrictive than the one resulting from the sequential consistent version of Peterson's Algorithm.

The analyzing part of the prototype now returns multiple error paths. One of which is the following (pseudo instructions have been removed for readability):

$i03, i04, i05, i09, i10, i18, i19, i20, i24, i25, i27, i12$

As can be seen, this path reaches a state where the two critical instructions, $i27$, and $i12$, are executed one after the other, meaning the interpreter has reached a state where both critical sections can be entered simultaneously. This happens because it skips the execution of $i00$ (setting $flag_0$) and $i01$ (setting $turn$) in `thread1` till it has already checked the condition of the while-loop and evaluates it as false. Since neither $i00$ nor $i01$ have been executed, $turn$ and $flag_0$ still have their default values of 0. Since the condition for the second thread to enter the critical section is already satisfied if $flag_0$ is 0, it can enter the critical section. Now that both threads are within the critical section, they can, back to back, execute the critical instructions $i12$ and $i27$. This can also be seen in Figure 5.5.

So the problem is that `thread1` can delay setting the flag and the $turn$ value. But why? The reason is quite simple: setting the values are both release operations, and while this means that no instruction before the release can move to after it, it does not limit any successor instructions to be moved before it. Hence, such instructions can be moved forward till either:

- Another instruction with memory dependencies to $turn$ or $flag_0$ is reached.
- A release instruction limits its delay.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
@flag_0 = dso_local global i32 0, align 4
@flag_1 = dso_local global i32 0, align 4
@turn = dso_local global i32 0, align 4
@critical = dso_local global i32 0, align 4

define dso_local void @thread1() #0 {
  store atomic i32 1, ptr @flag_0 release, align 4
  store atomic i32 1, ptr @turn release, align 4
  br label %1
1:
  %2 = load atomic i32, ptr @flag_1 acquire, align 4
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %4, label %7
4:
  %5 = load atomic i32, ptr @turn acquire, align 4
  %6 = icmp eq i32 %5, 1
  br label %7
7:
  %8 = phi i1 [ false, %1 ], [ %6, %4 ]
  br i1 %8, label %9, label %10
9:
  br label %1, !llvm.loop !6
10:
  store i32 19, ptr @critical, align 4
  store atomic i32 0, ptr @flag_0 release, align 4
  ret void
}
```

Figure 5.4: LLVM Version of Peterson's Algorithm Using Release Acquire

In the case of the first, the only other instruction that accesses the same memory is $i06$, the load instruction that fetches the value of *turn* for the condition. The problem is that this instruction is never reached in this particular path since $flag_1$ being false already satisfies the condition to enter the critical section, meaning the check for the value of *turn* is bypassed.

The only other release instruction for the second point is after the critical section. This gives other threads the time to enter the critical section before `thread0` sets the values to prevent them.

Since both cases do not prevent $i00$ and $i01$ from being delayed till after the critical section, the error path given by the prototype is correct.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

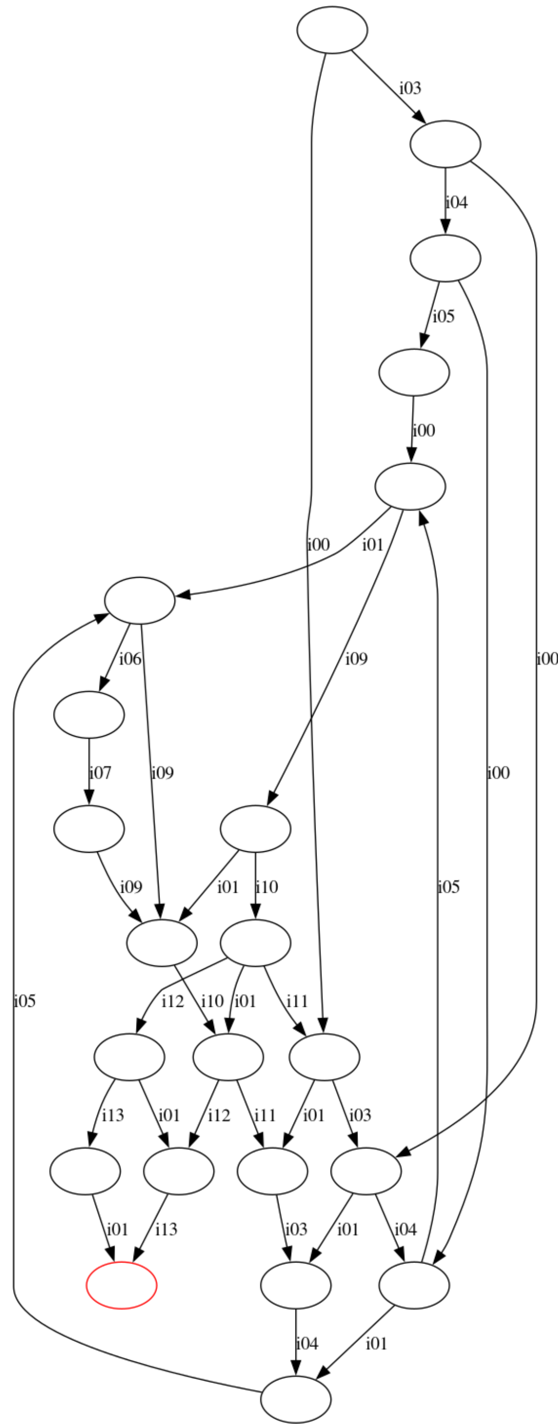


Figure 5.5: Single Thread Graphical Representation of Peterson's Algorithm Using Release Acquire With Pseudo Instructions Removed

```
@flag_0 = dso_local global i32 0, align 4
@flag_1 = dso_local global i32 0, align 4
@turn = dso_local global i32 0, align 4
@critical = dso_local global i32 0, align 4

define dso_local void @thread1() #0 {
  store atomic i32 1, ptr @flag_0 release, align 4
  store atomic i32 1, ptr @turn release, align 4
  fence acq_rel
  br label %1
1:
  %2 = load atomic i32, ptr @flag_1 acquire, align 4
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %4, label %7
4:
  %5 = load atomic i32, ptr @turn acquire, align 4
  %6 = icmp eq i32 %5, 1
  br label %7
7:
  %8 = phi i1 [ false, %1 ], [ %6, %4 ]
  br i1 %8, label %9, label %10
9:
  br label %1, !llvm.loop !6
10:
  store i32 19, ptr @critical, align 4
  store atomic i32 0, ptr @flag_0 release, align 4
  ret void
}
```

Figure 5.6: Corrected LLVM Release Acquire Version of Peterson's Algorithm

5.3 Peterson's Algorithm - Modified Release-Acquire Version

To modify Peterson's Algorithm to work correctly with release-acquire, an additional fence has to be added to ensure the two problematic instructions are not moved after the check. The modified code can be seen in Figure 5.6 and the resulting graph in Figure 5.7. The graph shows that the $i00$ and $i01$ instructions can no longer be moved after the critical section, and the analyzer run over the Kronecker Sum of the two threads detects no errors.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

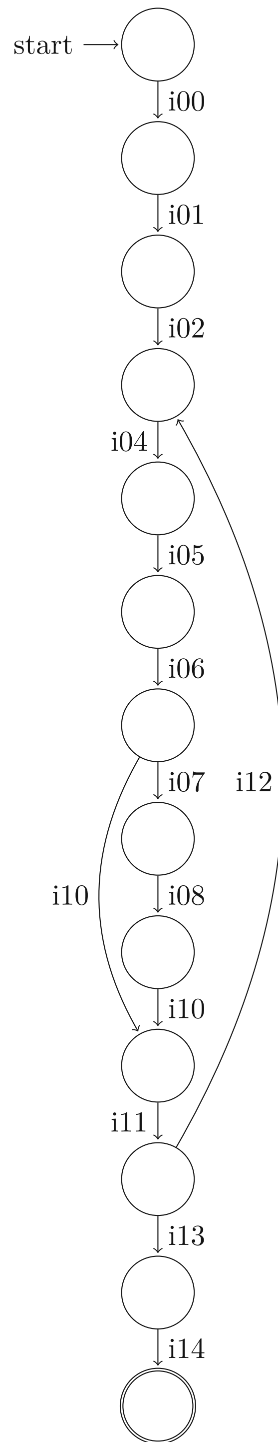


Figure 5.7: Single Thread of Corrected Peterson's Algorithm Using Release-acquire With Pseudo Instructions Removed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.4 Dekker's Algorithm - Sequential Consistent Version

Dekker's Algorithm is another synchronization algorithm known to work under sequential consistency. For the prototype to be able to operate on it, the code again has to be modified to use two separate variables, $flag_0$ and $flag_1$, instead of an array. The resulting C code can be seen in Figure 5.8 and its LLVM equivalent in Figure 5.9.

Dependencies in the first thread caused by sequential instructions are:

i00 – b01	i02 – i03	i02 – i04	i02 – b02	i02 – b08	i00 – i02	i16 – i02
i05 – i06	i05 – i07	i05 – b03	i05 – b07	i04 – i05	i03 – i05	i02 – i05
i08 – b04	i07 – i08	i06 – i08	i05 – i08	i10 – i11	i10 – i12	i10 – b05
i10 – b06	i08 – i10	i13 – i10	i14 – b07	i12 – i14	i11 – i14	i10 – i14
i18 – i19	i17 – i18	i04 – i18	i03 – i18	i02 – i18		

Dependencies caused by dependency chains are:

i02 – i03	i03 – i04	i05 – i06	i06 – i07	i10 – i11	i11 – i12

Dependencies caused by reading memory another instruction writes to, as well as dependencies caused by writing to memory other reads from:

i00 – i08	i14 – i08	i08 – i14	i05 – i18	i10 – i18	i00 – i19	i14 – i19

The resulting single-thread graph output by the prototype can be seen in Figure 5.10.

Again, as expected, the analyzing part of the prototype does not detect any errors.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
atomic_int turn = 0;
atomic_int flag_0 = 0;
atomic_int flag_1 = 0;
int critical = 0;

void thread1() {
    flag_0 = 1;
    while (flag_1) {
        if (turn == 1) {
            flag_0 = 0;
            while (turn == 1);
            flag_0 = 1;
        }
    }

    critical = 15;

    turn = 1;
    flag_0 = 0;
}
```

Figure 5.8: One Thread of Dekker's Algorithm in C

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
@flag_0 = dso_local global i32 0, align 4
@flag_1 = dso_local global i32 0, align 4
@turn = dso_local global i32 0, align 4
@critical = dso_local global i32 0, align 4

define dso_local void @thread1() #0 {
  store atomic i32 1, ptr @flag_0 seq_cst, align 4
  br label %1
1:
  %2 = load atomic i32, ptr @flag_1 seq_cst, align 4
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %4, label %14
4:
  %5 = load atomic i32, ptr @turn seq_cst, align 4
  %6 = icmp ne i32 %5, 0
  br i1 %6, label %7, label %13
7:
  store atomic i32 0, ptr @flag_0 seq_cst, align 4
  br label %8
8:
  %9 = load atomic i32, ptr @turn seq_cst, align 4
  %10 = icmp ne i32 %9, 0
  br i1 %10, label %11, label %12
11:
  br label %8, !llvm.loop !6
12:
  store atomic i32 1, ptr @flag_0 seq_cst, align 4
  br label %13
13:
  br label %1, !llvm.loop !8
14:
  store i32 15, ptr @critical, align 4
  store atomic i32 1, ptr @turn seq_cst, align 4
  store atomic i32 0, ptr @flag_0 seq_cst, align 4
  ret void
}
```

Figure 5.9: One Thread of Dekker's Algorithm in LLVM

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

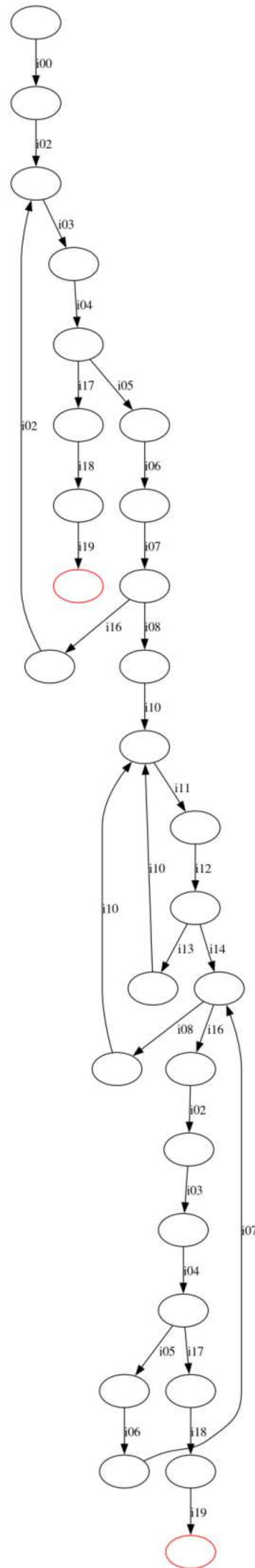


Figure 5.10: Graphical Representation of One Thread of Dekker's Algorithm

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.5 Dekker's Algorithm - Release-Acquire Version

Dekker's Algorithm can also be converted to release-acquire like Peterson's Algorithm. The resulting LLVM code can be seen in Figure 5.11. The critical instructions are $i17$ and $i38$.

This time, the analyzer again outputs multiple error paths; the shortest (with pseudo instructions removed) is:

$i02, i03, i04, i23, i24, i25, i17, i38$

The problem is again very similar to the one in the release-acquire version of Peterson's Algorithm. The $i00$ instruction that should set the global $flag_0$ is delayed till after the critical section. Again, this could happen in the real application since no instruction interacts with $flag_0$ between $i00$ and the critical instruction $i17$ except for $i08$, which is bypassed in this execution order. This means no memory dependencies are enforcing the instruction to be executed earlier. Furthermore, no release instruction executed on that path would require $i00$ not to be moved after it. This means delaying $i00$ is possible, and the error was detected correctly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
@flag_0 = dso_local global i32 0, align 4
@flag_1 = dso_local global i32 0, align 4
@turn = dso_local global i32 0, align 4
@critical = dso_local global i32 0, align 4

define dso_local void @thread1() #0 {
  store atomic i32 1, ptr @flag_0 release, align 4
  br label %1
1:
  %2 = load atomic i32, ptr @flag_1 acquire, align 4
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %4, label %14
4:
  %5 = load atomic i32, ptr @turn acquire, align 4
  %6 = icmp ne i32 %5, 0
  br i1 %6, label %7, label %13
7:
  store atomic i32 0, ptr @flag_0 release, align 4
  br label %8
8:
  %9 = load atomic i32, ptr @turn acquire, align 4
  %10 = icmp ne i32 %9, 0
  br i1 %10, label %11, label %12
11:
  br label %8, !llvm.loop !6
12:
  store atomic i32 1, ptr @flag_0 release, align 4
  br label %13
13:
  br label %1, !llvm.loop !8
14:
  store i32 15, ptr @critical, align 4
  store atomic i32 1, ptr @turn release, align 4
  store atomic i32 0, ptr @flag_0 release, align 4
  ret void
}
```

Figure 5.11: LLVM Version of Dekker's Algorithm Using Release Acquire

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

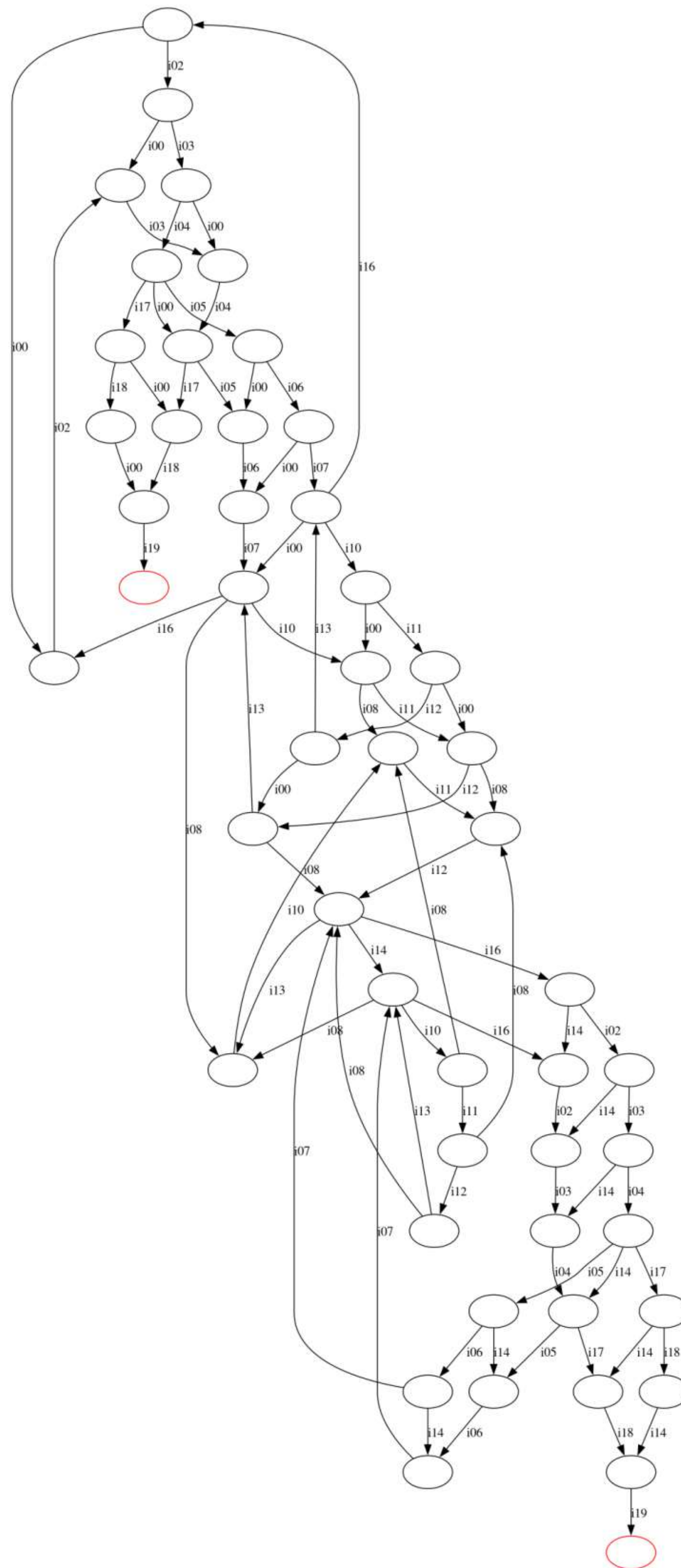


Figure 5.12: Single Thread Graphical Representation of Dekker's Algorithm Using Release Acquire With Pseudo Instructions Removed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.6 Dekker's Algorithm - Modified Release-Acquire Version

Modifying Dekker's Algorithm to work using the release-acquire model only requires one additional fence. The modified code can be seen in Figure 5.13 and the resulting graph in Figure 5.14. The graph shows that the $i00$ instruction is now guaranteed to be executed first, and the analyzer runs over the Kronecker Sum of the two threads and detects no errors.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
@flag_0 = dso_local global i32 0, align 4
@flag_1 = dso_local global i32 0, align 4
@turn = dso_local global i32 0, align 4
@critical = dso_local global i32 0, align 4

define dso_local void @thread1() #0 {
  store atomic i32 1, ptr @flag_0 release, align 4
  fence acq_rel
  br label %1
1:
  %2 = load atomic i32, ptr @flag_1 acquire, align 4
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %4, label %14
4:
  %5 = load atomic i32, ptr @turn acquire, align 4
  %6 = icmp ne i32 %5, 0
  br i1 %6, label %7, label %13
7:
  store atomic i32 0, ptr @flag_0 release, align 4
  br label %8
8:
  %9 = load atomic i32, ptr @turn acquire, align 4
  %10 = icmp ne i32 %9, 0
  br i1 %10, label %11, label %12
11:
  br label %8, !llvm.loop !6
12:
  store atomic i32 1, ptr @flag_0 release, align 4
  br label %13
13:
  br label %1, !llvm.loop !8
14:
  store i32 15, ptr @critical, align 4
  store atomic i32 1, ptr @turn release, align 4
  store atomic i32 0, ptr @flag_0 release, align 4
  ret void
}
```

Figure 5.13: Corrected LLVM Release Acquire Version of Dekker's Algorithm

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

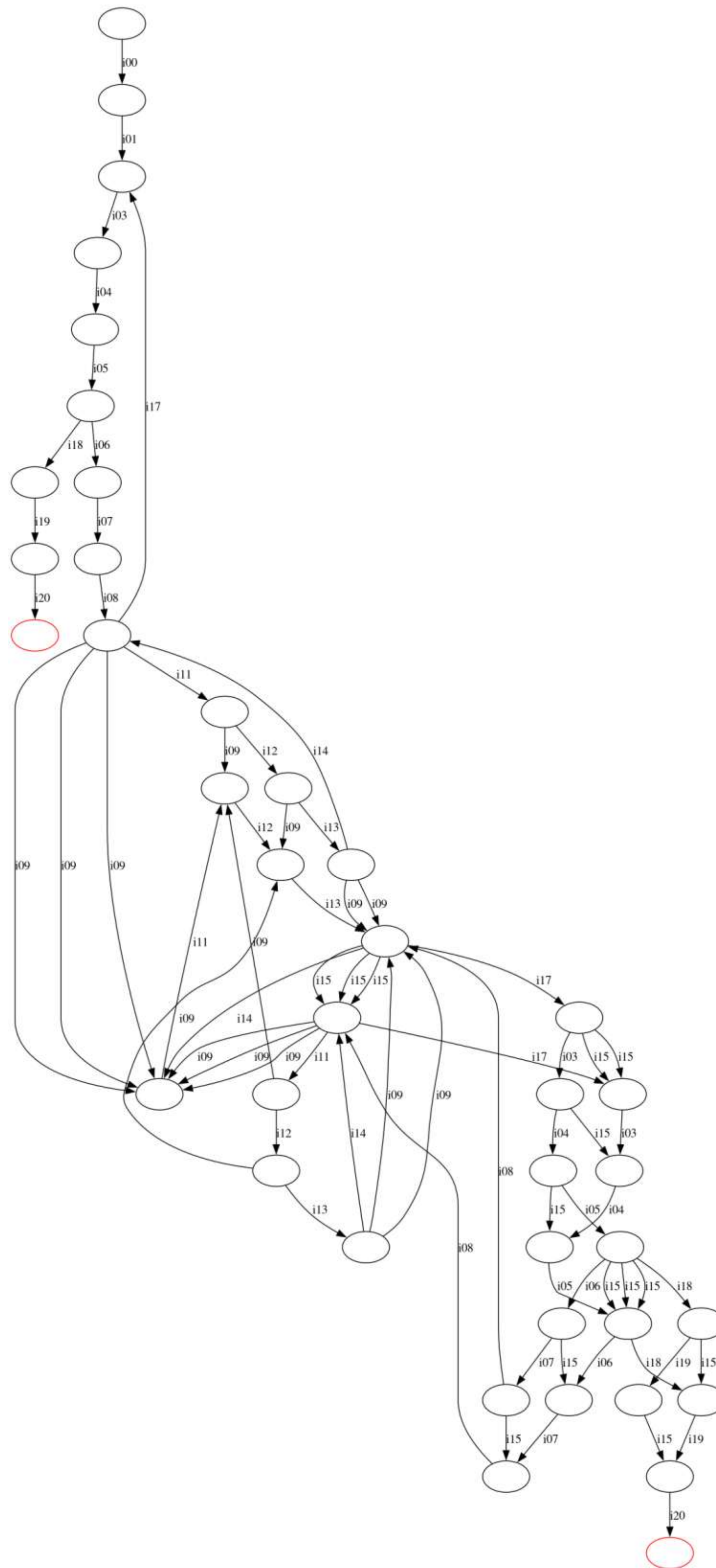


Figure 5.14: Single Thread of Corrected Dekker's Algorithm Using Release Acquire With Pseudo Instructions Removed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Related Work

As mentioned in Chapter 1, many approaches already exist to verify concurrent programs in weak memory models. In the following, some will be presented.

6.1 Dynamic Partial Order Reduction

One way of finding concurrency bugs is with stateless model checkers [AAJN18]. A stateless model checker explores all possible combinations of thread scheduling and detects possible concurrency bugs.

One big problem of stateless model checkers is that the exponentially growing number of possible scheduling orders requires much time. For this reason, stateless model checkers use different ways to reduce the amount of thread scheduling combinations.

Partial order reduction is one such technique. It divides different execution orders into equivalent classes, with each order in an equivalent class having all statements that interact with data of other threads executed in the same order. Since all members of one equivalence class execute similarly, only one of the orders in the equivalence class has to be checked.

For weak memory models, one equivalence class is called a Shasha-Snir trace. In such a trace, there are three different kinds of relations statements can have:

- program order: gives a total order for the events for a thread
- coherence: gives a total order for the writes to a shared variable
- read-from: connects every write to all the reads it depends on

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Different weak memory models use different kinds of relations differently. Even though this strongly reduces the amount of analyzed execution orders, it is far from optimal. The result might still be equivalent in cases where only coherence relations differ.

The paper focuses on an algorithm that weakens Shasha-Snir traces, so even fewer orders must be explored.

6.2 Iris

Iris [KDD+17] is a framework for higher-order concurrent separation logic. It allows users to customize the framework to suit any language featuring operational interleaving semantics, and it can then reason about programs using the logic selected.

Both GPS and RSL from the following sections can be encoded into Iris and used for program verification for programs using the Acquire-Release memory models.

6.2.1 GPS

GPS (or Ghosts, Protocols, and Separation) is a program logic designed to be a full suite of modern verification techniques for reasoning over weak memory models [TVD14]. Those verification techniques include ghost states, protocols, and separation logic. In the following, these three are explained.

Protocol

When threads interact, reasoning about the interference between them must be done. One common way to reason about them is invariants, meaning rules that hold over the entire duration of the interaction. One of the mechanisms used is called 'rely-guarantee,' and the 'rely on' is the state transitions the thread might perform and guaranteeing the environment's transitions.

For this, GPS supports protocols similar to concurrent logic protocols that are modified to be sound when under weak memory models.

Ghost States

Ghost states are used to enforce the program's correct behavior. Control flow, permissions, and execution history could otherwise not be properly handled.

The PL protocol already has a built-in form of ghost states that can, for example, summarize previous control flow. However, to support ownable/permissions resources, a system called ghosts in the form of partial commutative monoids is needed.

Separation and Ownership

Many parts of concurrent programs are still independent of each other, meaning many parts of a thread can be executed in parallel with parts of a different thread without the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

two threads having to interact since both are using (owning) different resources. Many program logics allow this ownership of resources to be transferred to other threads.

The GPS PL protocol also allows threads to transfer ownership of resources, but with the restriction, the thread receiving the owned resources must operate to synchronize. This is done to achieve soundness of the program logic.

6.2.2 Relaxed Separation Logic

The paper introduces a program logic to reason about concurrent programs running under the relaxed memory model of C11 [VN13]. The programming language is developed on top of concurrent separation logic, a program logic with the restriction that they only allow a thread to access data not owned by the thread using atomic operators with limited options to change ownership of data. This prevents data races.

Relaxed separation logic extends concurrent separation logic by allowing different ownership changes dependent on the used atomic. It adds proof rules for the different C11 atomic operators.

Relaxed separation logic is used for verification in the Linux kernel project [TDV15].

6.3 Context Bound Analysis

The following sections focus on a group of related papers that explore the use of Context Bound Analysis to reduce concurrent programs to a sequential one to apply analysis techniques that are known to work on sequentially consistent programs to detect errors in concurrent ones.

6.3.1 Context Bound to Sequential Analysis

This paper focuses on a reduction from a concurrent program to an equivalent sequential program that models all executions for a fixed amount of context switches [LR09].

It achieves this by dividing the threads into ordered chunks of execution contexts; within one execution context, only one thread is executed, and after that execution context, it switches to the following execution context or the next of a different thread. This switch is referred to as a context switch.

To be more precise, if a program has two threads (T_1, T_2). Both of those threads are divided into up to K chunks ($c_0^1, c_1^1, \ldots, c_K^1$ for T_1 and $c_0^2, c_1^2, \ldots, c_K^2$ for T_2). Then T_1 starts executing from c_0^1 for the first m chunks. After this, it switches control to T_2, which now starts executing from c_0^2, after which control is given back to T_1, and execution continues at c_m^1.

Before each swap, the local state is saved, and if the same thread resumes, it continues using the same local state but with all changes to global data made by other threads.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Due to the local state having to be remembered during the execution of other threads and the exponential number of orders, the different thread chunks can be executed, so this way of analysis takes much memory.

To reduce the amount of memory needed, the paper suggests instead changing the thread T_1 so more chunks can be executed before a view switch occurs. This is done by analyzing the effects of chunks (in the above example, the chunks $c_0^2 - c_n^2$) might have on the execution starting at c_m^1. This way, those effects can be applied to thread T_1 without switching to T_2 and, therefore, having to save the local state. Those assumed effects of T_2 on T_1 can be verified when executing T_2.

6.3.2 K-Bounded Computations

While the paper in 6.3.1 can reduce the memory required for verification, it only applies to multithreaded programs with a fixed, finite, and known amount of threads.

To solve this problem, K-bounded computation is introduced. K-bounded computation restricts the amount of context-switches of each thread to K but allows for the dynamic creation of new threads during execution [ABQ09]. Further, the amount of context switches of newly created threads is limited to the amount of context switches of the thread creating it reduced by 1.

6.3.3 VBMC

This paper [AAAK19] presents a prototype for verifying concurrent programs that use release-acquire semantics. Since the state reachability problem is still undecidable under release-acquire, a new version of context bounding, called view-bounding, is presented.

Since every thread in release acquire has its view on the system's current state due to writes not necessarily being synchronized with others immediately.

The prototype is a polynomial-time code-to-code translation between a program that uses release-acquire semantics and a second program that holds that for every bounded view-switching execution of the original program, there is a context-bound execution for the second one under sequential consistency.

This results in a reduction to a context-bounded state reachability problem for sequential consistency and allows for existing verification tools that assume sequential consistency to be used to verify concurrent programs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

CHAPTER 7

Discussion

The first research question of this thesis asked if Kronecker Algebra effectively captured the interactions between the threads of concurrent programs using the release acquire model. In the prototype, this was not achieved. Because of the explosion of time complexity, which will be explained more closely below, the prototype reduces its use of Kronecker Algebra to the creation of a control flow graph that models all possible execution orders of a singular thread.

But this does not mean that modelling the interactions between the threads is not possible if the prototype is given more time and/or the Kronecker Calculations are further optimized. In Section 8 there will be a short overview of how the system might be expanded so it also models the interactions between the threads.

The next question was to what extent Kronecker Algebra can assist in detecting and resolving data races in programs using the release acquire model. As seen in Section 5 the prototype successfully uses the generated control flow graph to verify basic algorithms and can help a user identify possible race conditions and therefore help with resolving such problems.

The last question was about which challenges and limitations are faced when trying to verify programs using Kronecker Algebra. As explained in Section 3.1.2 the system used in the prototype has the possibility of false negatives. While this type of false negative might be able to be fixed by modifying the way Kronecker Algebra is used to generate the Control Flow graph, a second problem, state explosion, is not that easily fixable.

As will be shown in Section 7.1, the time needed for the Kronecker Calculations is the biggest part of the total time needed for verification.

For this reason, while having a mathematical way to calculate the possible execution orders of a program is interesting, it seems unlikely it will be used outside of the verification of simple algorithms.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

7.1 Time

Table 7.1 shows the required time for the different examples. The tests were performed using an i5-1340P. The first thing it shows is that even though optimization focuses on reducing the amount of Kronecker Algebra, it still takes up most of the time in all examples. The second thing it shows is that the examples using Dekker's Algorithm take significantly longer than those using Peterson's Algorithm. This is primarily caused by the LLVM-compiled version of Dekker's Algorithm having more instructions and requiring more dependencies to be modeled.

Comparing the times to VBMC [AAAK19], this is significantly slower. Their prototype was able to find errors in the release-acquire versions of Dekker's and Peterson's Algorithms within 0.85 and 0.26 seconds, even though the benchmark was performed on hardware that is outperformed by the one used for the tests in Table 7.1. The benchmarks given in [AAAK19] also include times for further tools, namely those that outperform both VBMC and the prototype in this thesis when detecting errors within two threads of those interacting algorithms.

As was already mentioned in Section 4.7, a version of the prototype using lazy calculation was also created, but as can be seen in Table 7.2, this increases the time required for the calculation further.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

TU Bibliothek Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

example	peterson	r-a peterson	correct r-a peterson	dekker	r-a dekker	correct ra-dekker
dependencies (ms)	9.68	9.29	8.55	9.29	9.65	10.45
kronecker (s)	1.52	1.74	1.70	18.73	19.80	27.44
analysis (ms)	73.7	176.86	65.31	42.48	99.44	65.72
total (s)	1.60	1.93	1.78	18.78	19.91	27.52

Table 7.1: Required Time to Analyze the Examples

example	peterson	r-a peterson	correct r-a peterson	dekker	r-a dekker	correct ra-dekker
kronecker (s)	2.27	2.48	2.66	55.41	57.20	70.85

Table 7.2: Required Time for Kronecker Calculations using Lazy Calculations with a Balanced Tree

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 8

Conclusion and Future Work

In the thesis, we developed a way that uses Kronecker Algebra to build a Control Flow Graph of a program running under weak memory models that an interpreter can then use to detect data races within the program. The system was implemented in a prototype that takes a program written in LLVM bytecode. It correctly verified the correctness of Dekker's and Peterson's Algorithms under sequential consistency and detected errors in their trivial release-acquire form.

In the prototype developed, the exponential growth in the time needed to calculate the graph results in testing bigger algorithms already taking a significant amount of time, with testing entire subsections of a program within a reasonable time being impossible.

In future work, the maximum size of the tested code can most likely be increased by further optimizing the Kronecker calculations. As mentioned in Section 4.7, the library is most efficient when dealing with flat trees, and the main advantage of the used system only applies if it is working with very narrow trees. Since the order of Kronecker Sums has no influence on the result, the Kronecker Sums to create the blocks or, if the optimization that matrices of blocks are created in code and not calculated, the Kronecker Sums to merge the blocks into one graph can be optimized.

The library also provides ways of using caching calculated values to minimize recalculating them in the future. Thus it might be possible to reduce the time needed if values in key locations within the program are cached.

Furthermore, currently, the system uses Kronecker Algebra exclusively to build the program's Control Flow Graph by adding a single label per instruction to the graph. Instead of adding multiple versions of instruction, for example, in the case of conditional instructions, add one label representing the instruction evaluating to true and one in which it evaluates to false. One can then use dependencies to ensure that, depending on the last instruction assigned to the variable and the value it assigns to, only the label representing the conditional instruction evaluating the correct value is used.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

While such a system would most likely be capable of verifying programs using Kronecker Algebra and without the need for the interpreter, due to the inefficiency of Kronecker Algebra, this would reduce the program's size, and it can verify within a reasonable time even more.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

1.1 Race Condition Caused by Delayed Synchronization 2
1.2 Example How Optimization Might Influence Execution Order 2

2.1 Example Read-Modify Race Condition 11
2.2 Example Check-Then-Act Race Condition 12
2.3 C Implementation of Peterson's and Dekker's Algorithm (*turn* and *flag* Must Be Sequentially Consistent Atomics) 14
2.4 FSM F_0 15
2.5 FSM F_1 17
2.6 Result of the Kronecker Product of the Automata Shown in 2.4 and 2.5 . 18
2.7 Result of the Kronecker Product of the Automata Shown in 2.4 and 2.5 Without Unreachable Nodes 18
2.8 FSM F_2 21
2.9 FSM F_3 21
2.10 Result of the Synchronized Kronecker Product of the Automata F_2 and F_3 21
2.11 FSM F_4 22
2.12 FSM F_5 23
2.13 Result of Synchronized Kronecker Product of the Automata F_5 and F_3 . . 23
2.14 Examples FSM for Kronecker Sum 25
2.15 Result of Kronecker Sum Between F_6 and F_7 25
2.16 Examples FSM for Kronecker Skip 26
2.17 Result of Kronecker Skip Between F_8 and F_9 27

3.1 Example Creation of Blocks 32
3.2 Block Conversion with Loop 34
3.3 Example of a Loop That Can Be Modelled in Matrix Form 35
3.4 Examples of Loops Not Easily Modeled 36
3.5 Different Dependency Examples 38
3.6 Example of Applied Dependencies in Program With Rule 40
3.7 Visualization of Figure 3.5c With Applied Simple Rules 42

4.1 Example Where Kronecker Sum of Threads is Not Enough 52

5.1 One Thread of Peterson's Algorithm in C 56
5.2 LLVM Version of Peterson's Algorithm 57

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.3 Single Thread Graphical Representation of Peterson's Algorithm With Pseudo Instructions Removed 58
5.4 LLVM Version of Peterson's Algorithm Using Release Acquire 60
5.5 Single Thread Graphical Representation of Peterson's Algorithm Using Release Acquire With Pseudo Instructions Removed 61
5.6 Corrected LLVM Release Acquire Version of Peterson's Algorithm 62
5.7 Single Thread of Corrected Peterson's Algorithm Using Release-acquire With Pseudo Instructions Removed 63
5.8 One Thread of Dekker's Algorithm in C 65
5.9 One Thread of Dekker's Algorithm in LLVM 66
5.10 Graphical Representation of One Thread of Dekker's Algorithm 67
5.11 LLVM Version of Dekker's Algorithm Using Release Acquire 69
5.12 Single Thread Graphical Representation of Dekker's Algorithm Using Release Acquire With Pseudo Instructions Removed 70
5.13 Corrected LLVM Release Acquire Version of Dekker's Algorithm 72
5.14 Single Thread of Corrected Dekker's Algorithm Using Release Acquire With Pseudo Instructions Removed 73

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

List of Tables

2.1 List of Important LLVM Instructions [llva], (Optional Values Are Not Included in Usage) . . . 8
2.2 List of Possible ICMP Operators . . . 9

4.1 Average Execution Time in Seconds of Library for n Kronecker Skips . . . 51
4.2 Average Execution Time in Seconds of Custom Calculation for n Kronecker Skips . . . 51
4.3 Average Execution Time in Seconds of Library for n Kronecker Sums . . . 51
4.4 Average Execution Time in Seconds of Custom Calculation for n Kronecker Sums . . . 51

7.1 Required Time to Analyze the Examples . . . 81
7.2 Required Time for Kronecker Calculations using Lazy Calculations with a Balanced Tree . . . 81

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[AAAK19] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. Verification of programs under the release-acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1117–1132, New York, NY, USA, 2019. Association for Computing Machinery.

[AAC+12] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under tso. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 204–219, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[AAJN18] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[ABQ09] Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 107–123, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[BB14] Bernd Burgstaller and Johann Blieberger. Kronecker algebra for static analysis of Ada programs with protected objects. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 27–42. Springer, 2014.

[Bli15] Johann Blieberger. The Kronecker Booklet. `https://kronalg.blieberger.at/kb.pdf`, 2015.

[BM08] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 107–120, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Dij62] Edsger Wybe Dijkstra. Over de sequentialiteit van procesbeschrijvingen. 1962.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[KDD+17] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[llva] llvm. Llvm language reference manual.

[llvb] llvm. llvm-project. `https://github.com/llvm/llvm-project/tree/main`.

[LR09] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35:73–97, 2009.

[NM92] Robert HB Netzer and Barton P Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12:115–116, 1981.

[TDV15] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. *SIGPLAN Not.*, 50(6):110–120, jun 2015.

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, page 691–707, New York, NY, USA, 2014. Association for Computing Machinery.

[VN13] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for c11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 867–884, New York, NY, USA, 2013. Association for Computing Machinery.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.